

Министерство образования Республики Беларусь
Белорусский государственный университет
Механико-математический факультет
Кафедра веб-технологий и компьютерного моделирования

СОГЛАСОВАНО

Заведующий кафедрой

М.В. Игнатенко

«24» февраля 2026 г.

СОГЛАСОВАНО

Декан факультета

С.М. Босяков

«24» февраля 2026 г.

Технологии JavaEE

Электронный учебно-методический комплекс для специальности:
7-06-0533-04 «Математика и компьютерные науки»

Регистрационный № 2.4.3-24 / 779

Автор:

Блинов И.Н., кандидат физико-математических наук, доцент.

Рассмотрено и утверждено на заседании Совета Механико-математического факультета. Протокол № 6 от 24.02.2026 г.

Минск 2026

УДК 004.432.045:004.738.5JavaEE(075.8)
Б 695

Утверждено на заседании Научно-методического совета БГУ
Протокол № 7 от 27.02.2026 г.

Автор:

Блинов Игорь Николаевич, кандидат физико-математических наук, доцент, доцент кафедры веб-технологий и компьютерного моделирования механико-математического факультета БГУ

Рецензенты:

Серебряная Л.В., доцент, зав. кафедрой Экономической информатики факультета Цифровых технологий УО «Белорусский государственный экономический университет», кандидат технических наук

Левчук В.Д., доцент кафедры Программного обеспечения информационных технологий факультета компьютерных систем и сетей УО «Белорусский государственный университет информатики и радиоэлектроники», кандидат технических наук

Блинов, И. Н. Технологии JavaEE: электронный учебно-методический комплекс для специальности 7-06-0533-04 «Математика и компьютерные науки» / И. Н. Блинов ; БГУ, Механико-математический фак., Каф. веб-технологий и компьютерного моделирования. – Минск : БГУ, 2025. – 480 с. : ил. – Библиогр.: с. 480.

Электронный учебно-методический комплекс (ЭУМК) по учебной дисциплине «Технологии JavaEE» предназначен для магистрантов специальности 7-06-0533-04 «Математика и компьютерные науки». ЭУМК содержит тексты лекций, планы лабораторных занятий, перечень контрольных вопросов, списки рекомендованной литературы.

СОДЕРЖАНИЕ

Пояснительная записка	8
1. Теоретический раздел	11
1.1. Архитектура платформы Jakarta EE.....	11
1.1.1. История: от J2EE к Jakarta EE	12
1.1.2. Цели и Философия Jakarta EE	13
1.1.3. Управление Сообществом: Eclipse Foundation.....	13
1.1.4. Обзор Текущих Версий и Спецификаций.....	13
1.1.5. Заключение.....	15
1.2. Архитектура платформы Jakarta EE: Контейнерная модель	16
1.2.1. Web-контейнер (Web Container)	16
1.2.2. EJB-контейнер (EJB Container)	20
1.2.3. Application Client-контейнер (Application Client Container, ACC).....	23
1.3. Серверы приложений Jakarta EE: Обзор, установка, базовая конфигурация	24
1.3.1. WildFly (и JBoss EAP).....	24
1.3.2. GlassFish	26
1.3.3. Payara Server.....	27
1.3.4. Open Liberty.....	28
1.3.5. Apache TomEE.....	30
1.3.6. Заключение: Какой выбрать?	31
1.4. Профили, архитектура, паттерны и инструменты Jakarta EE	32
1.4.1. Профили Jakarta EE (Profiles).....	32
1.4.2. Многоуровневая архитектура корпоративных приложений.....	33
1.4.3. Основные паттерны проектирования в Jakarta EE	34
1.4.4. Инструменты разработки.....	37
1.5. Сервлеты	40
1.5.1. Запуск контейнера сервлетов и размещение проекта.....	47
1.5.2. Простая JSP–страница.....	50
1.5.3. Взаимодействие сервлета и JSP	52
1.5.4. Интерфейс ServletContext	57
1.5.5. Интерфейс HttpServletRequest.....	61
1.5.6. Интерфейс HttpServletResponse	65
1.5.7. Атрибуты и параметры	65
1.5.8. Многопоточность и электронная почта	66
1.6. Java Server Page	73
1.6.1. Жизненный цикл.....	75
1.6.2. Неявные объекты в expression language	78
1.6.3. Стандартные элементы action.....	81
1.6.4. Expression Language.....	86

1.6.5. EL операторы	92
1.6.6. Обработка ошибок	94
1.6.7. Взаимодействие	96
1.7. Сессии, события и фильтры.....	110
1.7.1. Сеанс (сессия).....	110
1.7.2. Файлы Cookie.....	116
1.7.3. Обработка событий	119
1.7.4. Фильтры.....	126
1.8. JSP Standard Tag Library	134
1.8.1. Автоматическое приведение типов и перехват исключений	140
1.8.2. Исключающие условия <c:choose>	143
1.8.3. Итераторы <c:forEach> и <c:forEachTokens>	144
1.8.4. Включение ресурсов	148
1.8.5. Динамические адреса и перенаправление.....	150
1.8.6. JSTL formatting.....	152
1.8.7. JSTL sql.....	159
1.8.8. JSTL xml	160
1.8.9. JSTL functions.....	166
1.8.10. Jakarta RESTful Web Services (JAX-RS)	168
1.9. Ядро корпоративных компонентов.....	181
1.9.1. Jakarta Contexts and Dependency Injection (CDI).....	181
1.9.2. Jakarta Enterprise Beans (EJB)	194
1.10. Управление данными и транзакциями	199
1.10.1. Концепция Object-Relational Mapping (ORM)	199
1.10.2. Jakarta Persistence (JPA): Сущности и базовые отображения ..	205
1.10.3. Первичные ключи (@Id и @GeneratedValue)	207
1.10.4. Базовые типы и их отображение.....	209
1.10.5. EntityManager и Persistence Context	212
1.10.6. Жизненный цикл сущностей	213
1.10.7. Сводная таблица по операциям и состояниям.....	217
1.10.8. Способы выполнения запросов в JPA	217
1.10.9. JPQL (Jakarta Persistence Query Language)	218
1.10.10. Criteria API.....	221
1.10.11. Native Queries (Нативные запросы)	222
1.10.12. Заключение: что когда использовать?.....	225
1.10.13. Отношения между сущностями в JPA	225
1.10.14. @ManyToOne (Многие-к-Одному).....	226
1.10.15. @OneToMany (Один-ко-Многим)	228
1.10.16. @OneToOne (Один-к-Одному).....	230
1.10.17. @ManyToMany (Многие-ко-Многим).....	232
1.10.18. Важные концепции: Cascade и FetchType	234
1.10.19. Наследование сущностей (Entity Inheritance)	236

1.10.20. Встраиваемые объекты (@Embeddable).....	242
1.10.21. Управление параллелизмом: Блокировки в JPA	245
1.10.22. Оптимистическая блокировка (Optimistic Locking).....	246
1.10.23. Пессимистическая блокировка (Pessimistic Locking)	248
1.10.24. Сравнительная таблица.....	251
1.10.25. Интеграция JPA с CDI и EJB.....	252
1.10.26. Популярные реализации JPA	257
1.10.27. Jakarta Transactions (JTA): Концепция транзакций. ACID	259
1.10.28. Управление транзакциями в Jakarta EE.....	265
1.10.29. Container-Managed Transactions (CMT) — Управление контейнером	265
1.10.30. Bean-Managed Transactions (BMT) — Ручное управление.....	268
1.10.31. Двухфазный коммит (2PC) и Распределенные транзакции ...	269
1.11. Интеграция и обмен сообщениями	273
1.11.1. Интеграция и обмен сообщениями: Jakarta Messaging (JMS)..	273
1.11.2. Модели обмена сообщениями в Jakarta Messaging (JMS)	278
1.11.3. Модель Point-to-Point (PTP) с использованием Очереди (Queues)	278
1.11.4. Модель Publish/Subscribe (Pub/Sub) с использованием Топиков (Topics).....	282
1.11.5. Сводная таблица различий	285
1.11.6. Основные компоненты JMS API.....	286
1.11.7. Типы сообщений в Jakarta Messaging (JMS).....	293
1.11.8. TextMessage.....	294
1.11.9. ObjectMessage.....	296
1.11.10. MapMessage.....	301
1.11.11. Транзакции в JMS	302
1.11.12. Управление контейнером (CMT) с JTA	303
1.11.13. Управление сессией (Session-Transacted).....	306
1.11.14. Заключение.....	308
1.11.15. Message-Driven Beans (MDB)	308
1.11.16. Пример: Система обработки заказов	311
1.11.17. Поток выполнения	315
1.11.18. Конфигурация MDB: @ActivationConfigProperty	316
1.11.19. Базовая конфигурация: указание источника сообщений	317
1.11.20. Конфигурация для работы с Топиками (Publish/Subscribe) ...	318
1.11.21. Управление подтверждением (Acknowledgement).....	321
1.11.22. Свойства, специфичные для брокера	321
1.11.23. Заключение.....	323
1.11.24. Жизненный цикл Message-Driven Bean (MDB).....	323
1.11.25. Схема жизненного цикла	326
1.11.26. Выводы и лучшие практики	327

1.11.27. Jakarta Connectors (JCA): Интеграция с EIS.....	328
1.11.28. Пример: Как это выглядит на практике	330
1.11.29. Выводы	333
1.12. Безопасность и валидация данных.....	333
1.12.1. Jakarta Security: Основные концепции	333
1.12.2. Примеры реализации в Jakarta EE	335
1.12.3. Современные механизмы аутентификации в Jakarta Security .	341
1.12.4. @BasicAuthenticationMechanism.....	342
1.12.5. @FormAuthenticationMechanism	343
1.12.6. @CustomFormAuthenticationMechanism	348
1.12.7. Создание своего собственного механизма.....	349
1.12.8. Заключение.....	350
1.12.9. Хранилища идентификационных данных (IdentityStore).....	350
1.12.10. Декларативные хранилища: @DataBaseIdentityStoreDefinition	352
1.12.11. Декларативные хранилища: @LdapIdentityStoreDefinition	355
1.12.12. Программное хранилище: реализация IdentityStore	356
1.12.13. Комбинирование нескольких хранилищ.....	357
1.12.14. Декларативная безопасность в Jakarta EE.....	358
1.12.15. @DeclareRoles.....	359
1.12.16. @RolesAllowed	360
1.12.17. @PermitAll	362
1.12.18. @DenyAll	363
1.12.19. Иерархия и приоритеты	364
1.12.20. Программная безопасность: SecurityContext.....	365
1.12.21. Сценарии использования программной безопасности	366
1.12.22. Заключение.....	371
1.12.23. OWASP Top 10 и защита в Jakarta EE	371
1.12.24. Заключение.....	377
1.12.25. Jakarta Bean Validation: Декларативная валидация	378
1.12.26. Стандартные аннотации ограничений.....	378
1.12.27. Интеграция с другими спецификациями Jakarta EE	380
1.12.28. Программная валидация	383
1.12.29. Стандартные ограничения (Constraints).....	386
1.12.30. Полный пример класса с использованием разных ограничений	389
1.12.31. Создание собственных ограничений (Custom Constraints)	390
1.12.32. Выводы	404
1.12.33. Группы валидации (Validation Groups)	406
1.12.34. Валидация методов и конструкторов (Executable Validation)	409
1.12.35. Интеграция Bean Validation с JPA, JAX-RS и CDI.....	413

1.12.36. Сводка по интеграции	421
1.13. Тестирование JEE приложений	421
1.13.1. Тестирование Jakarta EE приложений: Юнит-тестирование ...	421
1.13.2. Выводы по юнит-тестированию	428
1.13.3. Интеграционное тестирование с Arquillian	429
1.13.4. Выводы	438
1.13.5. Тестирование REST API	438
1.13.6. Заключение и рекомендации	447
1.14. JEE и микросервисная архитектура	448
1.14.1. Eclipse MicroProfile: Обзор спецификаций	448
1.14.2. MicroProfile Config (Конфигурация)	448
1.14.3. MicroProfile Health (Проверка состояния)	450
1.14.4. MicroProfile Metrics (Метрики)	452
1.14.5. MicroProfile Fault Tolerance (Отказоустойчивость)	453
1.14.6. MicroProfile JWT Authentication (Аутентификация по JWT) ...	454
1.14.7. MicroProfile OpenAPI (Документирование API)	455
1.14.8. MicroProfile OpenTracing (Распределенная трассировка)	458
1.14.9. Разработка легковесных Jakarta EE-приложений для микросервисов.....	459
1.14.10. Заключение.....	466
2. Практический раздел	467
2.1. Программа лабораторных занятий.....	467
2.2. Разработка информационной системы	468
3. Раздел контроля знаний.....	472
3.1. Перечень рекомендуемых средств диагностики	472
3.2. Примерный перечень заданий для управляемой самостоятельной работы магистрантов	472
3.3. Описание инновационных подходов и методов к преподаванию учебной дисциплины	472
3.4. Методические рекомендации по организации самостоятельной работы обучающихся.....	472
3.5. Примерный перечень вопросов к зачету.....	473
4. Вспомогательный раздел.....	475
4.1. Рекомендуемая литература	475
4.2. Учебно-методическая карта учебной дисциплины	476
4.3. Электронные ресурсы.....	480

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

Электронный учебно-методический комплекс (ЭУМК) по учебной дисциплине «Технологии JavaEE» предназначен для магистрантов специальности 7-06-0533-04 «Математика и компьютерные науки». Содержание разделов ЭУМК соответствует образовательным стандартам, структуре и тематике типового учебного плана.

Главные цели ЭУМК: помощь магистрантам в организации самостоятельной работы, повышение качества подготовки и усиление практико-ориентированности учебного процесса по дисциплине «Технологии JavaEE».

ЭУМК состоит из следующих разделов.

Теоретический. Включает аннотацию учебного пособия, написанного в соответствии с программой дисциплины. Материал данного пособия может быть использован для самостоятельной подготовки магистрантов к лабораторным занятиям, контрольным работам и зачёту в соответствии с учебной программой Технологии JavaEE учреждения высшего образования по учебной дисциплине для специальности 7-06-0533-04 «Математика и компьютерные науки», № М52-54-41 / уч. [Электронный ресурс]. – Режим доступа: <https://elib.bsu.by/handle/123456789/304759> – Дата доступа: 10.06.2025.

Практический. Содержит планы лабораторных работ. Данные материалы используются для подготовки к лабораторным занятиям и их организации, для самостоятельной работы над курсом.

Раздел контроля знаний представлен вопросами к зачёту.

Описаны формы диагностики и технология определения оценки по дисциплине с учетом текущей успеваемости.

Вспомогательный раздел включает рекомендуемую литературу и содержание учебной программы курса по отдельным темам, на основе которой построено изучение дисциплины и контроль знаний, вопросы к зачёту.

Учебная дисциплина «Технологии JavaEE» предназначен для магистрантов специальности 7-06-0533-04 «Математика и компьютерные науки». В учебной дисциплине изучаются теоретические основы баз данных, рассматриваются вопросы анализа и проектирования информационных систем. Изучается работа с программными средствами автоматизации проектирования информационных систем.

Изучение дисциплины должно способствовать формированию у магистрантов основ алгоритмического мышления и представления о современных подходах к программному решению научных и прикладных задач, позволяет применить знания, полученные при изучении проектирования информационных систем, для практического решения задач проектирования, программирования и использования информационных систем, возникающих в различных областях промышленности и науки.

Цели и задачи учебной дисциплины

Дисциплина «Технологии JavaEE» имеет прикладную направленность.

Цель учебной дисциплины — формирование у магистрантов устойчивых теоретических знаний и практических навыков в области разработки и эксплуатации информационных систем, использования средств автоматизированного проектирования и программных продуктов, реализующих функционирование и управление информационных систем.

Задачи учебной дисциплины

- получение знаний о языке Java;
- углубление знаний о концепциях, положенных в основу шаблонов проектирования программных интернетсистем;
- углубление знаний написания программного кода;
- формирование знаний, необходимых для более углубленного изучения технологий Jakarta EE.

Место учебной дисциплины в системе подготовки специалиста с высшим образованием.

Учебная дисциплина «Технологии JavaEE» относится к государственному компоненту.

Связи с другими учебными дисциплинами, включая учебные дисциплины компонента учреждения высшего образования, дисциплины специализации и др.

Учебная дисциплина «Технологии JavaEE» взаимосвязана с учебной дисциплиной «Технологии программирования».

Требования к компетенциям

Освоение учебной дисциплины «Технологии JavaEE» должно обеспечить формирование следующих компетенций:

базовые профессиональные компетенции:

БПК-3. Применять современные компьютерные математические системы для проведения вычислительного (компьютерного) эксперимента.

БПК-6. Применять современные технологии и базовые конструкции языков программирования для реализации алгоритмических прикладных задач и разработки веб-проектов.

БПК-9. Применять инновационные информационные технологии и современные языки программирования.

специализированные компетенции:

– СК-1.

Осуществлять анализ контекста и поставленной проблемы, аргументированно выбирать оптимальный способ ее решения, согласовывать частичные проекты решения в общую согласованную архитектуру, выполнять реализацию проекта с учетом накопленных и поступающих данных.

В результате изучения дисциплины магистрант должен *знать*:

- основные этапы построения информационных систем.

- принципы функционирования и взаимодействия распределенных систем на платформе JEE.
- передовые технологии разработки в области веб-программирования
- правила использования документации и учебных материалов от авторов языка Java.

Структура учебной дисциплины

Учебная дисциплина «Технологии JavaEE» изучается во 2 семестре дневной формы получения второй ступени высшего образования, во 2 семестре заочной формы получения второй ступени высшего образования для направления специальности 7-06-0533-04 «Математика и компьютерные науки».

На изучение учебной дисциплины «Технологии JavaEE» отводится всего 102 часа, в том числе:

– 68 аудиторных часов, из них лекции – 34 часа, лабораторные занятия – 30 часов, управляемая самостоятельная работа – 4 часа.

Трудоемкость учебной дисциплины составляет 3 зачетных единицы.

Форма текущей аттестации по данной дисциплине:

– зачет во 2 семестре.

Дисциплина изучается в _ семестре заочной форм обучения. На изучение учебной дисциплины «Технологии JavaEE» отводится всего 54 часа, в том числе:

– 12 аудиторных часа, из них лекции – 6 часов, лабораторные занятия – 6 часов.

Трудоемкость учебной дисциплины составляет 2 зачетных единицы.

Форма текущей аттестации по данной дисциплине:

– зачет во 2 семестре.

1. ТЕОРЕТИЧЕСКИЙ РАЗДЕЛ

В теоретическом разделе изложен краткий конспект лекций в соответствии с программой дисциплины.

Данное пособие ориентировано на магистрантов механико-математического факультета БГУ, изучающих современное программирование.

В первой части пособия рассмотрены основы серверных технологий на основе сервлетов.

Вторая часть пособия посвящена изучению JSP (*Java Server Pages*), технологии приема и передачи данных от клиента к серверу.

Третья часть пособия посвящает события, сессии, фильтры.

Четвертая часть пособия посвящена применению библиотек тегов.

1.1. Архитектура платформы Jakarta EE

История, цели, управление сообществом (Eclipse Foundation). Обзор текущих версий спецификаций.

Архитектура платформы Jakarta EE:

Контейнерная модель: Web-контейнер, EJB-контейнер, Application Client-контейнер.

Серверы приложений Jakarta EE (WildFly/JBoss EAP, GlassFish, OpenLiberty, Payara, TomEE и др.): обзор, установка, базовая конфигурация.

Профили Jakarta EE (Full Platform, Web Profile).

Многоуровневая архитектура корпоративных приложений:

Слой представления (Presentation Tier), слой бизнес-логики (Business Tier), слой доступа к данным (Data Tier / EIS Tier).

Отображение технологий Jakarta EE на эти слои.

Основные паттерны проектирования в Enterprise-приложениях:

MVC, Dependency Injection, DAO, DTO, Front Controller, Service Locator, Façade, и др. Их роль в Jakarta EE.

Инструменты разработки:

IDE (IntelliJ IDEA Ultimate, Eclipse for Enterprise Developers), системы сборки (Maven, Gradle) для Jakarta EE проектов.

Jakarta EE (ранее известная как Java Platform, Enterprise Edition или Java EE) — это набор спецификаций и API для создания и развертывания масштабируемых, надежных и безопасных корпоративных приложений на языке Java. Платформа предоставляет стандартную модель для разработки многоуровневых приложений, от веб-интерфейсов до бизнес-логики и доступа к данным.

1.1.1. История: от J2EE к Jakarta EE

Понимание истории Jakarta EE имеет решающее значение для понимания её текущей архитектуры и целей.

Этап 1: J2EE (Java 2 Platform, Enterprise Edition, 1999-2006)

- **Цель:** Стандартизировать разработку корпоративных приложений на Java, чтобы уйти от проприетарных решений от разных вендоров (IBM, BEA и др.).
- **Ключевые технологии:** Servlets, JSP (JavaServer Pages), EJB (Enterprise JavaBeans).
- **Проблема:** Ранние версии J2EE (1.2, 1.3, 1.4) были известны своей сложностью. Они требовали большого количества XML-конфигураций (дескрипторов развертывания), а EJB 2.x были сложны в написании и тестировании. Это привело к репутации платформы как "тяжеловесной" и появлению альтернативных легковесных фреймворков, таких как Spring Framework.

Этап 2: Java EE (Java Platform, Enterprise Edition, 2006-2017)

- **Цель:** Упростить разработку, сделать платформу более дружелюбной к разработчикам.
- **Ключевые изменения (Java EE 5):**
 - **Аннотации вместо XML:** Введение аннотаций (@EJB, @Stateless, @Entity и т.д.) резко сократило необходимость в XML-файлах.
 - **POJO-модель:** EJB 3.0 стали обычными Java-объектами (Plain Old Java Objects), что упростило их создание и тестирование.
 - **JPA (Java Persistence API):** Стандарт для ORM (Object-Relational Mapping), пришедший на смену сложным Entity Beans из EJB 2.x.
- **Дальнейшее развитие (Java EE 6, 7, 8):** Платформа продолжала развиваться, добавляя CDI (Contexts and Dependency Injection) для управления зависимостями, Bean Validation для валидации данных, JAX-RS для создания RESTful веб-сервисов и поддержку современных веб-стандартов (WebSocket, JSON).

Этап 3: Переход к Eclipse Foundation и рождение Jakarta EE (2017-н.в.)

- **Причина:** Компания Oracle, владевшая Java EE, решила передать платформу под управление независимой и нейтральной организации, чтобы ускорить её развитие и сделать процесс более открытым и управляемым сообществом. Такой организацией стал **Eclipse Foundation**.
- **Проблема с названием:** Oracle владеет торговой маркой "Java". Поэтому нельзя было продолжать использовать Java EE. Кроме того, спецификации использовали пакеты javax.*, которые также ассоциировались с брендом Java.
- **Решение:**

1. **Новое имя:** Платформа была переименована в **Jakarta EE**.
2. **Смена пространства имен (namespace):** Это самое важное техническое изменение. Все пакеты `javax.*`, относящиеся к Java EE, были переименованы в `jakarta.*`. Это был необходимый, но болезненный шаг, нарушивший обратную совместимость на уровне исходного кода.

1.1.2. Цели и Философия Jakarta EE

Современная Jakarta EE ставит перед собой следующие цели:

1. **Стандартизация и переносимость:** Предоставить набор стандартных API, чтобы приложения, написанные с их использованием, могли работать на любом совместимом сервере приложений (WildFly, Open Liberty, Payara, GlassFish и др.) без изменений.
2. **Управление сообществом:** Развитие платформы ведется открыто под эгидой Eclipse Foundation. В рабочую группу (Working Group) входят ключевые игроки индустрии: IBM, Red Hat, Oracle, Fujitsu, Payara и другие.
3. **Модернизация для облачных сред (Cloud-Native):** Адаптировать платформу для современных архитектур, таких как микросервисы, контейнеризация и бессерверные вычисления. Это включает в себя создание легковесных профилей и поддержку новых парадигм.
4. **Быстрый цикл релизов:** Перейти от многолетних циклов выпуска к более частым и предсказуемым релизам, чтобы быстрее реагировать на потребности рынка.

1.1.3. Управление Сообществом: Eclipse Foundation

- **Jakarta EE Working Group:** Основной орган управления, определяющий стратегию и дорожную карту развития платформы.
- **JESP (Jakarta EE Specification Process):** Формальный процесс, по которому любая компания или индивидуум может предложить новую спецификацию или изменение в существующую. Процесс полностью открыт.
- **Спецификации и реализации:** Каждая технология в Jakarta EE — это спецификация (документ и API) и как минимум одна совместимая реализация (например, Weld — это реализация CDI, Hibernate — реализация JPA).
- **TCK (Technology Compatibility Kit):** Набор тестов, который должен пройти сервер приложений, чтобы получить сертификат совместимости с Jakarta EE. Это гарантирует ту самую переносимость приложений.

1.1.4. Обзор Текущих Версий и Спецификаций

Ключевые версии после перехода:

- **Jakarta EE 8 (2019):** Функционально идентична Java EE 8. Целью этого релиза было проверить процесс выпуска под эгидой Eclipse Foundation. **Все еще использует пакеты javax.*.**
- **Jakarta EE 9 (2020) / 9.1 (2021): "Большой взрыв" (Big Bang) release.** Единственное, но кардинальное изменение — это переименование всех пакетов javax.* в jakarta.*. Новых функций нет. Этот релиз был технической необходимостью для дальнейшего развития.
- **Jakarta EE 10 (2022): Первый релиз с новыми функциями** под новым пространством имен. Включает в себя более 20 обновленных спецификаций. Ключевое нововведение — **Core Profile**.
- **Jakarta EE 11 (в разработке):** Ожидается поддержка виртуальных потоков (Project Loom из Java 21), что кардинально улучшит производительность и масштабируемость приложений.

Профили (Profiles)

Архитектура Jakarta EE не монолитна. Она разделена на профили, чтобы соответствовать разным потребностям.

1. **Full Platform:** Полный набор всех спецификаций. Подходит для крупных монолитных приложений.
2. **Web Profile:** Подмножество спецификаций, ориентированное на создание современных веб-приложений. Не включает "тяжелые" корпоративные технологии, такие как EJB Remote или Jakarta Connectors.
3. **Core Profile (новинка в Jakarta EE 10):** Минимальный набор спецификаций, предназначенный для создания легковесных микросервисов. Включает в себя CDI Lite, JAX-RS, JSON-P, JSON-B.

1.1.5. Заключение

Архитектура Jakarta EE прошла долгий путь от сложной и громоздкой J2EE до современной, гибкой и управляемой сообществом платформы. Ключевые аспекты ее архитектуры сегодня — это **стандартизация**, **модульность** (через профили), **ориентация на облачные технологии** и **открытый процесс разработки**. Смена пространства имен с `javax` на `jakarta` стала переломным моментом, открывшим дорогу для инноваций и быстрого развития платформы в будущем.

1.2. Архитектура платформы Jakarta EE: Контейнерная модель

В основе Jakarta EE лежит **контейнерная модель**. Контейнер — это не физический объект, а скорее **среда выполнения (runtime environment)**, которую предоставляет сервер приложений (например, WildFly, Open Liberty, GlassFish). Эта среда управляет жизненным циклом компонентов приложения и предоставляет им системные сервисы.

Ключевая идея: Разработчик фокусируется на написании **бизнес-логики** (что приложение должно делать), а контейнер берет на себя всю инфраструктурную "черновую работу" (как это будет работать). Это пример принципа **инверсии управления (Inversion of Control, IoC)**.

Вместо того чтобы вы вручную создавали объекты, управляли потоками, транзакциями и безопасностью, вы **декларативно** (с помощью аннотаций или XML) просите контейнер сделать это за вас.

Основные типы контейнеров в Jakarta EE:

1. **Web-контейнер**
2. **EJB-контейнер**
3. **Application Client-контейнер (ACC)**

Давайте рассмотрим каждый из них подробно.

1.2.1. Web-контейнер (Web Container)

Что это такое?

Web-контейнер — это компонент сервера приложений, который отвечает за управление веб-компонентами и обработку HTTP-запросов. Если ваше приложение имеет веб-интерфейс или предоставляет REST API, оно будет работать внутри Web-контейнера.

Основные задачи и сервисы:

- **Управление жизненным циклом:** Создание, инициализация (`init()`) и уничтожение (`destroy()`) сервлетов, фильтров и слушателей. Вы не вызываете `new MyServlet()` — это делает контейнер.
- **Диспетчеризация запросов:** Сопоставление URL-адреса входящего запроса с конкретным сервлетом или JAX-RS ресурсом.
- **Управление сессиями:** Предоставление механизма `HttpSession` для отслеживания состояния пользователя между запросами.
- **Безопасность:** Декларативная проверка прав доступа к веб-ресурсам (URL-шаблонам).
- **Поддержка веб-технологий:** Обработка JSP, JSF, Jakarta WebSocket и других веб-спецификаций.

- **Взаимодействие с другими контейнерами:** Позволяет веб-компонентам (например, сервлетам) получать доступ к бизнес-компонентам (EJB).

Какие компоненты в нем работают?

- Jakarta Servlet
- Jakarta Pages (JSP) и Jakarta Standard Tag Library (JSTL)
- Jakarta Faces (JSF)
- Jakarta Filters и Listeners
- Jakarta RESTful Web Services (JAX-RS) Endpoints
- Jakarta WebSocket Endpoints

Пример кода: Jakarta Servlet

Этот простой сервлет полностью управляется Web-контейнером.

```
import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import jakarta.servlet.http.HttpSession;
import java.io.IOException;

// @WebServlet - декларативная аннотация.
// Контейнер прочитает её и автоматически зарегистрирует сервлет по пути
/visitCounter

@WebServlet("/visitCounter")
public class VisitCounterServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {

        // 1. Контейнер предоставляет нам объект сессии. Нам не нужно его
        создавать.

        HttpSession session = req.getSession(true);

        Integer counter = (Integer) session.getAttribute("counter");
```

```
        if (counter == null) {
            counter = 1; // Первый визит
        } else {
            counter++; // Последующие визиты
        }

        session.setAttribute("counter", counter);

        // 2. Контейнер предоставляет поток для записи ответа.
        resp.setContentType("text/html");
        resp.getWriter().println("<h1>You have visited this page " + counter + "
times.</h1>");
    }

    // 3. Методы init() и destroy() также будут вызваны контейнером в нужный
момент.
    @Override
    public void init() throws ServletException {
        System.out.println("VisitCounterServlet has been initialized by the Web
Container!");
    }
}
```

Что делает контейнер?

1. При старте приложения находит класс `VisitCounterServlet`, видит аннотацию `@WebServlet` и создает его экземпляр.
2. Вызывает метод `init()`.
3. При поступлении GET-запроса на `/visitCounter`, вызывает метод `doGet()`, передавая ему готовые объекты `request` и `response`.
4. Предоставляет сервис `HttpSession` для управления состоянием.

1.2.2. EJB-контейнер (EJB Container)

Что это такое?

EJB-контейнер — это среда выполнения для компонентов бизнес-логики (Enterprise JavaBeans). Он предоставляет самые мощные корпоративные сервисы, такие как управление транзакциями, безопасность на уровне методов и др.

Основные задачи и сервисы:

- **Управление транзакциями (JTA):** Самый важный сервис. Контейнер может автоматически начинать, фиксировать (commit) или откатывать (rollback) транзакции для методов EJB. Это делается декларативно с помощью аннотаций (`@Transactional`).
- **Безопасность:** Проверка прав доступа к методам EJB (`@RolesAllowed`).
- **Управление параллелизмом (Concurrency):** Безопасное управление доступом к компонентам из нескольких потоков (особенно актуально для `@Singleton EJB`).
- **Пул экземпляров (Pooling):** Для `@Stateless` бинов контейнер держит пул готовых экземпляров, чтобы быстро обслуживать запросы, экономя на создании объектов.
- **Асинхронное выполнение:** Выполнение методов в отдельном потоке с помощью аннотации `@Asynchronous`.
- **Планировщик задач (Timers):** Запуск методов по расписанию (`@Schedule`).

Какие компоненты в нем работают?

- Session Beans (`@Stateless`, `@Stateful`, `@Singleton`)
- Message-Driven Beans (`@MessageDriven`)

Пример кода: Stateless EJB

Представим сервис для создания заказа, который взаимодействует с базой данных.

```
// EJB компонент
import jakarta.ejb.Stateless;
import jakarta.ejb.TransactionAttribute;
import jakarta.ejb.TransactionAttributeType;
import jakarta.persistence.EntityManager;
import jakarta.persistence.PersistenceContext;
```

@Stateless // Говорим контейнеру, что это Stateless EJB. Он будет управлять пулом экземпляров.

```
public class OrderService {
```

```
    // Контейнер внедрит сюда EntityManager для работы с базой данных (JPA)
```

```
    @PersistenceContext
```

```
    private EntityManager em;
```

```
    // @TransactionAttribute - декларативное управление транзакциями.
```

```
    // REQUIRED (по умолчанию) - контейнер убедится, что метод  
    // выполняется в транзакции.
```

```
    // Если транзакции нет, он её создаст.
```

```
    @TransactionAttribute(TransactionAttributeType.REQUIRED)
```

```
    public void createOrder(Order newOrder) {
```

```
        // Начало транзакции (неявно, управляется контейнером)
```

```
        em.persist(newOrder); // Сохраняем заказ
```

```
        // Если здесь произойдет ошибка (RuntimeException), контейнер  
        // автоматически
```

```
        // откатит транзакцию, и заказ не попадет в БД.
```

```
        // Конец транзакции (commit, управляется контейнером)
```

```
    }
```

```
}
```

```
// REST-ресурс (из Web-контейнера), который использует EJB
```

```
@jakarta.ws.rs.Path("/orders")
```

```
public class OrderResource {
```

```
    @jakarta.ejb.EJB // Просим EJB-контейнер внедрить экземпляр нашего  
    сервиса
```

```
private OrderService orderService;

@jakarta.ws.rs.POST
public void placeOrder(Order order) {
    // Просто вызываем бизнес-метод. Всю работу с транзакциями делает
    EJB-контейнер.
    orderService.createOrder(order);
}
}
```

Что делает контейнер?

1. Управляет жизненным циклом и пулом экземпляров OrderService.
2. Внедряет EntityManager с помощью аннотации @PersistenceContext.
3. Внедряет сам OrderService в OrderResource с помощью @EJB.
4. Перед вызовом метода createOrder **начинает транзакцию**.
5. После успешного завершения метода **фиксирует (commit) транзакцию**.
6. Если в методе возникает системное исключение, **откатывает (rollback) транзакцию**.

1.2.3. Application Client-контейнер (Application Client Container, ACC)

Что это такое?

Это наименее используемый сегодня контейнер. Он предназначен для запуска "толстых" клиентов или консольных Java-приложений, которым нужен доступ к сервисам сервера приложений, таким как EJB и Dependency Injection. Он запускается на клиентской машине, но взаимодействует с сервером.

Основные задачи и сервисы:

- **Внедрение зависимостей:** Позволяет клиентскому коду использовать `@EJB` или `@Inject` для получения удаленного доступа к компонентам на сервере.
- **Безопасность:** Обеспечивает аутентификацию и передачу контекста безопасности на сервер.
- **Поиск ресурсов (JNDI):** Предоставляет доступ к ресурсам сервера через JNDI.

Пример кода:

Предположим, у нас есть `OrderService EJB` на сервере. Мы можем написать консольный клиент для его вызова.

```
private static OrderService remoteOrderService;

public static void main(String[] args) {
    System.out.println("Application Client started.");

    // Создаем новый заказ
    Order newOrder = new Order("Laptop", 1);

    // Вызываем удаленный метод на сервере приложений.
    // ACC берет на себя всю сетевую коммуникацию, сериализацию и т.д.
    remoteOrderService.createOrder(newOrder);

    System.out.println("Order created successfully via remote EJB call.");
}
}
```

Вы не можете просто запустить этот класс командой `java OrderClient`. Его нужно упаковать в JAR и запустить с помощью специального скрипта, поставляемого с сервером приложений (например, `appclient.sh` в GlassFish/Payara). Этот скрипт запускает АСС, который подготавливает окружение (подключается к серверу, настраивает JNDI) и затем вызывает ваш `main` метод.

1.3. Серверы приложений Jakarta EE: Обзор, установка, базовая конфигурация

Введение: Что такое Сервер Приложений?

Сервер приложений (Application Server) — это среда выполнения (runtime), которая реализует спецификации Jakarta EE. Именно он предоставляет те самые **контейнеры** (Web, EJB), о которых мы говорили ранее.

Главное преимущество стандартизации: Вы пишете ваше приложение (например, `.war` файл) с использованием стандартных Jakarta EE API. В идеале, вы можете взять этот файл и развернуть его на **любом** совместимом сервере приложений, и он будет работать одинаково. На практике, конечно, есть нюансы в конфигурации, но код приложения остается переносимым.

Давайте рассмотрим ключевых игроков.

1.3.1. WildFly (и JBoss EAP)

- **Обзор:**

- Разработчик: Red Hat.
- Философия: WildFly — это передовой, бесплатный, open-source проект сообщества. JBoss EAP (Enterprise Application Platform) — это коммерческая, стабилизированная и поддерживаемая Red Hat версия WildFly. Для разработки обычно используют WildFly.
- Особенности: Очень мощный, быстрый, поддерживает полный профиль Jakarta EE. Имеет отличный инструментарий, включая мощный CLI (Command-Line Interface) для управления и скриптинга.
- Версии: WildFly очень быстро обновляется и часто является одной из первых реализаций новых спецификаций Jakarta EE.

- **Установка (самый простой способ):**

1. Скачайте ZIP-архив с официального сайта wildfly.org.
2. Распакуйте архив в удобную директорию (например, `C:\dev\wildfly` или `~/dev/wildfly`).
3. Всё! Установка завершена.

- **Запуск и остановка:**

- Перейдите в директорию `bin` внутри папки WildFly.

- **Запуск:** `./standalone.sh` (на Linux/macOS) или `standalone.bat` (на Windows).
- Сервер будет доступен по адресу `http://localhost:8080`.
- **Остановка:** Нажмите `Ctrl+C` в консоли, где запущен сервер.
- **Базовая конфигурация:**
 - **Главный файл конфигурации:** `standalone/configuration/standalone.xml`. Это большой XML-файл, который описывает все подсистемы сервера: веб, EJB, datasource, логирование и т.д.
 - **Развертывание (Deployment):** Самый простой способ — скопировать ваш `.war` или `.ear` файл в директорию `standalone/deployments/`. WildFly автоматически "подхватит" и развернет его.
 - **Пример: Настройка источника данных (Datasource)** Это самая частая задача. Вместо того чтобы прописывать данные для подключения к БД в коде, вы настраиваете их на сервере, а приложение получает их по JNDI-имени.
 - Откройте `standalone/configuration/standalone.xml` и найдите секцию `<subsystem xmlns="urn:jboss:domain:datasources:...">`. Добавьте в нее:

```

<!-- Внутри тега <datasources> -->

<datasource jndi-name="java:jboss/datasources/MyWebAppDS" pool-
name="MyWebAppDS" enabled="true" use-java-context="true">
    <connection-url>jdbc:h2:mem:mywebappdb;DB_CLOSE_DELAY=-
1</connection-url>
    <driver>h2</driver>
    <security>
        <user-name>sa</user-name>
        <password>sa</password>
    </security>
</datasource>

<drivers>
    <!-- Драйвер для H2 уже встроен в WildFly -->
    <driver name="h2" module="com.h2database.h2">
        <xa-datasource-class>org.h2.jdbcx.JdbcDataSource</xa-datasource-class>
    </driver>

```

</drivers>

Теперь в вашем приложении (в файле persistence.xml) вы можете ссылаться на этот datasource:

```
<!-- src/main/resources/META-INF/persistence.xml -->
<persistence-unit name="my-app-pu">
  <jta-data-source>java:boss/datasources/MyWebAppDS</jta-data-source>
  <!-- ... properties ... -->
</persistence-unit>
```

1.3.2. GlassFish

- **Обзор:**

- **Разработчик:** Изначально Sun Microsystems/Oracle, сейчас — **Eclipse Foundation**.
- **Философия:** GlassFish — это эталонная реализация (**Reference Implementation**) Jakarta EE. Это означает, что он является стандартом, с которым сравнивают себя другие серверы.
- **Особенности:** Прост в освоении, имеет удобную веб-консоль для администрирования (<http://localhost:4848>). Использует утилиту командной строки asadmin.

- **Установка:**

1. Скачайте ZIP с страницы проекта Eclipse GlassFish.
2. Распакуйте архив.

- **Запуск и остановка:**

- Перейдите в glassfish7/bin.
- **Запуск:** ./asadmin start-domain (на Linux/macOS) или asadmin.bat start-domain (на Windows).
- Приложения доступны на <http://localhost:8080>, админ-панель на <http://localhost:4848>.
- **Остановка:** ./asadmin stop-domain.

- **Базовая конфигурация:**

- **Главный** **файл:**
glassfish7/glassfish/domains/domain1/config/domain.xml.
Редактировать его вручную не рекомендуется.
- **Предпочтительный способ:** Использовать веб-консоль или утилиту asadmin.

- **Развертывание:** Скопировать .war в glassfish7/glassfish/domains/domain1/autodeploy/ или через админ-панель.
- **Пример: Настройка Datasource через asadmin**
 1. Положите JAR-файл JDBC-драйвера (например, postgresql.jar) в папку glassfish7/glassfish/lib.
 2. Создайте пул соединений:

Generated bash

```
./asadmin create-jdbc-connection-pool --datasourceclassname org.postgresql.ds.PGSimpleDataSource --restype javax.sql.DataSource --property user=postgres:password=secret:databaseName=myapp:serverName=localhost:portNumber=5432 MyWebAppPool
```

content_copydownload

Use code with caution.Bash

Создайте JNDI-ресурс, который будет использовать этот пул:

Generated bash

```
./asadmin create-jdbc-resource --connectionpoolid MyWebAppPool jdbc/MyWebAppDS
```

content_copydownload

Use code with caution.Bash

- Теперь в persistence.xml вы можете использовать JNDI-имя jdbc/MyWebAppDS.

1.3.3. Payara Server

- **Обзор:**
 - **Разработчик:** Payara Services Ltd.
 - **Философия:** "GlassFish на стероидах". Payara — это форк GlassFish с фокусом на **production-окружение**, микросервисы и облачные развертывания. Предлагает коммерческую поддержку.
 - **Особенности:** Включает в себя все от GlassFish, плюс дополнительные инструменты: встроенный JCache (на базе Hazelcast), улучшенный мониторинг, поддержка MicroProfile для микросервисов.
- **Установка и запуск:** Практически идентичны GlassFish. Скачиваете ZIP, распаковываете, запускаете через ./asadmin start-domain.

- **Базовая конфигурация:** Также идентична GlassFish (используется asadmin и та же админ-панель). Преимущество в том, что вы получаете больше "из коробки".

1.3.4. Open Liberty

- **Обзор:**
 - Разработчик: IBM.
 - Философия: Легковесный, модульный, динамический и ориентированный на облака/микросервисы.
 - Особенности:
 - **Динамическая конфигурация:** Вы можете менять конфигурацию сервера (server.xml) на лету, и сервер применит изменения **без перезапуска**. Это его киллер-фича.
 - **Модульность:** Сервер загружает только те компоненты Jakarta EE, которые вы объявили в конфигурации. Это приводит к очень быстрому старту и малому потреблению памяти.
 - **Нулевая миграция:** Конфигурация отделена от самого сервера, что упрощает обновление версий.

- **Установка:**

1. Скачайте ZIP с openliberty.io.
2. Распакуйте.

- **Запуск и остановка:**

- Перейдите в `wlp/bin`.
- **Запуск:** `./server run defaultServer` (или `./server start defaultServer` для фонового режима).
- Сервер доступен на `http://localhost:9080`, HTTPS на 9443.
- **Остановка:** `Ctrl+C` (если запущен через `run`) или `./server stop defaultServer`.

- **Базовая конфигурация:**

- **Главный файл:** `wlp/usr/servers/defaultServer/server.xml`. Он очень минималистичен по умолчанию.
- **Развертывание:** Скопировать `.war` в `wlp/usr/servers/defaultServer/apps/`.
- **Пример: Настройка Jakarta EE 10 Web Profile и Datasource**
Откройте `server.xml` и приведите его к такому виду:

Generated xml

```
<server description="My Awesome Server">
```

```
    <!-- 1. Включаем фичи, которые нам нужны. Сервер сам скачает и
    подключит их. -->
```

```
    <featureManager>
```

```
        <feature>jakartaee-10.0</feature> <!-- или webProfile-10.0 для веб-
    профиля -->
```

```
    </featureManager>
```

```
    <!-- 2. Настраиваем базовый HTTP-эндпоинт -->
```

```
    <httpEndpoint id="defaultHttpEndpoint" host="*" httpPort="9080"
    httpsPort="9443" />
```

```
    <!-- 3. Объявляем приложение -->
```

```
    <application location="my-awesome-app.war" type="war" context-
    root="/myapp"/>
```

```

<!-- 4. Настраиваем библиотеку с JDBC-драйвером -->
<library id="PostgreSQLLib">
    <fileset dir="\${shared.resource.dir}/postgresql" includes="postgresql-
42.5.1.jar"/>
</library>

<!-- 5. Настраиваем Datasource -->
<dataSource id="myDS" jndiName="jdbc/MyWebAppDS">
    <jdbcDriver libraryRef="PostgreSQLLib" />
    <properties.postgresql serverName="localhost" portNumber="5432"
databaseName="myapp" user="postgres" password="secret" />
</dataSource>
</server>

```

content_copydownload

Это показывает всю мощь и простоту конфигурации Open Liberty.

1.3.5. Apache TomEE

- **Обзор:**
 - **Разработчик:** Apache Software Foundation.
 - **Философия:** "Tomcat на стероидах". Берется самый популярный в мире веб-контейнер Apache Tomcat и к нему добавляются библиотеки (OpenEJB, OpenWebBeans, OpenJPA), чтобы сделать его совместимым с Jakarta EE (обычно Web Profile).
 - **Особенности:** Если вы уже знакомы с Tomcat, переход на TomEE будет очень простым. Он легкий и быстрый.
- **Установка и запуск:** Аналогичны Tomcat. Скачали, распаковали, запустили через bin/startup.sh или bin/startup.bat.
- **Базовая конфигурация:**
 - **Главный файл:** Конфигурация может быть в conf/tomee.xml или conf/server.xml.
 - **Развертывание:** Скопировать .war в webapps/.
 - **Пример: Настройка Datasource в conf/tomee.xml**

Generated xml

```

<tomee>

<Resource id="MyWebAppDS" type="DataSource">

```

```
JdbcDriver = org.h2.Driver
JdbcUrl = jdbc:h2:mem:mywebappdb
UserName = sa
Password = sa
JtaManaged = true
</Resource>
</tomee>
content_copydownload
Use code with caution.Xml
```

В persistence.xml вы будете ссылаться на этот ресурс по его ID: MyWebAppDS.

1.3.6. Заключение: Какой выбрать?

- **Для новичка: WildFly или Payara** — отличный выбор. Они полнофункциональны, имеют хорошую документацию и активные сообщества.
- **Для микросервисов и облака: Open Liberty** — явный фаворит благодаря своей динамичности и малому размеру. Payara Micro также является сильным конкурентом.
- **Если вы уже используете Tomcat: TomEE** — самый логичный и безболезненный шаг в мир Jakarta EE.

Главное, что нужно помнить — благодаря стандартам, ваше приложение не "прибито гвоздями" к одному серверу. Освоив один, вы сможете с минимальными усилиями переключиться на другой, изменив лишь специфичные для сервера файлы конфигурации.

1.4. Профили, архитектура, паттерны и инструменты Jakarta EE

Эта тема охватывает "каркас" любого корпоративного приложения:

- **ЧТО** мы можем использовать (Профили).
- **КАК** мы структурируем приложение (Многоуровневая архитектура).
- **С ПОМОЩЬЮ КАКИХ ПРИНЦИПОВ** мы пишем код (Паттерны).
- **КАКИМИ ИНСТРУМЕНТАМИ** мы пользуемся (Инструменты разработки).

1.4.1. Профили Jakarta EE (Profiles)

Платформа Jakarta EE не является монолитной. Она разделена на профили, чтобы разработчики могли использовать только те технологии, которые им нужны, избегая избыточности.

Jakarta EE Full Platform (Полная платформа)

- **Что это:** Полный набор всех спецификаций Jakarta EE. Включает в себя абсолютно все, от веб-технологий до интеграции с устаревшими системами.
- **Когда использовать:** Для крупных, традиционных монолитных корпоративных приложений, которым требуются расширенные возможности, такие как удаленный вызов EJB, взаимодействие с системами через коннекторы или распределенные транзакции между несколькими ресурсами.
- **Ключевые технологии, эксклюзивные для Full Platform:**
 - Jakarta Enterprise Beans (Remote) - удаленный вызов EJB.
 - Jakarta Connectors - для интеграции с EIS (Enterprise Information Systems).
 - Jakarta Messaging (JMS) - требуется в полном объеме.
 - Jakarta SOAP with Attachments.

Jakarta EE Web Profile (Веб-профиль)

- **Что это:** Подмножество полной платформы, ориентированное на создание современных веб-приложений и микросервисов. Это самый популярный и часто используемый профиль.
- **Когда использовать:** В 90% случаев. Идеально для REST API, веб-приложений с серверным рендерингом, бэкендов для мобильных приложений и микросервисов.
- **Включает в себя:**
 - **Веб-слой:** Servlet, JAX-RS, JSF, JSP, WebSocket, JSON-P, JSON-B.
 - **Бизнес-слой:** CDI, EJB Lite (только @Stateless, @Stateful, @Singleton для локального использования).

- **Слой данных:** JPA, Bean Validation, JTA (управление транзакциями).

Характеристика	Web Profile	Full Platform
Цель	Современные веб-приложения, микросервисы	Крупные, традиционные корпоративные приложения
Размер и сложность	Легче, быстрее старт, меньше потребление ресурсов	Тяжелее, включает все возможные спецификации
EJB	EJB Lite (только локальные бины)	Полная поддержка EJB, включая удаленные вызовы (Remote)
Интеграция	Нет Jakarta Connectors	Есть Jakarta Connectors для интеграции с EIS
Пример сервера	Apache TomEE, WildFly (в режиме Web Profile)	WildFly (по умолчанию), GlassFish, Payara Server

1.4.2. Многоуровневая архитектура корпоративных приложений

Это классическая модель разделения приложения на логические слои для улучшения управляемости, масштабируемости и переиспользования кода.

Слой Представления (Presentation Tier)

- **Ответственность:** Взаимодействие с пользователем. Отображение данных и передача действий пользователя на слой бизнес-логики.
- **Технологии Jakarta EE:**
 - **Jakarta RESTful Web Services (JAX-RS):** Для создания REST API, которые предоставляют данные (обычно в формате JSON) для "безголовых" приложений (frontend-фреймворки типа React/Angular, мобильные приложения).
 - **Jakarta Faces (JSF) / Jakarta Pages (JSP):** Для создания традиционных веб-приложений, где HTML-страницы генерируются на сервере.
 - **Jakarta Servlet:** Фундаментальная технология, на которой работают JAX-RS и JSF.
 - **Jakarta WebSocket:** Для интерактивной двунаправленной связи с клиентом.

Слой Бизнес-логики (Business Tier)

- **Ответственность:** Реализация основной логики приложения, бизнес-правил, координация работы со слоем данных, управление транзакциями. Это "мозг" приложения.

- **Технологии Jakarta EE:**

- **Jakarta Contexts and Dependency Injection (CDI):** Основа современного бизнес-слоя. Управляет жизненным циклом компонентов и внедрением зависимостей.
- **Jakarta Enterprise Beans (EJB):** Предоставляет мощные сервисы, такие как декларативное управление транзакциями (@Transactional), безопасность на уровне методов (@RolesAllowed) и асинхронное выполнение (@Asynchronous).
- **Jakarta Transactions (JTA):** Управляет границами транзакций.

Слой Доступа к Данным (Data Tier / Enterprise Information Systems Tier)

- **Ответственность:** Взаимодействие с системами хранения данных. Это могут быть реляционные базы данных, брокеры сообщений или устаревшие системы.
- **Технологии Jakarta EE:**
 - **Jakarta Persistence (JPA):** Стандарт для ORM (Object-Relational Mapping). Позволяет работать с объектами Java вместо написания SQL-запросов вручную. EntityManager - ключевой интерфейс.
 - **Jakarta Messaging (JMS):** Стандарт для асинхронного обмена сообщениями с брокерами (например, ActiveMQ, RabbitMQ).
 - **Jakarta Connectors:** Для интеграции с корпоративными информационными системами (EIS), такими как SAP или мейнфреймы.

1.4.3. Основные паттерны проектирования в Jakarta EE

Платформа Jakarta EE не просто позволяет использовать паттерны — она активно их реализует и поощряет их применение.

- **Dependency Injection (DI)**

- **Что это:** Объекты не создают свои зависимости, а получают их извне (от контейнера).
- **Роль в Jakarta EE:** Это фундаментальный принцип. Реализован через CDI (@Inject) и EJB (@EJB). Устраняет жесткие связи между компонентами, упрощает тестирование.
- **Пример:**

Generated java

```
@jakarta.enterprise.context.ApplicationScoped // CDI Bean
```

```
public class PaymentService {
```

```
    // Контейнер сам найдет и подставит сюда реализацию NotificationService
```

```
    @jakarta.inject.Inject
```

```

private NotificationService notificationService;

public void processPayment() {
    // ... логика оплаты ...
    notificationService.sendPaymentSuccess("user@example.com");
}
}
content_copydownload

```

Use code with caution.Java

- **Model-View-Controller (MVC)**

- **Что это:** Разделение приложения на три части: Модель (данные), Представление (UI) и Контроллер (логика обработки).
- **Роль в Jakarta EE:** Ярко выражен в Jakarta Faces (JSF).
 - **Model:** JPA Entity или DTO.
 - **View:** .xhtml файлы (страницы с UI-компонентами).
 - **Controller:** CDI Managed Beans (бины, управляемые CDI), которые обрабатывают действия пользователя.

- **DAO (Data Access Object)**

- **Что это:** Абстрагирует слой бизнес-логики от деталей работы с хранилищем данных.
- **Роль в Jakarta EE:** Несмотря на то, что EntityManager из JPA уже является мощным инструментом, принято создавать свои DAO-классы (часто в виде CDI-бинов), которые инкапсулируют логику запросов к конкретным сущностям.
- **Пример:**

Generated java

```

@jakarta.enterprise.context.ApplicationScoped
public class UserDao {
    @jakarta.persistence.PersistenceContext
    private EntityManager em; // EntityManager внедряется контейнером

    public User findById(Long id) {
        return em.find(User.class, id);
    }
}

```

```

    public void save(User user) {
        em.persist(user);
    }
}

```

content_copydownload

Use code with caution.Java

- **DTO (Data Transfer Object)**

- **Что это:** Простые Java-объекты (POJO) для передачи данных между слоями, особенно через сеть (например, в REST API). Они не содержат бизнес-логики.
- **Роль в Jakarta EE:** Крайне важны в JAX-RS. Вместо того чтобы выставлять наружу JPA-сущности (что небезопасно и может привести к утечке данных), их преобразуют в DTO.
- **Пример:**

Generated java

// JPA Entity (модель данных)

```

@Entity public class User { /* ... id, login, passwordHash ... */ }

```

// DTO (объект для передачи)

```

public class UserDto {
    private Long id;
    private String login;
    // Нет пароля!
    // ... getters/setters ...
}

```

// REST-ресурс

```

@Path("/users")

```

```

public class UserResource {

```

```

    @GET @Path("/{id}")

```

```

    @Produces(MediaType.APPLICATION_JSON)

```

```

public UserDto getUser(@PathParam("id") Long id) {
    User user = userService.findById(id);
    // Преобразуем Entity в DTO перед отправкой клиенту
    return mapUserToDto(user);
}
}

```

content_copydownload

Use code with caution.Java

- **Façade**
 - **Что это:** Предоставляет простой, унифицированный интерфейс для сложной подсистемы.
 - **Роль в Jakarta EE:** Идеальный кандидат для реализации этого паттерна — **Stateless EJB** или **CDI Bean**. Например, `OrderService` может быть фасадом, который под капотом работает с `InventoryService`, `CustomerService` и `PaymentGateway`.
- **Front Controller**
 - **Что это:** Единая точка входа для обработки всех запросов.
 - **Роль в Jakarta EE:** Этот паттерн **реализован самими фреймворками**. Вам не нужно его писать.
 - В **JSF** эту роль выполняет `FacesServlet`.
 - В **JAX-RS** эту роль выполняет его внутренняя реализация сервлета или фильтра, которая маршрутизирует запросы к нужным `@Path` методам.
- **Service Locator**
 - **Что это:** Паттерн для поиска сервисов (например, EJB) через централизованный реестр (JNDI).
 - **Роль в Jakarta EE:** **Устаревший паттерн**. Раньше он был необходим для получения EJB. Сегодня, благодаря **Dependency Injection** (`@Inject`), он практически не используется в коде приложения. Контейнер сам находит и внедряет сервисы, делая код намного чище.

1.4.4. Инструменты разработки

IDE (Интегрированные среды разработки)

1. IntelliJ IDEA Ultimate

- **Плюсы:** Считается одной из лучших IDE для Java-разработки. Предоставляет мощнейшую поддержку Jakarta EE: автодополнение для JPQL, CDI, JAX-RS; интеграцию с серверами приложений (запуск, отладка, развертывание в один клик); отличные инструменты для работы с базами данных.
- **Минусы:** Платная (требуется версия Ultimate).

2. Eclipse IDE for Enterprise Java and Web Developers

- **Плюсы:** Бесплатная, open-source. Является "домашней" IDE для проектов Eclipse Foundation, включая Jakarta EE и GlassFish. Имеет хорошую интеграцию с серверами и все необходимые инструменты.
- **Минусы:** Может показаться менее отзывчивой и интуитивной по сравнению с IDEA.

Системы сборки (Build Systems)

1. Maven

- **Что это:** Декларативная система сборки, стандарт де-факто в Java-мире. Управляет зависимостями, компиляцией, тестированием и упаковкой проекта в pom.xml файле.
- **Ключевой момент для Jakarta EE:** Зависимости на API платформы объявляются с областью видимости `<scope>provided</scope>`. Это говорит Maven, что библиотека нужна только для компиляции, а в итоговый .war файл её включать не надо, так как её предоставит сам сервер приложений.
- **Пример pom.xml:**

Generated xml

```
<properties>
```

```
  <maven.compiler.source>11</maven.compiler.source>
```

```
  <maven.compiler.target>11</maven.compiler.target>
```

```
  <jakartaee-api.version>10.0.0</jakartaee-api.version>
```

```
  <maven-war-plugin.version>3.3.2</maven-war-plugin.version>
```

```
</properties>
```

```
<dependencies>
```

```
  <!-- Основная зависимость на Jakarta EE API -->
```

```
  <dependency>
```

```
    <groupId>jakarta.platform</groupId>
```

```
    <artifactId>jakarta.jakartaee-api</artifactId>
```

```

        <version>${jakartaee-api.version}</version>
        <scope>provided</scope> <!-- Очень важно! -->
    </dependency>
</dependencies>

<build>
    <plugins>
        <!-- Плагин для сборки WAR-архива -->
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-war-plugin</artifactId>
            <version>${maven-war-plugin.version}</version>
        </plugin>
    </plugins>
</build>

```

content_copydownload

Use code with caution.Xml

2. Gradle

- **Что это:** Более современная и гибкая система сборки, использующая Groovy или Kotlin для написания скриптов сборки (build.gradle).
- **Ключевой момент для Jakarta EE:** Эквивалент <scope>provided в Gradle — это конфигурация compileOnly.
- **Пример build.gradle.kts (Kotlin DSL):**

Generated kotlin

```

plugins {
    java
    id("war") // Подключаем плагин для сборки WAR
}

```

```
group = "com.example"
```

```
version = "1.0-SNAPSHOT"
```

```

repositories {
    mavenCentral()
}

dependencies {
    // Аналог <scope>provided</scope>
    compileOnly("jakarta.platform:jakarta.jakartaee-api:10.0.0")

    // Зависимости для тестирования
    testImplementation("org.junit.jupiter:junit-jupiter-api:5.9.1")
    testRuntimeOnly("org.junit.jupiter:junit-jupiter-engine:5.9.1")
}

tasks.getByName<Test>("test") {
    useJUnitPlatform()
}

```

1.5. Сервлеты

Приложение Java, запускаемое и выполняемое в контейнере сервера приложений. Клиент общается с таким приложением посредством веб-браузера. Никаких дополнительных приложений на стороне клиента устанавливать не требуется. Сервлеты поддерживаются виртуальной машиной JVM, что предотвращает утечки памяти и обеспечивает функционирование garbage collection. Каждому клиенту сервлет выделяет независимый поток выполнения. Клиент посылает приложению HTTP-запрос, сервлет генерирует ответ и возвращает его клиенту в виде html-документа.

Сервлет:

- компонент приложений Jakarta Enterprise Edition;
- загружается application-сервером в контейнер;
- выполняется на стороне сервера;
- обрабатывает клиентские запросы;
- динамически генерирует ответы;
- находится в состоянии ожидания, если запросы отсутствуют;
- принимает\передает запросы от других сервлетов (Servlet chaining);
- поддерживает соединения с ресурсами.

Наибольшее распространение получили сервлеты, обрабатывающие клиентские запросы по протоколу HTTP. Технология сервлетов является оболочкой протокола HTTP и поддерживает его как транспорт передачи данных от клиента серверу и обратно. Контейнер сервлетов поддерживает также протокол HTTPS (HTTP и SSL) для защищаемых запросов.

Сервлеты в промышленном программировании используются для:

- приема входящих данных от клиента;
- взаимодействия с бизнес-логикой системы;
- динамической генерации ответа клиенту.

Все сервлеты реализуют общий интерфейс **Servlet** из пакета **jakarta.servlet**. Для обработки HTTP-запросов в web можно воспользоваться в качестве базового класса абстрактным классом **HttpServlet**

Жизненный цикл сервлета начинается с его создания, инициализации и загрузки в память контейнером сервлетов при старте контейнера либо в ответ на первый клиентский запрос. Сервлет готов к обслуживанию любого числа запросов.

Завершение существования происходит при выгрузке его из контейнера.

Первым вызывается метод **init()**. Он дает сервлету возможность инициализировать данные и подготовиться для обработки запросов. Чаще всего в этом методе размещается код, кэширующий данные фазы инициализации.

После этого сервлет можно считать запущенным, он находится в ожидании запросов от клиентов. Появившийся запрос обслуживается методом **void service(HttpServletRequest request, HttpServletResponse response)** сервлета, вызываемый контейнером, а все параметры запроса упаковываются в экземпляр **request** интерфейса **HttpServletRequest**, передаваемый в сервлет. Еще одним параметром этого метода является экземпляр **response** интерфейса **HttpServletResponse**, в который загружается информация для передачи клиенту. Для каждого нового клиента при обращении к сервлету создается независимый поток, в котором производится вызов метода **service()**. Метод **service()** предназначен для одновременной обработки множества запросов.

При выгрузке приложения из контейнера, то есть по окончании жизненного цикла сервлета, вызывается метод **destroy()**, в теле которого следует помещать код освобождения занятых сервлетом ресурсов.

Преимуществом сервлетов перед CGI или ASP является быстроедействие, переносимость на различные платформы, использование объектно-ориентированного языка высокого уровня Java, который расширяется большим числом классов и программных интерфейсов.

Сервлеты поддерживаются серверами приложений WebSphere, Weblogic, WildFly, Oracle OC4J Server, Glassfish, Geronimo, Apache Tomcat, на основе контейнеров сервлетов tomcat, jetty, grizzly, glassfish и являются частью платформы JEE.

Сервлеты реализуют интерфейс **Servlet**, в котором кроме рассмотренных выше методов **service()**, **init()**, **destroy()** предусмотрена реализация метода **ServletConfig getConfig()** — он возвращает объект, содержащий параметры конфигурации сервлета и дает доступ к среде выполнения.

При разработке сервлетов в качестве суперкласса в большинстве случаев используется не интерфейс **Servlet**, а абстрактный класс **HttpServlet**, отвечающий за обработку запросов HTTP.

Метод **service()** класса **HttpServlet** служит диспетчером для других методов, каждый из которых обрабатывает методы доступа к соответствующим ресурсам. В спецификации HTTP определены методы: **GET**, **HEAD**, **POST**, **PUT**, **DELETE**, **OPTIONS** и **TRACE**. Наиболее часто употребляются методы **GET** и **POST**, передающие на сервер запросы, а также параметры для их выполнения.

В задачу метода **service()** класса **HttpServlet** входит анализ полученного через запрос метода доступа к ресурсам и вызов метода со сходным именем, где перед именем добавляется префикс **do**: **doGet()**, **doPost()**, **doHead()**, **doPut()**, **doDelete()**, **doOptions()** и **doTrace()**. Разработчик должен переопределить нужный метод, разместив в нем функциональную логику.

Метод **GET** (`method="GET"`) используется для запроса содержимого указанного ресурса, изображения или гипертекстового документа. Вместе с запросом могут передаваться дополнительные параметры как часть URI, значения могут выбираться из полей формы или передаваться непосредственно через URL. При этом запросы кэшируются и имеют ограничения на размер (2048 символов). Этот метод является основным методом взаимодействия браузера клиента и веб-сервера.

Метод **POST** используется для передачи пользовательских данных в содержимом HTTP-запроса на сервер. Пользовательские данные упакованы в тело запроса согласно полю заголовка **Content-Type** и/или включены в URI запроса. При использовании метода **POST** под URI подразумевается ресурс, который будет обрабатывать запрос.

Метод **PUT** схож с методом **POST** за тем исключением, что здесь URI подразумевает ресурс, который будет создан или сохранен на сервере в результате выполнения **PUT**-запроса.

Метод **DELETE** предназначен для удаления целевого ресурса.

Оба эти действия на некоторых серверах могут запрещаться из-за угрозы внутренней безопасности.

Метод **HEAD** предполагает возврат сервером такого же ответа, как и при использовании **GET**, но без тела ответа. Метод обычно используется для того, чтобы проверить существование ресурса либо узнать, изменился ли запрашиваемый ресурс с момента последнего обращения.

Метод **OPTIONS** должен возвращать информацию о возможностях веб-сервера или параметрах соединения для конкретного ресурса.

Метод **TRACE** возвращает клиенту запрос в том виде, в каком он пришел на сервер — используется для отладки, определяя заголовки, добавляемые промежуточными серверами, а также для тестирования настроек соединения.

Методы **HEAD**, **OPTIONS** и **TRACE**, как правило, реализуются сервером веб-приложения прозрачно для программиста и не требуют какой-либо специальной обработки. Поэтому обычно при разработке веб-приложений они не рассматриваются.

Методы **GET** и **HEAD**, по принятым соглашениям, не должны иметь другого назначения, кроме как передача информации клиенту — они должны быть «безопасными» («safe»). То есть в результате их выполнения на сервере не должно происходить никаких «побочных эффектов», которые повлияют на состояние ресурсов сервера — таких, как создание, изменение, удаление данных. Также подразумевается, что запросы с «безопасными» методами могут быть выполнены браузером клиента в автоматическом режиме, без специального разрешения пользователя. Так как ссылки через элемент `<a>` в коде HTML по умолчанию подразумевают использование браузером **GET**-запросов, то довольно часто можно встретить нарушение «безопасности» **GET**-запросов из-за использования элемента `<a>` для выполнения каких-либо действий, изменяющих состояние сервера, например, ссылки вида:

```
<a href="http://example.com/do?action=delete">Delete database</a>
```

Далеко не всегда такое нарушение влечет негативные последствия, однако так как инфраструктура глобальных сетей предполагает соблюдение соглашения о «безопасности» **GET**-запросов, то при взаимодействии со следующими факторами могут возникать негативные последствия.

1. **Кэширующие прокси.** Так как **GET**-запрос (в отличие от **POST**) может быть кэширован промежуточным кэширующим прокси-сервером, то при обработке запроса прокси может не выполнить запрос к целевому серверу, а сразу вернуть результат. Таким образом, если запрос подразумевал какое-то действие над данными на сервере, то это действие не будет выполнено.
2. **Предзагрузка.** Некоторые браузеры для ускорения навигации выполняют загрузку страниц и других ресурсов по ссылкам еще до того, как пользователь выполнил переход по этим ссылкам. В этом случае при нарушении «безопасности» браузер выполнит запрос, который не был инициирован пользователем. Например, если на странице расположены ссылки, удаляющие какие-то объекты через **GET**-запросы, то при открытии этой страницы браузер может вызвать удаление всех этих объектов.

3. **Поисковые системы и другие клиенты, действующие в автоматическом режиме.** Разнообразные автоматические клиенты не могут отличить запросы, которые выполняют какие-либо действия от запросов, которые просто возвращают информацию. Поэтому они обычно ориентируются по методу запроса: **GET**-запросы выполняются и используются для дальнейшего анализа, а **POST**-запросы игнорируются. При нарушении «безопасности» **GET**-запросов возникает ситуация, аналогичная предзагрузке: выполняются те действия, которые при «нормальном» взаимодействии с приложением не должны были бы выполняться.
4. **Нарушение авторизации.** Если выполнение действия предполагает наличие определенных полномочий, то использование **GET**-запроса для этого предоставляет злоумышленнику возможность воспользоваться привилегиями другого пользователя (например, путем размещения ссылки с нужным **GET**-запросом в `<img>` элементе на произвольном сайте, который может посетить пользователь, обладающий нужными полномочиями с идентификацией через cookies) и тем самым выполнить неавторизованное действие.

Поэтому важно понимать различия между методами **GET** и **POST** и использовать **GET** только там, где выполняется только непосредственная передача данных без выполнения активных действий.

Методы **GET**, **HEAD**, **PUT** и **DELETE** должны быть идемпотентными («idempotence») в том смысле, что повторное (более одного раза) выполнение запросов с помощью этих методов будет иметь тот же эффект, что и однократное выполнение.

По умолчанию параметры передаются в запрос в формате:

```
?name1=value1&name2=value2
```

Однако форматы упаковки параметров могут быть самые разные, например, в случае передачи файлов с использованием формы

```
enctype="multipart/form-data"
```

Также часто вместо формата представления параметров вида

```
/article.jsp?id=1234&page=42
```

может быть использована так называемая Friendly URL форма, более понятная для человека, не являющегося программистом, и может быть представлена подобным образом:

```
/article/1234/page/4
```

либо

```
/article_1234/page_4
```

Однако такая форма представления параметров обычно требует дополнительной работы со стороны программиста.

Механизмами протокола HTTP не предусмотрено сохранение информации о своем клиенте. Однако веб-приложение может требовать информации о клиенте, особенно такое, реализация которого предусматривает аутентификацию клиента без необходимости ее подтверждения при смене страниц. Кроме поддержки распределенной сессии, о которой разговор пойдет в главе 17, существуют и примитивные механизмы получения сведений о клиенте:

- файлы cookie — текстовые файлы, ассоциированные с приложением и сохраняемые на компьютере клиента.
- добавление в строку GET-запроса дополнительных параметров.
- скрытые HTML-теги вида

```
<input type="hidden" name="command" value="Login"/>
```

которые необходимо добавлять в каждую страницу приложения для сохранения целостности ее архитектуры.

В следующем примере приведен готовый к выполнению, но не к практическому применению, шаблон сервлета:

```
package by.bsu.first.servlet;
import java.io.IOException;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import jakarta.servlet.annotation.WebServlet;
@WebServlet("/FirstServlettest")// test -> clazz
public class FirstServlet extends HttpServlet {
    public FirstServlet() {
    }
    public void init() throws ServletException {
    }
    protected void doGet(HttpServletRequest request, HttpServletResponse
response)
throws ServletException,
IOException {
    response.setContentType("text/html");
```

```

        response.getWriter().print("This is " + this.getClass().getName() +
                                     ", using the GET
method");
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse
response)
                               throws ServletException, IOException {
        response.setContentType("text/html");
        response.getWriter().print("This is " + this.getClass().getName() + ",
using the POST method");
    }

    public void destroy() {
        super.destroy(); // Just puts "destroy" string in log
    }
}

```

С версии Servlet API 3.0 регистрация сервлетов в файле-дескрипторе **web.xml** необязательна. Конфигурация определяется с помощью аннотации **@WebServlet** и в расширенном виде с указанием имени сервлета в контексте:

```
@WebServlet(name = "FirstServletname", urlPatterns = { "/FirstServlettest",  
"*.do" })
```

```
public class FirstServlet extends HttpServlet {  
}
```

Аннотация **@WebServlet** может содержать и другие методы-члены, о чем будет сказано ниже. Также аннотированию подлежат методы реализации функциональности сервлета.

Практика включения HTML-кода в код сервлета не считается хорошей, так как эти действия «уводят» сервлет от его основной роли — контроллера приложения. Это приводит к росту объема кода сервлета, который на определенном этапе становится неконтролируемым и реализует вследствие этого антишаблон «Волшебный сервлет». Приведенный выше сервлет имеет признаки антишаблона, так как содержит метод **print()** для формирования кода HTML. Сервлет должен использоваться только для контроля реализации бизнес-логики приложения и обязан быть отделен как от непосредственного формирования текста ответа на запрос, так и от данных, необходимых для этого. Признаки наличия антишаблонов все же будут встречаться ниже, но это отступление сделано только с точки зрения компактности примеров.

Сервлет является компонентом веб-приложения, который будет называться **FirstProject** и размещаться в папке **/WEB-INF/classes** проекта.

1.5.1. Запуск контейнера сервлетов и размещение проекта

Здесь и далее применяется веб-сервер Apache Tomcat в качестве обработчика страниц JSP и сервлетов. Последняя версия может быть загружена с ресурса <https://tomcat.apache.org/>.

При установке Tomcat предложит значение порта по умолчанию **8080**, но во избежание конфликтов с иными Application Server рекомендуется присвоить другое значение, например **8089**.

Ниже приведены необходимые действия по запуску сервлета из предыдущего примера с помощью сервера Tomcat, который установлен в каталоге **/Apache Software Foundation/Tomcat[версия]**. В этом же каталоге размещаются следующие подкаталоги:

/bin — содержит файлы запуска монитора и контейнера сервлетов вида **tomcat[номер].exe**, **tomcat[номер]w.exe** и некоторые необходимые для этого библиотеки;

/lib — содержит библиотеки служебных классов, в частности Servlet API и JSP API; используемые внешние библиотеки (если они есть), упакованные в JAR-файлы;

/conf — содержит конфигурационные файлы, в частности, конфигурационный файл сервера **server.xml**;

/logs — в этот каталог записываются log-файлы, инициированные сервером;

/webapps — в этот каталог помещаются папки с веб-приложениями, содержащие в свою очередь сервлеты и другие компоненты конкретного приложения.

В каталог **/webapps** необходимо поместить папку **/FirstProject** с вложенным в нее сервлетом **FirstServlet**. Кроме того, папка **/FirstProject** должна содержать каталог **/WEB-INF**, в котором помещаются подкаталоги:

/classes — содержит class-файл сервлета **by.bsu.first.servlet.FirstServlet**;

/src — содержит исходный файл сервлета **FirstServlet.java** (опционально); а также **web.xml** — дескриптор доставки (развертывания) приложения располагается в каталоге **/WEB-INF**.

В файле **web.xml** для версий Servlet API, не превышающих 2.5, необходимо прописать имя и путь к сервлету. Кроме того в дескрипторном файле можно определять параметры инициализации, MIME-типы, mapping сервлетов и JSP, стартовые страницы и страницы с сообщениями об ошибках, а также параметры для безопасной авторизации и аутентификации. Этот файл можно сконфигурировать так, что путь к сервлету в браузере не будет совпадать с истинным именем класса сервлета. Например:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  id="WebApp_ID" version="2.5">
  <display-name>FirstProject</display-name>
  <servlet>
    <display-name>FirstServletdisplay</display-name>
    <servlet-name>FirstServletname</servlet-name>
    <servlet-class>by.bsu.first.servlet.FirstServlet</servlet-class>
  </servlet>
```



```

<servlet-mapping>
    <servlet-name>FirstServletname</servlet-name>
    <url-pattern>/FirstServlettest</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>default</servlet-name>
    <url-pattern>/resources/*</url-pattern>
</servlet-mapping>
<session-config>
    <session-timeout>30</session-timeout>
</session-config>
<welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
</welcome-file-list>
</web-app>

```

Здесь указано имя сервлета **FirstServletname**, путь к откомпилированному классу сервлета **FirstServlet.class**, а также URL-имя сервлета **FirstServlettest**, по которому происходит его вызов.

Для версии сервлетов 3.0 и выше теги мэппинга не нужны, так как эта информация определяется аннотацией.

```

<?xml                                version="1.0"                                encoding="UTF-8"?>
<web-app                                xmlns="https://jakarta.ee/xml/ns/jakartaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
https://jakarta.ee/xml/ns/jakartaee/web-app_5_0.xsd"
    version="5.0">
    <display-name>FirstProject</display-name>
    <session-config>
        <session-timeout>30</session-timeout>
    </session-config>
    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>

```

</web-app>

Таким образом, требуется выполнить действия:

1. компиляцию сервлета с указанием в **-cp** пути к архиву **servlet-api.jar**;
2. полученный файл класса **FirstServlet.class** поместить в каталог **/FirstProject/ WEB-INF/classes/by/bsu/first/servlet**;
3. в каталог **/FirstProject/WEB-INF** поместить файл конфигурации **web.xml**;
4. в каталог **/FirstProject** поместить файл стартовой JSP-страницы **index.jsp**;
5. разместить папку **/FirstProject** в каталог **/webapps** сервера Tomcat;
6. стартовать Apache Tomcat;
7. запустить браузер и ввести адрес
`http://localhost:8080/FirstProject/FirstServlettest`
8. При обращении к сервлету из другого компьютера вместо **localhost** следует указать IP-адрес или имя компьютера;
9. если вызывать сервлет из **index.jsp**, то тег **FORM** должен выглядеть следующим образом:

```
<form action="FirstServlettest">
```

```
<input type="submit" value="Execute"> </form>
```

Для запуска проекта в браузере набирается строка

`http://localhost:8080/FirstProject/index.jsp`

или

`http://localhost:8080/FirstProject/`

Сервлет будет вызван из JSP-страницы по URL-имени **FirstServlettest**, и в результате в браузер будет выведено:



Рисунок 1 - Вывод сервлета после вызова метода doGet()

1.5.2. Простая JSP-страница

Технология Java Server Pages (JSP) обеспечивает разделение динамической и статической частей страницы, результатом чего является возможность изменения дизайна страницы, не затрагивая динамическое содержание. Это свойство используется при разработке и поддержке страниц, так как дизайнерам нет необходимости знать, как работать с динамическими данными.

JSP-код состоит из специальных тегов и выражений, которые указывают контейнеру соответствие между тегами и java-кодом для генерации сервлета или его части. Таким образом поддерживается документ, который одновременно содержит и статическую страницу, и теги Java, управляющие страницей. Статические части HTML-страниц посылаются в виде строк в метод **write()**. Динамические части включаются прямо в код сервлета. С момента формирования ответа сервера страница ведет себя как обычная HTML-страница с ассоциированным сервлетом.

Чтобы создать простейшую JSP, достаточно взять HTML-страницу и заменить расширение **html** на **jsp**. Только в этом случае для запуска страницы необходим специальный application server с контейнером сервлетов и особое размещение самой страницы.

1.5.3. Взаимодействие сервлета и JSP

Страницы JSP и сервлеты никогда не следует использовать в информационных системах друг без друга. Причиной являются принципиально различные роли, которые играют данные компоненты в приложении. Страница JSP ответственна за формирование пользовательского интерфейса и отображение информации, переданной с сервера. Сервлет выполняет роль контроллера запросов и ответов, то есть принимает запросы от всех связанных с ним JSP-страниц, вызывает соответствующую бизнес-логику для их (запросов) обработки и в зависимости от результата выполнения решает, какую JSP поставить этому результату в соответствие.

При необходимости расширения функциональности системы *не следует* создавать дополнительные сервлеты. Сервлет в приложении должен быть *один*. При создании нового учебного приложения во избежание путаницы всегда следует создавать новый проект.

Для демонстрации взаимодействия JSP-страниц и сервлета будет решена задача определения времени между загрузкой страницы в браузер и нажатием кнопки на этой же странице.

Страница JSP с последующим вызовом другой JSP-страницы, отображающей результаты выполнения запроса.

```

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<html>
  <head>
    <title>JSP Timing</title>
  </head>
  <body>
    <h5>Счетчик времени от запуска приложения до нажатия кнопки</h5>
    <jsp:useBean id="calendar" class="java.util.GregorianCalendar" />
    <form name="Simple" action="timeaction" method="POST">
      <input type="hidden" name="time"
        value="\${calendar.timeInMillis}" />
      <input type="submit" name="button" value="Посчитать время" />
    </form>
  </body>
</html>

```

В результате запуска стартовой страницы проекта в браузер будет выведено:

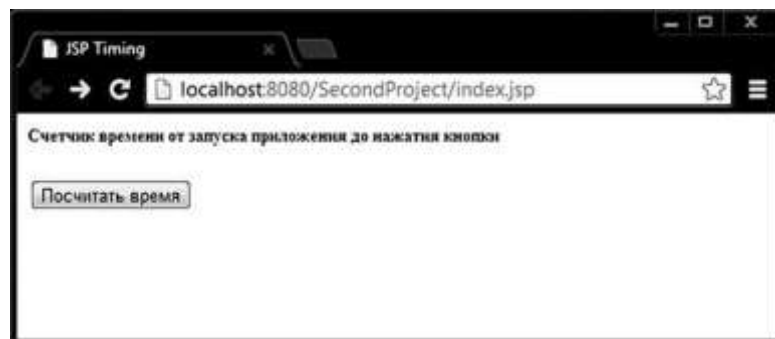


Рисунок 2 - Стартовая страница приложения

Кодировка для символов кириллицы задана с помощью директивы **page**. Action-тег **useBean** используется для создания объекта класса **GregorianCalendar** в области видимости JSP. Сервлет **TimeServlet** вызывается методом **POST**. Ему передается с переменной **calendar** информация о времени выполнения страницы.

```

package by.bsu.second.servlet;
import java.io.IOException;
import java.util.GregorianCalendar;

```

```

import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import jakarta.servlet.annotation.WebServlet;
@WebServlet("/timeaction")
public class TimeServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse
response)
                                                                    throws ServletException,
IOException {
        processRequest(request, response);
    }
    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse
response)
                                                                    throws
ServletException, IOException {
        processRequest(request, response);
    }
    protected void processRequest(HttpServletRequest request,
                                                                    HttpServletResponse response)
                                                                    throws ServletException,
IOException {
        GregorianCalendar gc = new GregorianCalendar();
        String timeJsp = request.getParameter("time");
        float delta = ((float) (gc.getTimeInMillis()-
Long.parseLong(timeJsp)))/1_000;
        request.setAttribute("res", delta);
        request.getRequestDispatcher("/jsp/result.jsp").forward(request,
response);
    }
}

```

```
}  
}
```

Обмен информацией между JSP и сервлетом чаще всего осуществляется с помощью атрибутов объектов **HttpServletRequest**, **HttpSession**, **ServletContext**. Вызов **result.jsp** из сервлета в данном случае производится методом **forward()** интерфейса **RequestDispatcher**.

Для вызова JSP по относительному пути применяется метод **forward()**, для обращения к JSP по абсолютному пути используется метод **sendRedirect()**. Отличие этих методов состоит в том, что с методом **forward()** передается уже существующий объект запроса **request**, а при вызове метода **sendRedirect()** формируется новый запрос. Информацию в последнем случае следует передавать с другими объектами. К тому же метод **forward()** срабатывает быстрее.

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<html>
  <head>
    <title>Result Page</title>
  </head>
  <body>
    <p>Кнопка нажата через ${res} сек </p>
  </body>
</html>
```

После обращения к сервлету и выполнения им действий по созданию атрибута объекта **request** будет вызвана страница **result.jsp**, интерпретирующая результат в виде:



Рисунок 3 - Вывод информации страницей result.jsp

В данном коде используется Expression Language (EL) для доступа к атрибутам запроса в виде **`${calendar.timeInMillis}`** и **`${res}`**.

Кроме сокращенной нотации вызова методов с префиксом **get** по умолчанию можно использовать полную форму вызова метода с помощью EL, а именно: **`${calendar.getTimeInMillis()}`** или **`${calendar.isLeapYear(2021)}`**. Методы, которые не возвращают значение, вызывать нельзя.

1.5.4. Интерфейс ServletContext

Приложению ставится в соответствие единственный экземпляр, ассоциированный с контейнером сервлетов, реализующий **ServletContext**. Контекст выполнения сервлета предоставляет средства для общения с application-сервером и позволяет получать информацию о среде выполнения, а также использовать ресурсы совместно с другими объектами приложения. В частности, можно добавить/извлечь/удалить атрибуты, извлечь параметры контекста или записать информацию в log-файл. Получить ссылку на объект **ServletContext** можно вызовом метода **getServletContext()** на экземпляре сервлета.

Ряд методов предназначен для управления атрибутами, с помощью которых передается информация между различными компонентами приложения (JSP, сервлетами):

void setAttribute(String name, Object value) — добавляет атрибут;

Object getAttribute(String name) — получает значение атрибута по имени;

Enumeration getAttributeNames() — получает список имен атрибутов;

void removeAttribute(String name) — удаляет атрибут из контекста.

Другими словами, используя объект, реализующий **ServletContext**, можно связывать объекты с экземплярами сервлета, сессии и запроса.

Приложению могут быть заданы параметры инициализации (параметры контекста), получить доступ к которым можно с помощью методов:

String getInitParameter(String paramName) — извлечение параметра контекста по имени;

Enumeration getInitParameterNames() — извлечение набора имен параметров контекста, например, из кода метода сервлета

```
String uploadPath = this.getServletContext().getInitParameter("upload_path");
```

Для получения значения, необходимо, чтобы дескрипторный файл приложения **web.xml** содержал тег **<context-param>** со значением и именем параметра в виде:

```
<context-param>
```

```
  <param-name>upload_path</param-name>
```

```
  <param-value>C:\\upload</param-value>
```

```
</context-param>
```

```
<context-param>
```

```
  <param-name>admin_mail</param-name>
```

```
  <param-value>blinov@gmail.com</param-value>
```

```
</context-param>
```

Экземпляр **ServletContext** может применяться для обмена информацией между сервлетами приложения. Но в большей части промышленных приложений применяется односерветная модель, поэтому использовать объект контекста следует только для хранения информации о конфигурации приложения или общей информации для всех клиентов.

Следующие методы позволяют получить из контекста сервлета базовую информацию:

String getContextPath() — возвращает путь к корню веб-приложения, находящегося в папке контейнера сервлетов и используется для доступа к jsp и другим компонентам веб-приложения;

String getRealPath(String path) — определение истинного маршрута файла относительно каталога, в котором сервер хранит текстовые и медиа файлы;

```
String path= this.getServletContext().getRealPath("/");
FileReader reader = new FileReader(path + "text/data.txt");
int code = reader.read();
char ch = (char)code;
request.setAttribute("code_char", code);
request.setAttribute("symbol", ch);
```

String getServerInfo() — возвращает имя и номер версии контейнера сервлетов;

String getServletContextName() — возвращает имя веб-приложения, заданное в дескрипторе приложения в теге **<display-name>**;

void log(String msg) — запись лога с сообщением **msg**;

void log(String msg, Throwable t) — запись лога с текстовым сообщением **msg** и сообщением, извлекаемым из исключения.

Интерфейс ServletConfig

Данный объект ассоциируется с самим сервлетом.

Параметры инициализации сервлета содержатся в экземпляре **ServletConfig**. Получить доступ к нему можно методом **getServletConfig()**, вызываемым на экземпляре сервлета.

Методы интерфейса:

Enumeration getInitParameterNames() — получение списка имен параметров инициализации сервлета;

String getInitParameter(String name) — определение значения конкретного параметра по его имени;

String getServletName() — определение имени сервлета, с которым связан текущий объект **ServletConfig**;

String getServletContext() — получение экземпляра контекста.

Чтобы задать параметры инициализации сервлета **EMailServlet**, который будет рассмотрен позже, необходимо в тег **<servlet>** дескрипторного файла **web.xml** вложить тег **<init-param>** с описанием имени и значения параметра в виде:

```
<servlet>
<servlet-name>EMailServlet</servlet-name>
<servlet-class>by.bsu.mail.servlet.EMailServlet</servlet-class>
  <init-param>
    <param-name>mail.smtps.host</param-name>
    <param-value>smtp.gmail.com</param-value>
  </init-param>
  <init-param>
    <param-name>mail.smtp.port</param-name>
    <param-value>465</param-value>
  </init-param>
  <init-param>
    <param-name>smtps.auth.user</param-name>
    <param-value>blinov@gmail.com</param-value>
  </init-param>
  <init-param>
    <param-name>smtps.auth.pass</param-name>
    <param-value>real_password</param-value>
  </init-param>
  <init-param>
    <param-name>mail.transport.protocol</param-name>
    <param-value>smtps</param-value>
  </init-param>
</servlet>
```

или для версии 3.0 и выше:

```
@WebServlet(
  urlPatterns = {"/mailservlet"},
  initParams = {
```

```

        @WebInitParam(name = "mail.smtps.host", value = "smtp.gmail.com"),
        @WebInitParam(name = "mail.smtp.port", value = "465"),
        @WebInitParam(name = "smtps.auth.user", value = "blinov@gmail.com"),
        @WebInitParam(name = "smtps.auth.pass", value = "real_password"),
        @WebInitParam(name = "mail.transport.protocol", value = "smtp"),
        @WebInitParam(name = "mail.host", value = "smtp.gmail.com"),
        @WebInitParam(name = "mail.smtp.auth", value = "true"),
        @WebInitParam(name = "mail.smtp.socketFactory.port", value = "465"),
        @WebInitParam(name = " mail.smtp.socketFactory.class", value =
"javax.net.ssl.SSLSocketFactory"),
        @WebInitParam(name = " mail.smtp.socketFactory.fallback", value =
"false"),
        @WebInitParam(name = " mail.smtp.quitwait", value = "false")
    }
)
public class MailServlet extends HttpServlet {
}

```

Тогда для доступа к параметрам инициализации сервлета и их дальнейшего использования в приложении, например, в коде метода **init()** сервлета следует применить:

```

ServletConfig config = this.getServletConfig();
// определение набора имен параметров инициализации
Enumeration<String> e = config.getInitParameterNames();
while(e.hasMoreElements()){
    // определение имени параметра инициализации
    String name = e.nextElement();
    // определение значения параметра инициализации
    String value = sc.getInitParameter(name);
}
или
this.getInitParameterNames()

```

```
.asIterator()
```

```
.forEachRemaining(o -> logger.log(Level.INFO, o + "=" + getInitParameter(o)));
```

1.5.5. Интерфейс **HttpServletRequest**

Информация от клиента веб-приложения отправляется серверу в виде объекта запроса типа **HttpServletRequest**. Данный интерфейс является производным от интерфейса **ServletRequest**. Используя методы интерфейса **ServletRequest**, можно получить: информацию о сервлете, доступ к сессии и параметрам запроса, управление атрибутами и кодировками, а также детали протокола HTTP.

При обращении к серверу с запросом от клиента передаются параметры и их значения. Для разбора параметров и извлечения их значений применяются методы:

String getParameter(String name) — определение значения параметра по его имени или **null**, если параметр с таким именем не задан;

String[] getParameterValues(String name) — определение всех значений параметра по его имени;

Enumeration getParameterNames() — определение ссылки на список имен всех параметров через объект типа **Enumeration**;

getParameterMap() —

String getCharacterEncoding() — определение символьной кодировки запроса;

void setCharacterEncoding(String code) — задание символьной кодировки запроса;

String getContentType() — определение MIME-типа (Multipurpose Internet Mail Extension) пришедшего запроса;

String getProtocol() — определение названия и версии протокола;

String getServerName(), getServerPort() — определение имени сервера, принявшего запрос, и порта, на котором запрос был принят сервером соответственно;

String getRemoteAddr(), getRemoteHost() — определение IP-адреса клиента, от имени которого пришел запрос, и его имени соответственно;

String getRemoteUser() — определение имени пользователя, выполнившего запрос;

String getQueryString() — извлечение строки HTTP-запроса;

String getMethod() — определение имени метода доступа к ресурсам, на основе которого построен запрос;

ServletInputStream **getInputStream()**, **BufferedReader** **getReader()** — получение ссылки на поток, ассоциированный с содержимым полученного запроса. Первый метод возвращает ссылку на байтовый поток **ServletInputStream**, а второй — на объект **BufferedReader**. В результате можно прочитать любой байт из полученного объекта-запроса. Если метод **getReader()** был вызван после вызова **getInputStream()** для этого запроса, то генерируется исключение **IllegalStateException** и наоборот.

Непосредственно в интерфейсе **HttpServletRequest** объявлен ряд методов, позволяющих связывать внешние ресурсы с запросом:

void setAttribute(String name, Object ob) — установка значения атрибута компонента, являющегося внутренним параметром для передачи информации между компонентами приложения, например, от сервлета к странице JSP или другому сервлету;

Enumeration **getAttributeNames()** — извлечение перечисления имен атрибутов;

Object **getAttribute(String name)** — извлечение значения переданного атрибута по имени;

void removeAttribute(String name) — удаление атрибута по имени;

Cookie[] **getCookies()** — извлечение массива cookie, полученного с запросом. Файл cookie — маленький символьный файл, сохраняемый приложением на стороне клиента;

Ниже рассматривается применение некоторых методов интерфейса **HttpServletRequest** для получения данных о запросе, посылаемом методом **GET** и генерации ответа клиенту. Вызов методов в сервлете **FirstServlet** проекта **FirstProject** на объекте **request**.

```
PrintWriter out = response.getWriter();  
out.println(request.getMethod());  
out.println(request.getRequestURL());  
out.println(request.getProtocol());  
out.println(request.getRemoteAddr());  
out.println(request.getContextPath());  
out.println(request.getScheme());
```

и передача результатов в браузер с помощью потока **PrintWriter** дадут следующие результаты:



Рисунок 4 - Извлечение информации из запроса

Из запроса также можно извлечь информацию о заголовке запроса, которая оказывается достаточно исчерпывающей:

```
Enumeration<String> names = request.getHeaderNames();  
while (names.hasMoreElements()) {  
    String headerName = names.nextElement();  
    String value = request.getHeader(headerName);  
    out.println(headerName + "=" + value);  
}
```

Последний пример следует попробовать вызывать на разных компьютерах с различными конфигурациями, используя несколько браузеров, чтобы оценить выводимую информацию.

Результат выполнения фрагмента кода может выглядеть:

referer=http://localhost:8089/tom2_war_exploded/index.jsp

cookie=JSESSIONID=C5AFB1AF7C0147279C2AE63179C745A4; Idea-
e935d2c9=5fe8e02c-7afc-4206-b75f-f8aff3c45f22

accept-language=ru-RU,ru;q=0.8,en-US;q=0.5,en;q=0.3

upgrade-insecure-requests=1

host=localhost:8089

connection=keep-alive

accept-encoding=gzip, deflate

accept=text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;
q=0.8

user-agent=Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:65.0)
Gecko/20100101 Firefox/65.0

Таким образом, при обращении клиента по адресу URL контейнер сервлета перехватывает запрос и вызывает метод **doGet()** или **doPost()**. Эти методы вызываются после конфигурации и создания объектов, реализующих интерфейсы **HttpServletRequest**, **HttpServletResponse**. Задача методов **doGet()** и **doPost()** — взаимодействие с HTTP-запросом клиента и создание HTTP-ответа, основанного на данных запроса и результатах взаимодействия с бизнес-логикой.

1.5.6. Интерфейс `HttpServletResponse`

Генерируемые сервлетами данные пересылаются серверу-контейнеру с помощью объектов, реализующих интерфейс **`ServletResponse`**, а сервер, в свою очередь, формирует и пересылает ответ клиенту, инициировавшему запрос.

В интерфейсе **`HttpServletResponse`**, наследующем интерфейс

`ServletResponse`, есть несколько важных методов:

`void setContentType(String type)` — установка MIME-типа генерируемых документов;

`void addCookie(Cookie cookies)` — добавление файла cookie к объекту ответа для пересылки на клиентский компьютер;

`void sendError(int sc, String msg)` — переход на error-page с сообщением о возникших ошибках, где **`sc`** — код ошибки, **`msg`** — текстовое сообщение;

`void sendRedirect(String location)` — переход по указанному адресу, без передачи экземпляра **`request`**.

Можно получить ссылки на потоки вывода одним из двух методов:

`ServletOutputStream getOutputStream()` — извлечение ссылки на поток **`ServletOutputStream`** для передачи бинарной информации;

`PrintWriter getWriter()` — извлечение ссылки на поток типа **`PrintWriter`** для передачи символьной информации;

Если метод **`getOutputStream()`** уже был вызван для этого ответа, то генерируется **`IllegalStateException`**. Обратное также верно.

```
httpResponse.setHeader("Cache-Control","no-cache, no-store, must-revalidate");
```

```
response.addHeader("Cache-Control", "post-check=0, pre-check=0");
```

```
httpResponse.setHeader("Pragma","no-cache");
```

```
httpResponse.setDateHeader ("Expires", 0);
```

1.5.7. Атрибуты и параметры

Какие объекты и каким образом могут получать, передавать, предоставлять информацию? Ответ представлен в таблице:

	Атрибут	Параметр
Носитель	page (PageContext) request(HttpServletReques t) session (HttpSession) application (ServletContext)	context (инициализация) servlet (инициализация) request (запрос)
Метод установки	setAttribute(String name, Object value)	добавить параметры можно только к request
Метод извлечения	getAttribute(String name)	getInitParameter(Stri ng name) getParameter(String name)
Возвращаемы й тип	Object	String и String[]

Рисунок 5 - Таблица носителей информации

Чаще всего для передачи информации между элементами системы используются объекты классов **HttpServletRequest** и **HttpSession**.

1.5.8. Многопоточность и электронная почта

Распределенное приложение может быть эффективным только в случае, если оно способно принимать информацию от физически удаленных клиентов и производить ее обработку максимально быстро. Увеличить скорость работы веб-приложения позволяет применение многопоточных режимов функционирования частей приложения.

Рассылка электронной почты, в том числе и автоматическая, является стандартным процессом при использовании веб-приложений. Собственный почтовый сервер создать легко, только необходимо указать адрес почтового сервера, который будет использован в качестве транспорта.

При построении простейшего почтового сервера требуются интерфейсы API JavaMail. Библиотека JavaMail содержит классы, позволяющие моделировать систему электронной почты. Класс **javax.mail.Session** представляет сеанс почтовой связи, класс **javax.mail.Message** — почтовое сообщение, класс **javax.mail.internet.InternetAddress** — адрес электронной почты.

Необходимую библиотеку, содержащую, в частности, архив **mail.jar**, следует загрузить по адресу oracle.com/technetwork/java/javamail/. Далее следует добавить этот файл в каталог jar-файлов серверу приложений (**lib** для Tomcat).

В случае автономного запуска без использования промежуточного сервиса для успешной работы почтового приложения бывает необходимо запустить почтовую программу James, являющуюся также одним из проектов **apache.org**.

Ниже приведена страница JSP, содержащая форму для заполнения основных полей: «**Кому**» — «**Send to**», «**Тема сообщения**» — «**Subject**», «**Текст письма**» — «**Body**»

```
<%@ page language="java" contentType= "text/html; charset=UTF-8"
pageEncoding="UTF-8"%>
```

```
<html>
  <head>
    <title>JavaMail</title>
  </head>
  <body>
    <form action="email-action" method="POST">
      <table>
        <tr>
          <td>Send to:</td>
          <td>
            <input type="text" name="to" />
          </td>
        </tr>
        <tr>
          <td>Subject:</td>
          <td>
            <input type="text" name="subject" />
          </td>
        </tr>
      </table>
      <hr />
      <textarea type="text" name="body" rows="5" cols="45">
        Message text
      </textarea>
      <br />
      <br />
    </form>
  </body>
</html>
```

```

        <input type="submit" value="Send message!" />
    </form>
</body>
</html>

```



Рисунок 6 - Формирование запроса на отправку письма

Параллельные процессы по отправке письма и предложению пользователю в это же самое время создать новое письмо организуются с применением потока в следующем сервлете.

```

package by.bsu.mail.servlet;
import java.io.IOException;
import java.util.Properties;
import jakarta.servlet.ServletContext;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import by.bsu.mail.action.MailThread;
import jakarta.servlet.annotation.WebServlet;

@WebServlet("/email-action")/// with init params
public class MailServlet extends HttpServlet {

```

```

@Override
protected void doPost(HttpServletRequest request, HttpServletResponse
response)

throws ServletException, IOException {

Properties properties = new Properties();
ServletContext context = getServletContext();
String filename = context.getInitParameter("mail");
// запрос параметров почтового сервера из mail.properties
// загрузка параметров почтового сервера в объект свойств
properties.load(context.getResourceAsStream(filename));
MailThread mailOperator = new MailThread(request.getParameter("to"),
request.getParameter("subject"),
request.getParameter("body"), properties);
// запуск процесса отправки письма в отдельном потоке
mailOperator.start();
// переход на страницу с предложением о создании нового письма
request.getRequestDispatcher("/send.jsp").forward(request, response);
}
}

```

Конфигурация почтового сервера размещена в файле **/config/mail.properties**.

```
#####
```

```
##### Session properties #####
##### mail.smtp.host=smtp.gmail.com
mail.smtp.port=465 mail.user.name=REAL~MAIL~NAME@gmail.com
mail.user.password=REAL~PASSWORD
```

```

<html>
  <head>
    <title>Send Result</title>
  </head>
  <body>
    <p>Sending in progress...</p>

```

```
<a href="index.jsp">Return to new mail page</a>
</body>
</html>
```

В результате в браузер будет выведено:



Рисунок 7 - Формирование запроса на отправку письма

Здесь **Return to new mail page** представляет собой активную ссылку, перенаправляющую при запуске на **index.jsp** для создания еще одного письма. Процесс же отправки письма будет функционировать независимо от дальнейшей работы приложения.

Недостатком такого применения многопоточности будет отсутствие информации об успешном отправлении письма или возникших при этом ошибках и исключениях. В данной ситуации такую информацию можно только сохранить в логах или выдавать на специальных страницах. Поэтому современные почтовые онлайн системы и не используют многопоточность в таком виде, так как пришлось бы делать достаточно сложный интерфейс по сообщению пользователю об ошибках при отправке почты, что сделало бы работу в системе более сложной и противоречивой.

```
package by.bsu.mail.action;
import java.util.Properties;
import javax.mail.Message;
import javax.mail.MessagingException;
import javax.mail.Session;
import javax.mail.Transport;
import javax.mail.internet.AddressException;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;
import by.bsu.mail.creator.SessionCreator;
```

```

public class MailThread extends Thread {
    private MimeMessage message;
    private String sendToEmail;
    private String mailSubject;
    private String mailText;
    private Properties properties;
    public MailThread(String sendToEmail, String mailSubject, String mailText,
        Properties properties) {
this.sendToEmail = sendToEmail;
this.mailSubject = mailSubject;
this.mailText = mailText;
this.properties = properties;
    }
    private void init() {
        // объект почтовой сессии
        Session mailSession = (new SessionCreator(properties)).createSession();
        mailSession.setDebug(true);
        // создание объекта почтового сообщения
        message = new MimeMessage(mailSession);
        try {
            // загрузка параметров в объект почтового сообщения
            message.setSubject(mailSubject);
            message.setContent(mailText, "text/html");
            message.setRecipient(Message.RecipientType.TO, new
                InternetAddress(sendToEmail));
        } catch (AddressException e) {
            System.err.print("Некорректный адрес:" + sendToEmail + " " + e);//log
        } catch (MessagingException e) {
            System.err.print("Ошибка формирования сообщения" + e);//log
        }
    }
}

```

```

    }
    }
    public void run() {
init();
    try {
        // send mail
        Transport.send(message);
    }
        catch (MessagingException e) {
            System.err.print("Ошибка при отправлении сообщения" + e);//log
        }
    }
}

package by.bsu.mail.creator;
import java.util.Properties;
import javax.mail.PasswordAuthentication;
import javax.mail.Session;

public class SessionCreator {
    private String smtpHost;
    private String smtpPort;
    private String userName;
    private String userPassword;
    private Properties sessionProperties;
    public SessionCreator(Properties configProperties) {
smtpHost = configProperties.getProperty("mail.smtp.host");
smtpPort = configProperties.getProperty("mail.smtp.port");
userName = configProperties.getProperty("mail.user.name");
userPassword = configProperties.getProperty("mail.user.password");
        // загрузка параметров почтового сервера в свойства почтовой сессии
    }
}

```



```

sessionProperties = new Properties();
sessionProperties.setProperty("mail.transport.protocol", "smtp");
sessionProperties.setProperty("mail.host", smtpHost);
sessionProperties.put("mail.smtp.auth", "true");
sessionProperties.put("mail.smtp.port", smtpPort);
sessionProperties.put("mail.smtp.socketFactory.port", smtpPort);
sessionProperties.put("mail.smtp.socketFactory.class",
    "javax.net.ssl.SSLSocketFactory");
sessionProperties.put("mail.smtp.socketFactory.fallback", "false");
sessionProperties.setProperty("mail.smtp.quitwait", "false");
}
public Session createSession() {
return Session.getDefaultInstance(sessionProperties,
new javax.mail.Authenticator() {
    protected PasswordAuthentication getPasswordAuthentication() {
        return new PasswordAuthentication(userName, userPassword);
    }
});
}
}

```

Параметры контекста сервлета в файле **web.xml** для данного приложения представлены в виде:

```

<context-param>
    <param-name>mail</param-name>
    <param-value>/WEB-INF/classes/config/mail.properties</param-value>
</context-param>

```

1.6. Java Server Page

Технология Java Server Pages (JSP) была разработана компанией Sun Microsystems (в настоящее время поглощена компанией Oracle), чтобы облегчить создание страниц с динамическим содержанием.

В то время как сервлеты наилучшим образом подходят для выполнения контролирующей функции приложения в виде обработки запросов и определения содержания и вида ответа, страницы JSP выполняют функцию отображения результатов работы приложения в виде текстовых документов типа HTML, XML, WML и некоторых других. JSP поддерживают как JavaScript, так и HTML-теги. JavaScript обычно используется, чтобы добавить функциональные возможности на уровне HTML-страницы.

Принято разделять динамическое и статическое содержимое JSP. *Динамические ресурсы.* Результаты их деятельности изменяются во время выполнения приложения. Обычно представлены в виде выражений Expression Language, библиотек тегов и тегов разработчика.

Статические ресурсы. Не изменяются сами в процессе работы (HTML, JavaScript, изображения и т. д.).

Смысл разделения динамического и статического содержания в том, что статические ресурсы могут находиться под управлением HTTP-сервера в то время, как динамические нуждаются в движке (JSP Engine) и иногда в доступе к уровню данных.

Рекомендуется разделить и разрабатывать параллельно две части приложения: часть, состоящая только из динамических ресурсов, и часть, состоящая только из статических ресурсов.

Некоторые преимущества использования JSP-технологии над другими методами создания динамического содержания страниц:

- разделение динамического и статического содержания.
- независимость от платформы. Технологии Java не зависят от платформы, следовательно, JSP могут выполняться практически на любом веб-сервере. Разрабатывать JSP можно на любой платформе;
- многократное использование компонентов. Использование JavaBeans и Enterprise JavaBeans (EJB) позволяет многократно использовать компоненты, что ускоряет создание веб-приложений;
- теги. Спецификация JSP содержит библиотеку стандартных тегов JSTL, позволяет разработчику создавать собственные теги, кроме того, теги обеспечивают возможность использования JavaBean и обращение к классам бизнес-логики.

Содержимое Java Server Pages (теги HTML, теги JSP и скрипты) переводится в сервлет код-сервером. Этот процесс ответствен за трансляцию как динамических, так и статических элементов, объявленных внутри файла JSP. Об архитектуре сайтов, использующих JSP/Servlet-технологии, часто говорят, как о thin-client (использование ресурсов клиента незначительно), потому что большая часть логики выполняется на сервере.

В спецификации JSP 1.2 были объявлены только пять основных тегов:

`<%@ директива %>` — используется для установки параметров серверной страницы JSP;

`<%! объявление %>` — (нежелателен в современном программировании) содержит поля и методы, которые вызываются в `expression`-блоке и становятся полями и методами генерируемого сервлета. Объявление не должно производить запись в выходной поток **out** страницы, но может быть использовано в скриптелях и выражениях;

`<% скриплет %>` — (нежелателен) вживление `java`-кода в JSP-страницу. Скриплеты обычно используют маленькие блоки кода и выполняются во время обработки запроса клиента. Когда все скриплеты собираются воедино в том порядке, в котором они записаны на странице, они должны представлять собой правильный код языка программирования. Контейнер помещает код `Java` в метод `_jspService(parameters)` на этапе трансляции;

`<%= вычисляемое выражение %>` — (нежелателен) содержит операторы языка `Java`, которые вычисляются, после чего результат вычисления преобразуется в строку **String** и посылается в поток **out**;

`<%-- JSP-комментарий --%>` — комментарий, который не отображается в исходных кодах JSP-страницы после этапа выполнения.

В конце прошлого тысячелетия существовала техника создания страницы на основе одного скриплета, в который вставлялся большой фрагмент кода, что в корне противоречило концепции JSP. В современных приложениях скриплеты, выражения и объявления не применяются вовсе из-за нарушения «теговой» структуры страницы.

1.6.1. Жизненный цикл

Процессы, выполняемые с файлом JSP при *первом* вызове: — браузер делает первый запрос к JSP;

- JSP-engine анализирует содержание файла JSP и создает сервлет с кодом, основанным на исходном тексте JSP, при этом engine транслирует статическое содержимое в методы вывода и помещает его в метод `_jspService()`. Полученный сервлет будет ответственен за генерацию статических элементов, определенных во время разработки. Динамические элементы транслируются в `java`-код;
 - код сервлета компилируется в файл **.class**,
 - создается объект;
 - выполняется метод **init()** построенного сервлета. В итоге сервлет на основе JSP загружен в контейнер сервлетов и готов к работе;
- вызывается метод `_jspService()`, и `jsp`-сервлет логически исполняется, формируя экземпляр **response**;

- комбинация статического HTML и графики вместе с результатами исполнения динамических элементов, определенных в оригинале JSP, пересылаются браузеру через выходной поток объекта ответа **HttpServletResponse**.

- метод **destroy()** один раз при выгрузке

Следующие обращения к файлу JSP просто вызовут метод **_jspService()** сервлета. Сервлет используется до тех пор, пока сервер не будет остановлен или сервлет не будет выгружен из контейнера. Результат работы JSP можно легко представить, зная правила трансляции JSP в сервлет, в частности, всего метод **_jspService()**.

```
<html>
  <head><title>Simple</title></head>
  <body>
    Hello, Bender
  </body>
</html>
```

то в результате запуска в браузер будет выведено:
Hello, Bender

а код сервлета на базе JSP будет выглядеть следующим образом:

```
package org.apache.jsp;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;

public final class simple_jsp extends org.apache.jasper.runtime.HttpJspBase
implements org.apache.jasper.runtime.JspSourceDependent {
    private static java.util.List _jspx_dependants;
    public Object getDependants() {
        return _jspx_dependants;
    }
    public void _jspService(HttpServletRequest request, HttpServletResponse
response) throws java.io.IOException, ServletException {
        JspFactory _jspxFactory = null;
        PageContext pageContext = null;
        HttpSession session = null;
        ServletContext application = null;
        ServletConfig config = null;
        JspWriter out = null;
        Object page = this;
        JspWriter _jspx_out = null;
        PageContext _jspx_page_context = null;
        try {
            _jspxFactory = JspFactory.getDefaultFactory();
            response.setContentType("text/html");
            pageContext = _jspxFactory.getPageContext(this, request, response, null, true,
8192, true);
            _jspx_page_context = pageContext;
            application = pageContext.getServletContext();
            config = pageContext.getServletConfig();
            session = pageContext.getSession();
```

```

        out = pageContext.getOut();
        _jspx_out = out;
        out.write("<html><head>\r\n");
        out.write("<title>Simple</title>\r\n");
        out.write("</head>\r\n");
        out.write("<body>\r\n");
        out.write("Hello, Bender\r\n");
        out.write("</body></html>");
    } catch (Throwable t) {
        if (!(t instanceof SkipPageException)) {
            out = _jspx_out;
            if (out != null && out.getBufferSize() != 0)
                out.clearBuffer();
            if (_jspx_page_context != null)
                _jspx_page_context.handlePageException(t);
        }
    } finally {
        if (_jspxFactory != null)
            _jspxFactory.releasePageContext(_jspx_page_context);
    }
}
}
}

```

Следует обратить внимание на объявление в методе **_jspService()** целого ряда локальных переменных. Ко всем этим переменным возможен доступ прямо из текста JSP по имени в выражениях Expression Language.

1.6.2. Неявные объекты в expression language

JSP expression language определяет свое множество неявных объектов, пересекающееся с множеством неявных объектов для JSP. Страница JSP с помощью скриплетов всегда имеет доступ ко всем локальным переменным и параметрам метода `_jspService()` сервлета, создаваемым jsp-engine на основе этой же страницы. Наиболее очевидные:

- **pageContext** — определяет контекст JSP-страницы и предоставляет доступ к другим неявным объектам. Объект класса **javax.servlet.jsp.PageContext**. Область видимости в пределах страницы;
- **servletContext** — определяет контекст для сервлета JSP страницы и любого веб-компонента, содержащегося в этом приложении;
- **param** — содержит все параметры, переданные странице в виде параметров формы или параметров адресной строки;
- **paramValues** — содержит список всех значений параметров, переданных на страницу;
- **initParam** — содержит конфигурационные параметры страницы, заданные в файле **web.xml**;
- **cookie** — содержит список переменных, переданных с файлом cookie;
- **request** — представляет запрос клиента. Обычно объект является экземпляром класса, реализующего интерфейс **javax.servlet.http.HttpServletRequest**. Для протокола, отличного от HTTP, это будет объект реализации интерфейса **javax.servlet.ServletRequest**.
- **response** — представляет собой ответ клиенту. Обычно объект является экземпляром класса, реализующего интерфейс **javax.servlet.http.HttpServletResponse**. Для протокола, отличного от HTTP, это будет объект реализации интерфейса **javax.servlet.ServletResponse**. Область видимости в пределах страницы;
- **config** — содержит параметры конфигурации сервлета и является экземпляром класса **javax.servlet.ServletConfig**. Область видимости в пределах страницы;
- **session** — создается контейнером в соответствии с протоколом HTTP и является экземпляром класса **javax.servlet.http.HttpSession**, предоставляет информацию о сессии клиента, если такая была создана. Область видимости в пределах сессии;
- **out** — содержит выходной поток сервлета. Информация, посылаемая в этот поток, передается клиенту. Объект является экземпляром класса **javax.servlet.jsp.JspWriter**. Область видимости в пределах страницы;
- **page** — явная ссылка **this** для текущего экземпляра данной страницы является объектом **java.lang.Object**. Область видимости в пределах страницы;
- **exception** — представляет исключение одного из подклассов **java.lang.Throwable**, которое передается странице, помеченной как `error page`. Область видимости в пределах страницы ошибок.

К неявным объектам возможно и обращение по имени, не допускающее никаких сокращений.

Существуют также объекты, которые обеспечивают доступ к различным областям видимости, а именно: **pageScope**, **requestScope**, **sessionScope**, **applicationScope**.

```
<%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
```

```
<html>
```

```
<head><title>EL for pageContext</title></head>
```

```
<body>
```

```
Путь к контексту : ${ pageContext.request.contextPath }
```

```
<br/> Имя хоста : ${ header["host"] }
```

```
<br/>Тип и кодировка контента : ${pageContext.response.contentType}
```

```
<br/> Кодировка ответа : ${pageContext.response.characterEncoding}
```

```
<br/> ID сессии : ${pageContext.request.session.id}
```

```
<br/>Время создания сессии в мсек :  
${pageContext.request.session.creationTime}
```

```
<br/>Время последнего доступа к сессии:  
${pageContext.request.session.lastAccessedTime}
```

```
<br/> Имя сервлета : ${pageContext.servletConfig.servletName}
```

```
</body>
```

```
</html>
```

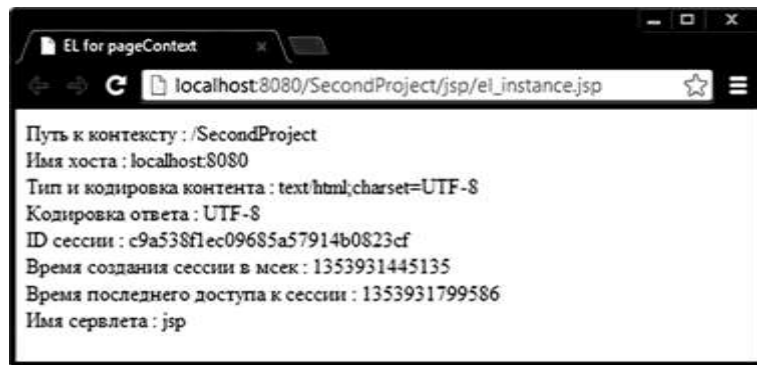



Рисунок 8 - Обращение к неявным объектам

1.6.3. Стандартные элементы action

Большинство стандартных тегов, содержащих символ %, в современном программировании не применяются. Используются action-теги версии JSP 2.0. Они позволяют создавать правильные JSP — документы. В тоже время декларации, выражения и скриптлеты остаются под негласным запретом из-за явного внедрения java-кода в JSP.

Action-теги:

- **<jsp:include>** — позволяет включать статические и динамические ресурсы в контекст генерируемой страницы при запросе вида

```
<jsp:include page="относительный URL" flush="true"/>
```

Включаемая страница не может объявлять собственные заголовки, определяющие тип или кодировку;

- **<jsp:declaration>** — (*нежелателен*) объявление, аналогичен тегу `<%! %>`;
- **<jsp:scriptlet>** — (*нежелателен*) скриптлет, аналогичен тегу `<% %>`;
- **<jsp:expression>** — (*нежелателен*) выражение, аналогичен тегу `<%= %>`;
- **<jsp:text>** — вывод текста;
- **<jsp:useBean>** — объявление экземпляра компонента Java Bean или класса, обладающего **public**-конструктором по умолчанию. Если экземпляр с указанным идентификатором не существует, то он будет создан в области видимости **scope** со значением **page** (страница), **request** (запрос), **session** (сессия) или **application** (приложение). Объявляется, как правило, с атрибутами **id** (имя объекта), **class** (полное имя класса), **type** (по умолчанию **class**).

```
<jsp:useBean id="ob" scope="request" class="by.bsu.test.Message" />
```

что идентично java-коду:

```
by.bsu.test.Message ob = new by.bsu.test.Message();
```

Создан объект **ob** класса **Message**, и в дальнейшем через этот объект можно вызывать доступные методы класса. По умолчанию область видимости устанавливается в значение **page**. Специфика компонентов **JavaBean** в том, что если компонент имеет поле **text**, экземпляр компонента имеет параметр **text**, а метод, устанавливающий значение, должен называться **setText(String txt)**, возвращающий значение — **getText()**.

```
package by.bsu.test;

public class Message implements Serializable {
    private Integer id;
    private String text = "";
    public Message() {}
    public Message(Integer id, String text) {
        this.id = id;
        this.text = text;
    }
    public void setText(String text) {
        this.text = text;
    }
    public String getText() {
        return text;
    }
    public Integer getId() {
        return id;
    }
    public void setId(Integer id) {
        this.id = id;
    }
}
```

```

@Override
public String toString() {
    return "Message [id=" + id + ", text=" + text + "];"
}
}

```

В теге **useBean** можно создавать только объекты классов, у которых определен конструктор по умолчанию;

- **<jsp:setProperty>** — позволяет устанавливать значения полей указанного в атрибуте **text** объекта. Если установить значение **property** в «*», то значения свойств компонента **JavaBean** будут установлены таким образом, что будет определено соответствие между именами параметров и именами методов-установщиков (**setter-ов**) компонента:

```
<jsp:setProperty name="ob" property="text" value="hello" />
```

что идентично java-коду:

```
ob.setText("hello");
```

или с автоматическим преобразованием строки в число

```
<jsp:setProperty name="ob" property="id" value="717" />
```

идентично:

```
ob.setId(Integer.parseInt("717"));
```

что возможно только для интегральных и логических (**boolean**) типов;

- **<jsp:getProperty>** — извлекает значения поля указанного объекта, преобразует его в строку и отправляет в неявный объект **out**:

```
<jsp:getProperty name="ob" property="text" />
```

- **<jsp:forward>** — позволяет передать запрос другой странице или сервлету:

```
<jsp:forward page="относительный URL"/>
```

- **<jsp:params>** — группирует параметры внутри тега **jsp:plugin**;
- **<jsp:param>** — добавляет параметры в объект запроса, например, в элементах **forward**, **include**, **plugin**;
- **<jsp:fallback>** — указывает содержимое, которое будет использоваться браузером клиента, если подключаемый модуль не сможет запуститься. Используется внутри элемента **plugin**.

В качестве примера можно привести следующий фрагмент:

Элементы `<jsp:attribute>`, `<jsp:body>`, `<jsp:invoke>`, `<jsp:doBody>`, `<jsp:element>`, `<jsp:output>` используются, в основном, при включении в страницу пользовательских тегов и библиотек тегов.

Современные тенденции оформления клиентской части стремятся к XML-формату, например, в виде **xhtml**. Такой формат менее подвержен грамматическим ошибкам разработчика, так как осуществляется проверка на *well-formed*. Предпочтительно создавать JSP-страницу в виде JSP-документа — корректного XML-документа, который ссылается на определенное пространство имен, содержит стандартные действия JSP, пользовательские теги и теги ядра JSTL, XML-эквиваленты директив JSP. JSP-документы сохраняются с расширением **.jspx**. Директива **page** для обычной JSP:

```
<%@ page contentType="text/html"%>
```

для JSP-документа:

```
<jsp:directive.page contentType="text/html" />
```

Если к директиве **page** добавить атрибут **session="false"**, то атрибуты сессии будут недоступны.

Директива **include**:

```
<%@ include file="header.jsp"%>
```

Action-тег:

```
<jsp:include file="header.jsp" />
```

Директива **taglib** для обычной JSP:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
```

для JSP-документа:

```
<jsp:root xmlns:c="http://java.sun.com/jsp/jstl/core"/>
```

Выше были представлены директива и action-тег **include**. При включении статических ресурсов в страницу оба механизма работают идентично. При включении динамических ресурсов (другой JSP или JSPF-фрагмента) директива **include** сразу конвертирует их непосредственно в создаваемый сервлет и включает в процесс исполнения, таким образом, исходная и включаемая страница могут пользоваться одним и тем же пространством имен. Применение action-тега **include** формирует вид запрашиваемой страницы уже после выполнения сгенерированного сервлета и статически включает в его объект **response**.

Таким образом, для включения динамического содержимого в JSP-страницу каждый раз, когда та получает запрос, используется директива **include**. В этом случае включаемые сегменты имеют доступ к объектам **page**, **request**, **session** и **application** исходной страницы и ко всем атрибутам, которые имеют эти объекты. Если использовать action-тег **jsp:include**, то изменения включаемого сегмента отразятся только после изменения исходной страницы (контейнер JSP перекомпилирует исходную страницу).

Для иллюстрации отличий директивы и action-тега **include** следует объявить некоторую jsp с обращением к переменной с именем **calendar**, которая на данной странице не объявлялась, а именно:

```
<%@ page contentType="text/html; charset=utf-8" pageEncoding="utf-8"%>
<html>
  <body>
    <hr/>
    Время в миллисекундах: ${calendar.timeInMillis}
    <hr/>
  </body>
</html>
```

Теперь в **index.jsp** будут использованы и директива, и action-тег.

```
<%@ page contentType="text/html; charset=utf-8" pageEncoding="utf-8"%>
<html>
  <head><title>Index</title></head>
  <body>
    <jsp:useBean          id="calendar"          scope="page"
    class="java.util.GregorianCalendar"/>
    Directive
    <%@ include file="time.jsp"%>
    <br/>
    Action-tag
    <jsp:include page="time.jsp"/>
  </body>
</html>
```

В результате имеется: если страница вставлена с помощью директивы **include**, то ее объявления получают доступ к переменным в области видимости основной страницы.

Другой способ импорта статического содержимого представляет тег **<c:import>** из библиотеки Java Standard Tag Library (JSTL).

Защита от кнопки back браузера (в фильтр):

```
httpResponse.setHeader("Cache-Control","no-cache, no-store, must-revalidate");  
response.addHeader("Cache-Control", "post-check=0, pre-check=0");  
httpResponse.setHeader("Pragma","no-cache");  
httpResponse.setDateHeader ("Expires", 0);
```

1.6.4. Expression Language

В JSP API введено понятие Expression Language (EL). EL используется для упрощения доступа к данным (атрибутам, параметрам и т. п.), хранящимся в различных областях видимости: **page**, **request**, **session**, **application** и вычисления простых выражений.



Рисунок 9 - Отличия директивы и action-тега include

EL вызывается при помощи конструкции **$\${attribute\_name}$** .

Начиная с версии спецификации JSP 2.0, EL является частью JSP и поддерживается без всяких сторонних библиотек.

EL-идентификатор ссылается на переменную, возвращаемую вызовом вида **pageContext.findAttribute(attribute_name)**. В общем случае переменная может быть сохранена в любой области видимости: **page** (**PageContext**), **request** (**HttpServletRequest**), **session** (**HttpSession**), **application** (**ServletContext**). Перечислены в порядке возрастания длительности существования. Если переменная не найдена, возвращается **null**. Значение **null** в поток вывода не передается. Также возможен доступ к параметрам запроса через предопределенные объекты **param** и **paramValues**, а также к заголовкам запроса через **requestHeaders** и некоторым другим, как уже было сказано.

Данные, как правило, состоят из объектов, соответствующих спецификации JavaBeans, или представляют собой контейнеры — такие как, **List**, **Set**, **Map**, массивы и др. EL предоставляет доступ к этим объектам при помощи операторов «.» и «[]». Способ применения этих операторов зависит от типа объекта. К элементу массива или списка доступ производится обычной индексацией с оператором «[]». К элементам типа **Map**: либо указанием ключа после оператора «точка», либо обращением к ключу также по имени, но в операторе «[]».

Обращение к экземпляру класса производится с неявным вызовом метода **toString()** в виде:

```
${ ob }
```

Обращение к значению поля экземпляра класса производится с неявным вызовом метода **getИмяПоля()** в виде:

```
${ об.имяПоля }
```

то есть на объекте будет вызван метод **getИмя()**.

```
<%@ page contentType="text/html; charset=UTF-8" %> <html>
```

```
<head><title>EL for object</title></head>
```

```
<body>
```

```
<jsp:useBean id="ob" scope="page" class="by.bsu.test.Message" />
```

```
<jsp:setProperty name="ob" property="text" value="Bender" />
```

```
${ ob.text } in Rio
```

```
</body>
```

```
</html>
```

С помощью оператора «точка» можно вызывать другие методы класса, к которому принадлежит объект. В этом случае сигнатура метода должна быть представлена полностью без каких-либо сокращений.

Если JSP вызывается из сервлета, передача объекта в качестве значения атрибута запроса осуществляется в виде:

```
Message obj = new Message(); obj.setText("Bender"); request.setAttribute("ob", obj);
```

В результате запуска этой страницы в браузер будет выведено:

Bender in Rio

Создать объект-список и получить доступ к нему, его элементам и полям элементов возможно с помощью следующего кода:

```
<%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<html>
  <head><title>List</title></head>
  <body>
    Элемент списка: ${ lst[0] } <br/>
    Значение поля элемента списка: ${ lst[0].text } <br/>
    Весь список: ${ lst }
  </body>
</html>
```

При вызове **el_list.jsp** из сервлета, что значительно более естественно, чем создание объекта в самой странице, передача объекта осуществляется с помощью экземпляра запроса в виде:

```
MessageList<Message> list = new MessageList<>(); request.setAttribute("lst", list);
```

где класс **MessageList** представляет собой реализацию списка в виде наследования или агрегирования (объявление в качестве поля класса):

```
package by.bsu.test;
import java.util.ArrayList;
public class MessageList extends ArrayList<Message> {
  {
    /* логический блок вставлен для упрощения создания коллекции
    в реальном коде его быть не должно!*/
    this.add(new Message(721, "Hello"));
    this.add(new Message(315, "Bye"));
  }
  public MessageList() {
```



```

    }
    @Override
    public String toString() {
        String str = "";
        for (Message msg : this) {
            str += msg;
        }
        return str;
    }
}

```

В браузер будет выведено:



Рисунок 10 - Использование EL для доступа к элементам списка

Создать объект-карту типа **Map** и получить доступ к нему, его элементам и полям элементов возможно с помощью следующего кода:

```
<%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<html>
  <head><title>Map</title></head>
  <body>
    Доступ по прямому значению ключа: ${ map.Blinov } <br/>
    Доступ по ключу: ${ map["Romanchik"] } <br/>
    Доступ к полю ключа: ${ map["Romanchik"].id } <br/>
    Вся карта: ${ map }
  </body>
</html>
```

Если **el_map.jsp** вызывается из сервлета, добавление объекта к запросу производится в виде:

```
MessageMap<String, Message> messMap = new MessageMap<String, Message>
(); request.setAttribute("map", messMap);
```

где класс **MessageMap** представляет примитивнейшую реализацию карты отображения:

```
package by.bsu.test;
import java.util.HashMap;
public class MessageMap extends HashMap<String, Message> {
    {
        // упрощение
        put("Blinov", new Message(1, "Work"));
        put("Romanchik", new Message(865, "Holiday"));
    }
}
```



Рисунок 11 - EL для доступа к значениям карты отображения

Чтобы организовать обработку коллекции через цикл, а не посредством прямого индексирования, следует применить тег **<c:forEach>** из библиотеки JSTL, о нем будет рассказано в главе 18.

Параметры запроса, определенные на одной странице, доступны на другой странице при прямом переходе с первой страницы на вторую.

```
<%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<html>
  <head><title>Parameters</title></head>
  <body>
    <form action="el_param_browse.jsp">
      First Name: <input type="text" name="firstname"><br/>
      e-mail 1: <input type="text" name="mail"><br/>
      e-mail 2: <input type="text" name="mail"><br/>
      <input type="submit" name="submit" value="submit">
    </form>
  </body>
```

EL с помощью неявных объектов **param** и **paramValues** получает доступ к параметрам, определенным на вызывающей странице:

```
<%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<html>
    <head><title>EL for parameter</title></head>
    <body>
        The firstname is ${ param.firstname }<br/>
        e-mail 1 is ${ param.mail }<br/>
        e-mail 1 is ${ paramValues.mail[0] }<br/>
        e-mail 2 is ${ paramValues.mail[1] }
    </body>
</html>
```

Операторы в EL поддерживают наиболее часто используемые манипуляции данными.

1.6.5. EL операторы

Стандартные операторы отношения: == (или **eq**), != (или **ne**), < (или **lt**), > (или **gt**), <= (или **le**), >= (или **ge**).

Арифметические операторы: +, -, *, / (или **div**), % (или **mod**).

Логические операторы: && (или **and**), || (или **or**), ! (или **not**).

Применение некоторых из операторов приведено в следующем коде, где для наглядности после знака равно приведен потенциальный результат, ожидаемый в окне браузера:

```
<%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<html>
    <head><title>EL operation</title></head>
    <body>
        ${ 1 > (7/3) } = false
        ${ 7.0 >= 5 } = true
        ${ 100.0 == 100.000000000000000001 } = true
        ${ 'Z' < 'a' } = true
        ${ 'dog' gt 'doc' } =
        ${ 7.0E3 + 1.4 } = 7001.
```

$\{ 17 \bmod 7 \} = 3$

</body>

</html>

Оператор **empty** используется для проверки значения переменной на **null**, или «пустое значение». Термин «пустое значение» зависит от типа проверяемого объекта. Это понятие включает нулевую длину для строки или нулевой размер для коллекции или массива.

```
<%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<html>
```

```
<head><title>operator empty</title></head>
```

```
<body>
```

```
<jsp:useBean id="ob" scope="page" class="by.bsu.test.Message" />
```

```
Message is not valid: ${not empty ob and empty ob.text }
```

```
</body>
```

```
</html>
```

В браузер будет выведено: **Message is not valid: true**

Если в JSP отсутствует объявление или доступ к объекту **ob** типа **Message**, то при попытке обращения **\${ob}** в результате ничего выведено не будет. Поток вывода игнорирует значения типа **null**. Более того, при обращении к полю т.е get-методу несуществующего объекта вида **\${ob.text}** генерации исключения не произойдет. Если же объект существует в какой-либо области видимости, а обращение происходит к несуществующему полю, то будет сгенерировано исключение.

1.6.6. Обработка ошибок

При выполнении веб-приложений, как и любых других, могут возникать ошибки и исключительные ситуации. Возникает вопрос об их обработке. Если исключение в java-коде не обрабатывается, то оно попадает в контейнер сервлетов и получает код 500 с генерацией сообщения вида «500 Internal Server Error». Возникает также при вызове сервлетом метода **sendError(500)** на объекте **HttpServletResponse**. Такие действия производятся, как правило, для исключений времени выполнения. Исключения, генерируемые веб-приложением и не перехватываемые фильтром, сервлетом или JSP. При отсутствии запрашиваемой страницы генерируется ошибка с кодом 404. При запрете доступа — код 403. При отсутствии ответа сервера в течение определенного времени — код 504. При передаче слишком длинной строки запроса — код 414. Для обработки исключений в зависимости от типа в приложении могут использоваться обычные JSP-страницы, специальные JSP для обработки ошибок или HTML-страницы. Для настройки соответствия ошибок и обработчиков используется элемент **error-page** файла **web.xml**. Например, для обработки ошибки по коду:

```
<error-page>
  <error-code>404</error-code>
  <location>/jsp/errors/error404.jsp</location>
</error-page>
```

или для обработки страницей ошибок конкретного типа необработанного ранее исключения

```
<error-page>
  <exception-type>java.lang.RuntimeException</exception-type>
  <location>/jsp/errors/error_runtime.jsp</location>
</error-page>
```

В элементе **error-code** указывается код ошибки, в элементе **exception-type** — тип исключения.

Чтобы страница распознавалась приложением как «error page», следует установить значение атрибута **isErrorPage** в **true**:

```
<%@ page isErrorPage="true" %>
```

Ниже приведен пример с генерацией кода ошибки 500. При запуске страницы в коде соответствующего ей сервлета генерируется исключение **javax.el.PropertyNotFoundException**, и управление передается странице **error_runtime.jsp** в соответствии с тегом **<error-page>** в **web.xml**.

```

<%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<html>
  <head><title>Генерация исключения</title></head>
  <body>
    <jsp:useBean id="ob" class="java.lang.String"/>
    ${ ob.toString }
  </body>
</html>

```

Генерация исключения произойдет из-за обращения к несуществующему свойству экземпляра класса **String**.

Страница, вызываемая при ошибках, может иметь статический вид, но при необходимости сообщает о типе и месте возникшего исключения в понятной для клиента приложения форме. Информация о коде и типе ошибки, а также о месте ее возникновения, извлекается из неявного объекта **pageContext**, точнее, из его поля **errorData**, являющегося экземпляром класса **javax.servlet.jsp.ErrorData**.

```

<%@ page isErrorPage="true" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"%>
<html><title>Error Page</title>
  <body>
    Request from ${pageContext.errorData.requestURI} is failed <br/>
    Servlet name: ${pageContext.errorData.servletName} <br/>
    Status code: ${pageContext.errorData.statusCode} <br/>
    Exception: ${pageContext.exception} <br/>
    Message from exception: ${pageContext.exception.message}
  </body>
</html>

```

В результате запуска страницы **index.jsp** в браузер будет выведена информация об ошибке в виде:

Servlet name: jsp

Status code: 500

Exception: javax.el.PropertyNotFoundException: Property 'toString' not found on type java.lang.String

Message from exception: Property 'toString' not found on type java.lang.String

Для указания конкретной error page, обрабатывающей ошибки, которые возникают при выполнении только этой страницы, можно использовать директиву **page** с атрибутом **errorPage**:

`<%@ page errorPage="path" %>`, где **path** — это путь к странице-обработчику ошибок из обычной исполняемой страницы, например:

`<%@ page errorPage="/errors/error_runtime.jsp" %>`

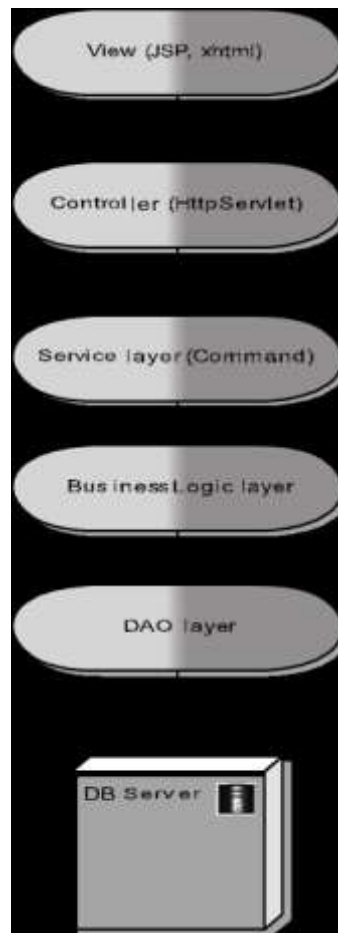


Рисунок 12 - Слои веб-приложения

1.6.7. Взаимодействие

JSP — сервлет — JSP

В большинстве приложений используются не сер-влеты или JSP по отдельности, а их архитектурное взаимодействие. Страница JSP представляет вид для результатов выполнения запроса клиента, а сервлет отвечает за вызов классов бизнес-логики и передачу результатов выполнения бизнес-логики в соответствующую JSP и ее вызов. Т.е. сервлеты не генерируют от-вета сами, а только выступают в роли контроллера за-просов. Такая архитектура построения приложений носит название MVC. Model — классы бизнес-логики и длительного хранения (БД), View — страницы JSP, Controller — сервлет.

Применение шаблона на практике приводит к тому, что трехуровневая базовая модель распадается на мно-гоуровневую. Число и назначение слоев может существенно отличаться в зависимости от разработанного архитектурного решения.

В предлагаемом ниже подходе добавляются: уро-вень служб, обрабатывающих объект запроса, уровень логики, имплементирующий конкретные бизнес-про-цессы приложения и уровень организации взаимодействия с базой данных.

Реализацию достаточно простой, но эффективной тех-нологии построения распределенного приложения можно рассмотреть на примере решения задачи проверки логина и пароля пользователя с выводом приветствия в случае положительного результата. Вход в приложение осуществляется обычно через индексную страницу,

в данном случае **index.jsp**.

```
<%@ page contentType="text/html" %>
<html>
  <body>
    <jsp:forward page="/jsp/login.jsp"/>
  </body>
</html>
```

«Индексная» страница является формальной и выполняет **forward** запроса на первую активную страницу приложения. Следующая страница **login.jsp** со-держит форму для ввода логина и пароля для аутентификации в системе:

```
<%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<html>
  <head>
    <title>Login</title>
  </head>
  <body>
```

```

<form method="POST" action="controller">
    <input type="hidden" name="command" value="login" />
    Login:<br/>
    <input type="text" name="login" value="" />
    <br/>Password:<br/>
    <input type="password" name="password" value="" />
    <br/>
    ${errorLoginPassMessage} <br/>
    ${wrongAction} <br/>
    ${nullPage} <br/>
    <input type="submit" value="Login" />
</form>
<hr/>
</body>
</html>

```

Страница содержит скрытое поле **hidden**, которое должно содержать указание на действие, связанное с функционалом этой страницы. В данном случае — запуск процесса аутентификации пользователя. В атрибут **errorLoginPassMessage** будет передаваться сообщение после неудачной попытки ввода логина-пароля. В атрибут **wrongAction** сообщение о несуществующей команде.

Демонстрационное приложение содержит страницу **main.jsp**, которая отображается пользователю в случае успешной аутентификации в приложении:



Рисунок 13 - Страница аутентификации

```

<%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<html>
    <head>

```

```

        <title>Welcome</title>
    </head>
    <body>
        <h3>Welcome</h3>
        <hr/>
        ${user}, hello
        <hr/>
        <a href="controller?command=logout">Logout</a>
    </body>
</html>

```

Страница **main.jsp** также объявляет команду, для передачи параметра с командой здесь использована строка с запросом типа **GET**, но это лишь демонстрация возможностей. В реальном примере следует передавать команду в скрытом поле и только по методу **POST**. Метод **doGet()** в сервлете оставить пустым.

Если пароль введен неправильно, то атрибут **errorLoginPassMessage** получит соответствующее значение и будет выведено:

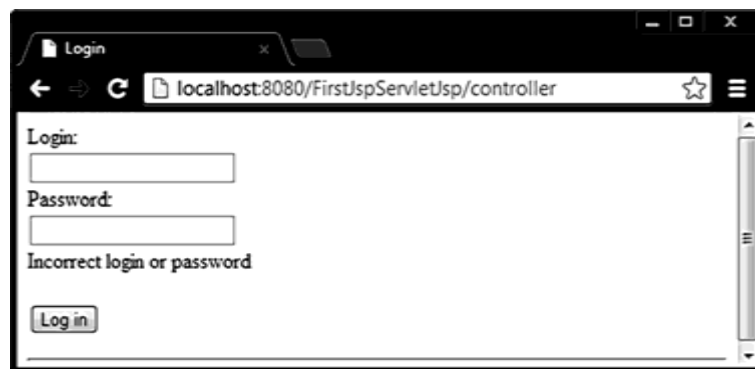


Рисунок 14 - Вывод сообщения при ошибке аутентификации

Страница **error.jsp**, последняя из созданных, загружается пользователю в случае возникновения ошибок:

```

<%@ page isErrorPage="true" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"%>
<html>
    <head>
        <title>Error Page</title>
    </head>

```

```

<body>
    Request from ${pageContext.errorData.requestURI} is failed <br/>
    Servlet name or type: ${pageContext.errorData.servletName} <br/>
    Status code: ${pageContext.errorData.statusCode} <br/>
    Exception: ${pageContext.errorData.throwable}
</body>
</html>

```

В эту страницу следует добавить возможность перехода на какую-либо область приложения. Такие страницы должны быть в любом приложении, в данном случае она также может содержать команду, позволяющую пользователю продолжить взаимодействие с системой.

Код сервлета-контроллера **Controller**, который не будет изменяться при добавлении новых страниц.

```

package by.bsu.example.servlet;

@WebServlet("/controller")

public class Controller extends HttpServlet {

    protected void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
        processRequest(request, response)
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
        processRequest(request, response)
    }

    private void processRequest(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
        String page = null;
        // определение команды, пришедшей из JSP
        Command command = ActionFactory.defineCommand(request
/*
        * вызов реализованного метода execute() и передача параметров
        * классу-обработчику конкретной команды
        */

```

```

page = command.execute(request);
// метод возвращает страницу ответа
// page = null
// поэкспериментировать
if (page != null) {
    RequestDispatcher dispatcher =
getServletContext().getRequestDispatcher(page);
    // вызов страницы ответа на запрос
    dispatcher.forward(request, response);
} else {
    // установка страницы с сообщением об ошибке
    page = ConfigurationManager.getProperty("path.page.index");
    request.getSession().setAttribute("nullPage",
MessageManager.getProperty("message.nullpage"));
    response.sendRedirect(request.getContextPath() + page);
}
}
}

```

Для определения типа и создания экземпляра-команды используется элемент реализации шаблона **Factory Method** класс **ActionFactory**.

На самом деле экземпляр **request** не должен свободно передаваться в бизнес-логику приложения во избежание некорректной модификации, поэтому следует предварительно извлекать необходимую информацию из экземпляра-запроса и сохранять ее в объекте класса, специально для этого предназначенном, а именно:

```

public class SessionRequestContent {
    private HashMap<String, Object> requestAttributes;
    private HashMap<String, String[]> requestParameters;
    private HashMap<String, Object> sessionAttributes;
    private boolean invalidateSession
    public SessionRequestContent(HttpServletRequest request) {
        // извлечение информации из запроса & session
    }
}

```

```

public void restore(HttpServletRequest request) {
    // добавление в запрос & session данных для передачи в jsp
}
}

```

По окончании выполнения команд бизнес-логики, полученные данные следует передать в **request** для передачи клиенту. В такой ситуации метод **defineCommand()** в качестве параметра должен будет объявлять класс **SessionRequestContent**, а не **HttpServletRequest**.

При большом числе клиентов приложения решение определять все команды в перечислении не повлечет за собой создание большого количества объектов-команд, так как все клиенты будут пользоваться общими экземплярами команд.

```

package by.bsu.example.command.client;
import by.bsu.example.command.Command;
import by.bsu.example.command.LoginCommand;
import by.bsu.example.command.LogoutCommand;
public enum CommandType {
    LOGIN {
        {
            this.command = new LoginCommand();
        }
    },
    LOGOUT {
        {
            this.command = new LogoutCommand();
        }
    };
    Command command;
    public Command getCommand() {
        return command;
    }
}

```

```

package by.epam.web.command;

import java.util.Arrays;
import java.util.Optional;

public class CommandProvider {
    private CommandProvider() {
    }

    public static Optional<Command> defineCommand(String commandName) {
        return Arrays.stream(CommandType.values())
            .filter(command
                                ->
            command.getName().equalsIgnoreCase(commandName))
            .findFirst()
            .map(CommandType::getCommand);
    }
}

```

Выполнение всех команд будет осуществляться при помощи простой реализации шаблона Command. Интерфейс Command определяет одно действие execute(), реализации интерфейса определяют в имплементированном методе бизнес-логику выполнения соответствующей команды.

Ниже приведен код класса бизнес-логики **UserService**, который должен выполнять проверку правильности введенных логина и пароля с помощью запроса в БД, но для демонстрации простой логики работы приложения используется «заглушка» — логин и пароль хранятся в константах класса, чего никогда не следует делать в реальном проекте:

```

package by.bsu.example.command;

import javax.servlet.http.HttpServletRequest;
import by.bsu.example.logic.LoginLogic;

import by.bsu.example.resource.ConfigurationManager;
import by.bsu.example.resource.MessageManager;
public class LoginCommand implements Command {

    private static final String PARAM_NAME_LOGIN = "login

```

```

private static final String PARAM_NAME_PASSWORD = "password";
private AbstractService service = new UserService();
@Override
public String execute(HttpServletRequest request) {
    String page = null;
    // извлечение из запроса логина и пароля
    String login = request.getParameter(PARAM_NAME_LOGIN);
    String pass = request.getParameter(PARAM_NAME_PASSWORD);
    if (service.checkLogin(login, pass))
        request.setAttribute("user", login);
    // определение пути к main.jsp
    page = ConfigurationManager.getProperty("path.page.main");
    } else {
        request.setAttribute("errorLoginPassMessage",
            MessageManager.getProperty("message.loginerror"));
        page = ConfigurationManager.getProperty("path.page.login");
    }
    return page;
}
}

```

```

package by.bsu.example.logic;

public class UserService {
    private final static String ADMIN_LOGIN = "admin";
    private final static String ADMIN_PASS = "Qwe123";
    public boolean checkLogin(String enterLogin, String enterPass) {
        // валидация параметров логина и пароля
        // шифрование пароля
        return ADMIN_LOGIN.equals(enterLogin)
            && ADMIN_PASS.equals(enterPass);
    }
}

```



```
}
```

Команда **EmptyCommand** будет выполнена, если к сервлету произошло обращение без команды.

```
package by.bsu.example.command;
import javax.servlet.http.HttpServletRequest;
import by.bsu.example.resource.ConfigurationManager;
public class EmptyCommand implements Command { // DefaultCommand
    @Override
    public String execute(HttpServletRequest request) {
        /* в случае ошибки или прямого обращения к контроллеру
        * переадресация на страницу ввода логина */
        String page = ConfigurationManager.getProperty("path.page.login");
        return page;
    }
}
```

```
package by.bsu.example.commands;
import javax.servlet.http.HttpServletRequest;
import by.bsu.example.resource.ConfigurationManager;
public class LogoutCommand implements Command {
    @Override
    public String execute(HttpServletRequest request) {
        String page = ConfigurationManager.getProperty("path.page.index");
        // уничтожение сессии
        request.getSession().invalidate();
        return page;
    }
}
```

Служебные классы, извлекающие из properties-файлов необходимую для функционирования приложения информацию:

```
package by.bsu.example.resource;
import java.util.ResourceBundle;
```

```

public class MessageManager {
    private      final      static      ResourceBundle      resourceBundle
=ResourceBundle.getBundle("resources.messages");
    // класс извлекает информацию из файла messages.properties
    private MessageManager() { }
    public static String getProperty(String key) {
        return resourceBundle.getString(key);
    }

package by.bsu.example.resource;
import java.util.ResourceBundle;
public class ConfigurationManager {
    private      final      static      ResourceBundle      resourceBundle      =
ResourceBundle.getBundle("resources.config");
    // класс извлекает информацию из файла config.properties
    private ConfigurationManager() { }
    public static String getProperty(String key) {
        return resourceBundle.getString(key);
    }
}

```

Далее приведено содержимое файла *config.properties*:

```

#####
## Application configuration ##
#####
path.page.index=/index.jsp
path.page.login=/jsp/login.jsp
path.page.main=/jsp/main.jsp
path.page.error=/jsp/error/error.jsp
#####

```

В общем случае генерация исключения в приложении и обращение по несуществующему адресу относятся к разл ичным типам, поэтому следует созда-вать отдельные error page. Данное приложение очень простое, поэтому вторая страница ошибок явно лишняя.

Запуск приложения может производиться из командной строки браузера вида:

`http://localhost:8080/FirstJspServletJsp/index.jsp`

или просто

`http://localhost:8080/FirstJspServletJsp/`

Во втором случае также будет произведено обращение к **index.jsp**, так как этот адрес указан в элементе **<welcome-file-list>** де-скриптора приложения.

При большом количестве клиентов, работающих с приложением, к его базе данных выполняется большое количество запросов. Установление соединения с базой данных является дорогостоящей по требуемым ресурсам операцией. Эффективный способ решения данной проблемы — организация пула (pool) используемых соединений, которые не закрываются физически, а хранятся в очереди и предоставляются повторно для других запросов.

Пул соединений — это одна из стратегий предоставления соединений приложению (не единственная, да и самих стратегий организации пула существует несколько десятков).

Далее необходимо сконфигурировать параметры пула соединений для Apache Tomcat в файле **context.xml**.

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<Context antiJARLocking="true" path="/FirstJspServletJsp">
```

```
  <Resource
```

```
    name="jdbc/testphones"
```

```
    author="Container"
```

```
    type="javax.sql.DataSource"
```

```
    maxActive="32"
```

```
    maxIdle="8"
```

```
    maxWait="10000"
```

```
    username="root"
```

```
    password="pass"
```

```
    driverClassName="com.mysql.cj.jdbc.Driver"
```

```
    url="jdbc:mysql://localhost:3306/testphones"
```

```
/>  
</Context>
```

где:

- **name** — имя ресурса, совпадающее по значению с именем, объявленным в **<res-ref-name>** дескриптора **web.xml**;
указание на тип интерфейса **type** также должно совпадать со значением в **<res-type>**;
- **username** — имя владельца базы данных;
- **password** — пароль владельца;
- **driverClassName** — класс драйвера JDBC;
- **url** — адрес физического расположения БД;
- **maxActive** — размер пула, то есть максимальное число активных соединений, при превышении — ожидание освободившегося;
- **maxIdle** — максимальное число свободных объектов-соединений;
- **maxWait** — время (миллисекунды) ожидания предоставления объекта-соединения из пула соединений при невозможности немедленной выдачи соединения, например, если весь пул выбран. По истечении этого времени при не-предоставлении соединения генерируется исключение.

Непосредственно в коде приложения найти и запустить данный пул соединений можно с помощью JNDI (Java Naming Directory Interface).

Класс **javax.naming.InitialContext** как часть JNDI API обеспечивает взаимодействие с каталогом именованных объектов, а именно: хранение в некоторой области необходимых классов объектов и предоставление к ним именованного доступа по запросу приложения (не только к БД, но и вообще к любому доступному через JNDI). В каталоге ресурсов можно сопоставить объект источника данных **javax.sql.DataSource** с некоторым именем, предварительно создав объект **DataSource**.

Разделяемый доступ к источнику данных можно организовать, например, путем объявления статической переменной типа **DataSource** из пакета **javax.sql**, однако в JEE принято использовать для этих целей каталог или имя контекста приложения. Источник данных типа **DataSource** — это компонент, предоставляющий соединение с приложением СУБД.

Доступ к внешнему ресурсу предоставляется с помощью метода **lookup()** по его, ресурса, имени. Методу **lookup()** передается имя, всегда начинающееся с имени корневого контекста.

```
Context envCtx = (Context) (new InitialContext().lookup("java:comp/env"));
```

```
DataSource ds = (DataSource) envCtx.lookup("jdbc/testphones");
```

Соединение извлекается из пула очень просто:

```
Connection cn = ds.getConnection();
```

После выполнения запроса соединение завершается, и его объект должен быть возвращен обратно в пул вызовом:

```
cn.close();
```

Производители СУБД для облегчения создания пула соединений определяют собственный класс на основе интерфейса **DataSource**. Подобные классы есть и у MySQL и у Oracle. В этом случае пул соединений может быть создан с помощью класса **oracle.jdbc.pool.OracleDataSource**:

```
OracleDataSource ods = new OracleDataSource();
ods.setURL("jdbc:oracle:thin:@server:1521/testphones ");
ods.setUser("root");
ods.setPassword("pass");
ods.setConnectionCacheName(CACHE_NAME);
Properties cacheProps = new Properties();
cacheProps.setProperty("MinLimit", "0");
cacheProps.setProperty("MaxLimit", "4");
cacheProps.setProperty("InitialLimit", "1");
cacheProps.setProperty("ConnectionWaitTimeout", "5");
cacheProps.setProperty("ValidateConnection", "true");
ods.setConnectionProperties(cacheProps);
Connection cn = ods.getConnection();
```

1.7. Сессии, события и фильтры

1.7.1. Сеанс (сессия)

При посещении клиентом веб-ресурса и выполнении вариантов запросов контекстная информация о клиенте не хранится. В протоколе HTTP нет возможностей для сохранения информации о предыдущих действиях клиента. При этом возникают проблемы в распределенных системах с различными уровнями доступа для разных пользователей. Например, действия, которые может делать администратор системы, не может выполнять гость. В данном случае необходима проверка прав пользователя при переходе с одной страницы на другую. В иных случаях необходима информация о предыдущих запросах клиента. В JEE существует несколько способов хранения текущей информации о клиенте или о нескольких соединениях клиента с сервером.

Сеанс (сессия) между клиентом и сервером, устанавливаемая на определенное время, за которое клиент может отправить на сервер сколько угодно запросов. Сеанс устанавливается непосредственно между клиентом и веб-сервером в момент получения первого запроса к веб-приложению. Каждый клиент устанавливает с сервером свой собственный сеанс, который сохраняется до окончания работы с приложением.

Сеанс используется для обеспечения хранения данных при последовательном выполнении нескольких запросов различных веб-страниц или на обработку информации, введенной в пользовательскую форму в результате нескольких HTTP-соединений (например, клиент совершает несколько покупок в Интернет-магазине; студент отвечает на несколько тестов в системе дистанционного обучения). Как правило, при работе с сессией возникают следующие проблемы:

- поддержка распределенной сессии (синхронизация/репликация данных, уникальность идентификаторов ит. д.); — обеспечение безопасности;
- инвалидация сессии (expiration), предупреждение пользователя об уничтожении сессии и\или возможность ее продления (watchdog).

Чтобы открыть явный доступ к экземпляру сеанса/сессии пользователя веб-приложения, используются методы **getSession(boolean create)** или **getSession()** интерфейса **HttpServletRequest**. Экземпляр запроса возвращает ссылку на объект сессии. Метод не создает сессию, а только дает доступ с помощью запроса к экземпляру сессии **HttpSession**, соответствующему данному пользователю.

Если для метода **getSession(boolean create)** входной параметр равен **true**, то сервлет-контейнер проверяет наличие активного сеанса, установленного с данным клиентом. В случае успеха метод возвращает дескриптор этого сеанса. В противном случае метод устанавливает/создает новый сеанс:

```
HttpSession session = request.getSession(true);
```

после чего начинается сбор информации о клиенте.

Если метод **getSession(boolean param)** вызывать с параметром **false**, то ссылка на активный сеанс будет возвращена, если он существует, если же сеанс уничтожен или не был активирован, то метод возвратит **null**. Метод **getSession()** без параметров работает так же, как и **getSession(true)**.

Сессия содержит информацию о дате и времени создания последнего обращения к сессии, которая может быть извлечена с помощью методов **getCreationTime()** и **getLastAccessedTime()**.

Метод **String getId()** возвращает уникальный идентификатор JSESSIONID (автоматически записывается в cookie), который получает каждый сеанс при создании. Метод **isNew()** возвращает **false** для уже существующего сеанса и **true** — для нового сеанса. Может быть изменен в процессы работы приложения. Задать время (в секундах) бездействия сессии, по истечении которого экземпляр сессии будет уничтожен, можно методом **setMaxInactiveInterval(long sec)**.

Чтобы сохранить значения переменной в текущем сеансе, используется метод **setAttribute(String name, Object value)** класса **HttpSession**, прочесть — **getAttribute(String name)**, удалить — **removeAttribute(String name)**. Список имен всех переменных, сохраненных в текущем сеансе, можно получить, используя метод **Enumeration getAttributeNames()**, работающий так же, как и соответствующий метод интерфейса **HttpServletRequest**.

Принудительно завершить сеанс можно методом **invalidate()**. В результате для сеанса будут уничтожены все связи с используемыми объектами, и данные, сохраненные в старом сеансе, будут потеряны для клиента и всех приложений.

```
package by.bsu.control.listener;

import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import jakarta.servlet.http.HttpSession;
import java.io.IOException;
import jakarta.servlet.annotation.WebServlet;
@WebServlet(urlPatterns = {"/sessionservlet"})
public class SessionControlServlet extends HttpServlet {
    protected void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
        HttpSession session = request.getSession(true);
        if (session.getAttribute("role") == null) {
```

```

        session.setAttribute("role", "moderator");
        // guest
    }
    /* количество запросов, которые были сделаны к данному сервлету
    текущим в рамках текущей пользовательской сессии */
    Integer counter = (Integer) session.getAttribute("counter");
    if (counter == null) {
        session.setAttribute("counter", 1);
    } else {
        /* увеличивает счетчик обращений к текущему сервлету и
        кладет его в сессию */
        counter++;
        session.setAttribute("counter", counter);
    }
    request.setAttribute("lifecycle", "CONTROL request LIFECYCLE");
    request.getRequestDispatcher("/jsp/sessionattr.jsp").forward(request,
    response);
}
}

```

Задача страницы **sessionattr.jsp** — отразить информацию о роли пользователя и счетчике обращений.

```

<%@ page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"%>
<html><head><title>Session Attribute</title></head>
<body>
    Role: ${role}
    <br /><hr />
    Counter: ${counter}
    <br /><hr />
    MaxInactiveInterval: ${pageContext.session.maxInactiveInterval}<br/>
    ID session: ${pageContext.session.id}<br/>
    Lifecycle: ${lifecycle}<br/>

```



```
<a href="index.jsp">Back to index.jsp</a>
</body></html>
```

В результате в браузер будет выведено:

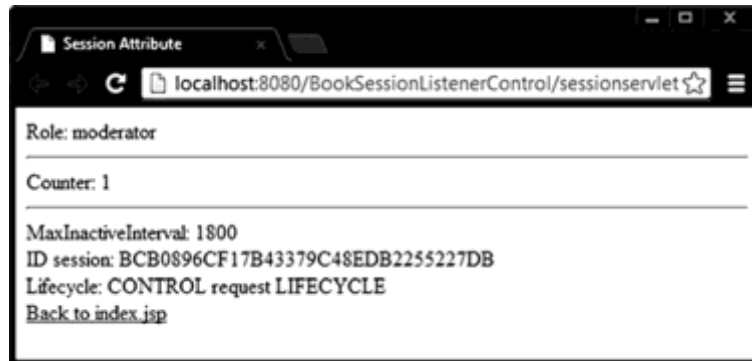


Рисунок 15 - Информация о сессии

В качестве данных сеанса выступают: счетчик обращений — объект типа **Integer** и роль пользователя. В ответ на пользовательский запрос сервлет **SessionServlet** возвращает страницу **sessionattr.jsp**, на которой отображаются все атрибуты сессии, а также идентификационный номер сессии и время инвалидации сессии. Время жизни сессии при отсутствии активности пользователя задано с помощью тега **session-config** в **web.xml** в виде:

```
<session-config>
  <session-timeout>30</session-timeout>
</session-config>
```

где время ожидания задается в минутах.

```
<%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<html>
  <head><title>Index Page</title></head>
  <body>
    Lifecycle: ${lifecycle}
    <a href="sessionervlet">Session Servlet</a>
  </body>
</html>
```

В следующем примере произведено моделирование процесса ликвидации сессии при отсутствии активности за определенный промежуток времени. Реализация корректно работает только в случае, если приложением пользуется один клиент. Для нескольких клиентов алгоритм будем немного сложнее.

```

package by.bsu.timeoutsession;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import jakarta.servlet.http.HttpSession;
import java.io.IOException;

public class TimeoutServlet extends HttpServlet {
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        HttpSession session = null;
        if (SessionLocator.flag) {
            // "создание" сессии и установка времени инвалидации
            session = req.getSession();
            int timeLive = 10; // десять секунд!
            session.setMaxInactiveInterval(timeLive);
            SessionLocator.flag = false;
        } else {
            // если сессия не существует, то ссылка на нее не будет получена
            session = req.getSession(false);
            if (session == null) {
                SessionLocator.flag = true;
            }
        }
        req.setAttribute("messages", SessionLocator.addMessage(session));
        req.getRequestDispatcher("/jsp/time.jsp").forward(req, resp);
    }
}

```

```

package by.bsu.timeoutsession;
import java.util.ArrayList;

```

```

import java.util.Date;
import jakarta.servlet.http.HttpSession;
public class SessionLocator {
    private final static String BR = "<br/><hr/>";
    public static boolean flag = true;
    public static ArrayList<String> addMessage(HttpSession session) {
        ArrayList<String> messages = new ArrayList<>();
        if (session != null) { // если сессия существует
            messages.add("Creation Time : " + new
Date(session.getCreationTime()) + BR);
            messages.add("Session id : " + session.getId() + BR);
            messages.add("Session alive!" + BR);
        } else {
            // если сессии уже не существует
            messages.add("Session disabled!" + BR);
        }
        return messages;
    }
}

```

При первом запуске в браузер будет выведено (см. рисунок 15).

Если повторить запрос к сервлету менее чем через 10 секунд, вывод будет идентично повторен. Если же запрос повторить более чем через десять секунд, сессия будет автоматически уничтожена, и в браузер будет выведено следующее сообщение:



Рисунок 16 - Жизненный цикл сессии. Сессия «not alive»

1.7.2. Файлы Cookie

Для хранения информации на компьютере клиента используются возможности класса **`jakarta.servlet.http.Cookie`**.

Cookie — это небольшие блоки текстовой информации, которые сервер посылает клиенту для сохранения в файлах `cookies`. Клиент может запретить браузеру прием файлов `cookies`. Браузер возвращает информацию обратно на сервер как часть заголовка HTTP, когда клиент выполняет переход на следующую страницу или повторно заходит на тот же веб-ресурс. Cookies могут быть ассоциированы не только с сервером, но и также с доменом; в этом случае браузер посылает их на все серверы указанного домена. Этот принцип лежит в основе одного из протоколов обеспечения единой идентификации пользователя (Single Signon), где серверы одного домена обмениваются идентификационными маркерами (token) с помощью общих `cookies`.

Cookie были созданы в компании Netscape как средства отладки, но теперь используются повсеместно. Файл `cookie` — это файл небольшого размера для хранения информации, который создается серверным приложением и размещается на компьютере пользователя. Браузеры накладывают ограничения на размер файла `cookie` и общее количество `cookie`, которые могут быть установлены на пользовательском компьютере приложениями одного веб-сервера.

Чтобы послать `cookie` клиенту, сервлет должен создать объект класса **`Cookie`**, указав конструктору имя и значение блока, и добавить их в объект `response`. Конструктор использует имя блока в качестве первого параметра, а его значение — в качестве второго.

```
Cookie cookie = new Cookie("locale", "en_US"); response.addCookie(cookie);
```

Извлечь информацию `cookie` из запроса можно с помощью метода **`getCookies()`** объекта **`HttpServletRequest`**, который возвращает массив объектов, составляющих этот файл.

```
Cookie[] cookies = request.getCookies();
```

После этого для каждого объекта класса **`Cookie`** можно вызвать метод **`getValue()`**, который возвращает строку **`String`** с содержимым блока `cookie`. В данном случае этот метод вернет значение «`en_US`».

Экземпляр **`Cookie`** имеет целый ряд параметров: путь, домен, номер версии, время жизни, комментарий. Одним из ключевых является срок жизни в секундах от момента первой отправки клиенту, определить который следует методом **`setMaxAge(long sec)`**. Если параметр не указан, то `cookie` существует только до момента прерывания контакта клиента с приложением.

В предложенном примере cookie добавляются в сервлете, в сервлете же иницируется процесс их извлечения и передачи информации, содержащейся в файле страницы **maincookie.jsp**.

```
<%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<html><head><title>from Cookie </title></head>
<body>
    ${messages}
    <a href= "cookiecontroller">Messages from Cookie</a>
</body></html>
```

При старте приложения **\${messages}** ничего не выведет, так как атрибут **messages** не существует. Сервлет **CookieController** добавляет cookie к объекту-ответу и пытается извлечь cookie из объекта-запроса. При первом запуске извлекать нечего, так как запрос пришел без cookie.

```
package by.bsu.servlet.cookie;
import java.io.IOException;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import by.bsu.servlet.action;

public class CookieController extends HttpServlet {
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
        CookieAction.setCookie(response); // добавление cookie
        // извлечение cookie и добавление информации к request
        request.setAttribute("messages", CookieAction.addToRequest(request));
        request.getRequestDispatcher("/jsp/maincookie.jsp").forward(request, response);
    }
}

package by.bsu.servlet.action;
import java.util.ArrayList;
```

```

import java.util.Date;
import jakarta.servlet.http.HttpSession;
public class CookieAction {
    private static int number = 1;
    public static void setCookie(HttpServletResponse resp) {
        String name = "JamesBond";
        String role = "00" + number++;
        Cookie c = new Cookie(name, role);
        c.setMaxAge(60 * 60 * 24 * 30); // время жизни файла cookie
        resp.addCookie(c); // добавление cookie к объекту-ответу
        value = resp.getLocale().toString();
        Cookie loc = new Cookie("locale", value);
        resp.addCookie(loc);
    }
    public static ArrayList<String> addToRequest(HttpServletRequest request) {
        ArrayList<String> messages = new ArrayList<>();
        Cookie[ ] cookies = request.getCookies();
        if (cookies != null) {
            messages.add("Number cookies : " + cookies.length);
            for (int i = 0; i < cookies.length; i++) {
                Cookie c = cookies[i];
                messages.add(c.getName() + " = " + c.getValue());
            } // end for
        } // end if return messages;
    }
}

```

При первом запуске приложения cookie, естественно, переданы не будут, так как они еще не созданы, и браузер еще не получал ни одного ответа.

При повторном обращении к сервлету cookie уже будет передан с запросом и в браузер будет выведено:

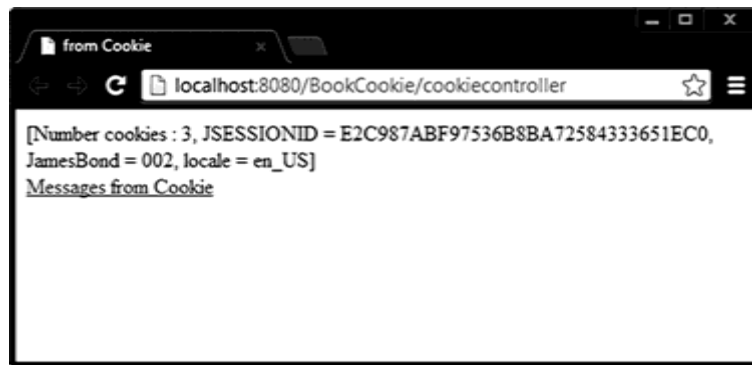


Рисунок 17 - Информация из файла cookie

Файл cookie, как это видно из указанного примера, может быть перезаписан при генерации каждого нового объекта-ответа.

1.7.3. Обработка событий

Изменение состояния некоторых объектов приложения представляют интерес и в чисто информационных целях, и в целях реализации реакции на происходящее. Например, можно отслеживать время входа и выхода из системы пользователей, действия, выполняемые пользователем. Осуществлять инициализацию ресурсов в момент запуска системы, создания пользовательской сессии, и просто служить источником дополнительной информации.

Интерфейсы и их методы	
ServletContextListener	contextInitialized(ServletContextEvent e) contextDestroyed(ServletContextEvent e)
HttpSessionListener	sessionCreated(HttpSessionEvent e) sessionDestroyed(HttpSessionEvent e)
ServletContextAttributeListener	attributeAdded(ServletContextAttributeEvent e) attributeRemoved(ServletContextAttributeEvent e) attributeReplaced(ServletContextAttributeEvent e)
HttpSessionAttributeListener	attributeAdded(HttpSessionBindingEvent e) attributeRemoved(HttpSessionBindingEvent e) attributeReplaced(HttpSessionBindingEvent e)
HttpSessionBindingListener	valueBound(HttpSessionBindingEvent e) valueUnbound(HttpSessionBindingEvent e)
HttpSessionActivationListener	sessionWillPassivate(HttpSessionEvent e) sessionDidActivate(HttpSessionEvent e)
ServletRequestListener	requestDestroyed(ServletRequestEvent e) requestInitialized(ServletRequestEvent e)
HttpSessionIdListener	void sessionIdChanged(HttpSessionEvent event, String oldSessionId)

Рисунок 18 - Интерфейсы событий и сигнатуры их методов

Существует несколько интерфейсов, которые позволяют следить за событиями, связанными с сеансом, контекстом и запросом сервлета, генерируемыми во время жизненного цикла веб-приложения:

- **jakarta.servlet.ServletContextListener** — обрабатывает события создания/удаления контекста сервлета;
- **jakarta.servlet.ServletContextAttributeListener** — обрабатывает события со-здания/удаления/модификации атрибутов контекста сервлета;
- **jakarta.servlet.http.HttpSessionListener** — обрабатывает события создания/удаления HTTP-сессии;
- **jakarta.servlet.http.HttpSessionIdListener** — обрабатывает событие изменения идентификатора HTTP-сессии;
- **jakarta.servlet.http.HttpSessionAttributeListener** — обрабатывает события создания/удаления/модификации атрибутов HTTP-сессии;
- **jakarta.servlet.http.HttpSessionBindingListener** — обрабатывает события привязывания/разъединения объекта с атрибутом HTTP-сессии;
- **jakarta.servlet.http.HttpSessionActivationListener** — обрабатывает события, связанные с активацией/деактивацией HTTP-сессии;
- **jakarta.servlet.ServletRequestListener** — обрабатывает события создания/удаления запроса;
- **jakarta.servlet.ServletRequestAttributeListener** — обрабатывает события создания/удаления/модификации атрибутов запроса сервлета.

Регистрация блока прослушивания производится в дескрипторном файле приложения или аннотированием класса, реализующего интерфейс **Listener**.

Демонстрация обработки событий изменения состояния атрибутов сессии рассматривается на примере #1 данной главы. Обрабатываться будут события добавления атрибута **role**, а также добавления и изменения атрибута для счет-чика обращений к сервлету **counter**. Для этого необходимо создать класс, реализующий интерфейс **HttpSessionAttributeListener** и реализовать необходимые для выполнения поставленной задачи методы.

```
package by.bsu.control.listener;

import jakarta.servlet.annotation.WebListener;
import jakarta.servlet.http.HttpSessionAttributeListener;
import jakarta.servlet.http.HttpSessionBindingEvent;

@WebListener
public class HttpSessionListenerImpl implements HttpSessionAttributeListener {

    public void attributeRemoved(HttpSessionBindingEvent ev) {

    }

    public void attributeAdded(HttpSessionBindingEvent ev) {
```

```

        // запись в log-файл или иные действия
        System.out.println("add: " + ev.getClass().getSimpleName() + " : " +
ev.getName() + " : " + ev.getValue());
    }
    public void attributeReplaced(HttpSessionBindingEvent ev) {
        // запись в log-файл или иные действия
        System.out.println("replace: " + ev.getClass().getSimpleName() + " : " +
ev.getName() + " : " + ev.getValue());
    }
}

```

Экземпляр класса события **HttpSessionBindingEvent** дает доступ к самой сессии методом **getSession()**.

Зарегистрировать обработку событий можно также в элементе **<listener>** дескрипторного файла **web.xml**:

```

<listener>
  <listener-class>
    by.bsu.control.listener.SessionListenerImpl
  </listener-class>
</listener>

```

В результате запуска сервлета **SessionControlServlet** и двух обращений к нему в консоль будет выведено:

```

add: HttpSessionBindingEvent : role : moderator
add: HttpSessionBindingEvent : counter : 1
replace: HttpSessionBindingEvent : counter : 1
replace: HttpSessionBindingEvent : counter : 2

```

Интерфейс **ServletRequestListener** добавлен в Servlet API, начиная с версии 2.4. С его помощью можно отследить события создания запроса при обращении к сервлету и его уничтожения.

```

package by.bsu.control.listener;
import jakarta.servlet.ServletContext;
import jakarta.servlet.ServletRequestEvent;
import jakarta.servlet.ServletRequestListener;
import jakarta.servlet.annotation.WebListener;

```

```

import jakarta.servlet.http.HttpServletRequest;
@WebListener
public class ServletRequestListenerImpl implements ServletRequestListener {
    public void requestInitialized(ServletRequestEvent ev) {
        /* будет использован для получения информации о запросе
        HttpServletRequest req = (HttpServletRequest) ev.getServletRequest();
        String uri = "Request Initialized for " + req.getRequestURI();
        */
        String id = "Request Initialized with ID=" + req.getRequestedSessionId();
        System.out.println(uri + "\n" + id);
        ServletContext context = ev.getServletContext(); // счетчик числа
        созданных запросов
        Integer reqCount = (Integer) req.getSession().getAttribute("counter");
        if (reqCount == null) {
            reqCount = 0;
        }
        context.log(uri + "\n" + id + ", Request Counter =" + reqCount);
    }
    public void requestDestroyed(ServletRequestEvent ev) {
        System.out.println("Request                Destroyed:                "+
        ev.getServletRequest().getAttribute("lifecycle"));
    }
}

```

С помощью экземпляра **ServletRequestEvent**, как видно из примера, можно получить доступ к экземпляру запроса **HttpServletRequest**, а через него и к сессии пользователя, и к контексту приложения.

После вызова страницы **index.jsp** будет создан новый **request** с нулевым значением атрибута **lifecycle**:

Request Initialized for /BookSessionListenerControl/index.jsp

Request Initialized with ID=944B8FBDB21C7DCDF9D37745C90C4EC3
Request Destroyed: null

При переходе к сервлету атрибут будет добавлен:

Request Initialized for /BookSessionListenerControl/sessionservlet

Request Initialized with ID=944B8FBDB21C7DCDF9D37745C90C4EC3

Request Destroyed: CONTROLrequest LIFECYCLE

После обратного перехода к **index.jsp** объект **request**, а вместе с ним атрибут **lifecycle**, будет уничтожен:

Request Initialized for /BookSessionListenerControl/index.jsp

Request Initialized with ID=944B8FBDB21C7DCDF9D37745C90C4EC3

Request Destroyed: null

```
package by.bsu.control.listener;
import jakarta.servlet.ServletContext;
import jakarta.servlet.ServletContextEvent;
import jakarta.servlet.ServletContextListener;
import jakarta.servlet.annotation.WebListener;
@WebListener
public class ServletContextListenerImpl implements ServletContextListener {
    public void contextInitialized(ServletContextEvent ev) {
        ServletContext context = ev.getServletContext();
        String mailFeedback = context.getInitParameter("feedback");
        context.log("Context Initialized with parameter: " + mailFeedback);
        System.out.println("Context Initialized with parameter: " + mailFeedback);
    }
    public void contextDestroyed(ServletContextEvent ev) {
    }
}
```

Параметры инициализации представлены в **web.xml** в виде:

```
<context-param>
    <param-name>feedback</param-name>
    <param-value>blinov@gmail.com</param-value>
</context-param>
```

Так как событие инициализации контекста приложения производится только один раз за его жизненный цикл, то за все время работы приложения в log-файл только один раз будут выведены записи:

Dec 31, 2012 11:57:05 PM org.apache.catalina.core.ApplicationContext

log INFO: Context Initialized with parameter: blinov@gmail.com

1.7.4. Фильтры

Реализация интерфейса **Filter** позволяет создать объект, который перехватывает запрос, может трансформировать заголовок и содержимое запроса клиента. Фильтры не создают запрос или ответ, а только модифицируют их. Фильтр выполняет предварительную обработку запроса, прежде чем тот попадает в сервлет, с последующей (если необходимо) обработкой ответа, исходящего из сервлета. Фильтр может взаимодействовать с разными типами ресурсов, в частности, и с сервлетами, и с JSP-страницами.

Основные действия, которые может выполнить фильтр:

- перехват инициализации\вызова сервлета или jsp и определение содержания запроса прежде, чем сервлет будет инициализирован\вызван;
- блокировка\перенаправление дальнейшего прохождения пары request-response;
- изменение заголовка и данных запроса и ответа;
- взаимодействие с внешними ресурсами; — построение цепочек фильтров;
- фильтрация более одного сервлета и/или jsp.

При программировании фильтров следует обратить внимание на интерфейс-сы **Filter**, **FilterChain** и **FilterConfig** из пакета **jakarta.servlet**. Сам фильтр определяется реализацией интерфейса **Filter**. Основным методом этого интерфейса является метод **doFilter(ServletRequest req, ServletResponse res, FilterChain chain)**, которому передаются объекты запроса, ответа и цепочки фильтров. Он вызывается каждый раз, когда запрос/ответ проходит через список фильтров **FilterChain**. В данный метод помещается реализация задач, обозначенных выше.

Кроме того необходимо реализовать метод **void init(FilterConfig config)**, который принимает параметры инициализации и настраивает конфигурационный объект фильтра **FilterConfig**. Метод **void destroy()** вызывается при завершении работы фильтра, в тело которого помещаются команды освобождения используемых ресурсов.

Жизненный цикл фильтра начинается с однократного вызова метода **init()**, затем контейнер вызывает метод **doFilter()** столько раз, сколько запросов будет сделано непосредственно к данному фильтру. При отключении фильтра вызывается метод **destroy()**.

С помощью метода **doFilter(ServletRequest request, ServletResponse response, FilterChain chain)** каждый фильтр получает текущий запрос и ответ, а также список фильтров **FilterChain**, предназначенных для обработки. Если в **FilterChain** не осталось необработанных фильтров, то продолжается передача запроса/ответа. Затем фильтр вызывает **chain.doFilter()** для передачи управления следующему фильтру.

Фильтр может модифицировать ответ сервера клиенту. Одним из распространенных приемов использования фильтра является модификация кодировки запроса и/или ответа. Когда страница посылает в сервер запрос клиента, используется установленная в запросе кодировка. В настоящее время некоторыми браузерами кодировка устанавливается по умолчанию и не зависит от кодировки страницы, обращающейся к серверу, и к тому же вообще не передается серверу приложений. Для корректной интерпретации передаваемых с запросом, например, символов кириллицы, необходимо перехватить запрос и заменить его кодировку.

```
<%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<html>
  <head><title>Encoding Filter</title></head>
  <body>
    Request encoding: ${ pageContext.request.characterEncoding }
  </body>
</html>
```

Если фильтр не подключать, то кодировка запроса не будет определена, потому что браузер всегда передает значение **null** для кодировки запроса.

Реализация интерфейса **Filter** для поставленной задачи, изменяющей кодировку запроса и ответа на кодировку, заданную параметром фильтра.

```

package by.bsu.barrier.filter;
import java.io.IOException;
import jakarta.servlet.Filter;
import jakarta.servlet.FilterChain;
import jakarta.servlet.FilterConfig;
import jakarta.servlet.ServletException;
import jakarta.servlet.ServletRequest;
import jakarta.servlet.ServletResponse;
import jakarta.servlet.annotation.WebFilter;
import jakarta.servlet.annotation.WebInitParam;

@WebFilter(filterName = "Encoding", urlPatterns = { "/"* }, initParams = {
@WebInitParam(name = "encoding", value = "UTF-8", description = "Encoding
Param") })

public class EncodingFilter implements Filter {
    private String code;

    public void init(FilterConfig fConfig) throws ServletException {
        code = fConfig.getInitParameter("encoding");
    }

    public void doFilter(ServletRequest request, ServletResponse response,
    FilterChain chain) throws IOException, ServletException {
        String codeRequest = request.getCharacterEncoding();
        if (codeRequest == null || !codeRequest.equalsIgnoreCase(code)) {
            request.setCharacterEncoding(code);
        }
        chain.doFilter(request, response);
    }
}

```



```

public void destroy() {
    code = null;
}
}

```

Параметры инициализации задаются внутри объявления аннотации **@WebFilter** аннотацией **@WebInitParam**. Аналогичная конфигурация для предыдущей версии определяется в дескрипторе **web.xml**:

```

<filter>
  <filter-name>encodingfilter</filter-name>
  <filter-class>
    by.bsu.sample.filter.EncodingFilter
  </filter-class>
  <init-param>
    <param-name>encoding</param-name>
    <param-value>UTF-8</param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>encodingfilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>

```

Для использования фильтра для всех сервлетов и JSP используется шаблон **"/***", только для JSP — **"*.jsp"**, для каталога — **"/jsp/admin/*"**, для конкретной jsp — **"/index.jsp"**, для сервлета — **"/controller"**.

Фильтры применяются для обеспечения безопасности приложения, а именно, легко решают задачу запрещения несанкционированного прямого обращения к JSP. Следствием установки шаблона **urlPatterns** в значение **"/jsp/*"** будет вызов фильтра при любом прямом обращении из строки браузера. Фильтр в итоге перенаправит вызов на указанную страницу.

```

package by.bsu.barrier.filter;
import java.io.IOException;
import jakarta.servlet.Filter;
import jakarta.servlet.FilterChain;
import jakarta.servlet.FilterConfig;
import jakarta.servlet.ServletException;
import jakarta.servlet.ServletRequest;
import jakarta.servlet.ServletResponse;
import jakarta.servlet.annotation.WebFilter;
import jakarta.servlet.annotation.WebInitParam;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
@WebFilter( urlPatterns = { "/jsp/*" },
initParams = { @WebInitParam(name = "INDEX_PATH", value = "/index.jsp") })
public class PageRedirectSecurityFilter implements Filter {
    private String indexPath;

    public void init(FilterConfig fConfig) throws ServletException {
        indexPath = fConfig.getInitParameter("INDEX_PATH");
    }

    public void doFilter(ServletRequest request, ServletResponse response,
        FilterChain chain) throws IOException, ServletException {
        HttpServletRequest httpRequest = (HttpServletRequest) request;
        HttpServletResponse httpResponse = (HttpServletResponse) response;
        httpResponse.sendRedirect(httpRequest.getContextPath() + indexPath);
        chain.doFilter(request, response);
    }

    public void destroy() {
    }
}

```

Фильтр также успешно перехватывает обращения к сервлету. Например, обращение к сервлету из любого источника перехватывается следующим фильтром. Фильтр проверяет роль пользователя, извлекая его значение из сессии. Если роль не задана, то фильтр присваивает ему значение **GUEST** и перенаправляет клиента на страницу **guest.jsp**.

```
package by.bsu.barrier.filter;
import java.io.IOException;
import jakarta.servlet.Filter;
import jakarta.servlet.FilterChain;
import jakarta.servlet.FilterConfig;
import jakarta.servlet.RequestDispatcher;
import jakarta.servlet.ServletException;
import jakarta.servlet.ServletRequest;
import jakarta.servlet.ServletResponse;
import jakarta.servlet.annotation.WebFilter;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import jakarta.servlet.http.HttpSession;
import by.bsu.barrier.client.ClientType;
@WebFilter(urlPatterns = { "/controller" }, servletNames = { "MainServlet" })
public class ServletSecurityFilter implements Filter {
    public void destroy() {
    }

    public void doFilter(ServletRequest request, ServletResponse response,
        FilterChain chain) throws IOException, ServletException {
        HttpServletRequest req = (HttpServletRequest) request;
        HttpServletResponse resp = (HttpServletResponse) response;
        HttpSession session = req.getSession();
        ClientType type = (ClientType) session.getAttribute("userType");
        if (type == null || session == null) {
            type = ClientType.GUEST;
            session.setAttribute("userType", type);
        }
    }
}
```

```
        session.setAttribute("locale", "fr_CA");
        RequestDispatcher dispatcher = request.getServletContext()
            .getRequestDispatcher("/jsp/guest.jsp");
        dispatcher.forward(req, resp);
        return;
    }
    // pass the request along the filter chain
    chain.doFilter(request, response);
}
public void init(FilterConfig fConfig) throws ServletException {
}
}
```

```

package by.bs barrier.client;

public enum ClientType {
    GUEST, USER, ADMINISTRATOR
}

@WebFilter(urlPatterns = {"/*"})
public class CurrentPageFilter implements Filter {
    private static final String REFERER = "referer";
    private static final String PATH_REGEX = "/controller.+"
    public void destroy() {
    }

    public void doFilter(ServletRequest req, ServletResponse resp, FilterChain chain)
    throws ServletException, IOException {
        HttpServletRequest request = (HttpServletRequest) req;
        HttpSession session = request.getSession(true);
        String url = request.getHeader(REFERER);
        //or request.getRequestURI();
        String path = substringPathWithRegex(url);
        session.setAttribute("current_page", path);
        chain.doFilter(req, resp);
    }

    private String substringPathWithRegex(String url) {
        Pattern pattern = Pattern.compile(PATH_REGEX);
        String path = null;
        if (url != null) {
            Matcher matcher = pattern.matcher(url);
            if (matcher.find()) {
                path = matcher.group(0);
            } else {
                path = PagePath.PATH_PAGE_MAIN;
            }
        }
    }
}

```

```

    }
    return path;
}

public void init(FilterConfig config) throws ServletException {
}
}

```

Фильтры в общем случае не перехватывают запросы, перенаправленные с помощью методов **forward()** и **include()** экземпляра **RequestDispatcher**. Элемент **dispatcherTypes** позволяет указать фильтру способ получить управление: непосредственно от сервлета или клиента значением **DispatcherType.REQUEST**, в результате перенаправления с использованием метода **forward()** значением **DispatcherType.FORWARD** или метода **include()** значением **DispatcherType.INCLUDE**) и некоторых других:

```

@WebFilter(dispatcherTypes = {
    DispatcherType.REQUEST,
    DispatcherType.FORWARD,
    DispatcherType.INCLUDE
}, urlPatterns = { "/jsp/*" })

```

Конфигурировать доступ можно любым необходимым набором значений.

```

@WebFilter(dispatcherTypes = { DispatcherType.FORWARD }, urlPatterns = {
"/jsp/admin/*" } )

```

1.8. JSP Standard Tag Library

Повторное использование кода — обычная техника программирования. В большинстве случаев такой код инкапсулируется в методе, при создании JSP — в стандартом теге. Наиболее стандартные и востребованные решения были определены и собраны в библиотеку JSTL.

JSP-страницы, включающие элементы action (стандартные действия) и пользовательские теги, не могут быть технологичными без использования JSTL (JSP Standard Tag Library). Создание страниц с применением JSTL позволяет упростить разработку и отказаться от вживления java-кода в JSP. Как было показано ранее, страницы со скриптами трудно читаемы, что вызывает проблемы как у про-граммиста, так и у веб-дизайнера, не обладающего глубокими познаниями в Java.

Библиотека тегов JSTL состоит из пяти групп тегов: основные теги — **core**, теги форматирования — **formatting**, теги для работы с SQL — **sql**, теги для обработки XML — **xml**, функции-теги для обработки строк — **functions**.

JSTL предоставляет следующие возможности:

- поддержку Expression Language, что позволяет разработчику писать простые выражения внутри атрибутов тега и предоставляет «прозрачный» доступ к переменным в различных областях видимости страницы;
- организацию условных переходов и циклов, основанную на тегах, а не на скриптовом языке;
- простое формирование доступа (URL) к различным ресурсам; — интернационализацию JSP;
- взаимодействие с базами данных (*не рекомендуется*); — обработку XML;
- форматирование и разбор строк посредством библиотеки функций.

Библиотека	Число тегов	Описание
core	14	<i>Основные:</i> if/then выражения и конструкции множественного выбора; вывод; создание и удаление контекстных переменных; управление свойствами JavaBeans компонентов; перехват исключений; forEach для итерирования коллекций, массивов; создание URL и импорт их содержимого.
formatting	12	<i>Интернационализация и форматирование:</i> локализация текста и структуры сообщений; форматирование и анализ чисел, денежных единиц и дат.

Библиотека	Число тегов	Описание
sql	6	<i>Доступ к БД:</i> описание источника данных; выполнение запросов, обновление данных и транзакций; обработка результатов запроса.
xml	10	<i>XML-анализ и преобразование:</i> преобразование XML; до-ступ и преобразование XML через XPath и XSLT.
fn	16	<i>Строки:</i> обработка объектов типа String и определение размера коллекций.

Библиотеку JSTL версии 3.0.0 (или более позднюю версию) подключаются в виде

```
<dependency>
  <groupId>jakarta.servlet.jsp.jstl</groupId>
  <artifactId>jakarta.servlet.jsp.jstl-api</artifactId>
  <version>3.0.0</version>
</dependency>
```

Или

```
<dependency>
  <groupId>org.glassfish.web</groupId>
  <artifactId>jakarta.servlet.jsp.jstl</artifactId>
  <version>3.0.1</version>
</dependency>
```

Expression Language подключается

```
<dependency>
  <groupId>jakarta.el</groupId>
  <artifactId>jakarta.el-api</artifactId>
  <version>5.0.1</version>
</dependency>
```


При указании значения параметра **xmlns** элемента **root** (для JSP-страницы значение параметра директивы **taglib uri="jakarta.tags.core"**) необходимо быть внимательным, так как при неправильном указании адреса JSP-страница не сможет получить доступ к тегам JSTL. Проверить корректность значения параметра **uri** (оно же справедливо и для параметра **xmlns**) можно в файле подключаемой библиотеки (например **c.tld**). Простейшая JSP с применением тега JSTL, выводящим в браузер приветствие, будет выглядеть следующим образом:

```
<%@ page contentType="text/html; charset=UTF-8" %>
<%@ taglib prefix="c" uri="jakarta.tags.core" %>
<html>
  <body>
    <c:out value="Welcome to JSTL"/>
  </body>
</html>
```

Тег **<c:out/>** отправляет значение параметра **value** в поток **JspWriter**.

JSTL core

Библиотека **core** содержит в себе наиболее часто используемые решения.

Теги общего назначения:

<c:out /> — вычисляет и выводит значение выражения;

<c:set /> — создает и устанавливает переменную в указанную область видимости;

<c:remove /> — удаляет переменную из указанной области видимости;

<c:catch /> — перехватывает исключение.

Теги условного перехода:

<c:if /> — тело тега выполняется, если значение выражения **true**;

<c:choose /> (**<c:when />**, **<c:otherwise />**) — то же, что и **<c:if />** с поддержкой нескольких условий и действия, производимого по умолчанию.

Итераторы:

<c:forEach /> — выполняет тело тега для каждого элемента коллекции, массива, итерируемого объекта;

<c:forEachTokens /> — выполняет тело тега для каждой лексемы в строке.

Теги обработки URL:

<c:redirect/> — перенаправление на указанный адрес **URL**;

<c:import/> — добавляет на JSP содержимое указанного веб-ресурса;
<c:url/> — формирует адрес с учетом контекста приложения `request.getContextPath()`;

<c:param/> — добавляет параметр к запросу, сформированному при помощи **<c:url/>**.

Ниже приведено несколько примеров, иллюстрирующих применение основных тегов из группы **core**. Существует аналогия между тегом **<c:set>** и тегом **<jsp:useBean>**. Оба создают и помещают экземпляры в заданную область видимости. Но **<jsp:useBean>** только непосредственно создает экземпляр конкретного типа, а **<c:set>**, создав ссылку, позволяет извлекать значение, например, из параметров запроса, сессии и т. д.

```
<%@ page contentType="text/html; charset=UTF-8" %>
<%@ taglib prefix="c" uri="jakarta.tags.core" %>
<html>
  <body>
    <c:set var="user" value="guest" scope="page"/>
    <c:if test="${ not empty user and user eq 'guest' }">
      User is Guest
    </c:if>
    <br/>
  </body>
</html>
```

В браузер будет выведено:

User is Guest

Тег **<c:if>** вычисляет значение атрибута **test**. Если результатом будет истина, как в приведенном выше примере, то тело тега выполняется, если ложь — игнорируется. Где понятию «ложь» соответствует любое значение, отличное от **true**.

Сохранить результаты вычисления в атрибуте **test** можно в другом атрибуте **var** с возможностью следующего использования в контексте страницы.

```

<%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"
%>
<%@ taglib prefix="c" uri="jakarta.tags.core" %>
<html>
  <head>
    <title>Core: if</title>
  </head>
  <body>
    <c:if test="\${ 5 < 7 }" var = "firstOperation" scope = "page">
      first if : is TRUE
    </c:if>
    <br/>
    <c:out value="First Operation variable is : ${ firstOperation }"/>
    <br/>
    <c:if test="\${ firstOperation } + some bt" var = "secondOperation">
      second if : is TRUE
    </c:if>
    <br/>
    <c:out value="Second Operation variable is : ${ secondOperation }"/>
  </body>
</html>

```

В результате в браузер будет выведено:

first if : is TRUE

First Operation variable is : true

Second Operation variable is : false

Тег **<c:set>** может присваивать значение и без помощи атрибута **value**, просто объявляя его в теле тега:

```

<c:set var="user" scope="page">
  guest
</c:set>

```

Такая запись необходима в ситуации, когда размещение значения в атрибут **var** невозможно: например, при обязательном наличии в значении двойных или одинарных кавычек.

```
<c:set var="users" scope="page">
    "guest" 'client' "admin"
</c:set>
<c:out value="\${users}"/>
```

Тег **<c:remove>** удаляет переменную из области видимости и при попытке доступа к ней после выполнения тега результатом будет **null**.

```
<%@ page contentType="text/html; charset=UTF-8" %>
<%@ taglib prefix="c" uri="jakarta.tags.core" %>
<html>
<body>
    <c:set var="user" scope="page">
        "guest" 'client' "admin"
    </c:set>
    User is set= <c:out value="\${user}" />
    <c:remove var="user"/>
    <br/>
    User after remove=<c:out value="\${user}"/>
</body>
</html>
```

После срабатывания тега **<c:remove>** атрибут **user** будет удален из области видимости и при попытке его поиска будет возвращено значение **null**, в браузере не отображаемое.

1.8.1. Автоматическое приведение типов и перехват исключений

Данные не всегда имеют тот же тип, который ожидается в EL-операторе. EL использует набор правил для автоматического приведения типов. Например, если оператор ожидает параметр типа **Double**, то значение идентификатора будет приведено к типу **Double**.

```

<%@ page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8" %>
<%@ taglib prefix="c" uri="jakarta.tags.core" %>
<html>
  <body>
    <c:set var="number" value="42.0" scope="session"/>
    <c:if test="\${ number < 90.0 }">
      <c:out value ="Number \${ number }"/> is smaller than (90.0)
    </c:if>
  </body>
</html>

```

Параметр из переменной **number** передается в виде строки **String**, и преобразованное к числу значение будет корректным при использовании переменной. Причем совпадение по типу должно быть точным. Если ожидается тип **Long**, а передается строка в виде **Double**, приведение типов будет некорректным, что приведет к генерации исключения, как в предлагаемом коде

```

<c:set var="number" value="42.0" scope="session"/>
<c:if test="\${ number < 90 }">
  <c:out value ="Number \${ number }"/> is smaller than (90)
</c:if>

```

где число **42.0** не может быть приведено к ожидаемому типу **Long**. В итоге страница выполнена не будет и нормальная работа остановлена генерацией ошибки с кодом 500.

Для предотвращения такой ситуации и нормального завершения выполнения страницы следует использовать тег **<c:catch>**.

```

<%@ page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8" %>
<%@ taglib prefix="c" uri="jakarta.tags.core" %>
<html>
  <head>
    <title>Core: catch</title>
  </head>
  <body>

```

```

<c:set var="number" value="42.0" scope="session" />
<c:catch var="formatException">
    <c:if test="\${ number < 90 }">
        <c:out value="Number \${ number }" /> is smaller than (90)
    </c:if>
</c:catch>
<br/>
<c:if test="\${not empty formatException }">
    Wrong format number: <c:out value="\${ number }" /> isn't Long
    <br />
    <hr />
        Generated exception
    <hr />
    \${ formatException }
</c:if>
</body>
</html>

```

В итоге разработчик будет предупрежден о некорректных данных и сможет принять меры по недопущению некорректной информации в указанный параметр.

Тег **<c:catch>** введен для избирательной обработки одного или нескольких исключений, генерация которых не должна приводить к переходу на заданные страницы обработки ошибок. Страница, на которой произошло исключение, будет корректно исполнена до конца.

Выражение **<c:if test="\\${not empty formatException}">** позволяет определить, происходила ошибка или нет. В случае возникновения исключения переменная в атрибуте **var** тега **<c:catch>** инициализируется экземпляром **javax.el.ELException**, доступ к которому может быть получен в контексте страницы, как это и сделано в приведенном примере.



Рисунок 19 - Генерация и перехват исключения

1.8.2. Исключающие условия <c:choose>

Операции по управлению условным переходом существенно расширяются благодаря тегу **<c:choose>**. Выполнение производится только одного из всего набора действий тега после определения истинности условия. Все остальные выражения, даже истинные, игнорируются, после чего управление передается тегу, следующему после тега **</c:choose>**. Если ни одно из действий **<c:when>** не выполнено, то есть **test** не принял значение **true**, то управление переходит к единственному оператору **<c:otherwise>**, который может и отсутствовать.

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8" %>
<%@ taglib prefix="c" uri="jakarta.tags.core" %>
<html>
  <head>
    <title>Core:choose</title>
  </head>
  <body>
    <c:set var="number" value="42"/>
    <c:choose>
      <c:when test="\${ number > 60 }" >
        <c:out value="число \${ number } больше 60"/>
      </c:when>
      <c:when test="\${ number > 40 }" >
        <c:out value="число \${ number } больше 40"/>
      </c:when>
      <c:when test="\${ number > 10 }" >
```

```

        <c:out value="число ${ number } больше 10"/>
    </c:when>
    <c:otherwise>
        <c:out value="число ${ number } меньше 10"/>
    </c:otherwise>
</c:choose>

</body>
</html>

```

Если необходимо повторить функциональность if-then-else, следует использовать в теге **<c:choose>** только один **<c:when>** вместе с **<c:otherwise>**.

1.8.3. Итераторы **<c:forEach>** и **<c:forTokens>**

Возможности EL предоставляют способ доступа к элементам массивов, списков, множеств, карт отображений, свойств и других итерируемых объектов. Однако часто возникает задача вывести всю коллекцию или ее часть, которая может оказаться значительной по размеру. К тому же, в следующем запросе эта коллекция может иметь уже другой размер. Для корректной работы с такими объектами используется тег **<c:forEach>**.

Пусть к экземпляру запроса в методе **doGet()** сервлета **BaseServlet** добавлен атрибут **lst**, представляющий список:


```

MessageList messages = new MessageList();
request.setAttribute("lst", messages);
request.getRequestDispatcher("jstl_foreach.jsp").forward(request, response);
где экземпляр класса MessageList предварительно заполнен значениями:
public class MessageList extends ArrayList<Message> {
    {
        // логический блок вставлен для упрощения создания коллекции
        // в реальном коде его быть не должно!
        this.add(new Message(72, "Hello"));
        this.add(new Message(31, "Bye"));
        this.add(new Message(11, "Good Bye"));
        this.add(new Message(37, "Wow"));
        this.add(new Message(92, "Hey"));
        this.add(new Message(77, "Hi"));
        this.add(new Message(71, "Ok"));
    }
}

```

Тег **<c:forEach>** имеет атрибуты **items** для размещения коллекции и **var** для доступа к элементу коллекции на каждой итерации. Необязательный атрибут **varStatus** содержит номер итерации, доступ к которому осуществляется методом **getCount()**:

```

<%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"
%>
<%@ taglib prefix="c" uri="jakarta.tags.core" %>
<html>
<head>
    <title>Core: forEach</title>
</head>
<body>
    <table>
        <c:forEach var="elem" items="${lst}" varStatus="status">

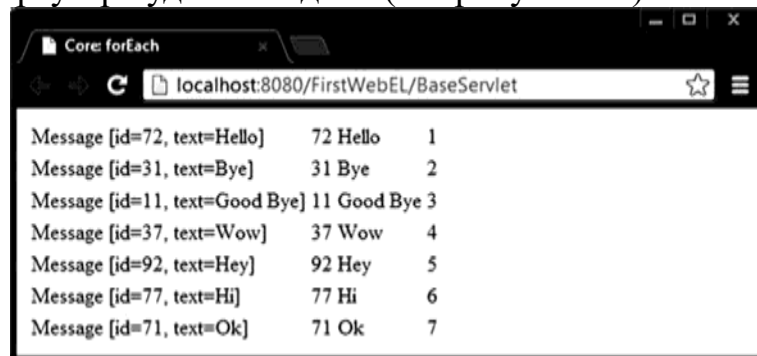
```

```

<tr>
    <td><c:out value="\${ elem }" /></td>
    <td><c:out value="\${ elem.id }" /></td>
    <td><c:out value="\${ elem.text }" /></td>
    <td><c:out value="\${ status.count }" /></td>
</tr>
</c:forEach>
</table>
</body>
</html>

```

В результате в браузер будет выведено (см. рисунок 20).



Message [id=72, text=Hello]	72 Hello	1
Message [id=31, text=Bye]	31 Bye	2
Message [id=11, text=Good Bye]	11 Good Bye	3
Message [id=37, text=Wow]	37 Wow	4
Message [id=92, text=Hey]	92 Hey	5
Message [id=77, text=Hi]	77 Hi	6
Message [id=71, text=Ok]	71 Ok	7

Рисунок 20 - Обработка списка в теге foreach

Цикл может начинаться не с первого элемента коллекции и заканчиваться не последним. Атрибуты **begin** (должен принимать значение ≥ 0) и **end** (должен быть $\geq$ **begin**) обеспечивают ограниченное действие цикла. Тег **<c:forEach>** в виде

```
<c:forEach var="elem" items="\${ lst }" varStatus="status" begin="2" end="4">
```

Изменит вывод предыдущего примера на следующий.

Также можно изменять размер шага движения по коллекции, то есть выводить, например, через один, два или более элементов. Такое использование позволяет организовать атрибут **step**.

```
<c:forEach var="elem" items="\${ lst }" varStatus="status" step="2">
```

Вывод исходного примера изменится в очередной раз.

Атрибут **varStatus** в каждом из случаев будет содержать относительную нумерацию в порядке доступа к элементам в цикле вне зависимости от их порядка в самой коллекции.

Если необходимо разбить на части объект типа **String** с использованием разделителей, применяется тег **<c:forTokens>**. Кроме атрибутов **var** и **items** с аналогичными свойствами здесь применяется атрибут **delims** для определения списка символов-разделителей, отбрасываемых при выделении подстрок. При его отсутствии строка не разбивается.

```
<%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"
%>

<%@ taglib prefix="c" uri="jakarta.tags.core" %>

<html>

<head>

<title>Core: forTokens</title>

</head>

<body>

<c:set var="str" value="45, 76 - 32 : 77 . 91" />

<c:forTokens var="token" items="{ str }" delims=".,-:)">

<c:out value="{ token }" />

<br/>

</c:forTokens>

</body>

</html>
```

Результатом выполнения будет вывод в браузер столбца чисел:

45 76 32 77 91

Регулярные выражения для определения разделителей не применяются.

1.8.4. Включение ресурсов

В реальных проектах JSP-страницы часто состоят из набора статических и динамических элементов, находящихся в разных местах контекста и вне его. Заголовок всех страниц приложения (header) и нижняя часть (footer) могут быть одинаковыми. Чтобы избежать плохой практики «сору-paste», следует выделить повторяющиеся части в отдельные страницы или фрагменты страниц и импортировать по необходимости.

Тег **<c:import>** позволяет включать содержимое ресурса, указанного в единственном обязательном атрибуте **url**. Адрес может быть внешним или внутренним, относительным или абсолютным по отношению к данному контексту.

Примеры использования:

- **<c:import url="\WEB-INF\jspf\footer.jspf" charEncoding="utf-8"/>**
- **<c:import url="\jspf\header.jspf" charEncoding="utf-8" >**
<param name="title" value="import header info"/>
<c:import/>
- **<c:import url="ftp://somestore.com/project/sample.war"/>**
- **<c:import url="\js\calendar.js" />**ег **<c:import>** получает доступ к источнику, чтение информации из которого происходит непосредственно без буферизации. Контент включается построчно в исходную JSP. По сравнению с action-тегом **<jsp:include>** и директивой **<%@include %>** тег **<c:import>** обеспечивает более совершенное включение динамических ресурсов.

Пусть существует некоторое приложение, состоящее из четырех простых страниц: **index.jsp**, **admin.jsp**, **header.jsp** и **footer.jsp**.

Стартовая страница импортирует общие страницы **header.jsp** и **footer.jsp** и содержит простой переход на страницу **admin.jsp**, которая, в свою очередь также импортирует указанные страницы.

```
<%@ page contentType="text/html; charset=utf-8" pageEncoding="utf-8" %>
<%@ taglib uri="jakarta.tags.core" prefix="c" %>
<html>
  <head><title>Index page</title></head>
  <body>
    <c:import url="jsp\admin\fragment\header.jsp" />
```

```

        <br/>
        <a href="jsp\admin\admin.jsp" >GoTo admin page</a>
        <c:import url="jsp\common\footer.jsp" />
    </body>
</html>

```

Если путь к файлу начинается из корневой директории проекта, то перед первой папкой не следует ставить символ «\» текущего контекста.

Страницы **header.jsp** и **footer.jsp** содержат выражение для извлечения из области видимости запроса значения **admin**. Это значение никто, ни в одной из приведенных страниц не добавлял. Значению в итоге будет присвоен **null**, которое поток вывода игнорирует.

```

<%@ page contentType="text/html; charset=utf-8" pageEncoding="utf-8" %>
<html>
<body>
    <hr/>
    Hello, ${requestScope.admin}
    <hr/>
</body>
</html>

<%@ page contentType="text/html; charset=utf-8" pageEncoding="utf-8" %>
<html>
<body>
    <hr/>
    Copyright, 2001-2020, ${requestScope.admin}
</body>
</html>

```

Страница **admin.jsp** содержит тег **<c:set>** для добавления атрибута **admin** в область видимости запроса. При импорте страницы получают доступ к этому значению и осуществляют его вывод.

```

<%@ page contentType="text/html; charset=utf-8" pageEncoding="utf-8" %>
<%@taglib uri="jakarta.tags.core" prefix="c" %>
<html>

```

```

<head><title>Admin page</title></head>
<body>
    <c:set var="admin" value="Blinov" scope="request"/>
    <c:import url="fragment\header.jsp" />
    <form action="\${pageContext.request.contextPath}/index.jsp">
        Content admin page
        <br/>
        <input type="submit" value="to Index">
    </form>
    <c:import url="..\common\footer.jsp" />
</body>
</html>

```

При возврате на **index.jsp** будет создан новый экземпляр **request**, поэтому на этой странице переменная **admin** существовать не будет.

Кодировка включаемой страницы по умолчанию всегда имеет значение ISO-8859-1. Если включаемая страница содержит кириллические символы, то кодировку следует указывать явно при импорте:

```

<c:import url="fragment\header.jsp" charEncoding="utf-8" />

```

1.8.5. Динамические адреса и перенаправление

Запрос можно перенаправить на другую страницу, применив тег **<c:redirect>**. Тег не является часто используемым, так как навигация между страницами при следовании принципам шаблона MVC обычно сильно ограничена.

```

<%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"
%>
<%@ taglib uri="jakarta.tags.core" prefix="c" %>
<html>
<body>
    <c:redirect url="jsp/admin/destination.jsp">
    <c:param name="firstname" value="Ostap"/>
    <c:param name="lastname" value="Bender"/>
</c:redirect>

```

```
</body>
```

```
</html>
```

При запуске данной страницы будет сразу же осуществлен переход на страницу **destination.jsp**, и пользователь приложения увидит только результаты ее работы, а именно:

```
<%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8" %>
```

```
<%@ taglib uri="jakarta.tags.core" prefix="c" %>
```

```
<html>
```

```
<head><title>Destination page</title></head>
```

```
<body>
```

```
<c:forEach var="ps" items="{param}">
```

```
<c:out value="{ps.key} : {ps.value}"/>
```

```
<br/>
```

```
</c:forEach>
```

```
</body>
```

```
</html>
```

В результате в браузер будут выведены имена переданных параметров и их значения:

```
lastname: Bender
```

```
firstname: Ostap
```

Тег **<c:param>** позволяет объявлять и передавать с запросом параметры, которые доступны на целевой странице и передаются со строкой запроса в виде:

```
http://localhost:8080/FirstProject/jsp/admin/destination.jsp?firstname=Ostap&lastname=Bender
```

Тег **<c:url>** позволяет формировать переменную с адресом, значение которого может извлекаться из контекста или формироваться динамически:

```
<%@ page contentType="text/html; charset=utf-8" pageEncoding="utf-8" %>
```

```
<%@ taglib uri="jakarta.tags.core" prefix="c" %>
```

```
<html>
```

```
<head>
```

```

<title>Index page</title>
</head>
<body>
    <c:url value="jsp/admin/destination.jsp" var="newUrl" />
    <a href='<c:out value="\${ newUrl }" />'>Go</a>
</body>
</html>

```

1.8.6. JSTL formatting

Библиотека содержит теги форматирования и интернационализации для разработки локализованных приложений. При грамотном использовании страницы вообще не будут содержать текст, а вся информация (надписи на кнопках и ссылках, заголовки, сообщения и пр.) будет извлекаться из файлов свойств.

Теги интернационализации:

<fmt:setLocale/> — устанавливает региональные установки для страницы на основе объекта класса **Locale**;

<fmt:setBundle/>, **<fmt:bundle/>** — устанавливают объект **ResourceBundle**, используемый на странице. В зависимости от установленного значения локали выбирается файл ресурсов (как правило, `properties`), соответствующий указанному языку, стране или региону;

<fmt:message/> — извлекает локализованное сообщение из ресурса, определенного тегами **<fmt:setBundle/>** или **<fmt:bundle/>**;

<fmt:requestEncoding /> — с атрибутом **value="utf-8"** устанавливает кодировку входящего запроса.

Теги форматирования дат и чисел:

<fmt:timeZone/>, **<fmt:setTimeZone/>** — устанавливает часовой пояс, используемый для форматирования;

<fmt:formatNumber/>, **<fmt:formatDate/>** — форматирует числа/даты с учетом установленной локали (региональных установок) либо указанного шаблона;

<fmt:parseNumber/>, **<fmt:parseDate/>** — переводит строковое представление числа/даты в объекты подклассов **Number** / **Date**.

Ниже приведены три примера на использование тегов из группы **fmt**.

Документ **formatdate.jsp** выводит на экран текущую дату и время с учетом установленного объекта класса **Locale**.


```

%>
<%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"
%>
<%@ taglib prefix="fmt" uri="jakarta.tags.fmt" %>
<html>
  <head><title>Data Format</title></head>
  <body>
    <jsp:useBean id="now" class="java.util.Date" />
    <fmt:setLocale value="en-EN"/>
    Вывод даты в формате English<br/>
    Сегодня: <fmt:formatDate value="{now}" /><br/>
    <fmt:setLocale value="be-BY"/>
    Вывод даты в формате Belarus<br/>
    Сегодня: <fmt:formatDate value="{now}" /><br/>
    Стил ь времени:
    (short): <fmt:formatDate value="{now}" type="time" timeStyle="short"
    /><br/>
    (medium):    <fmt:formatDate      value="{now}"      type="time"
    timeStyle="medium"/><br/>
    (long): <fmt:formatDate value="{now}" type="time" timeStyle="long"
    /><br/>
  </body>
</html>

```

В следующем примере реализован еще один способ вывода времени и даты:

```

%>
<%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"
%>
<%@ taglib uri="jakarta.tags.fmt" prefix="fmt" %>
<html>
  <head><title>timezone</title></head>
  <body>
    <jsp:useBean id="now" class="java.util.Date" />
    Вывод даты и времени с помощью тега<br/> fmt:formatDate и
    установки TimeZone

```

```

<br/>
<fmt:setLocale value="be-BY"/>
<fmt:timeZone value="GMT+3:00">
    Locale: BY
    <fmt:formatDate value="\${now}" type="both"
dateStyle="medium" timeStyle="long"/><br/>
</fmt:timeZone>
<fmt:setLocale value="pl-PL"/>
<fmt:timeZone value="GMT+1:00">
    Locale: PL :
    <fmt:formatDate value="\${now}" type="both" dateStyle="full"
timeStyle="medium"/><br/>
</fmt:timeZone>
</body>
</html>

```

Результат работы страницы смотреть на рисунке 21.



Рисунок 21 - Вывод даты и времени при задании часовых поясов

Страница **formatnumber.jsp** выводит числа в соответствии с региональными установками.

```

<%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8" %>
<%@ taglib prefix="c" uri="jakarta.tags.core" %>
<%@ taglib prefix="fmt" uri="jakarta.tags.fmt" %>
<html>
    <head><title>Format Number</title></head>

```

```

<body>
    <c:set var="currentNumber" value="118000"/>
    <c:out value="Вывод формата числа : ${currentNumber}"/>
    <br/>
    Формат      (по      умолчанию)      :      <fmt:formatNumber
value="${currentNumber}" />
    <br/>
    Процентный      формат      :      <fmt:formatNumber
value="${currentNumber}" type="percent"/>
    <br/>
    <fmt:setLocale value="be-BY"/>
    Белорусские      рубли      :      <fmt:formatNumber
value="${currentNumber}" type="currency"/>
    <br/>
    <fmt:setLocale value="pl-PL"/>
    Польская валюта:
    <fmt:formatNumber value="${currentNumber}" type="currency"/><br/>
    Французская валюта :
    <fmt:setLocale value="fr-FR"/>
    <fmt:formatNumber value="${currentNumber}" type="currency"/><br/>
</body>
</html>

```

Результат работы страницы смотреть на рисунке 21.

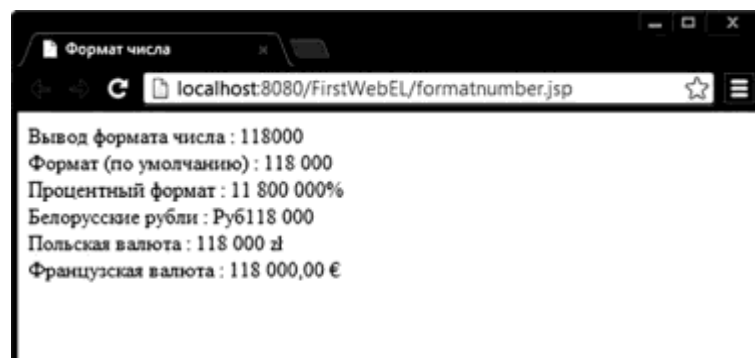


Рисунок 21 - Обычный, денежный и процентный форматы числа

Теги **<fmt:setLocale>** и **<fmt:setBundle>** позволяют задать локализацию страницы в целом и исключить из текста страниц конкретные текстовые сообщения, заменяя их ссылками на имена ключей файла ресурсов с расширением **properties**.

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8" %>

<%@ taglib prefix="fmt" uri="jakarta.tags.fmt" %>
<%@ taglib prefix="c" uri="jakarta.tags.core" %>
<fmt:setBundle basename="resource.messages" var="rb" />
<fmt:setLocale value="en_US" scope="session" /> // value="${locale}"
<fmt:setBundle basename="prop.pagecontent" var="rb" />
<html>
  <head>
    <title>
      <fmt:message key="label.title" bundle="${rb}" />
    </title>
  </head>
  <body>
    <fmt:message key="label.welcome" bundle="${rb}" />
    <hr/>
    <fmt:message key="footer.copyright" bundle="${rb}" />
  </body>
</html>
```

Результат работы страницы смотреть на рисунке 22.

Где файлы **pagecontent_en_US.properties** и **pagecontent_be_BY.properties** содержат информацию о значении ключей в соответствии с заданным для этих файлов языком и страной:



Рисунок 22 - Локализованная страница

```
# Locale en_US
```

```
label.title=Example
```

```
label.welcome=Welcome!
```

```
footer.copyright=Copyright 2001-2013, Blinov
```

а также:

```
# Locale be_BY
```

```
label.title=Пример
```

```
label.welcome=Добро пожаловать!
```

```
footer.copyright=Все права защищены 2001-2013, Блинов
```

Тег **<fmt:setBundle>** задает возможность работы с конкретным файлом ресурсов для всей области видимости страницы.

Тег **<fmt:bundle>** задает возможность работы в своей области видимости с конкретным префиксом, например **label.**, из своей области видимости. То есть информация из ключей, начинающихся с других префиксов, будет недоступна

```
<%@ page contentType="text/html; charset=utf-8" pageEncoding="utf-8" %>
```

```
<%@ taglib uri="jakarta.tags.fmt" prefix="fmt" %>
```

```
<fmt:setLocale value="be_BY" scope="session" />
```

```
<fmt:bundle basename="prop.pagecontent" prefix = "label." >
```

```
<html>
```

```
<head>
```

```
<title><fmt:message key="title" /></title>
```

```
</head>
```

```
<body>
```

```
<fmt:message key="welcome" />
```

```
</body>  
</html>  
</fmt:bundle>
```

Атрибут **prefix** можно опустить, тогда ключи нужно указывать полностью.
В результате в браузер будет выведено сообщение:

Добро пожаловать!

Можно попытаться получить доступ к информации с другим ключом:

```
<%@ page contentType="text/html; charset=utf-8" pageEncoding="utf-8" %>
<%@ taglib uri="jakarta.tags.fmt" prefix="fmt" %>
<fmt:setLocale value="ru_RU" scope="session" />
<fmt:bundle basename="prop.pagecontent" prefix = "label." >
  <html>
    <head>
      <title><fmt:message key="title" /></title>
    </head>
    <body>
      <fmt:message key="welcome" /> <br/>
      <fmt:message key="footer.copyright" />
    </body>
  </html>
</fmt:bundle>
```

В результате в браузер уже будет получено:

Добро пожаловать!

??? label. footer.copyright???

При попытке по ключу **footer.copyright** тег **<fmt:message>** пытается найти соответствующее значение с определенным для него префиксом **label** и, не найдя его в файле ресурсов, выдает сообщение **??? label. footer.copyright???**, аналогичное сообщение будет выдано , если не найден сам файл ресурсов.

1.8.7. JSTL sql

Используется для выполнения запросов SQL непосредственно из JSP и обработки результатов запроса в JSP.

<%@taglib uri="jakarta.tags.sql" prefix="sql" %> — для обычной страницы JSP.

Теги:

- **<sql:dateParam>** — определяет параметры даты для **<sql:query>** либо **<sql:update>**;
- **<sql:param>** — определяет параметры **<sql:query>** либо **<sql:update>**;
- **<sql:query>** — выполняет запрос к БД;

- **<sql:setDataSource>** — устанавливает data source (для пула соединений) для **<sql:query>**, **<sql:update>**, и **<sql:transaction>** тегов;
- **<sql:transaction>** — объединяет внутренние теги **<sql:query>** и **<sql:update>** в одну транзакцию;
- **<sql:update>** — выполняет запрос на вставку/удаление/изменение БД.

В промышленном программировании данная библиотека не используется из-за прямого доступа из JSP в СУБД, что является явным нарушением шаблона MVC.

1.8.8. JSTL xml

Используется для импорта, парсинга и обработки данных из XML-документов в документе JSPX.

Подключение библиотеки для XML формата JSP:

```
<jsp:root version="1.2" xmlns:x= "jakarta.tags.xml">
```

Список тегов:

- **<x:set>** — XML-версия тега **<c:set>**; **<x:out>** — XML-версия тега **<c:out>**;
- **<x:forEach>** — XML-версия тега **<c:forEach>**; **<x:if>** — XML-версия тега **<c:if>**;
- **<x:choose>** — XML-версия тега **<c:choose>**; **<x:when>** — XML-версия тега **<c:when>**; **<x:otherwise>** — XML-версия тега **<c:otherwise>**; **<x:parse>** — разбор XML-документа;
- **<x:transform>** — трансформация XML-документа с применением XSLT-преобразования;
- **<x:param>** — XML-версия тега **<c:param>**, определяющая параметры для другого тега **<x:transform>**.

XML-документ может быть импортирован из внешнего файла или может быть размещен непосредственно в коде JSP, что практикуется крайне редко.

Тег **<x:parse var="doc">** выполняет разбор документа, представленного в виде тела тега, и помещает построенный объект в переменную **doc**, объявленную в атрибуте **var** тега **parse**.

В XML-документе представлена информация о нескольких студентах, причем данные об одном студенте находятся в теге **<student>**. Вывод информации о всех студентах производится тегом

```
<x:forEach select="$doc/students/student" var="stud">
```

где элемент **<student>** со всем его содержимым ассоциируется с переменной **stud** после выбора из объекта **doc** с помощью простого оператора **\$doc/students/student**. Далее на каждой итерации цикла из объекта **stud** выбираются элементы **<x:out select="\$stud/name"/>** и атрибуты **<x:out select="\$stud/@login"/>**, которые выводятся в браузер тегом **<x:out>**.


```

<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" xmlns:x="jakarta.tags.xml"
version="2.0">
  <jsp:directive.page contentType="text/html; charset=UTF-8"/>
  <html>
    <head><title>XML inside</title></head>
    <body>
      <!--using tag x:parse-->
      <x:parse var="doc">
        <students>
          <student login="mit" faculty="mmf">
            <name>Mitar Alex</name>
            <telephone>2456474</telephone>
            <address>
              <country>Belarus</country>
              <city>Minsk</city>
              <street>Kalinovsky 45</street>
            </address>
          </student>
          <student login="pus" faculty="mmf">
            <name>Pashkun Alex</name>
            <telephone>3453789</telephone>
            <address>
              <country>Belarus</country>
              <city>Brest</city>
              <street>Knorina 56</street>
            </address>
          </student>
        </students>
      </x:parse>
      <!--using tags x:forEach and x:out-->

```

```

<x:forEach select="$doc/students/student" var="stud">
    Name: <x:out select="$stud/name" /><br/>
    Login: <x:out select="$stud/@login" /><br/>
    Faculty: <x:out select="$stud/@faculty" /><br/>
    Country: <x:out select="$stud/address/country" /><br/>
    City: <x:out select="$stud/address/city" /><br/>
    Street: <x:out select="$stud/address/street" /><br/>
    Telephone: <x:out select="$stud/telephone" /><br/>
    <hr/><br/>
</x:forEach>
</body>
</html>
</jsp:root>

```

Размещение XML-информации непосредственно в документе JSPX снижает гибкость приложения, поэтому разумным решением будет импортировать XML-документ, размещенный вне кода страницы.

```

<jsp:root
    xmlns:jsp="http://java.sun.com/JSP/Page"
    xmlns:x="jakarta.tags.xml"
    xmlns:c="jakarta.tags.core" version="2.0">
    <jsp:directive.page contentType="text/html; charset=UTF-8" />
    <html>
        <head>
            <title>XML outside</title>
        </head>
        <body>
            <x:parse var="doc">
                <c:import url="../xml/students.xml" />
            </x:parse>
            <!--using tag x:set-->

```

```

Student      name:      <x:set      var="studentName"
select="$doc/students/student[1]/name"/>
<x:out select="$studentName/." />

</body>

</html>

</jsp:root>

```

Результатом будет вывод в браузер строки:

Student name: Mitar Alex

Тег выбор по условию **<x:if>** работает аналогично тегу из библиотеки **core**, только использует свой синтаксис поиска соответствия в экземпляре документа.

```

<jsp:root
  xmlns:jsp="http://java.sun.com/JSP/Page"
  xmlns:x="jakarta.tags.xml"
  xmlns:c="jakarta.tags.core" version="2.0">
  <jsp:directive.page contentType="text/html; charset=UTF-8"/>
  <html>
    <head><title>XML x:if</title></head>
    <body>
      <x:parse var="doc">
        <c:import url="../xml/students.xml"/>
      </x:parse>
      <!--using x:if tag-->
      Find students from Minsk:
      <x:forEach select="$doc/students/student" var="stud">
        <x:if select="$stud/address/city = 'Minsk' ">
          <x:out select="$stud/name" /><br/>
        </x:if>
      </x:forEach>
    </body>
  </html>

```

</jsp:root>

Результатом будет вывод в браузер информации о результатах поиска:

Find students from Minsk: Mitar Alex

Тег **<x:choose>** также аналогичен тегу из базовой библиотеки:

<jsp:root

xmlns:jsp="http://java.sun.com/JSP/Page"

xmlns:x="jakarta.tags.xml"

xmlns:c="jakarta.tags.core" version="2.0">

<jsp:directive.page contentType="text/html; charset=UTF-8"/>

<html>

<head><title>XML choose</title></head>

<body>

<c:import url="../xml/students.xml" var="studXML"/>

<x:parse var="doc" doc="{studXML}"/>

<!--using tags x:choose -->

<x:forEach select="\$doc/students/student" var="stud">

<x:out select="\$stud/name" />:

<x:choose>

<x:when select="\$stud/address/city = 'Minsk'">

This student from Minsk

</x:when>

<x:when select="\$stud/address/city = 'Brest'">

This student from Brest

</x:when>

<x:otherwise>

This student not from Minsk and Brest

</x:otherwise>

</x:choose>

</x:forEach>

```

</body>
</html>
</jsp:root>

```



Рисунок 23 - Выбор информации по нескольким условиям

Результатом будет вывод в браузер (см. рисунок 23).

Преобразование XML-документа в другую форму документа выполняется тегом **<x:transform>**. Для трансформации используется XSL-таблица. В приведенном примере XML-документ преобразуется в HTML-форму с таблицей, в которую выбирается часть информации из XML.

```

<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page"
xmlns:x=jakarta.tags.xml xmlns:c="jakarta.tags.core" version="2.0">
  <jsp:directive.page contentType="text/html; charset=UTF-8"/>
  <html>
    <head>
      <title>XML transform</title>
    </head>
    <body>
      <!--using tag x:transform-->
      <c:import url="../xml/students.xml" var="studXML"/>
      <c:import url="../xml/studentsXSL.xsl" var="studXSL"/>
      <x:transform doc="{studXML}" xslt="{studXSL}"/>
    </body>
  </html>
</jsp:root>

```

```

<xsl:stylesheet
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0"

```

```

<xsl:output method="xml" indent="yes"/>
<xsl:template match="/">
    <html>
        <body>
            <table border="0" bgcolor="white">
                <tr>
                    <td>Name</td>
                    <td>City</td>
                </tr>
                <xsl:for-each select="students/student">
                    <tr>
                        <td><xsl:value-of select="name" /></td>
                        <td><xsl:value-of
                            select="address/city"/></td>
                    </tr>
                </xsl:for-each>
            </table>
        </body>
    </html>
</xsl:template>
</xsl:stylesheet>

```

Результатом будет вывод таблицы в браузер (см. рисунок 24).



Рисунок 24 - XSL трансформация XML-документа

1.8.9. JSTL functions

Теги из этой библиотеки выполняют роль, подобную смыслу библиотеки **fmt**, только не для чисел и дат, а для строк. Теги библиотеки используют префикс **fn** и во многом копируют известные методы класса **String** и не составляют особой сложности в применении.

Подключение библиотеки осуществляется директивой **taglib**:

```
<%@ taglib prefix="fn" uri="jakarta.tags.functions" %>
```

Список тегов:

- **`{fn:length(arg)}`** — подсчитывает число элементов в коллекции или длину строки;
- **`{fn:toUpperCase(String str)}`** и **`{fn:toLowerCase(String str)}`** — изменяет регистр строки;
- **`{fn:substring(String str, int from, int to)}`** — извлекает подстроку; **`{fn:substringAfter(String str, String after)}`** и **`{fn:substringBefore(String str, String before)}`** — извлекает подстроку до или после указанной во втором аргументе;
- **`{fn:trim(String str)}`** — обрезает все пробелы по краям строки; **`{fn:replace(String str, String str1, String str2)}`** — заменяет все вхождения строки **`str1`** на строку **`str2`**;
- **`{fn:split(String str, String delim)}`** — разбивает строку на подстроки, используя **`delim`**, как разделитель;
- **`{fn:join(String[] arr, String delim)}`** — соединяет массив в строку, вставляя между элементами подстроку **`delim`**;
- **`{fn:escapeXml(String xmlString)}`** — удаляет из обрабатываемой строки теги;
- **`{fn:indexOf(String str, String searchString)}`** — возвращает индекс первого вхождения строки **`searchString`**;
- **`<c:if test="{fn:startsWith(String str, String part)}">`** — возвращает истину, если строка начинается с подстроки **`part`**;
- **`<c:if test="{fn:endsWith(String str, String part)}">`** — возвращает истину, если строка заканчивается на подстроку **`part`**;
- **`<c:if test="{fn:contains(String name, String searchString)}">`** — возвращает истину, если строка содержит подстроку **`searchString`**;
- **`<c:if test="{fn:containsIgnoreCase(String name, String searchString)}">`** — возвращает истину, если строка содержит подстроку **`searchString`**, без учета регистра.

1.8.10. Jakarta RESTful Web Services (JAX-RS)

JAX-RS — это спецификация Jakarta EE, которая предоставляет стандартный API для создания веб-сервисов, следующих архитектурному стилю REST (Representational State Transfer). Она позволяет разработчикам легко создавать RESTful-сервисы с помощью набора аннотаций для маппинга POJO (Plain Old Java Objects) на веб-ресурсы.

Принципы REST. Архитектура RESTful сервисов.

REST — это не протокол, а набор архитектурных принципов (ограничений) для построения распределенных систем. Сервис считается RESTful, если он следует этим принципам:

1. **Клиент-серверная архитектура:** Четкое разделение ответственности. Клиент отвечает за пользовательский интерфейс, а сервер — за хранение и обработку данных. Это позволяет им развиваться независимо.
2. **Отсутствие состояния (Stateless):** Сервер не хранит никакого состояния клиента между запросами. Каждый запрос от клиента должен содержать всю информацию, необходимую для его выполнения. Это повышает надежность, масштабируемость и предсказуемость.
3. **Кэширование (Cacheable):** Ответы от сервера должны явно помечаться как кэшируемые или некаэшируемые. Это позволяет клиентам и промежуточным узлам кэшировать ответы, снижая нагрузку и улучшая производительность.
4. **Единообразный интерфейс (Uniform Interface):** Это ключевой принцип, который упрощает и стандартизирует взаимодействие. Он включает в себя:
 - **Идентификация ресурсов:** Каждый ресурс однозначно идентифицируется с помощью URI (например, /users/123).
 - **Манипуляция ресурсами через представления:** Клиент взаимодействует не с самим ресурсом, а с его представлением (например, в формате JSON или XML).
 - **Самодостаточные сообщения (Self-descriptive messages):** Каждый запрос и ответ содержит достаточно информации для своей обработки (например, HTTP-заголовки Content-Type и Accept).
 - **HATEOAS (Hypermedia as the Engine of Application State):** Ответы от сервера могут содержать ссылки, которые направляют клиента к возможным дальнейшим действиям.

Архитектура RESTful-сервиса строится вокруг **ресурсов**. Ресурс — это любая сущность, которой можно дать имя (например, пользователь, заказ, товар). Взаимодействие с ресурсами происходит с помощью стандартных HTTP-методов:

- **GET:** Получение ресурса.
- **POST:** Создание нового ресурса.
- **PUT:** Полное обновление существующего ресурса.
- **DELETE:** Удаление ресурса.
- **PATCH:** Частичное обновление ресурса (менее распространен).

JAX-RS API: аннотации

JAX-RS использует аннотации для связывания Java-классов и методов с URI и HTTP-методами.

- **@Path**: Указывает относительный URI для класса (корневой путь) или метода.
- **@GET**, **@POST**, **@PUT**, **@DELETE**: Связывают метод с соответствующим HTTP-глаголом.
- **@Produces**: Определяет MIME-тип данных, которые возвращает метод (например, `application/json`).
- **@Consumes**: Определяет MIME-тип данных, которые метод может принять в теле запроса.
- **@PathParam**: Извлекает значение из сегмента URI.
- **@QueryParam**: Извлекает значение из параметра запроса в URI (после `?`).
- **@HeaderParam**: Извлекает значение из HTTP-заголовка.
- **@FormParam**: Извлекает значение из данных формы (`application/x-www-form-urlencoded`).
- **@BeanParam**: Позволяет агрегировать несколько параметрных аннотаций в один Java-бин, что делает сигнатуры методов чище.

Пример простого ресурса:

```
// Book.java - наша модель данных
public class Book {
    public String id;
    public String title;
    public String author;
    // конструкторы, геттеры, сеттеры
}

// BookResource.java - наш JAX-RS ресурс
import jakarta.ws.rs.*;
import jakarta.ws.rs.core.MediaType;
import jakarta.ws.rs.core.Response;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

@Path("/books") // Корневой URI для всех методов этого класса
@Produces(MediaType.APPLICATION_JSON) // Все методы по умолчанию
будут возвращать JSON
@Consumes(MediaType.APPLICATION_JSON) // Все методы по умолчанию
будут принимать JSON
public class BookResource {
    // Простое хранилище в памяти для примера
    private static Map<String, Book> books = new ConcurrentHashMap<>();

    static {
        Book book1 = new Book();
        book1.id = "1";
        book1.title = "War and Peace";
        book1.author = "Leo Tolstoy";
    }
}
```

```

        books.put(book1.id, book1);
    }

    @GET // Обрабатывает GET-запросы к /books
    public Response getAllBooks(@QueryParam("author") String author) {
        if (author != null) {
            // Логика фильтрации по автору
            return Response.ok(books.values().stream()
                .filter(b -> b.author.equalsIgnoreCase(author))
                .toList()).build();
        }
        return Response.ok(books.values()).build();
    }

```

```

    @GET
    @Path("/{id}") // Обрабатывает GET-запросы к /books/{id}, например
    /books/1
    public Response getBookById(@PathParam("id") String id) {
        Book book = books.get(id);
        if (book != null) {
            return Response.ok(book).build();
        } else {
            return Response.status(Response.Status.NOT_FOUND)
                .build();
        }
    }
}

```

```

    @POST // Обрабатывает POST-запросы к /books
    public Response createBook(Book book, @HeaderParam("X-Request-ID")
String requestId) {
        System.out.println("Processing request with ID: " + requestId);
        book.id = String.valueOf(books.size() + 1);
        books.put(book.id, book);
        // Возвращаем 201 Created и ссылку на новый ресурс
        return Response.status(Response.Status.CREATED)
            .entity(book).build();
    }

    @PUT
    @Path("/{id}") // Обрабатывает PUT-запросы к /books/{id}
    public Response updateBook(@PathParam("id") String id, Book updatedBook)
    {
        if (!books.containsKey(id)) {
            return Response.status(Response.Status.NOT_FOUND)
                .build();
        }
        updatedBook.id = id;
        books.put(id, updatedBook);
        return Response.ok(updatedBook).build();
    }

```

```

@DELETE
@Path("/{id}") // Обрабатывает DELETE-запросы к /books/{id}
public Response deleteBook(@PathParam("id") String id) {
    Book removedBook = books.remove(id);
    if (removedBook != null) {
        return Response.noContent().build();
        // 204 No Content
    } else {
        return Response.status(Response.Status.NOT_FOUND)
            .build();
    }
}
}

```

Ресурсы, подресурсы, провайдеры

- **Ресурсы (Resources):** Это классы, аннотированные `@Path`, которые обрабатывают запросы. `BookResource` из примера выше — это ресурс.
- **Подресурсы (Sub-resources):** Это механизм для создания иерархических REST API. Метод ресурса может не обрабатывать запрос сам, а делегировать его другому объекту-ресурсу (подресурсу). Это полезно для моделирования отношений "один-ко-многим".

Пример подресурса:

```

// Допустим, у каждого автора есть свои книги
// AuthorsResource.java
@Path("/authors")
public class AuthorsResource {
    @Path("/{authorId}/books")
    public BookResource getBooksForAuthor(@PathParam("authorId") String
authorId) {
        // Здесь можно было бы передать authorId в конструктор BookResource,
        // чтобы он работал только с книгами этого автора.
        System.out.println("Delegating to BookResource for author: " + authorId);
        return new BookResource(); // Возвращаем экземпляр другого ресурса
    }
}

```

```
}  
}
```

// Теперь запрос GET /authors/tolstoy/books/1 будет обработан методом `getBookById()` в `BookResource`

- **Провайдеры (Providers):** Это специальные компоненты, которые расширяют или настраивают поведение JAX-RS. Они помечаются аннотацией `@Provider`. Основные типы:
 - `MessageBodyReader` / `MessageBodyWriter`: Для преобразования HTTP-тела в/из Java-объектов. Вы будете использовать их для нестандартных форматов данных.
 - `ExceptionHandler`: Для перехвата исключений и формирования кастомных HTTP-ответов об ошибках.
 - `ContextResolver`: Для предоставления контекстно-зависимых объектов, например, настроенного `ObjectMapper` для Jackson.

Обработка данных: JSON и XML

- **JSON-B (Jakarta JSON Binding):** Стандартный API для преобразования Java-объектов в JSON и обратно. Он работает по принципу "соглашение вместо конфигурации". Если в вашем проекте есть зависимость JSON-B, JAX-RS автоматически будет использовать его для `application/json`. Для этого POJO не требует специальных аннотаций.
- **Jackson:** Самая популярная сторонняя библиотека для работы с JSON. Она более гибкая и функциональная, чем JSON-B. Чтобы использовать Jackson, нужно добавить его зависимость и зарегистрировать его провайдер.
- **XML (JAXB - Jakarta XML Binding):** Стандартный API для работы с XML. Чтобы JAX-RS мог автоматически преобразовывать объект в XML, класс должен быть аннотирован `@XmlRootElement`.

Пример POJO для JAXB:

```
import jakarta.xml.bind.annotation.XmlRootElement;  
import jakarta.xml.bind.annotation.XmlElement;
```

```
@XmlRootElement(name = "book")  
public class Book {  
    @XmlElement  
    public String id;  
    @XmlElement  
    public String title;  
    @XmlElement  
    public String author;  
}
```

Обработка исключений

Для централизованной обработки исключений используются `ExceptionHandler`. Это провайдер, который перехватывает определенный тип исключения и преобразует его в `Response`.

Пример:

```
// Создадим свое исключение
public class BookNotFoundException
extends RuntimeException {
    public BookNotFoundException(String message) {
        super(message);
    }
}

// Создадим маппер для него
import jakarta.ws.rs.core.Response;
import jakarta.ws.rs.ext.ExceptionMapper;
import jakarta.ws.rs.ext.Provider;

@Provider // Регистрируем маппер в JAX-RS
public class BookNotFoundMapper implements
ExceptionMapper<BookNotFoundException> {
    @Override
    public Response toResponse(BookNotFoundException exception) {
        // Создаем кастомный JSON-ответ
        String errorJson = "{\"error\": \"" + exception.getMessage() + "\"}";
        return Response.status(Response.Status.NOT_FOUND)
            .entity(errorJson)
            .type(MediaType.APPLICATION_JSON)
            .build();
    }
}

// Теперь в ресурсе можно просто бросать это исключение
@GET
```

```

@Path("/{id}")
public Response getBookById(@PathParam("id") String id) {
    Book book = books.get(id);
    if (book != null) {
        return Response.ok(book).build();
    } else {
        throw new BookNotFoundException("Book with id " + id + " not found.");
    }
}

```

Валидация данных (интеграция с Bean Validation)

JAX-RS отлично интегрируется со спецификацией Jakarta Bean Validation для автоматической валидации входящих данных.

1. Добавьте аннотации валидации к полям вашего POJO.
2. В методе ресурса пометьте параметр, который нужно валидировать, аннотацией `@Valid`.

Пример:

```

import jakarta.validation.constraints.NotNull;
import jakarta.validation.constraints.Size;

public class Book {
    public String id;

    @NotNull(message = "Title cannot be null")
    @Size(min = 2, max = 100, message = "Title must be between 2 and 100 characters")
    public String title;

    @NotNull(message = "Author cannot be null")
    public String author;
}

```

```
// В ресурсе
import jakarta.validation.Valid;

@POST
public Response createBook(@Valid Book book) { // @Valid запускает
валидацию
    // ... логика создания книги
    // Если валидация не пройдет, JAX-RS бросит ConstraintViolationException,
    // который можно обработать через ExceptionMapper.
    return Response.status(Response.Status.CREATED).entity(book).build();
}
```

Безопасность REST-сервисов

Безопасность — это обширная тема. JAX-RS интегрируется со спецификацией Jakarta Security.

- **Аутентификация:** Проверка личности клиента. Популярные механизмы для REST:
 - **JWT (JSON Web Token):** Клиент получает токен от сервера аутентификации и передает его в каждом запросе в заголовке `Authorization: Bearer <token>`. Сервер проверяет подпись токена.
 - **OAuth2:** Это протокол авторизации, который часто используется для предоставления доступа к API от имени пользователя.
- **Авторизация:** Проверка прав доступа аутентифицированного пользователя. Jakarta Security предоставляет декларативные аннотации:
 - `@RolesAllowed({"ADMIN", "USER"})`: Разрешает доступ только пользователям с указанными ролями.
 - `@PermitAll`: Разрешает доступ всем.
 - `@DenyAll`: Запрещает доступ всем.

Пример:

```
import jakarta.annotation.security.RolesAllowed;

@DELETE
@Path("/{id}")
@RolesAllowed("ADMIN") // Только пользователи с ролью ADMIN могут
удалять книги
```

```

public Response deleteBook(@PathParam("id") String id) {
    // ... логика удаления ...
    return Response.noContent().build();
}

```

Клиентский API JAX-RS

JAX-RS также предоставляет стандартный API для создания клиентов, которые могут обращаться к REST-сервисам. Он имеет удобный текущий (fluent) интерфейс.

Пример использования клиентского API:

```

import jakarta.ws.rs.client.Client;
import jakarta.ws.rs.client.ClientBuilder;
import jakarta.ws.rs.client.Entity;
import jakarta.ws.rs.core.MediaType;
import jakarta.ws.rs.core.Response;

public class BookApiClient {
    public static void main(String[] args) {
        // Создаем клиент
        Client client = ClientBuilder.newClient();

        // 1. Получение книги по ID
        String bookId = "1";
        Book book = client.target("http://localhost:8080/api/books")
            .path(bookId)
            .request(MediaType.APPLICATION_JSON)
            .get(Book.class); // Выполняем GET и сразу преобразуем ответ
                             // в Book

        System.out.println("Fetched Book: " + book.title);

        // 2. Создание новой книги
    }
}

```

```

Book newBook = new Book();
newBook.title = "The Master and Margarita";
newBook.author = "Mikhail Bulgakov";
Response response = client.target("http://localhost:8080/api/books")
    .request(MediaType.APPLICATION_JSON)
    .post(Entity.json(newBook)); // Отправляем POST с телом в
    формате JSON

System.out.println("Create book status: " + response.getStatus());
if (response.getStatus() == 201) {
    Book createdBook = response.readEntity(Book.class);
    System.out.println("Created book ID: " + createdBook.id);
}

// Не забываем закрывать клиент
client.close();
}
}

```

1.9. Ядро корпоративных компонентов

1.9.1. Jakarta Contexts and Dependency Injection (CDI)

CDI — это, пожалуй, самая важная спецификация в современной Jakarta EE. Она предоставляет мощный и гибкий фреймворк для связывания компонентов приложения, управления их жизненным циклом и реализации принципов Inversion of Control.

1. Принципы инверсии управления (IoC) и внедрения зависимостей (DI)

- **Проблема:** В традиционном подходе объект сам создает свои зависимости.

```

public class OrderService {
    private final NotificationService notificationService = new
    EmailNotificationService(); // Жесткая связь!
    // ...

```

}

Это плохо:

1. **Жесткая связь:** OrderService навсегда связан с EmailNotificationService. Заменить его на SMS-сервис сложно.
2. **Сложность тестирования:** Нельзя подменить EmailNotificationService на мок-объект в юнит-тесте.

- **Решение (IoC/DI):**

- **Inversion of Control (IoC):** Принцип "Не звоните нам, мы позвоним вам". Вы не управляете созданием и поиском зависимостей, это делает сторонняя среда (контейнер).
- **Dependency Injection (DI):** Конкретная реализация IoC. Контейнер "внедряет" (предоставляет) нужные зависимости в ваш компонент

@ApplicationScoped

```
public class OrderService {  
    @Inject // Просим контейнер предоставить зависимость  
    private NotificationService notificationService;  
    // ...  
}
```

Теперь OrderService не знает, какая именно реализация NotificationService будет использована. Контейнер решает это на основе конфигурации.

2. CDI Бины и Области видимости (Scopes)

- **CDI Бин (Bean):** Любой Java-класс, который контейнер может создавать и которым может управлять. Чтобы класс стал бином, достаточно, чтобы он имел конструктор без аргументов или был помечен одной из аннотаций области видимости.
- **Область видимости (Scope):** Определяет, как долго живет экземпляр бина и сколько экземпляров будет создано.
 - **@RequestScoped:** Один экземпляр на один HTTP-запрос. Идеально для хранения данных, относящихся к одному действию пользователя.

@RequestScoped

```
public class LoginHandler { /* ... */ }
```

- **@SessionScoped:** Один экземпляр на одну HTTP-сессию пользователя. Идеально для "корзины покупок" или данных аутентифицированного пользователя.

@SessionScoped

```
public class ShoppingCart implements Serializable {  
    /* ... */  
} // Должен быть Serializable!
```

- @ApplicationScoped: **Один экземпляр на все приложение (синглтон).** Создается при первом обращении или при старте приложения и живет до его остановки. Для кэшей, конфигураций.

@ApplicationScoped

```
public class AppConfiguration { /* ... */ }
```

- @ConversationScoped: **Живет дольше одного запроса, но короче сессии.** Управляется программистом. Используется для многошаговых "мастеров" (wizards). Требуется явного старта и окончания "беседы". Редко используется.
- @Dependent: **Псевдо-область видимости.** Бин живет столько же, сколько бин, в который он внедрен. Если бин не внедрен никуда, создается новый экземпляр при каждом запросе.

3. Механизмы внедрения

CDI предоставляет три способа внедрить зависимость с помощью @Inject.

1. **Внедрение в поле (Field Injection):** Самый простой и читаемый.

@ApplicationScoped

```
public class MyService {  
    @Inject  
    private MyRepository repository;  
}
```

2. **Внедрение через конструктор (Constructor Injection): Предпочтительный способ.** Делает зависимости явными и позволяет создавать неизменяемые (immutable) объекты. Упрощает юнит-тестирование.

@ApplicationScoped

```
public class MyService {  
    private final MyRepository repository;
```

@Inject // Аннотация @Inject обязательна

```
public MyService(MyRepository repository) {
```

```
        this.repository = repository;
    }
}
```


3. **Внедрение через сеттер (Setter Injection):** Используется для опциональных зависимостей или для разрешения циклических зависимостей.

@ApplicationScoped

```
public class MyService {  
    private MyRepository repository;
```

@Inject

```
    public void setRepository(MyRepository repository) {  
        this.repository = repository;  
    }  
}
```

4. Квалификаторы (@Qualifier)

- **Проблема:** Что если у нас есть несколько реализаций одного интерфейса?

```
public interface Notifier { void send(String message); }
```

```
@ApplicationScoped public class EmailNotifier implements Notifier { /* ... */ }
```

```
@ApplicationScoped public class SmsNotifier implements Notifier { /* ... */ }
```

При попытке @Inject private Notifier notifier; CDI выдаст ошибку неоднозначности (ambiguous dependency).

- **Решение:** Квалификаторы — это аннотации, которые помогают CDI выбрать нужный бин.

1. Создаем аннотации-квалификаторы:

```
@Qualifier
```

```
@Retention(RUNTIME)
```

```
@Target({TYPE, METHOD, FIELD, PARAMETER})
```

```
public @interface Email {}
```

```
@Qualifier
```

```
@Retention(RUNTIME)
```

```
@Target({TYPE, METHOD, FIELD, PARAMETER})
```

```
public @interface SMS {}
```

2. Помечаем ими реализации:

```
@Email @ApplicationScoped public class EmailNotifier implements Notifier {  
/* ... */ }
```

```
@SMS @ApplicationScoped public class SmsNotifier implements Notifier { /* ...  
*/ }
```

3. Используем при внедрении:

```
public class BillingService {
```

```
    @Inject @Email // Внедряем конкретно EmailNotifier
```

```
    private Notifier emailNotifier;
```

```
    @Inject @SMS // Внедряем конкретно SmsNotifier
```

```
    private Notifier smsNotifier;
```

```
}
```

5. Альтернативы (@Alternative)

- **Назначение:** Позволяют подменять реализацию бина для всего приложения через конфигурацию, без изменения кода. Идеально для переключения между "боевой" и "тестовой" реализацией.

1. Определяем альтернативу:

```
// Боевая реализация
```

```
public class RealPaymentGateway implements PaymentGateway { /* ... */ }
```

// Тестовая реализация

@Alternative

public class MockPaymentGateway implements PaymentGateway { /* ... */ }

2. Активируем ее в файле src/main/webapp/WEB-INF/beans.xml:

```
<beans xmlns="https://jakarta.ee/xml/ns/jakartaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
https://jakarta.ee/xml/ns/jakartaee/beans_4_0.xsd"
  version="4.0">
  <alternatives>
    <class>com.example.MockPaymentGateway</class>
  </alternatives>
</beans>
```

Теперь везде, где запрашивается PaymentGateway, CDI будет внедрять MockPaymentGateway.

6. Продюсеры (@Produces) и Диспозеры (@Disposes)

- **Проблема:** Как сделать CDI-бином объект из сторонней библиотеки, который мы не можем аннотировать (например, java.sql.Connection), или объект, требующий сложной логики создания?
- **Решение:** Метод-продюсер (@Produces).

@ApplicationScoped

```
public class ResourceFactory {
```

```
    // Этот метод будет вызван CDI, когда кто-то попросит внедрить Logger
```

```
    @Produces
```

```
    public Logger createLogger(InjectionPoint ip) {
```

```
        // InjectionPoint позволяет узнать, куда внедряется бин
```

```
        return Logger.getLogger(ip.getMember()
```

```
            .getDeclaringClass().getName());
```

```
    }
```

```
    // Продюсер для сложного объекта
```

```
    @Produces @ApplicationScoped @Named("appConfig")
```

```
    public Properties loadProperties() throws IOException {
```

```
        Properties props = new Properties();
```

```
        props.load(getClass().getResourceAsStream("/app.properties"));
```

```
        return props;
```

```
    }
```

```
}
```

- **Диспоузеры (@Disposes):** Для каждого продюсера можно определить метод-диспоузер, который "уничтожит" созданный объект, когда его скоуп закончится.

```
public class ConnectionFactory {
    @Produces @RequestScoped
    public Connection createConnection() throws SQLException {
        // ... создает соединение
        return connection;
    }

    // Этот метод будет вызван в конце HTTP-запроса для закрытия соединения
    public void closeConnection(@Disposes Connection connection) throws
    SQLException {
        if (connection != null) {
            connection.close();
        }
    }
}
```

7. Перехватчики (@Interceptor) и Декораторы (@Decorator)

- **Назначение:** Реализация сквозной функциональности (Cross-Cutting Concerns) в стиле АОП.
- **Перехватчики (@Interceptor):** Для технических задач (логирование, аудит, метрики, кэширование).

1. Создаем аннотацию-привязку:

```
@InterceptorBinding
@Retention(RUNTIME) @Target({METHOD, TYPE})
public @interface Logged {}
```

2. Создаем перехватчик:

@Logged // Привязываем к аннотации

@Interceptor

public class LoggingInterceptor {

 @AroundInvoke // Главный метод перехватчика

 public Object logMethodCall(InvocationContext ctx) throws Exception {

 System.out.println("Entering method: " + ctx.getMethod().getName());

 Object result = ctx.proceed(); // Вызываем оригинальный метод

 System.out.println("Exiting method: " + ctx.getMethod().getName());

 return result;

 }

}

3. Активируем в beans.xml:

```
<beans ...>
  <interceptors>
    <class>com.example.LoggingInterceptor</class>
  </interceptors>
</beans>
```

4. Используем на бизнес-методе:

```
public class MyBusinessService {
  @Logged
  public void doSomething() { /* ... */ }
}
```

- **Декораторы (@Decorator):** Для изменения или дополнения **бизнес-логики**. В отличие от перехватчиков, декоратор реализует тот же интерфейс, что и декорируемый бин.

5. Создаем декоратор:

```
// Декорируемый интерфейс и его реализация
public interface Calculator { int add(int a, int b); }
public class SimpleCalculator implements Calculator { /* ... */ }

// Декоратор
@Decorator
public abstract class AuditingCalculator implements Calculator {
  @Inject @Delegate // Внедряем оригинальный объект
  private Calculator delegate;

  @Override
  public int add(int a, int b) {
    System.out.println("Audit: calculating " + a + " + " + b);
    return delegate.add(a, b); // Вызываем оригинальный метод
  }
}
```

6. Активируем в beans.xml:

```
<beans ...>
  <decorators>
    <class>com.example.AuditingCalculator</class>
  </decorators>
</beans>
```

8. События (Events)

- **Назначение:** Обеспечивают слабую связанность (loose coupling) между компонентами. Один компонент "выстреливает" событие, а другие, "слушающие" его, реагируют. Отправитель не знает о получателях.

1. Создаем объект-событие (Payload): Обычный POJO.

```
public class UserLoggedInEvent {
    public final String username;
    public UserLoggedInEvent(String username) {
        this.username = username;
    }
}
```


2. "Выстреливаем" событие:

```
public class LoginService {  
    @Inject  
    private Event<UserLoggedInEvent> loginEvent;  
  
    public void login(String username, String password) {  
        // ... логика аутентификации ...  
        if (success) {  
            loginEvent.fire(new UserLoggedInEvent(username));  
        }  
    }  
}
```

3. "Слушаем" событие:

```
public class AuditService {  
    public void onUserLogin(@Observes UserLoggedInEvent event) {  
        System.out.println("AUDIT: User " + event.username + " logged in.");  
    }  
}  
  
public class WelcomeEmailSender {  
    public void sendWelcomeEmail(@Observes UserLoggedInEvent event) {  
        // ... отправляем приветственное письмо ...  
    }  
}
```

9. Интеграция CDI с другими спецификациями

CDI — это "клей", который связывает всю платформу.

- **JAX-RS:** Ресурсы (`@Path`) являются CDI-бинами, вы можете `@Inject` в них любые другие бины.
- **JPA:** Вы можете `@Inject EntityManager` в ваши CDI-бины, хотя чаще используется `@PersistenceContext` в EJB.
- **JSF:** Управляемые бины (Managed Beans) в JSF полностью заменены на CDI-бины с нужными областями видимости (`@RequestScoped`, `@SessionScoped`).

1.9.2. Jakarta Enterprise Beans (EJB)

EJB — это не конкурент CDI, а дополнение к нему. EJB — это компоненты, которым контейнер предоставляет мощные системные сервисы, недоступные обычным CDI-бинам.

Когда использовать EJB? Когда вам нужны:

1. Декларативное управление транзакциями (самый частый случай).
2. Декларативная безопасность на уровне методов.
3. Асинхронное выполнение методов.
4. Запуск задач по расписанию (таймеры).
5. Удаленный вызов методов (RMI).

Обзор EJB Lite и Full EJB

- **EJB Lite (часть Web Profile):** Включает только Session Beans (`@Stateless`, `@Stateful`, `@Singleton`), `@Asynchronous`, `@Schedule` и поддержку локальных вызовов. Этого достаточно для 95% приложений.
- **Full EJB (часть Full Platform):** Добавляет Message-Driven Beans (для интеграции с JMS) и поддержку удаленных интерфейсов.

Session Beans

- **Назначение:** Для бизнес-логики, которая не хранит состояние, специфичное для клиента, между вызовами.
- **Пул экземпляров:** Контейнер держит пул идентичных экземпляров `@Stateless` бина. Когда приходит вызов, контейнер берет свободный экземпляр из пула, использует его и возвращает обратно. Это очень эффективно.
- **Параллелизм:** По своей природе потокобезопасны, так как каждый вызов обрабатывается отдельным экземпляром.

- **Пример (сервис с управлением транзакциями):**

@Stateless // Говорим контейнеру, что это Stateless EJB

```
public class OrderService {
```

@PersistenceContext // Внедрение EntityManager (JPA)

```
    private EntityManager em;
```

// По умолчанию, этот метод будет обернут в транзакцию (REQUIRED)

```
    public void createOrder(Order order) {
```

// Если здесь произойдет ошибка, контейнер откатит транзакцию

```
        em.persist(order);
```

```
    }
```

```
}
```

- **Назначение:** Для бизнес-логики, которая **хранит состояние для конкретного клиента** на протяжении "беседы" (conversation). Классический пример — корзина покупок.

- **Жизненный цикл:** Один экземпляр создается для одного клиента и связан с ним.

- **Пассивация/Активация:** Если экземпляров много, контейнер может сохранить состояние бина на диск (пассивация), чтобы освободить память, и восстановить его при следующем обращении (активация).

- **Пример:**

@Stateful

```
public class ShoppingCartEJB {
```

```
    private List<Item> items = new ArrayList<>();
```

```
    public void addItem(Item item) {
```

```
        items.add(item);
```

```
    }
```

@Remove // После вызова этого метода контейнер уничтожит экземпляр бина

```
    public void checkout() {
```

```

        // ... логика оформления заказа ...
        items.clear();
    }
}

```

- **Назначение: Один экземпляр на все приложение.** Для кэширования данных, управления общими ресурсами.
- **Инициализация:** Можно заставить контейнер создать бин при старте приложения с помощью `@Startup`.
- **Управление параллелизмом: Критически важно!** Так как экземпляр один, доступ к нему из разных потоков нужно контролировать.
 - `@ConcurrencyManagement(ConcurrencyManagementType.CONTAINER)` (по умолчанию): Контейнер управляет блокировками.
 - `@Lock(LockType.WRITE)` (по умолчанию): Эксклюзивный доступ. Только один поток может выполнять метод.
 - `@Lock(LockType.READ)`: Разделяемый доступ. Много потоков могут одновременно выполнять метод.

- **Пример:**

@Singleton

@Startup // Создать при старте приложения

```
public class AppCache {  
    private Map<String, Object> cache = new ConcurrentHashMap<>();
```

@Lock(LockType.READ) // Многие могут читать одновременно

```
    public Object getFromCache(String key) {  
        return cache.get(key);  
    }
```

@Lock(LockType.WRITE) // Только один поток может писать

```
    public void addToCache(String key, Object value) {  
        cache.put(key, value);  
    }
```

```
}
```

Таймеры EJB (@Schedule, TimerService)

- **Назначение:** Автоматический запуск методов по расписанию.
- **Пример:** Ежедневный отчет в полночь.

@Stateless

```
public class ReportGenerator {
```

@Schedule(hour = "0", minute = "0", second = "0", persistent = false)

```
    public void generateDailyReport() {
```

```
        System.out.println("Generating daily report...");
```

```
        // ... логика ...
```

```
    }
```

```
}
```

Асинхронные методы в EJB (@Asynchronous)

- **Назначение:** Для длительных операций, которые не должны блокировать вызывающий поток.
- **Возвращаемое значение:** Метод должен возвращать void или Future<V>.
- **Пример:**

```
@Stateless
```

```
public class ArchiveService {
```

```
    @Asynchronous
```

```
    public Future<String> createArchive(List<File> files) {
```

```
        // ... длительный процесс архивации ...
```

```
        String archivePath = "/path/to/archive.zip";
```

```
        return new AsyncResult<>(archivePath); // AsyncResult - это реализация  
Future из EJB
```

```
    }
```

```
}
```

```
// Вызывающий код
```

```
public class MyController {
```

```
    @EJB
```

```
    private ArchiveService archiveService;
```

```
    public void onArchiveButtonClick() {
```

```
        Future<String> future = archiveService.createArchive(files);
```

```
        // можно делать другую работу, не дожидаясь завершения ...
```

```
        // Позже можно проверить, готов ли результат: future.isDone()
```

```
    }
```

```
}
```

Интеграция EJB с CDI и JPA

EJB и CDI полностью интегрированы. EJB — это CDI-бины.

- Вы можете `@Inject` EJB в любой другой CDI-бин.
- Вы можете `@Inject` CDI-бины в ваши EJB.
- JPA (`@PersistenceContext`) прекрасно работает внутри EJB для управления транзакциями.

Локальные и удаленные интерфейсы

- **Локальный интерфейс (`@Local`):** (по умолчанию для EJB) Для вызовов внутри одного и того же приложения (JVM). Очень быстрая передача по ссылке.
- **Удаленный интерфейс (`@Remote`):** Для вызовов между разными приложениями (JVM), возможно, по сети. Требуется сериализация данных. В современных архитектурах часто заменяется на REST API.

1.10. Управление данными и транзакциями

1.10.1. Концепция Object-Relational Mapping (ORM)

Проблема: Несоответствие парадигм (Object-Relational Impedance Mismatch)

В разработке программного обеспечения мы сталкиваемся с фундаментальным конфликтом между двумя мирами:

1. **Объектно-ориентированный мир (Java, C#, Python и др.):**
 - Данные представлены в виде **объектов**.
 - Объекты имеют **свойства** (поля) и **поведение** (методы).
 - Ключевые концепции: **наследование** (класс `Dog` наследуется от `Animal`), **полиморфизм**, **инкапсуляция**.
 - Объекты могут иметь сложные связи: один-ко-многим (`Order` имеет список `OrderItem`), многие-ко-многим (`Student` связан с `Course`).
2. **Реляционный мир (SQL-базы данных: PostgreSQL, MySQL, Oracle и др.):**
 - Данные представлены в виде **таблиц**.
 - Таблицы состоят из **строк** и **столбцов** (колонок).
 - Данные являются "плоскими" и типизированными (числа, строки, даты).
 - Связи реализуются через **внешние ключи (foreign keys)**.
 - Нет концепций наследования или полиморфизма в том же виде.

Это несоответствие (impedance mismatch) порождает массу проблем. Чтобы сохранить объект User в базу данных, нам нужно вручную:

- Написать SQL-запрос INSERT INTO users (name, email) VALUES (?, ?);.
- Получить данные из полей объекта user.getName(), user.getEmail().
- Установить эти данные в параметры SQL-запроса, заботясь о типах и защите от SQL-инъекций.
- Чтобы прочесть данные, нужно выполнить SELECT ..., получить ResultSet, итерироваться по нему и вручную создавать объект new User(), заполняя его поля.

Этот "клей" между объектами и таблицами называется **кодом доступа к данным (Data Access Layer)**, и его написание — это рутинная, подверженная ошибкам и скучная работа.

Решение: ORM (Object-Relational Mapping)

ORM — это технология (или библиотека, фреймворк), которая автоматизирует процесс преобразования данных между объектно-ориентированной моделью и реляционной базой данных. Она создает "виртуальную объектную базу данных" поверх реляционной.

Ключевая идея ORM: Разработчик должен работать **только с объектами**, а ORM-фреймворк сам позаботится о генерации и выполнении SQL-запросов, преобразовании данных и управлении соединениями.

Как это работает: Основные компоненты ORM

ORM-решение обычно состоит из нескольких частей:

1. Механизм маппинга (Mapping):

Это правила, которые описывают, как Java-класс соответствует таблице в БД.

- Какой класс соответствует какой таблице (User -> users).
- Какое поле класса соответствует какому столбцу (userName -> user_name).
- Какое поле является первичным ключом (id -> id).
- Как связаны классы (Order и OrderItem) и как это отражается в БД (через order_id foreign key).

Этот маппинг может быть задан через **аннотации** (самый популярный способ) или через XML-файлы.

2. **API для управления сущностями (Entity Management API):**

Это набор методов для выполнения CRUD-операций (Create, Read, Update, Delete) с объектами. Вместо INSERT/UPDATE/SELECT/DELETE вы используете методы вроде save(), find(), merge(), remove().

3. **Язык запросов (Query Language):**

Поскольку писать SQL-запросы для сложных выборок все равно нужно, ORM предоставляет объектно-ориентированный язык запросов, который работает с классами и полями, а не с таблицами и столбцами. Примеры: JPQL (в Jakarta Persistence) или HQL (в Hibernate). ORM-фреймворк транслирует эти запросы в нативный SQL для конкретной СУБД.

4. **Управление транзакциями и кэширование:**

Хорошие ORM-решения помогают управлять транзакциями и предоставляют многоуровневое кэширование для повышения производительности (кэш первого уровня — на уровне сессии, кэш второго уровня — глобальный).

5. Пример "До" и "После" использования ORM

Давайте рассмотрим простой пример с классом Product.

Класс Product:

```
public class Product {  
    private Long id;  
    private String name;  
    private double price;  
    // getters and setters...  
}
```

Таблица в БД:

```
CREATE TABLE products (  
    id BIGINT PRIMARY KEY,  
    name VARCHAR(255),  
    price DECIMAL(10, 2)  
);
```

```

public void saveProduct(Product product)
    throws SQLException {
    String sql = "INSERT INTO products (id, name, price) VALUES (?, ?, ?)";
    try (
        Connection conn = dataSource.getConnection();
        PreparedStatement ps = conn.prepareStatement(sql)
    ) {
        ps.setLong(1, product.getId());
        ps.setString(2, product.getName());
        ps.setDouble(3, product.getPrice());
        ps.executeUpdate();
    }
}

```

```

public Product findProductById(Long id) throws SQLException {
    String sql = "SELECT id, name, price FROM products WHERE id = ?";
    try (
        Connection conn = dataSource.getConnection();
        PreparedStatement ps = conn.prepareStatement(sql)
    ) {
        ps.setLong(1, id);
        try (ResultSet rs = ps.executeQuery()) {
            if (rs.next()) {
                Product product = new Product();
                product.setId(rs.getLong("id"));
                product.setName(rs.getString("name"));
                product.setPrice(rs.getDouble("price"));
                return product;
            }
        }
    }
}

```

```
}  
return null;  
}
```

Проблемы: Много шаблонного кода, ручное управление ресурсами, риск ошибок (забыл закрыть ResultSet), SQL-строки разбросаны по коду.

Сначала мы "мапим" наш класс с помощью аннотаций.

Класс Product, размеченный для JPA:

```
import jakarta.persistence.*;
```

```
@Entity // 1. Говорим, что это сущность, управляемая JPA
```

```
@Table(name = "products") // 2. Указываем, с какой таблицей она связана
```

```
public class Product {
```

```
    @Id // 3. Указываем, что это поле - первичный ключ
```

```
    private Long id;
```

```
    private String name;
```

```
    private double price;
```

```
    // getters and setters...
```

```
}
```

Теперь код для работы с данными становится невероятно простым.

```
// EntityManager - это главный интерфейс JPA для работы с сущностями.
```

```
// Его предоставляет сервер приложений или сам ORM-фреймворк.
```

```
@PersistenceContext
```

```
private EntityManager em;
```

```
@Transactional // Управление транзакциями (например, через EJB)
```

```
public void saveProduct(Product product) {
```

```
    // Просто передаем объект! JPA сама сгенерирует нужный INSERT.
```

```
    em.persist(product);
```

```
}
```

```

public Product findProductById(Long id) {
    // Просто указываем класс и ключ. JPA сама сгенерирует SELECT.
    return em.find(Product.class, id);
}

```

@Transactional

```

public void updateProductPrice(Long id, double newPrice) {
    Product product = em.find(Product.class, id); // Находим продукт
    if (product != null) {
        product.setPrice(newPrice); // Просто меняем поле объекта
    }
    // Когда транзакция завершится, JPA увидит, что объект изменился,
    // и автоматически сгенерирует UPDATE-запрос.
}

```

Преимущества:

- **Меньше кода:** Никакого ручного SQL.
- **Надежность:** Меньше вероятность ошибок.
- **Переносимость:** Код не зависит от конкретной СУБД (PostgreSQL, MySQL). Если мы захотим сменить базу данных, нам нужно будет поменять только строку конфигурации, а не переписывать весь SQL.
- **Объектно-ориентированные запросы:**

// Пример запроса на JPQL (Java Persistence Query Language)

```

public List<Product> findExpensiveProducts(double minPrice) {
    // Запрос написан в терминах классов и полей, а не таблиц и столбцов
    String jpql = "SELECT p FROM Product p WHERE p.price > :minPrice";
    return em.createQuery(jpql, Product.class)
        .setParameter("minPrice", minPrice)
        .getResultList();
}

```

Преимущества и недостатки ORM

Преимущества:

- **Ускорение разработки:** Разработчики фокусируются на бизнес-логике, а не на SQL.
- **Независимость от СУБД:** Легкая смена базы данных.
- **Объектно-ориентированный подход:** Работа с данными ведется в привычной парадигме.
- **Встроенные оптимизации:** Кэширование, ленивая загрузка (lazy loading) и другие механизмы могут значительно повысить производительность.
- **Безопасность:** Защита от SQL-инъекций "из коробки" при использовании параметризованных запросов.

Недостатки:

- **Сложность:** ORM — это сложная технология. Чтобы использовать ее эффективно, нужно понимать, как она работает "под капотом".
- **Проблема N+1 запроса:** Классическая ловушка для новичков, когда из-за неоптимальной настройки связей ORM генерирует огромное количество лишних запросов к БД.
- **Снижение производительности (Overhead):** Для очень простых и высоконагруженных операций ORM может создавать небольшой оверхед по сравнению с "чистым" JDBC.
- **"Черный ящик":** Иногда трудно понять, какой именно SQL генерирует ORM, и сложно оптимизировать очень специфичные, сложные запросы. В таких случаях ORM обычно позволяет выполнять и нативный SQL.

Заключение

Несмотря на недостатки, преимущества ORM в большинстве современных приложений многократно перевешивают их. Технология ORM, и в частности ее стандарт **Jakarta Persistence (JPA)**, является краеугольным камнем разработки корпоративных приложений на Java, позволяя создавать более надежные, поддерживаемые и переносимые системы.

1.10.2. Jakarta Persistence (JPA): Сущности и базовые отображения

Сущности (@Entity)

Сущность (Entity) — это основной строительный блок JPA. Это обычный Java-класс (POJO - Plain Old Java Object), который представляет строку в таблице базы данных. Чтобы JPA-провайдер (например, Hibernate) "узнал" о таком классе и начал им управлять, класс необходимо пометить аннотацией `@jakarta.persistence.Entity`.

Требования к классу-сущности:

- Он должен быть помечен аннотацией `@Entity`.
- Он должен иметь **конструктор без аргументов** (может быть `public` или `protected`). JPA-провайдер использует его для создания экземпляров объектов при извлечении их из базы данных.
- Класс не должен быть `final`, а его методы и поля не должны быть `final`. Это нужно для того, чтобы JPA-провайдер мог создавать прокси-объекты для реализации ленивой загрузки и других оптимизаций.
- Он должен иметь поле, помеченное как первичный ключ (`@Id`).

Пример минимальной сущности:

```
import jakarta.persistence.Entity;
import jakarta.persistence.Id;
```

`@Entity` // 1. Объявляем класс как JPA-сущность

```
public class Product {
```

`@Id` // 2. Указываем, что это поле является первичным ключом

```
private Long id;
```

```
private String name;
```

// 3. JPA требует конструктор без аргументов

```
public Product() {}
```

// getters and setters...

```
}
```

По умолчанию JPA будет искать таблицу с таким же именем, как у класса (в данном случае `Product`). Чтобы явно указать имя таблицы и другие ее атрибуты, используется аннотация `@Table`.

```
import jakarta.persistence.Entity;
```

```
import jakarta.persistence.Id;
```

```
import jakarta.persistence.Table;
```

`@Entity`

`@Table(name = "products", schema = "inventory")` // Явно указываем имя таблицы и схему

```
public class Product {
```

```
    // ...
```

```
}
```

1.10.3. Первичные ключи (`@Id` и `@GeneratedValue`)

Каждая сущность **обязана** иметь первичный ключ. Поле, представляющее первичный ключ, помечается аннотацией `@jakarta.persistence.Id`.

В большинстве случаев мы не хотим назначать ID вручную. Мы хотим, чтобы база данных генерировала его автоматически. Для этого используется аннотация `@jakarta.persistence.GeneratedValue`.

У `@GeneratedValue` есть атрибут `strategy`, который определяет способ генерации ключа.

Стратегии генерации ключей (`GenerationType`)

1. `GenerationType.IDENTITY` (Самый распространенный для MySQL, PostgreSQL)

- **Как работает:** Полагается на механизм автоинкрементных колонок в базе данных (например, `AUTO_INCREMENT` в MySQL или `SERIAL/IDENTITY` в PostgreSQL).
- **Поведение:** JPA выполняет SQL `INSERT` и только после этого получает сгенерированный базой данных ID.
- **Когда использовать:** Когда ваша БД поддерживает автоинкрементные столбцы.

`@Id`

`@GeneratedValue(strategy = GenerationType.IDENTITY)`

`private Long id;`

2. `GenerationType.SEQUENCE` (Самый распространенный для Oracle, PostgreSQL)

- **Как работает:** Использует специальный объект в базе данных — **последовательность (sequence)** — для генерации уникальных значений.
- **Поведение:** JPA сначала запрашивает следующее значение у последовательности, а затем использует его при выполнении `INSERT`. Это позволяет выполнять пакетные вставки (`batch inserts`) более эффективно, чем `IDENTITY`.
- **Когда использовать:** Когда ваша БД поддерживает последовательности. Это часто является наиболее производительным вариантом.

@Id

```
@SequenceGenerator(name = "product_seq_gen", sequenceName =  
"product_id_seq", allocationSize = 1)
```

```
@GeneratedValue(strategy = GenerationType.SEQUENCE, generator =  
"product_seq_gen")
```

```
private Long id;
```

- @SequenceGenerator: определяет настройки для последовательности.
- sequenceName: имя последовательности в БД.
- allocationSize: сколько ID "резервировать" за один запрос к БД. 1 означает, что для каждого нового объекта будет делаться запрос к sequence. Большие значения улучшают производительность при массовых вставках.

3. GenerationType.TABLE (Устаревший, не рекомендуется)

- **Как работает:** JPA симулирует последовательность с помощью отдельной таблицы в базе данных.
- **Поведение:** Перед вставкой JPA блокирует строку в этой таблице, обновляет значение и использует его. Это **крайне неэффективно** из-за блокировок.
- **Когда использовать:** Только если ваша БД не поддерживает ни IDENTITY, ни SEQUENCE. В современных системах таких практически не осталось.

4. GenerationType.AUTO (По умолчанию)

- **Как работает:** Позволяет JPA-провайдеру (например, Hibernate) самому выбрать наиболее подходящую стратегию для используемой СУБД.
- **Поведение:** Для PostgreSQL он выберет SEQUENCE, для MySQL — IDENTITY.
- **Когда использовать:** Если вы хотите полностью положиться на провайдера и не заботиться о деталях. Однако явное указание стратегии делает код более предсказуемым.

1.10.4. Базовые типы и их отображение

JPA автоматически отображает большинство стандартных типов Java на соответствующие типы SQL без каких-либо дополнительных аннотаций.

- **Примитивы и их обертки:** int, long, double, Boolean -> INTEGER, BIGINT, DOUBLE, BOOLEAN и т.д.
- **Строки:** String -> VARCHAR.
- **Большие числа:** BigInteger, BigDecimal -> NUMERIC.

- **Даты и время (до Java 8):** java.util.Date, java.util.Calendar.
- **Даты и время (Java 8+):** java.time.LocalDate, java.time.LocalTime, java.time.LocalDateTime. Современные JPA-провайдеры поддерживают их "из коробки".

Для тонкой настройки отображения столбца используется аннотация @jakarta.persistence.Column.

Основные атрибуты @Column:

- name: Имя столбца в БД (полезно для соответствия javaCamelCase -> db_snake_case).
- nullable: true (по умолчанию) или false (создает ограничение NOT NULL).
- unique: true или false (создает ограничение UNIQUE).
- length: Длина для строковых столбцов (например, VARCHAR(255)).
- precision, scale: Точность и количество знаков после запятой для DECIMAL/NUMERIC.

Полный пример сущности

```
import jakarta.persistence.*;
import java.time.LocalDateTime;
```

```
@Entity
```

```
@Table(name = "articles")
```

```
public class Article {
```

```
    @Id
```

```
    @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
    private Long id;
```

```
    @Column(name = "title", length = 200, nullable = false)
```

```
    private String title;
```

```
    @Lob // Указывает, что поле должно храниться как Large Object (CLOB для
текста, BLOB для байт)
```

```
    @Column(name = "content", nullable = false)
```

```
    private String content;
```

```
@Column(name = "publication_date")
private LocalDateTime publicationDate; // Современный тип даты и времени
(Java 8+)
```

```
@Column(name = "is_published")
private boolean published;
```

```
@Enumerated(EnumType.STRING) // Настоятельно рекомендуется хранить
Enum как строку
@Column(name = "category", length = 50)
private Category category;
```

```
public enum Category {
    TECHNOLOGY,
    BUSINESS,
    LIFESTYLE
}
// Конструктор без аргументов, геттеры и сеттеры...
}
```

Разбор примера:

- **@Id @GeneratedValue**: Первичный ключ, генерируемый базой данных.
- **@Column(name = "title", ...)**: Поле title будет отображено на столбец title типа VARCHAR(200), который не может быть NULL.
- **@Lob**: Поле content будет отображено на текстовый тип данных для хранения больших объемов текста (например, TEXT в MySQL/PostgreSQL или CLOB в Oracle).
- **LocalDateTime**: JPA (Hibernate 5+) автоматически отобразит этот тип на TIMESTAMP в базе данных без дополнительных аннотаций.
- **@Enumerated(EnumType.STRING)**: Это очень важный момент.
 - EnumType.STRING (рекомендуется): В базе данных будет храниться строковое представление enum ('TECHNOLOGY', 'BUSINESS'). Это надежно.

- EnumType.ORDINAL (по умолчанию, **опасно**): В базе будет храниться порядковый номер enum (0, 1, 2). Если вы измените порядок или добавите новый enum в середину, ваши данные в БД станут некорректными! **Всегда используйте EnumType.STRING.**

1.10.5. EntityManager и Persistence Context

Что такое EntityManager?

EntityManager (Менеджер Сущностей) — это основной интерфейс, который ваше приложение использует для взаимодействия с JPA. Он предоставляет API для выполнения CRUD-операций (Create, Read, Update, Delete) над сущностями.

Можно думать о EntityManager как о "воротах" в мир персистентности. Вы получаете его от контейнера (через @PersistenceContext) и используете для всех операций с базой данных.

@Stateless // Например, в EJB

```
public class ProductService {
```

```
    // Контейнер внедрит сюда EntityManager, связанный с нужной базой данных.
```

```
    @PersistenceContext(unitName = "my-app-pu")
```

```
    private EntityManager em;
```

```
    public Product findProduct(Long id) {  
        return em.find(Product.class, id);  
    }
```

```
    public void saveProduct(Product product) {  
        em.persist(product);  
    }  
}
```

Что такое Persistence Context?

Persistence Context (Контекст Персистентности) — это "сердце" EntityManager. Это, по сути, кэш первого уровня, набор всех уникальных сущностей, которые EntityManager загрузил из базы данных или которые были сохранены в текущей транзакции.

Ключевые характеристики Persistence Context:

- **Он отслеживает изменения:** Любое изменение, которое вы вносите в поля управляемой (**managed**) сущности, будет автоматически отслежено Persistence Context.
- **Он гарантирует уникальность:** Внутри одного Persistence Context для одной и той же записи в базе данных (с тем же ID) всегда будет существовать **только один экземпляр** объекта.
- **Он привязан к транзакции:** Жизнь Persistence Context обычно совпадает с жизнью транзакции. Когда транзакция начинается, создается новый Persistence Context. Когда транзакция завершается (commit), все изменения, отслеженные контекстом, синхронизируются с базой данных. После этого контекст уничтожается.

Важно: EntityManager — это всего лишь API для доступа к Persistence Context. Именно Persistence Context выполняет всю "магическую" работу по отслеживанию объектов.

1.10.6. Жизненный цикл сущностей

Каждая сущность (экземпляр @Entity класса) в JPA может находиться в одном из четырех состояний. Понимание этих состояний критически важно.

Состояние 1: New (Новая / Transient)

- **Что это:** Объект только что был создан с помощью оператора new. Он еще никак не связан с Persistence Context и не имеет представления в базе данных.
- **Характеристики:**
 - Не имеет постоянного ID (если ID генерируется).
 - Persistence Context ничего о нем не знает.
 - Изменения в этом объекте не будут сохранены в БД.
 - Если на объект не останется ссылок, он будет уничтожен сборщиком мусора.
- **Пример кода:**

```
public void createNewProduct() {  
    // Состояние: New (Transient)  
    Product newProduct = new Product();  
    newProduct.setName("Super Gadget");  
    newProduct.setPrice(99.99);  
    // На данный момент newProduct - это просто обычный Java-объект.  
    // В базе данных его нет.
```

```
}
```

Состояние 2: Managed (Управляемая / Persistent)

- **Что это:** Сущность, которая связана с активным Persistence Context. Она либо была только что сохранена (`em.persist()`), либо загружена из базы данных (`em.find()`, `em.createQuery()`).
- **Характеристики:**
 - **Главная особенность:** Любые изменения в полях этой сущности будут **автоматически отслежены** Persistence Context.
 - Имеет постоянный ID.
 - Гарантированно представляет конкретную строку в БД.
 - Когда транзакция завершается, JPA автоматически сгенерирует SQL UPDATE для всех измененных управляемых сущностей (это называется **dirty checking** или "проверка на грязь").
- **Как перейти в состояние Managed:**
 - Вызвать `em.persist(entity)` для сущности в состоянии New.
 - Загрузить сущность через `em.find(Product.class, id)` или запрос.
 - Вызвать `em.merge(entity)` для сущности в состоянии Detached.
- **Пример кода:**

```
@Transactional
```

```
public void updateProductPrice(Long productId, double newPrice) {
```

```
    // 1. Загружаем продукт. Он переходит в состояние Managed.
```

```
    Product product = em.find(Product.class, productId);
```

```
    if (product != null) {
```

```
        // 2. Просто меняем поле у управляемого объекта.
```

```
        // НЕ НУЖНО вызывать em.update() или что-то подобное!
```

```
        product.setPrice(newPrice);
```

```
    }
```

```
}
```

```
// 3. Здесь транзакция завершается (commit). JPA видит, что поле 'price' у
```

```
// управляемого объекта 'product' изменилось, и автоматически выполняет:
```

```
// SQL: UPDATE products SET price = ? WHERE id = ?
```

Состояние 3: Detached (Отсоединенная)

- **Что это:** Сущность, которая **ранее была** в состоянии Managed, но Persistence Context, который ею управлял, был закрыт (например, транзакция завершилась).

- **Характеристики:**
 - Объект все еще существует в памяти Java.
 - У него есть ID.
 - Изменения в этом объекте **больше не отслеживаются** JPA.
 - Часто встречается в многоуровневых приложениях, когда сущность передается со слоя бизнес-логики на слой представления (например, в веб-форму), а затем возвращается обратно.
- **Как перейти в состояние Detached:**
 - Завершение транзакции (самый частый случай).
 - Явный вызов `em.detach(entity)`.
 - Явный вызов `em.close()` или `em.clear()`.
- **Пример кода:**

// Метод, который возвращает сущность на слой представления

```
public Product getProductForEdit(Long productId) {
```

```
    // Транзакция начинается и сразу заканчивается внутри этого метода
```

```
    Product product = em.find(Product.class, productId);
```

```
    return product; // Возвращается уже отсоединенный (Detached) объект.
```

```
}
```

// Метод, который получает измененный объект со слоя представления

```
@Transactional
```

```
public void saveEditedProduct(Product detachedProduct) {
```

```
    // detachedProduct находится в состоянии Detached. JPA не знает о его изменениях.
```

```
    // Чтобы сохранить изменения, его нужно "вернуть" в Persistence Context.
```

```
    // Используем em.merge()
```

```
    Product managedProduct = em.merge(detachedProduct);
```

```
    // merge() делает следующее:
```

```
    // 1. Загружает из БД свежую версию объекта с тем же ID.
```

```
    // 2. Копирует состояние из detachedProduct в загруженный managedProduct.
```

```
    // 3. Возвращает управляемую копию (managedProduct).
```

```
// 4. Дальнейшие изменения нужно делать уже на managedProduct.  
}
```

Состояние 4: Removed (Удаленная)

- **Что это:** Управляемая сущность, которая была помечена для удаления из базы данных с помощью `em.remove()`.
- **Характеристики:**
 - Сущность все еще связана с Persistence Context до конца транзакции.
 - Когда транзакция завершается (`commit`), JPA выполняет SQL DELETE.
 - После завершения транзакции объект переходит в состояние Detached (или становится доступным для сборщика мусора).
- **Пример кода:**

```
@Transactional
```

```
public void deleteProduct(Long productId) {  
    // 1. Находим сущность, она становится Managed.  
    Product product = em.find(Product.class, productId);  
    if (product != null) {  
        // 2. Помечаем для удаления. Сущность переходит в состояние  
        Removed.  
        em.remove(product);  
    }  
}  
  
// 3. Здесь транзакция завершается, и JPA выполняет:  
// SQL: DELETE FROM products WHERE id = ?
```


1.10.7. Сводная таблица по операциям и состояниям

Операция	Начальное состояние	Конечное состояние	Действие SQL при коммите
new Product()	(нет)	New	(нет)
em.persist(product)	New	Managed	INSERT
em.find(Product.class, id)	(нет)	Managed	SELECT (если нет в кеше)
em.remove(product)	Managed	Removed	DELETE
em.merge(product)	Detached	Managed (возвращает новый управляемый экземпляр)	UPDATE (если есть изменения)
em.detach(product)	Managed	Detached	(нет)
<i>Конец транзакции</i>	Managed/Removed	Detached	(различные)

1.10.8. Способы выполнения запросов в JPA

Когда простого `em.find(Entity.class, id)` недостаточно и вам нужно выполнить более сложный запрос, JPA предлагает три механизма:

5. **JPQL (Jakarta Persistence Query Language):** Объектно-ориентированный язык запросов, похожий на SQL. Самый распространенный способ.
6. **Criteria API:** Типобезопасный, программный способ построения запросов. Идеален для динамических запросов.
7. **Native Queries (Нативные запросы):** Позволяет выполнять "сырой" SQL, специфичный для вашей базы данных.

1.10.9. JPQL (Jakarta Persistence Query Language)

Что это?

JPQL — это язык запросов, который очень похож на SQL, но работает не с таблицами и столбцами, а с сущностями (классами) и их полями. JPQL-запросы являются переносимыми между разными базами данных, так как JPA-провайдер (Hibernate) сам транслирует их в нативный SQL для конкретной СУБД.

Ключевые отличия от SQL:

- **Регистрозависимость:** Имена сущностей и их полей регистрозависимы. Ключевые слова (SELECT, FROM, WHERE) — нет.
- **Имена сущностей:** Используются имена классов, а не таблиц. SELECT u FROM User u, а не SELECT * FROM users.
- **Навигация по связям:** Можно перемещаться по объектным связям с помощью точки, например, o.customer.address.city.

Синтаксис и примеры

Для выполнения JPQL-запросов используется метод `entityManager.createQuery()`.

Базовый SELECT: Выбрать всех пользователей.

```
// Выбрать всех пользователей и вернуть их в виде списка объектов User
```

```
String jpql = "SELECT u FROM User u";
```

```
TypedQuery<User> query = em.createQuery(jpql, User.class);
```

```
List<User> users = query.getResultList();
```

- **TypedQuery<User>:** Типизированный запрос, который гарантирует, что результат будет состоять из объектов User.

SELECT с условием (WHERE) и параметрами: Найти пользователя по email.

```
// Используйте именованные параметры (:emailParam) для безопасности и читаемости
```

```
String jpql = "SELECT u FROM User u WHERE u.email = :emailParam";
```

```
TypedQuery<User> query = em.createQuery(jpql, User.class);
```

```
query.setParameter("emailParam", "john.doe@example.com");
```

```
// Используем getSingleResult(), если уверены, что результат один.
```

```
// Если ничего не найдено, будет брошено NoResultException.
```

```
// Если найдено больше одного, будет брошено NonUniqueResultException.
```

SELECT с JOIN (для связей): Найти все заказы конкретного пользователя.

```
// Предполагаем, что у сущности Order есть поле "private User customer;"
String jpql = "SELECT o FROM Order o WHERE o.customer.name = :customerName";
TypedQuery<Order> query = em.createQuery(jpql, Order.class);
query.setParameter("customerName", "John Doe");
List<Order> orders = query.getResultList();
```

SELECT с явным JOIN и FETCH: Загрузить заказы вместе с их позициями одним запросом (решение проблемы "N+1 запроса").

```
// JOIN FETCH говорит JPA: "загрузи связанные сущности orderItems сразу"
String jpql = "SELECT o FROM Order o JOIN FETCH o.orderItems WHERE o.id = :orderId";
TypedQuery<Order> query = em.createQuery(jpql, Order.class);
query.setParameter("orderId", 1L);
Order order = query.getSingleResult();
```

// Теперь order.getOrderItems() не вызовет дополнительный запрос к БД

SELECT с проекцией (выборка конкретных полей): Выбрать только имена и email пользователей.

```
// Возвращаемый тип - Object[]
String jpql = "SELECT u.name, u.email FROM User u";
Query query = em.createQuery(jpql);
// Используем нетипизированный Query
List<Object[]> results = query.getResultList();
for (Object[] result : results) {
    String name = (String) result[0];
    String email = (String) result[1];
    System.out.println("Name: " + name + ", Email: " + email);
}
```

```
// Более удобный способ - DTO Projection (с помощью конструктора)
String jpqlDto = "SELECT new com.example.dto.UserDto(u.name, u.email) FROM User u";
TypedQuery<UserDto> queryDto = em.createQuery(jpqlDto, UserDto.class);
```

```
List<UserDto> userDtos = queryDto.getResultList();
```

JPQL UPDATE выполняется для всех сущностей, удовлетворяющих условию, и **обходит Persistence Context**. Это означает, что сущности, уже загруженные в память, не будут обновлены. Это операция для массовых изменений.

```
@Transactional
```

```
public int increasePriceForCategory(Category category, double percentage) {  
    String jpql = "UPDATE Product p SET p.price = p.price * (1 + :increase)  
WHERE p.category = :cat";  
    Query query = em.createQuery(jpql);  
    query.setParameter("increase", percentage / 100.0);  
    query.setParameter("cat", category);  
  
    // executeUpdate() возвращает количество измененных строк  
    int updatedCount = query.executeUpdate();  
    return updatedCount;  
}
```

Аналогично UPDATE, DELETE работает в обход Persistence Context.

@Transactional

```
public int deleteInactiveUsers(LocalDateTime cutOffDate) {  
    String jpql = "DELETE FROM User u WHERE u.lastLoginDate < :cutOff";  
    Query query = em.createQuery(jpql);  
    query.setParameter("cutOff", cutOffDate);  
  
    int deletedCount = query.executeUpdate();  
    return deletedCount;  
}
```

1.10.10. Criteria API

Что это?

Criteria API — это программный, строго типизированный способ построения запросов. Вместо написания строк JPQL, вы строите запрос, вызывая методы Java.

Преимущества:

- **Типобезопасность:** Ошибки обнаруживаются на этапе компиляции, а не во время выполнения. Вы не можете ошибиться в имени поля.
- **Динамические запросы:** Идеально подходит для построения запросов "на лету" на основе множества опциональных фильтров (например, в форме поиска с кучей полей).

Недостатки:

- **Многословность:** Код получается гораздо более громоздким по сравнению с JPQL.

Пример: Поиск продуктов по нескольким критериям

Представим, что у нас есть форма поиска с полями "название", "минимальная цена" и "категория". Все поля опциональны.

```
public List<Product> findProductsByCriteria(String name, Double minPrice,  
Category category) {  
    // 1. Получаем "строителей"  
    CriteriaBuilder cb = em.getCriteriaBuilder();  
    CriteriaQuery<Product> cq = cb.createQuery(Product.class);  
    // 2. Определяем "корень" запроса (аналог FROM Product p)  
    Root<Product> product = cq.from(Product.class);
```

```

// 3. Создаем список предикатов (условий WHERE)
List<Predicate> predicates = new ArrayList<>();

// 4. Динамически добавляем условия
if (name != null && !name.isEmpty()) {
    predicates.add(cb.like(
        product.get("name"), "%" + name + "%")
    );
}
if (minPrice != null) {
    predicates.add(cb.greaterThanOrEqualTo(product.get("price"),
minPrice));
}
if (category != null) {
    predicates.add(cb.equal(product.get("category"), category));
}

// 5. Собираем запрос
cq.where(predicates.toArray(new Predicate[0]));

// 6. Выполняем запрос
TypedQuery<Product> query = em.createQuery(cq);
return query.getResultList();
}

```

В современных проектах для этого часто используют фреймворки вроде Spring Data JPA или спецификацию Jakarta Data, которые значительно упрощают написание таких запросов.

1.10.11. Native Queries (Нативные запросы)

Что это?

Это способ выполнить "сырой" SQL-запрос, специфичный для вашей базы данных.

Когда использовать:

- Когда вам нужна функция, специфичная для вашей СУБД, которой нет в JPQL (например, рекурсивные запросы WITH RECURSIVE в PostgreSQL, или CONNECT BY в Oracle, или хинты для оптимизатора).
- Для выполнения DDL-операций (хотя это плохая практика).
- Для вызова хранимых процедур.

Недостатки:

- **Потеря переносимости:** Запрос будет работать только в той СУБД, для которой он написан.
- **Риск SQL-инъекций:** Если вы конкатенируете строки вместо использования параметров, вы открываете уязвимость.

Примеры

- Простой нативный запрос:

```
String sql = "SELECT * FROM users WHERE email = ?";
```

```
Query query = em.createNativeQuery(sql, User.class); // Указываем, в какой класс мапить результат
```

```
query.setParameter(1, "john.doe@example.com"); // Позиционные параметры
```

```
User user = (User) query.getSingleResult();
```

- Нативный запрос с маппингом в DTO (через @SqlResultSetMapping)

Иногда нужно получить результат, который не соответствует ни одной сущности.

// 1. В файле orm.xml или прямо над сущностью определяем маппинг

```
@SqlResultSetMapping(  
    name = "UserStatsMapping",  
    classes = @ConstructorResult(  
        targetClass = UserStatsDto.class,  
        columns = {  
            @ColumnResult(name = "user_id", type = Long.class),  
            @ColumnResult(name = "user_name"),  
            @ColumnResult(name = "order_count", type = Long.class)  
        }  
    )  
)
```

)

@Entity

public class User { /*...*/ }

// 2. Используем этот маппинг в коде

```
public List<UserStatsDto> getUserStats() {  
    String sql = "SELECT u.id as user_id, u.name as user_name, count(o.id) as  
order_count " +  
        "FROM users u LEFT JOIN orders o ON u.id = o.customer_id " + "GROUP  
BY u.id, u.name";  
    Query query = em.createNativeQuery(sql, "UserStatsMapping");  
    return query.getResultList();  
}
```


1.10.12. Заключение: что когда использовать?

Метод	Лучший сценарий использования	Плюсы	Минусы
JPQL	90% случаев: Статические или умеренно-динамические запросы.	Переносимый, лаконичный, легко читаемый.	Нетипобезопасный (ошибки в строках), ограниченный функционал.
Criteria API	Динамические запросы: Построение запроса на основе множества опциональных критериев (сложные фильтры).	Типобезопасный , хорошо подходит для динамики.	Очень многословный, сложно читать.
Native Query	Специфичные для БД запросы: Использование уникальных функций СУБД, вызов хранимых процедур, сложная оптимизация.	Полная мощь SQL , нет ограничений.	Непереносимый , потенциально небезопасный, обходит некоторые кэши JPA.

1.10.13. Отношения между сущностями в JPA

В реляционных базах данных отношения между таблицами строятся с помощью внешних ключей (foreign keys). В JPA мы моделируем эти отношения на уровне объектов с помощью специальных аннотаций.

JPA поддерживает четыре типа отношений:

1. **@ManyToOne**: Многие-к-Одному (самое частое)
2. **@OneToMany**: Один-ко-Многим
3. **@OneToOne**: Один-к-Одному
4. **@ManyToMany**: Многие-ко-Многим

Давайте разберем каждое на примерах.

1.10.14. @ManyToOne (Многие-к-Одному)

Это самый распространенный и простой тип отношений.

- **Концепция:** Множество экземпляров одной сущности ("многие") ссылаются на один экземпляр другой сущности ("один").
- **Пример:** Множество Order (заказов) могут принадлежать одному Customer (клиенту).
- **Реализация в БД:** В таблице orders будет столбец с внешним ключом customer_id, который ссылается на id в таблице customers.

Код

// Сущность "один" (владелец отношения)

@Entity

@Table(name = "customers")

public class Customer {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

// ...

}

// Сущность "многие" (сторона, где находится внешний ключ)

@Entity

@Table(name = "orders")

public class Order {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private LocalDateTime orderDate;

@ManyToOne(fetch = FetchType.LAZY) // Указываем тип отношения

`@JoinColumn(name = "customer_id")` // Указываем столбец с внешним ключом

```
private Customer customer;  
// ...  
}
```

Разбор аннотаций:

- `@ManyToOne`: Устанавливает тип отношения.
- `@JoinColumn(name = "customer_id")`: **Критически важная аннотация.** Она говорит JPA, какой столбец в таблице orders использовать в качестве внешнего ключа для связи с таблицей customers. Это **владеющая сторона** отношения.

1.10.15. @OneToMany (Один-ко-Многим)

Это обратная сторона отношения @ManyToOne.

- **Концепция:** Один экземпляр сущности ссылается на коллекцию экземпляров другой сущности.
- **Пример:** Один Customer может иметь коллекцию Order.
- **Важно:** Отношение @OneToMany почти всегда является **двунаправленным** (bidirectional), то есть оно настраивается в паре с @ManyToOne.

Код (продолжение предыдущего примера)

Добавим коллекцию заказов в класс Customer.

```
@Entity
```

```
@Table(name = "customers")
```

```
public class Customer {
```

```
    @Id
```

```
    @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
    private Long id;
```

```
    private String name;
```

```
    // mappedBy указывает на поле в классе Order, которое "владеет" этим  
    // отношением
```

```
    @OneToMany(mappedBy = "customer", cascade = CascadeType.ALL,  
    orphanRemoval = true)
```

```
    private List<Order> orders = new ArrayList<>();
```

```
    // ... геттеры, сеттеры, и вспомогательные методы для синхронизации
```

```
    public void addOrder(Order order) {
```

```
        orders.add(order);
```

```
        order.setCustomer(this);
```

```
    }
```

```
}
```

Разбор аннотаций:

- @OneToMany: Устанавливает тип отношения.

- `mappedBy = "customer"`: **Самый важный атрибут!** Он говорит JPA: "Это отношение **не является владеющим**. Ищи конфигурацию внешнего ключа в поле `customer` класса `Order`". Без `mappedBy` JPA попытается создать промежуточную таблицу для связи, что неверно для данного типа отношений.
- `cascade`: Описывает каскадирование операций (см. ниже).
- `orphanRemoval = true`: Если вы удаляете заказ из коллекции `customer.getOrders().remove(order)`, то этот заказ будет удален и из базы данных.

1.10.16. @OneToOne (Один-к-Одному)

- **Концепция:** Один экземпляр сущности связан ровно с одним экземпляром другой.
- **Пример:** У одного User может быть только один UserProfile.
- **Реализация в БД:** Обычно реализуется через общий первичный ключ (shared primary key) или через уникальный внешний ключ.

Код

```
// Владеющая сторона (здесь будет внешний ключ)
@Entity
@Table(name = "users")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String username;

    @OneToOne(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
    @JoinColumn(name = "profile_id", referencedColumnName = "id") //
    // Внешний ключ в таблице users
    private UserProfile profile;
    // ...
}
```

```
// Зависимая сторона
@Entity
@Table(name = "user_profiles")
public class UserProfile {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String bio;

    // Для двунаправленной связи
    @OneToOne(mappedBy = "profile")
    private User user;
    // ...
}
```

1.10.17. @ManyToMany (Многие-ко-Многим)

- **Концепция:** Множество экземпляров одной сущности могут быть связаны с множеством экземпляров другой.
- **Пример:** Один Student может посещать множество Course, и один Course может иметь множество Student.
- **Реализация в БД:** Всегда требует создания **промежуточной таблицы (join table)**, которая хранит пары внешних ключей (student_id, course_id).

Код

```
@Entity
```

```
@Table(name = "students")
```

```
public class Student {
```

```
    @Id
```

```
    @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
    private Long id;
```

```
    private String name;
```

```
    @ManyToMany(cascade = { CascadeType.PERSIST, CascadeType.MERGE })
```

```
    @JoinTable(
```

```
        name = "student_courses", // Имя промежуточной таблицы
```

```
        joinColumns = @JoinColumn(name = "student_id"),
```

```
        // Внешний ключ на эту сущность
```

```
        inverseJoinColumns = @JoinColumn(name = "course_id")
```

```
        // Внешний ключ на другую сущность
```

```
    )
```

```
    private Set<Course> courses = new HashSet<>();
```

```
    // ...
```

```
}
```



```

@Entity
@Table(name = "courses")
public class Course {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String title;

    @ManyToMany(mappedBy = "courses") // Обратная сторона, не владеющая
    private Set<Student> students = new HashSet<>();
    // ...
}

```

Разбор аннотаций:

- **@ManyToMany:** Устанавливает тип отношения.
- **@JoinTable:** Конфигурирует промежуточную таблицу.
 - **Использование Set вместо List:** Рекомендуется для @ManyToMany и @OneToMany, чтобы избежать проблем с дубликатами и производительностью.

1.10.18. Важные концепции: Cascade и FetchType

Каскадирование (Cascading)

Атрибут `cascade` в аннотациях отношений определяет, какие операции, выполненные над родительской сущностью, должны быть "каскадно" применены к связанным с ней дочерним сущностям.

- **CascadeType.PERSIST**: Когда вы сохраняете родителя (`em.persist(parent)`), дочерние сущности тоже сохраняются.
- **CascadeType.MERGE**: Когда вы обновляете отсоединенного родителя (`em.merge(parent)`), изменения в дочерних сущностях тоже обновляются.
- **CascadeType.REMOVE**: Когда вы удаляете родителя (`em.remove(parent)`), дочерние сущности тоже удаляются.
- **CascadeType.REFRESH**: Когда вы обновляете состояние родителя из БД (`em.refresh(parent)`), дочерние сущности тоже обновляются.
- **CascadeType.ALL**: Включает в себя все вышеперечисленные операции. Удобно, но используйте с осторожностью.

Пример:

Если в отношении `Customer <-> Order` указано `cascade = CascadeType.ALL`, то код:

```
Customer customer = new Customer();
```

```
customer.addOrder(new Order()); // Добавляем новый, еще не сохраненный заказ
```

```
em.persist(customer); // Сохранит и клиента, и его заказ
```

...будет работать. Без `CascadeType.PERSIST` вы бы получили ошибку, так как пытаетесь сохранить `Customer`, который ссылается на несохраненный `Order`.

Типы загрузки (Fetch Types)

Атрибут fetch определяет, **когда** JPA должен загружать связанные сущности из базы данных.

1. FetchType.LAZY (Ленивая загрузка)

- **Поведение:** Связанные сущности **не загружаются** из БД вместе с родительской. Вместо них подставляется специальный прокси-объект. Данные будут загружены только в тот момент, когда вы **впервые обратитесь** к этому полю (например, вызовете `customer.getOrders()`).
- **Когда использовать:** Почти всегда для коллекций (`@OneToMany`, `@ManyToMany`) и часто для `@OneToOne`. Это позволяет избежать загрузки огромного графа объектов и предотвращает проблему "N+1 запроса".
- **Проблема LazyInitializationException:** Если вы попытаетесь получить доступ к ленивой коллекции после того, как `EntityManager` (и транзакция) закрыт, вы получите эту ошибку. Решение: использовать JOIN FETCH в JPQL-запросе или паттерн "Open Session in View".

2. FetchType.EAGER (Жадная/немедленная загрузка)

- **Поведение:** Связанные сущности загружаются из БД **сразу же**, одним SELECT запросом (с помощью JOIN), вместе с родительской сущностью.
- **Когда использовать:** Только для отношений *-to-one (`@ManyToOne`, `@OneToOne`), когда вы **уверены**, что связанные данные понадобятся вам почти всегда. Использовать EAGER для коллекций — очень плохая практика, которая почти всегда ведет к проблемам с производительностью.

Правила по умолчанию в JPA:

- `@ManyToOne` и `@OneToOne`: по умолчанию **EAGER**.
- `@OneToMany` и `@ManyToMany`: по умолчанию **LAZY**.

Рекомендация: Всегда явно указывайте fetch тип. Для всех коллекций ставьте FetchType.LAZY. Для *-to-one отношений ставьте LAZY, если данные не нужны постоянно.

```
// Хорошая практика
```

```
@ManyToOne(fetch = FetchType.LAZY)
```

```
private Customer customer;
```

```
// Плохая практика, может привести к загрузке тысяч объектов
```

```
@OneToMany(fetch = FetchType.EAGER)
```

```
private List<Order> orders;
```

1.10.19. Наследование сущностей (Entity Inheritance)

Обычное Java-наследование (class Dog extends Animal) можно отобразить и на реляционные таблицы. JPA предлагает три основные стратегии для этого.

Предположим, у нас есть иерархия классов:

```
// Абстрактный базовый класс
```

```
@Entity
```

```
// Здесь будет аннотация стратегии наследования
```

```
public abstract class BillingDetails {
```

```
    @Id
```

```
    @GeneratedValue
```

```
    private Long id;
```

```
    private String owner;
```

```
    // ...
```

```
}
```

// Конкретный наследник 1

@Entity

```
public class CreditCard extends BillingDetails {  
    private String cardNumber;  
    private String expiryMonth;  
    private String expiryYear;  
    // ...  
}
```

// Конкретный наследник 2

@Entity

```
public class BankAccount extends BillingDetails {  
    private String accountNumber;  
    private String bankName;  
    // ...  
}
```

Как это хранить в базе данных?

1. Стратегия 1: SINGLE_TABLE (Одна таблица на всю иерархию)

- **Как работает:** Все классы иерархии хранятся в **одной общей таблице**. Для различения типов используется специальный **столбец-дискриминатор**.
- **Аннотации:**
 - @Inheritance(strategy = InheritanceType.SINGLE_TABLE) на базовом классе.
 - @DiscriminatorColumn(name = "billing_type") на базовом классе для указания имени столбца-дискриминатора.
 - @DiscriminatorValue("CC") или @DiscriminatorValue("BA") на дочерних классах для указания значения в столбце-дискриминаторе.

- **Код:**

```
@Entity
```

```
@Table(name = "billing_details")
```

```
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
```

```
@DiscriminatorColumn(name = "billing_type", discriminatorType =  
DiscriminatorType.STRING)
```

```
public abstract class BillingDetails { /* ... */ }
```

```
@Entity
```

```
@DiscriminatorValue("CREDIT_CARD") // Значение для этого типа
```

```
public class CreditCard extends BillingDetails { /* ... */ }
```

```
@Entity
```

```
@DiscriminatorValue("BANK_ACCOUNT") // Значение для этого типа
```

```
public class BankAccount extends BillingDetails { /* ... */ }
```

- **Структура таблицы billing_details:**

id(PK)		billing_type	cardNumber	expiryMonth	expiryYear	accountNumber	bankName
1	John	CREDIT_CARD	1234..	12	2025	NULL	NULL
2	Jane	BANK_ACCOUNT	NULL	NULL	NULL	5678...	MyBank

- **Плюсы:**
 - **Производительность:** Полиморфные запросы (найти все BillingDetails) очень быстрые, так как требуют сканирования всего одной таблицы. Не нужны JOIN'ы.
- **Минусы:**
 - **Модель данных:** Все поля всех подклассов находятся в одной таблице, что приводит к большому количеству NULL значений. Нельзя сделать поле cardNumber обязательным (NOT NULL) на уровне БД, так как для банковских счетов оно будет NULL.

Стратегия 2: JOINED (Соединенные таблицы)

- **Как работает:** Для каждого класса в иерархии (включая базовый) создается **своя таблица**. Связь между таблицей наследника и таблицей родителя осуществляется через общий первичный ключ, который одновременно является и внешним ключом.
- **Аннотации:** @Inheritance(strategy = InheritanceType.JOINED) на базовом классе.
- **Код:**

```
@Entity
```

```
@Table(name = "billing_details")
```

```
@Inheritance(strategy = InheritanceType.JOINED)
```

```
public abstract class BillingDetails { /* ... */ }
```

```
@Entity
```

```
@Table(name = "credit_cards")
```

```
@PrimaryKeyJoinColumn(name = "billing_details_id") // Указываем имя
PK/FK
```

```
public class CreditCard extends BillingDetails { /* ... */ }
```

```
@Entity
```

```
@Table(name = "bank_accounts")
```

```
@PrimaryKeyJoinColumn(name = "billing_details_id")
```

```
public class BankAccount extends BillingDetails { /* ... */ }
```

- **Структура таблиц:**

billing_details:

id (PK)	owner
1	John
2	Jane

credit_cards:

billing_details_id (PK, FK)	cardNumber	expiryMonth	expiryYear
1	1234...	12	2025

bank_accounts:

billing_details_id (PK, FK)	accountNumber	bankName
2	5678...	MyBank

- **Плюсы:**
 - **Чистая модель данных:** Данные нормализованы, нет лишних NULL. Можно использовать NOT NULL ограничения.
- **Минусы:**
 - **Производительность:** Полиморфные запросы требуют JOIN'ов между таблицами, что может быть медленнее, чем SINGLE_TABLE.

Стратегия 3: TABLE_PER_CLASS (Таблица на каждый конкретный класс)

- **Как работает:** Для каждого **конкретного** (не абстрактного) класса создается своя таблица, которая содержит **все поля** этого класса, включая унаследованные. Базовый класс таблицы не имеет.
- **Аннотации:** `@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)` на базовом классе. =
- **Код:**

```
@Entity
```

```
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
```

```
public abstract class BillingDetails { /* ... */ }
```

```
@Entity
```

```
@Table(name = "credit_cards")
```

```
public class CreditCard extends BillingDetails {  
    /* ... */  
}
```

```
@Entity
```

```
@Table(name = "bank_accounts")
```

```
public class BankAccount extends BillingDetails {  
    /* ... */  
}
```

- **Структура таблиц:**

credit_cards:

id (PK)	owner	cardNumber	expiryMonth	expiryYear
1	John	1234...	12	2025

bank_accounts:

id (PK)	owner	accountNumber	bankName
2	Jane	5678...	MyBank

- **Плюсы:**
 - Позволяет иметь чистую модель для каждого подкласса.
- **Минусы:**
 - **Производительность:** Полиморфные запросы являются самыми медленными, так как требуют выполнения UNION между несколькими таблицами.
 - **Ограничения:** Невозможно использовать IDENTITY для генерации ключей. Внешние ключи на базовый класс BillingDetails не могут быть созданы.
 - **Рекомендация:** Эту стратегию обычно советуют избегать.

1.10.20. Встраиваемые объекты (@Embeddable)

Что это?

@Embeddable — это способ сгруппировать несколько полей в отдельный переиспользуемый класс, который **не является сущностью** (@Entity) и не имеет своей собственной таблицы. Вместо этого его поля "встраиваются" как столбцы в таблицу владеющей им сущности.

Когда использовать?

- Для представления составных частей, которые не имеют собственного ID и не могут существовать отдельно от своего владельца.
- Для уменьшения дублирования кода. Классический пример — Address.

Пример

Предположим, у нас есть User и Company, и у обоих есть адрес. Вместо дублирования полей street, city, zipcode в обоих классах, мы создадим встраиваемый класс Address.

1. Создаем встраиваемый класс:

```
import jakarta.persistence.Embeddable;
```

```
@Embeddable // 1. Помечаем класс как встраиваемый
```

```
public class Address {
```

```
    private String street;
```

```
    private String city;
```

```
    private String zipcode;
```

```
    // Важно: должен быть конструктор без аргументов
```

```
    public Address() {}
```

```
// getters and setters...  
}
```

2. Встраиваем его в сущности:

```
@Entity
```

```
@Table(name = "users")
```

```
public class User {
```

```
    @Id
```

```
    @GeneratedValue
```

```
    private Long id;
```

```
    private String name;
```

```
    @Embedded // 2. Указываем, что сюда будет встроен объект Address
```

```
    private Address homeAddress;
```

```
    // ...
```

```
}
```

```
@Entity
```

```
@Table(name = "companies")
```

```
public class Company {
```

```
    @Id
```

```
    @GeneratedValue
```

```
    private Long id;
```

```
    private String companyName;
```

```
    @Embedded
```

```
    private Address legalAddress;
```

```
// ...  
}
```

Структура таблицы users:

JPA "развернет" поля из Address прямо в таблицу users.

id (PK)	name	street	city	zipcode
1	John	123 Main St	Anytown	12345

Переопределение имен столбцов (@AttributeOverride)

Проблема: Что, если у сущности должно быть два адреса? Например, домашний и рабочий.

@Entity

```
public class User {
```

```
// ...
```

@Embedded

```
private Address homeAddress;
```

@Embedded

```
private Address workAddress; // Конфликт! Оба захотят создать столбцы  
street, city...
```

```
}
```

JPA выдаст ошибку, так как попытается создать столбцы с одинаковыми именами.

Решение: `@AttributeOverrides` и `@AttributeOverride`.

`@Entity`

```
public class User {
```

```
    // ...
```

```
    @Embedded
```

```
    @AttributeOverrides({
```

```
        @AttributeOverride(name = "street", column = @Column(name =  
"home_street")),
```

```
        @AttributeOverride(name = "city", column = @Column(name =  
"home_city")),
```

```
        @AttributeOverride(name = "zipcode", column = @Column(name =  
"home_zipcode"))
```

```
    })
```

```
    private Address homeAddress;
```

```
    @Embedded
```

```
    @AttributeOverrides({
```

```
        @AttributeOverride(name = "street", column = @Column(name =  
"work_street")),
```

```
        @AttributeOverride(name = "city", column = @Column(name =  
"work_city")),
```

```
        @AttributeOverride(name = "zipcode", column = @Column(name =  
"work_zipcode"))
```

```
    })
```

```
    private Address workAddress;
```

```
}
```

- `@AttributeOverride`: `name` — это имя поля в классе `@Embeddable`, `column` — это новое определение столбца в таблице сущности.

Теперь таблица `users` будет выглядеть так, как и ожидается, без конфликтов имен.

1.10.21. Управление параллелизмом: Блокировки в JPA

Представьте классическую ситуацию:

1. **Пользователь А** загружает данные о продукте (цена: 100) для редактирования.
2. **Пользователь Б** в это же время загружает данные о том же продукте (цена: 100).
3. **Пользователь А** меняет цену на 120 и сохраняет. В базе теперь цена 120.
4. **Пользователь Б** меняет цену на 90 (он-то видел старую цену 100) и сохраняет. В базе теперь цена 90.

Проблема: Изменение, сделанное пользователем А, **потеряно (Lost Update)**.

ЈРА предлагает два фундаментальных подхода для решения этой проблемы: **оптимистические** и **пессимистические** блокировки.

1.10.22. Оптимистическая блокировка (Optimistic Locking)

Философия: "Мы надеемся (оптимистично), что конфликты будут редкими, поэтому не будем блокировать данные в базе. Вместо этого мы просто проверим перед сохранением, не изменил ли их кто-то другой за это время".

Как работает:

1. В таблицу сущности добавляется специальный **столбец-версия** (version column).
2. Когда сущность загружается из БД, ЈРА запоминает значение этого столбца (например, version = 1).
3. Когда приходит время обновить сущность, ЈРА выполняет UPDATE с условием WHERE:

UPDATE products SET price = ?, version = version + 1 WHERE id = ? AND version = ?

Вместо последнего ? подставляется **старое значение версии**, которое было загружено (в нашем случае 1).

4. Два возможных исхода:

- **Успех:** Если никто не менял запись, условие version = 1 выполнится. Запись обновится, и version станет 2.
- **Конфликт:** Если кто-то другой уже успел обновить запись, ее версия в базе уже 2. Условие version = 1 не выполнится, UPDATE не затронет ни одной строки. ЈРА обнаружит это и бросит исключение OptimisticLockException.

Приложение должно перехватить это исключение и решить, что делать: сообщить пользователю об ошибке, предложить обновить данные и попробовать снова, или выполнить слияние изменений.

Реализация в JPA:

Используется аннотация `@Version` на поле, которое будет хранить номер версии. Тип поля может быть `int`, `long`, `short` или `java.sql.Timestamp`.

```
@Entity
```

```
@Table(name = "products")
```

```
public class Product {
```

```
    @Id
```

```
    @GeneratedValue
```

```
    private Long id;
```

```
    private String name;
```

```
    private double price;
```

```
    @Version // 1. Добавляем поле-версию
```

```
    private long version;
```

```
    // getters and setters
```

```
}
```

Пример сценария:

```
// --- Транзакция 1 (Пользователь А) ---
```

```
EntityManager em1 = ...
```

```
em1.getTransaction().begin();
```

```
Product p1 = em1.find(Product.class, 1L); // Загружен продукт с id=1, version=1
```

```
p1.setPrice(120.0);
```

```
// Пока не коммитим...
```

```
// --- Транзакция 2 (Пользователь Б) ---
```

```
EntityManager em2 = ...
```

```
em2.getTransaction().begin();
```

```
Product p2 = em2.find(Product.class, 1L); // Загружен тот же продукт с id=1, version=1
```

```
p2.setPrice(90.0);
```

```
em2.getTransaction().commit(); // <-- COMMIT пользователя Б
```

```
// SQL: UPDATE products SET price=90.0, version=2 WHERE id=1 AND  
version=1
```

```
// Успех! В базе теперь price=90, version=2.
```

```
// --- Возвращаемся к Транзакции 1 ---
```

```
em1.getTransaction().commit(); // <-- COMMIT пользователя A
```

```
// SQL: UPDATE products SET price=120.0, version=2 WHERE id=1 AND  
version=1
```

```
// Запрос ничего не обновит, так как в базе version уже 2.
```

```
// JPA бросит OptimisticLockException!
```

Когда использовать:

- В **95% случаев**. Особенно в веб-приложениях, где транзакции короткие.
- Когда вероятность одновременного редактирования одной и той же записи мала.
- Когда целостность данных важна, но блокировка записи в БД на длительное время нежелательна.

1.10.23. Пессимистическая блокировка (Pessimistic Locking)

Философия: "Мы предполагаем (пессимистично), что конфликты очень вероятны, поэтому мы заблокируем запись в базе данных с самого начала, чтобы никто другой не мог ее изменить, пока мы с ней работаем".

Как работает:

1. Когда вы загружаете сущность, вы явно указываете JPA, что хотите наложить на нее блокировку.
2. JPA транслирует это в команду `SELECT ... FOR UPDATE` (или аналогичную, в зависимости от СУБД).
3. База данных **блокирует строку** на физическом уровне.
4. Любая другая транзакция, которая попытается прочитать эту строку с блокировкой или изменить ее, будет **остановлена** и будет **ждать**, пока первая транзакция не завершится (commit или rollback) и не снимет блокировку.

Реализация в JPA:

Блокировка запрашивается при вызове `em.find()` или `query.setLockMode()`. Тип блокировки задается через `LockModeType`.

- `LockModeType.PESSIMISTIC_READ` (`SELECT ... FOR SHARE` в PostgreSQL):

- Разрешает другим транзакциям **читать** запись, но не изменять ее.
- Это разделяемая (shared) блокировка. Несколько транзакций могут одновременно держать READ-блокировку.
- LockModeType.PESSIMISTIC_WRITE (SELECT ... FOR UPDATE в PostgreSQL):
 - Запрещает другим транзакциям и читать (с блокировкой), и изменять запись.
 - Это эксклюзивная (exclusive) блокировка.

Пример кода:

```
// --- Транзакция 1 (начало) ---
EntityManager em1 = ...
em1.getTransaction().begin();

// Загружаем продукт и СРАЗУ ЖЕ блокируем его в базе данных
Product p1 = em1.find(Product.class, 1L,
LockModeType.PESSIMISTIC_WRITE);

// --- Транзакция 2 (пытается вмешаться) ---
EntityManager em2 = ...
em2.getTransaction().begin();

// Эта операция ЗАВИСИТ. Транзакция 2 будет ждать, пока Транзакция 1
не завершится.
Product p2 = em2.find(Product.class, 1L,
LockModeType.PESSIMISTIC_WRITE);

// --- Возвращаемся к Транзакции 1 ---
p1.setPrice(p1.getPrice() - 10.0); // Списываем деньги со счета
em1.getTransaction().commit(); // Блокировка снимается

// --- Транзакция 2 "отвисает" ---
// Теперь p2 загружен, и Транзакция 2 может продолжить свою работу
// с уже обновленными данными.
```

Когда использовать:

- Когда вероятность конфликта **очень высока**, и цена ошибки велика (например, финансовые операции, управление складскими остатками).
- Когда "потерянное обновление" недопустимо, и лучше заставить пользователя подождать, чем выдать ошибку.
- Когда транзакции короткие. Длинные пессимистические блокировки могут серьезно снизить производительность всей системы и привести к дэдлокам (взаимным блокировкам).

1.10.24. Сравнительная таблица

Характеристика	Оптимистическая блокировка (@Version)	Пессимистическая блокировка (LockModeType)
Философия	Конфликты редки. Проверяем в конце.	Конфликты вероятны. Блокируем сразу.
Механизм	Столбец-версия в таблице. Проверка при UPDATE.	Блокировка строк на уровне базы данных (SELECT ... FOR UPDATE).
Реакция на конфликт	Бросает OptimisticLockException.	Вторая транзакция ждет завершения первой.
Производительность	Высокая. Нет реальных блокировок в БД.	Низкая. Может серьезно снизить параллелизм. Риск дэдлоков.
Масштабируемость	Хорошо масштабируется.	Плохо масштабируется.
Сценарий	Веб-приложения, CRUD-операции, "длинные" беседы с пользователем.	Финансовые расчеты, списание остатков, критические секции кода.
Лучший выбор для...	Большинства приложений.	Редких, но очень важных операций

1.10.25. Интеграция JPA с CDI и EJB

JPA не существует в вакууме. В Jakarta EE она тесно интегрирована с CDI (для управления зависимостями) и EJB (для управления транзакциями), создавая мощную и удобную среду для разработки слоя доступа к данным.

1. Интеграция с CDI (Contexts and Dependency Injection)

CDI — это "клей", который связывает компоненты приложения. Основная точка интеграции с JPA — это внедрение EntityManager.

Проблема: EntityManager не является потокобезопасным. Каждый поток (например, каждый HTTP-запрос) должен работать со своим собственным экземпляром EntityManager. Создавать его вручную — плохая идея.

Решение: CDI и EJB-контейнеры умеют управлять EntityManager и предоставлять его "контекстно-зависимым" образом.

Способ 1: Использование @PersistenceContext (традиционный, в EJB)

Аннотация @PersistenceContext — это стандартный способ запросить у контейнера EntityManager, привязанный к определенному persistence unit (описанному в persistence.xml).

```
import jakarta.ejb.Stateless;
import jakarta.persistence.EntityManager;
import jakarta.persistence.PersistenceContext;
```

@Stateless // Это EJB, который предоставляет транзакционные гарантии

```
public class ProductRepository {
```

```
    // Контейнер внедрит сюда прокси, который для каждой транзакции
```

```
    // будет предоставлять правильный EntityManager.
```

```
    @PersistenceContext(unitName = "my-app-pu")
```

```
    private EntityManager em;
```

```
    public Product findById(Long id) {
```

```
        return em.find(Product.class, id);
```

```
    }
```

```
// Транзакция управляется EJB-контейнером (@Transactional по
умолчанию)
public void save(Product product) {
    em.persist(product);
}
}
```

Способ 2: Использование CDI-продюсера (современный подход)

В чистой CDI-среде (без EJB) можно создать свой собственный продюсер (@Produces), который будет предоставлять EntityManager как обычный CDI-бин. Это делает код более гибким и менее зависимым от EJB.

1. Создаем продюсер:

```
import jakarta.enterprise.context.ApplicationScoped;
import jakarta.enterprise.context.RequestScoped;
import jakarta.enterprise.inject.Disposes;
import jakarta.enterprise.inject.Produces;
import jakarta.persistence.EntityManager;
import jakarta.persistence.EntityManagerFactory;
import jakarta.persistence.PersistenceUnit;
```

```
@ApplicationScoped
```

```
public class EntityManagerProducer {
```

```
    // Внедряем фабрику EntityManager'ов, которая является потокобезопасной
```

```
    @PersistenceUnit(unitName = "my-app-pu")
```

```
    private EntityManagerFactory emf;
```

```
    // Метод-продюсер. Будет создавать EntityManager для каждого запроса.
```

```
    @Produces
```

```
    @RequestScoped // Связываем жизнь EntityManager'a с HTTP-запросом
```

```
    public EntityManager createEntityManager() {
```

```
        return emf.createEntityManager();
```

```
    }
```

// Метод-диспоузер. Будет вызван в конце запроса для закрытия EntityManager'a.

```
public void closeEntityManager(@Disposes EntityManager em) {  
    if (em.isOpen()) {  
        em.close();  
    }  
}  
}
```

2. Используем @Inject для внедрения:

```
import jakarta.enterprise.context.ApplicationScoped;  
import jakarta.inject.Inject;  
// Нет зависимости от EJB!
```

```
@ApplicationScoped  
public class ProductRepositoryCDI {  
    @Inject  
    private EntityManager em;  
    // Теперь мы можем внедрять EntityManager как обычный CDI-бин  
    // ... те же методы findById, save ...  
    // ВАЖНО: для метода save нужно будет управлять транзакцией вручную  
    // или использовать перехватчик @Transactional из CDI.  
}
```

2. Интеграция с EJB (Enterprise JavaBeans)

Это самая мощная и распространенная связка в традиционных Jakarta EE приложениях. EJB предоставляет для JPA ключевой сервис — **декларативное управление транзакциями**.

Как это работает:

1. Вы создаете EJB-компонент (@Stateless, @Stateful).
2. Внедряете в него EntityManager через @PersistenceContext.
3. Помечаете бизнес-методы аннотациями управления транзакциями (или используете поведение по умолчанию).

Когда вы вызываете метод ЕJB, контейнер делает следующее:

- **Начинает транзакцию** в базе данных.
- **Связывает** с этой транзакцией EntityManager.
- Выполняет ваш код.
- Если метод завершился успешно, контейнер выполняет **commit** транзакции (и все изменения, сделанные через EntityManager, сохраняются в БД).
- Если в методе произошло системное исключение (RuntimeException), контейнер выполняет **rollback** транзакции (и все изменения отменяются).

Пример: Сервис для перевода денег

@Stateless

```
public class BankTransferService {  
    @PersistenceContext  
    private EntityManager em;  
    // По умолчанию  
    @TransactionAttribute(TransactionAttributeType.REQUIRED)  
    // Это значит, что метод должен выполняться в транзакции.  
    // Если ее нет, контейнер создаст новую.  
    public void transfer(Long fromAccountId, Long toAccountId, BigDecimal  
amount) {  
        // Начало транзакции (неявно, управляется EJB-контейнером)  
        Account fromAccount = em.find(Account.class, fromAccountId);  
        Account toAccount = em.find(Account.class, toAccountId);  
        if (fromAccount.getBalance().compareTo(amount) < 0) {  
            // Бросаем исключение. Контейнер перехватит его и  
            // АВТОМАТИЧЕСКИ сделает ROLLBACK транзакции.  
            throw new InsufficientFundsException("Not enough money on  
account " + fromAccountId);  
        }  
        fromAccount.setBalance(fromAccount.getBalance().subtract(amount));  
        toAccount.setBalance(toAccount.getBalance().add(amount));  
        // В этом коде нет ни em.persist(), ни em.merge(), потому что  
        // fromAccount и toAccount - это управляемые (managed) сущности.
```

```
// JPA сам отследит изменения в их полях.  
} // Конец метода. Если не было исключений, контейнер делает COMMIT.  
// JPA генерирует два SQL UPDATE запроса.  
}
```

Вывод: Связка JPA + EJB избавляет вас от ручного управления транзакциями, делая код намного чище и надежнее. Вы просто пишете бизнес-логику.

1.10.26. Популярные реализации JPA

JPA — это **спецификация**, то есть набор интерфейсов и правил. Чтобы она работала, нужна конкретная **реализация** — библиотека, которая реализует эти интерфейсы.

Два самых популярных JPA-провайдера:

1. Hibernate

- **Разработчик:** Red Hat.
- **История:** Hibernate появился до JPA и был настолько популярен, что сама спецификация JPA во многом основана на его идеях.
- **Статус:** **Де-факто стандарт** и самая популярная реализация JPA в мире. Hibernate используется по умолчанию в большинстве серверов приложений, включая WildFly/JBoss EAP и Spring Boot.
- **Особенности:**
 - **Зрелость и стабильность:** Существует очень давно, хорошо протестирован, имеет огромное сообщество и тонны документации.
 - **Расширенные возможности:** Предоставляет множество функций сверх стандарта JPA, таких как расширенная поддержка типов, гибкие генераторы ID, фильтры, аудит (через Hibernate Envers) и многое другое.
 - **HQL (Hibernate Query Language):** Собственный язык запросов, который является надмножеством JPQL и предлагает больше возможностей.
- **Когда использовать:** В большинстве случаев. Если вы не знаете, что выбрать, выбирайте Hibernate.

2. EclipseLink

- **Разработчик:** Eclipse Foundation (изначально Oracle).
- **История:** EclipseLink — это потомок проекта Oracle TopLink.
- **Статус:** Является эталонной реализацией (**Reference Implementation**) спецификации JPA. Это означает, что он является стандартом, с которым должны сверяться другие провайдеры для прохождения тестов на совместимость (TCK). Используется по умолчанию в серверах GlassFish и Payara.
- **Особенности:**
 - **Строгое соответствие стандарту:** Если что-то работает в EclipseLink, оно с высокой вероятностью будет работать и в других реализациях.
 - **Гибкое кэширование:** Имеет очень мощные и настраиваемые опции для кэширования.
 - **Поддержка не-SQL баз данных:** Имеет расширения для работы с NoSQL базами данных (JPA on NoSQL).

- **Когда использовать:** Если вы работаете с GlassFish/Payara, или если вам требуется строгое соответствие стандарту, или если вам нужны его уникальные расширенные возможности (например, продвинутое кэширование).

Как выбрать?

Для большинства приложений выбор между Hibernate и EclipseLink не будет иметь решающего значения, так как вы будете писать код с использованием стандартного JPA API. Переключиться с одного провайдера на другой обычно можно, изменив одну строку в persistence.xml.

Пример persistence.xml с указанием провайдера:

```
<persistence xmlns="https://jakarta.ee/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence
https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd"
version="3.0">

  <persistence-unit name="my-app-pu" transaction-type="JTA">
    <!-- Явное указание Hibernate как провайдера -->
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
    <!-- Или EclipseLink -->
    <!--
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider> -->
    <jta-data-source>
      java:jboss/datasources/MyWebAppDS
    </jta-data-source>
    <properties>
      <!-- Специфичные для Hibernate свойства -->
      <property name="hibernate.hbm2ddl.auto" value="update"/>
      <property name="hibernate.show_sql" value="true"/>
      <property name="hibernate.format_sql" value="true"/>
    </properties>
  </persistence-unit>
</persistence>
```

Обычно провайдер не указывается явно, и сервер приложений использует тот, который у него настроен по умолчанию (например, WildFly использует Hibernate).

1.10.27. Jakarta Transactions (JTA): Концепция транзакций. ACID

Концепция транзакций

Представьте себе операцию перевода денег с одного банковского счета на другой. Она состоит из двух шагов:

1. Снять деньги со счета А.
2. Зачислить деньги на счет Б.

Что произойдет, если после первого шага система выйдет из строя (например, отключится электричество)? Деньги со счета А списались, а на счет Б не поступили. Они просто "исчезли". Это катастрофа.

Транзакция — это последовательность операций, которая рассматривается как **единая, неделимая (атомарная) работа**. Она должна быть выполнена либо **полностью и успешно**, либо **не выполнена вовсе**, и система должна вернуться в состояние, в котором она была до начала транзакции.

- **Commit (Фиксация):** Если все операции внутри транзакции завершились успешно, транзакция фиксируется. Все изменения, сделанные в ее рамках, становятся постоянными и видимыми для других.
- **Rollback (Откат):** Если хотя бы одна операция внутри транзакции завершилась с ошибкой, вся транзакция откатывается. Все изменения, сделанные в ее рамках, отменяются, как будто ничего и не происходило.

В нашем примере с переводом денег, оба действия — списание и зачисление — должны быть объединены в одну транзакцию.

Свойства транзакций: ACID

Чтобы транзакции были надежными, они должны обладать четырьмя ключевыми свойствами, которые формируют акроним **ACID**:

1. Atomicity (Атомарность)

- **Что это:** "Все или ничего". Транзакция является неделимой единицей. Она не может быть выполнена частично.
- **Пример:** В переводе денег либо и списание, и зачисление пройдут успешно (commit), либо оба будут отменены (rollback). Не может быть ситуации, когда только списание прошло, а зачисление — нет.

2. Consistency (Согласованность)

- **Что это:** Транзакция переводит базу данных из одного **согласованного (корректного)** состояния в другое. Все бизнес-правила и ограничения целостности (например, NOT NULL, UNIQUE, внешние ключи, баланс не может быть отрицательным) должны соблюдаться как до, так и после транзакции.
- **Пример:** Если в нашей банковской системе есть правило, что общий баланс всех счетов должен оставаться неизменным, то успешная транзакция перевода денег не нарушит это правило. Сумма денег в системе останется той же, она просто переместится. Если бы деньги "исчезли", это было бы нарушением согласованности.

3. Isolation (Изолированность)

- **Что это:** Параллельно выполняющиеся транзакции не должны мешать друг другу. Для каждой транзакции должно создаваться впечатление, что она выполняется в системе одна, без других параллельных транзакций. Промежуточные, незафиксированные результаты одной транзакции не должны быть видны другим.
- **Пример:** Если транзакция А переводит деньги со счета X на счет Y, то другая транзакция Б, которая в это же время считает общую сумму на счетах X и Y, не должна видеть промежуточное состояние, когда деньги уже списаны с X, но еще не зачислены на Y. Она должна увидеть либо состояние до перевода, либо состояние после.

4. Durability (Долговечность)

- **Что это:** Если транзакция была успешно зафиксирована (commit), ее результаты должны быть **постоянными** и сохраниться даже в случае системного сбоя (перезагрузка сервера, отказ оборудования).
- **Пример:** Если система подтвердила, что перевод денег выполнен, то эти деньги должны остаться на счете Б, даже если сразу после этого сервер выключится из розетки. Обычно это гарантируется записью изменений в журнал транзакций на диск перед фактическим коммитом.

Jakarta Transactions (JTA) в контексте Jakarta EE

JTA — это стандартная Java API спецификация, которая определяет, как приложения должны управлять транзакциями. JTA предоставляет высокоуровневый, не зависящий от конкретной реализации (СУБД, брокера сообщений) способ работы с транзакциями.

Ключевые роли JTA:

- **Декларативное управление транзакциями:** Это самый частый способ использования JTA. Вы не пишете код для `begin()`, `commit()`, `rollback()`. Вместо этого вы просто аннотируете свои бизнес-методы (обычно в EJB), и контейнер делает всю работу за вас.
- **Управление распределенными транзакциями:** Это "суперсила" JTA. Она позволяет координировать транзакции, которые затрагивают **несколько ресурсов**, например, две разные базы данных или базу данных и брокер сообщений. Это делается с помощью двухфазного протокола фиксации (Two-Phase Commit, 2PC).

Примеры кода

Это самый простой и рекомендуемый способ. Вся работу делает контейнер.

```
import jakarta.ejb.Stateless;
import jakarta.ejb.TransactionAttribute;
import jakarta.ejb.TransactionAttributeType;
import jakarta.persistence.EntityManager;
import jakarta.persistence.PersistenceContext;
```

```
@Stateless
```

```
public class OrderProcessingService {
```

```
    @PersistenceContext
```

```
    private EntityManager em;
```

```
    // Этот метод будет обернут в транзакцию контейнером.
```

```
    // Атомарность, Согласованность и Долговечность гарантируются.
```

```
    // Изолированность обеспечивается базой данных на уровне по умолчанию.
```

```
    @TransactionAttribute(TransactionAttributeType.REQUIRED)
```

```
    // REQUIRED - поведение по умолчанию
```

```

public void placeOrder(Order order, StockItem stockItem) {
    // --- НАЧАЛО ТРАНЗАЦИИ (неявно) ---
    // Шаг 1: Сохраняем заказ
    em.persist(order);
    // Шаг 2: Обновляем остаток на складе
    stockItem.setQuantity(stockItem.getQuantity() - order.getQuantity());
    em.merge(stockItem);
    // Симуляция ошибки
    if (order.getQuantity() > 100) {
        // Это RuntimeException. Контейнер его перехватит
        // и автоматически вызовет ROLLBACK.
        throw new IllegalStateException("Cannot order more than 100
items at once.");
    }

    // --- КОНЕЦ ТРАНЗАЦИИ (неявно) ---
    // Если исключения не было, контейнер вызовет COMMIT.
    // И заказ, и остаток на складе будут сохранены.
    // Если было исключение, будет ROLLBACK, и ни одно из изменений
не сохранится.
}
}

```

Здесь все свойства ACID обеспечиваются автоматически:

- **A:** `persist(order)` и `merge(stockItem)` либо оба пройдут, либо оба откатятся.
- **C:** Если в БД есть ограничение, что остаток на складе не может быть отрицательным, и мы попытаемся его нарушить, БД вернет ошибку, контейнер сделает `rollback`, и база останется в согласованном состоянии.
- **I:** Другие транзакции не увидят наполовину созданный заказ.
- **D:** После успешного `commit` данные сохраняются даже при сбое.

Иногда требуется более тонкий контроль над границами транзакции. В этом случае можно управлять ею вручную. Это используется редко.

```

import jakarta.ejb.Stateless;
import jakarta.ejb.TransactionManagement;

```

```

import jakarta.ejb.TransactionManagementType;
import jakarta.inject.Inject;
import jakarta.transaction.UserTransaction;

@Stateless
@TransactionManagement(TransactionManagementType.BEAN) // Говорим
контейнеру: "я сам буду управлять транзакциями"
public class ManualTransactionService {
    @Inject
    private UserTransaction utx; // Внедряем объект для управления транзакцией

    @PersistenceContext
    private EntityManager em;

    public void performComplexOperation() {
        try {
            utx.begin(); // Явно начинаем транзакцию
            // ... выполнение нескольких шагов ...
            em.persist(new Log("Step 1 complete"));
            // ... еще логика ...
            if (someCondition) {
                // Если все хорошо, фиксируем
                utx.commit();
            } else {
                // Если что-то пошло не так, откатываем
                utx.rollback();
            }
        } catch (Exception e) {
            try {
                // Всегда пытаемся откатить транзакцию в блоке catch
                utx.rollback();
            }
        }
    }
}

```

```

    } catch (Exception rollbackEx) {
        // Логируем ошибку отката
    }
    throw new RuntimeException("Operation failed", e);
}
}
}

```

Как видите, ручное управление гораздо сложнее и более подвержено ошибкам. В 99% случаев следует предпочитать декларативный подход (CMT).

Вывод:

Концепция транзакций и гарантии ACID являются основой для построения надежных систем. Jakarta Transactions (JTA) предоставляет стандартный механизм в Jakarta EE для работы с ними, а связка с EJB делает использование транзакций декларативным, простым и мощным.

1.10.28. Управление транзакциями в Jakarta EE

Jakarta EE предоставляет два подхода к управлению транзакциями, реализуемых через спецификацию **JTA (Jakarta Transactions)**:

1. **CMT (Container-Managed Transactions)** — Управление транзакциями контейнером.
2. **BMТ (Bean-Managed Transactions)** — Управление транзакциями вручную компонентом (бином).

1.10.29. Container-Managed Transactions (CMT) — Управление контейнером

Это предпочтительный, самый простой и самый распространенный способ.

Идея: Вы не пишете код для начала, фиксации или отката транзакций. Вы **декларативно** (с помощью аннотаций) описываете транзакционное поведение ваших методов, а EJB-контейнер или CDI-контейнер (через перехватчик) делает всю работу за вас.

Использование в EJB

В EJB-компонентах (`@Stateless`, `@Stateful`, `@Singleton`) управление транзакциями включено по умолчанию.

```
import jakarta.ejb.Stateless;
import jakarta.persistence.EntityManager;
import jakarta.persistence.PersistenceContext;
```

```
@Stateless // Говорим, что это EJB
```

```
public class AccountService {
```

```
    @PersistenceContext
```

```
    private EntityManager em;
```

```
    // Контейнер автоматически обернет этот метод в транзакцию
```

```
    public void deposit(Long accountId, BigDecimal amount) {
```

```
        // Контейнер начинает транзакцию (неявно)
```

```
        Account account = em.find(Account.class, accountId);
```

```
        account.setBalance(account.getBalance().add(amount));
```

```

        // Контейнер фиксирует транзакцию (неявно)
    }
}

```

Использование в CDI бинах с @Transactional

Начиная с JTA 1.2 (часть Java EE 7), аннотацию @Transactional можно использовать и на обычных CDI-бинах. Это позволяет получить декларативное управление транзакциями без необходимости использовать EJB.

```

import jakarta.enterprise.context.ApplicationScoped;
import jakarta.inject.Inject;
import jakarta.persistence.EntityManager;
import jakarta.transaction.Transactional; // Важно: импорт из jakarta.transaction

```

```

@Transactional // Обычный CDI бин
public class AccountServiceCDI {
    @Inject
    private EntityManager em; // EntityManager должен быть предоставлен
    (например, через продюсер)

```

@Transactional // Явно просим контейнер управлять транзакцией для этого метода

```

    public void deposit(Long accountId, BigDecimal amount) {
        // Логика та же, что и в EJB примере
        Account account = em.find(Account.class, accountId);
        account.setBalance(account.getBalance().add(amount));
    }
}

```

Важно: Чтобы @Transactional работал на CDI-бинах, в вашем beans.xml должен быть активирован перехватчик транзакций, или сервер приложений должен делать это по умолчанию.

Атрибуты транзакций

С помощью атрибутов аннотации @Transactional (или @TransactionAttribute для EJB) можно тонко настроить поведение.

@Transactional(TransactionAttributeType.XXX)

- **REQUIRED** (по умолчанию):
 - Если транзакция уже существует, метод выполняется в ее контексте.
 - Если транзакции нет, **контейнер создает новую**.
 - *Сценарий*: Большинство бизнес-методов.
- **REQUIRES_NEW**:
 - Контейнер **всегда создает новую, независимую транзакцию**.
 - Если уже была активная транзакция, она приостанавливается на время выполнения метода.
 - *Сценарий*: Запись в лог или аудита, которая должна быть сохранена, даже если основная транзакция откатится.

@Stateless

```
public class AuditService {  
    @Transactional(Transaction.TxType.REQUIRES_NEW)  
    public void logAction(String message) {  
        // Этот метод выполнится в своей собственной транзакции  
    }  
}
```

- **SUPPORTS**:
 - Если транзакция существует, метод выполняется в ней.
 - Если транзакции нет, метод выполняется **без транзакции**.
 - *Сценарий*: Методы, которые только читают данные и не требуют транзакционных гарантий, но могут быть вызваны из транзакционного контекста.
- **NOT_SUPPORTED**:
 - Метод **всегда выполняется без транзакции**.
 - Если активная транзакция существует, она приостанавливается.
 - *Сценарий*: Вызов не-транзакционных систем, обращение к внешним API.
- **MANDATORY**:
 - Метод **обязан** быть вызван в контексте существующей транзакции.
 - Если транзакции нет, контейнер бросит исключение.
 - *Сценарий*: Внутренние методы-хелперы, которые не имеют смысла без транзакции.
- **NEVER**:
 - Метод **не должен** быть вызван в контексте транзакции.
 - Если транзакция существует, будет брошено исключение.

1.10.30. Bean-Managed Transactions (BMT) — Ручное управление

Этот подход используется **редко** и только тогда, когда требуется полный, гранулярный контроль над границами транзакций.

Идея: Вы вручную внедряете объект `UserTransaction` и вызываете его методы `begin()`, `commit()` и `rollback()`.

Как использовать:

1. Пометить бин (обычно EJB) аннотацией `@TransactionManagement(TransactionManagementType.BEAN)`.
2. Внедрить `UserTransaction` через `@Inject` или `@Resource`.

```
import jakarta.ejb.Stateless;
import jakarta.ejb.TransactionManagement;
import jakarta.ejb.TransactionManagementType;
import jakarta.inject.Inject;
import jakarta.persistence.EntityManager;
import jakarta.persistence.PersistenceContext;
import jakarta.transaction.UserTransaction;
```

```
@Stateless
```

```
@TransactionManagement(TransactionManagementType.BEAN) // Указываем,
что управляем транзакциями сами
```

```
public class BatchProcessingService {
```

```
    @PersistenceContext
```

```
    private EntityManager em;
```

```
    @Inject
```

```
    private UserTransaction utx; // Внедряем интерфейс для управления
транзакцией
```

```
    public void processBatch(List<Item> items) throws Exception {
```

```
        for (Item item : items) {
```

```
            try {
```

```

        utx.begin(); // Начинаем новую транзакцию для каждого
        элемента
        processItem(item);
        utx.commit(); // Фиксируем, если обработка успешна
    } catch (Exception e) {
        // Если ошибка, откатываем только транзакцию для этого
        элемента
        utx.rollback();
        System.err.println("Failed to process item " + item.getId() +
            ". Continuing with next.");
        // Не пробрасываем исключение дальше, чтобы
        продолжить цикл
    }
}

private void processItem(Item item) {
    // ... сложная логика обработки ...
    em.merge(item);
}
}

```

Когда использовать ВМТ?

- Для обработки больших пакетов данных, где вы хотите иметь отдельную транзакцию на каждую запись.
- Когда одна бизнес-операция требует нескольких отдельных, независимых транзакций.
- Когда нужно управлять временем жизни транзакции, выходящим за рамки одного метода.

1.10.31. Двухфазный коммит (2PC) и Распределенные транзакции

Это самая мощная возможность JTA, которая позволяет поддерживать **атомарность (ACID)** при работе с **несколькими транзакционными ресурсами**.

Ресурсы: База данных, брокер сообщений (JMS), коннектор к ERP-системе и т.д.

Проблема:

Представим, что нужно выполнить операцию:

1. Сохранить заказ в базе данных Oracle.
2. Отправить сообщение об этом заказе в очередь JMS (например, ActiveMQ).

Что если после сохранения в БД сервер упадет до отправки сообщения? Заказ есть, а уведомления о нем нет. Система в несогласованном состоянии.

Решение: Распределенная транзакция под управлением JTA. Менеджер транзакций JTA (который является частью сервера приложений) координирует все участвующие ресурсы с помощью **двухфазного протокола фиксации (Two-Phase Commit, 2PC)**.

Как работает 2PC (упрощенно):

- **Фаза 1: Голосование (Prepare)**

1. Менеджер транзакций говорит всем участникам (БД Oracle, брокеру JMS): "Приготовьтесь к фиксации. Можете ли вы гарантировать, что сможете закоммитить свою часть работы?".
2. Каждый участник выполняет свою работу, но **не фиксирует ее окончательно**. Он записывает все изменения во внутренние журналы и отвечает менеджеру: "Да, я готов" (VOTE_COMMIT) или "Нет, у меня ошибка" (VOTE_ROLLBACK).

- **Фаза 2: Решение (Commit/Rollback)**

1. Если **ВСЕ участники ответили "Да"**: Менеджер транзакций рассылает всем команду: "Фиксируйте!". Каждый участник применяет свои изменения и делает их постоянными.
2. Если **хотя бы ОДИН участник ответил "Нет" (или не ответил)**: Менеджер транзакций рассылает всем команду: "Откатывайте!". Все участники отменяют свои изменения.

Главная гарантия: Все участники либо вместе успешно закоммитят свои изменения, либо вместе их откатят.

Пример кода (концептуальный):

С точки зрения разработчика, код **не меняется!** Вы просто используете разные ресурсы в рамках одной CMT-транзакции.

@Stateless

```
public class DistributedOrderService {
```

```
    @PersistenceContext(unitName = "OracleDS") // EntityManager для Oracle
```

```
    private EntityManager em;
```

`@Resource(lookup = "java:/jms/ConnectionFactory") // Фабрика соединений
для JMS`

`private ConnectionFactory connectionFactory;`

`@Resource(lookup = "java:/jms/queue/OrderQueue") // Очередь JMS`

`private Queue orderQueue;`

`@Transactional // JTA обернет все это в одну распределенную транзакцию`

`public void placeOrderAndNotify(Order order) throws JMSException {`

`// --- НАЧАЛО РАСПРЕДЕЛЕННОЙ ТРАНЗАКЦИИ (неявно) ---`

`// Шаг 1: Работа с первым ресурсом (База данных)`

`em.persist(order);`

`// Шаг 2: Работа со вторым ресурсом (Брокер сообщений)`

`try (Connection connection = connectionFactory.createConnection();`

`Session session = connection.createSession();`

`MessageProducer producer = session.createProducer(orderQueue)) {`

`TextMessage message = session.createTextMessage("New order: "
+ order.getId());`

`producer.send(message);`

`}`

`// --- КОНЕЦ РАСПРЕДЕЛЕННОЙ ТРАНЗАКЦИИ (неявно) ---`

`// JTA-менеджер проведет 2PC для Oracle и JMS.`

`// Либо и заказ сохранится, и сообщение отправится.`

`// Либо и то, и другое будет отменено.`

`}`

`}`

Требования:

- Сервер приложений должен быть настроен для работы с JTA.

- Все используемые ресурсы (драйверы баз данных, JMS-провайдеры) должны быть **XA-совместимыми**. XA — это стандарт, который описывает, как ресурс может участвовать в распределенных транзакциях. Большинство корпоративных драйверов (Oracle, PostgreSQL, SQL Server) имеют XA-версии.

1.11. Интеграция и обмен сообщениями

1.11.1. Интеграция и обмен сообщениями: Jakarta Messaging (JMS)

Основы асинхронного обмена сообщениями

В традиционной **синхронной** модели взаимодействия (например, вызов REST API или EJB-метода) клиент отправляет запрос и **блокируется**, ожидая ответа от сервера.

Проблемы синхронного подхода:

- **Сильная связанность:** Клиент и сервер должны быть доступны одновременно. Если сервер не работает, клиент получает ошибку.
- **Низкая отказоустойчивость:** Сбой сервера напрямую влияет на клиента.
- **Плохая масштабируемость:** Если сервер медленно обрабатывает запросы, клиенты простаивают. В пиковые нагрузки сервер может "захлебнуться" от количества одновременных запросов.

Асинхронная модель решает эти проблемы, вводя промежуточное звено — **брокер сообщений (Message Broker)**.

Как это работает:

1. **Отправитель (Producer)** создает **сообщение** (кусочек данных) и отправляет его брокеру сообщений. Отправив сообщение, он **не ждет** ответа и может сразу продолжать свою работу.
2. **Брокер сообщений** — это специальное ПО (например, ActiveMQ, RabbitMQ, Kafka), которое надежно хранит сообщение в **пункте назначения (Destination)**.
3. **Получатель (Consumer)** подключается к брокеру и забирает сообщения из пункта назначения для обработки. Он может делать это в своем собственном темпе.

Преимущества асинхронного подхода:

- **Слабая связанность (Decoupling):** Отправитель и получатель не знают друг о друге. Они знают только о брокере и формате сообщений. Их можно разрабатывать, развертывать и масштабировать независимо.
- **Отказоустойчивость:** Если получатель временно недоступен, сообщения накапливаются в брокере. Когда получатель снова заработает, он обработает все накопившиеся сообщения. Никакие данные не теряются.
- **Масштабируемость:** Можно легко добавить больше получателей для обработки сообщений из одной очереди, тем самым распределяя нагрузку.

- **Сглаживание пиковых нагрузок:** Если отправители генерируют сообщения быстрее, чем получатели могут их обработать, брокер выступает в роли буфера, сглаживая пики.

Брокеры сообщений (Message Brokers)

Брокер сообщений — это сердце асинхронной системы. Jakarta EE не предоставляет сам брокер, но определяет стандартный API (JMS) для работы с ним.

Популярные брокеры, поддерживающие JMS:

- **Apache ActiveMQ:** Классический, очень популярный и полнофункциональный JMS-брокер. Часто встраивается в серверы приложений, такие как TomEE.
- **Artemis:** Следующее поколение ActiveMQ, также от Apache.
- **IBM MQ:** Мощный коммерческий брокер от IBM.
- **RabbitMQ:** Очень популярный брокер, реализующий протокол AMQP, но имеющий JMS-клиент для взаимодействия.

JMS позволяет вашему приложению работать с любым из этих брокеров, не меняя код, а лишь изменив конфигурацию подключения.

Ключевые концепции Jakarta Messaging (JMS)

JMS API определяет несколько основных интерфейсов и две модели обмена сообщениями.

Основные объекты JMS API:

- **ConnectionFactory:** Фабрика для создания подключений к брокеру. Конфигурируется на сервере приложений.
- **Connection:** Подключение к брокеру.
- **Session:** Однопоточный контекст для отправки и получения сообщений.
- **Destination:** Пункт назначения, куда отправляются и откуда читаются сообщения. Бывает двух типов: Queue и Topic.
- **MessageProducer:** Объект для отправки сообщений в Destination.
- **MessageConsumer:** Объект для получения сообщений из Destination.
- **Message:** Само сообщение. Может быть разных типов (TextMessage, ObjectMessage, BytesMessage и т.д.).

Модели обмена сообщениями в JMS

- **Концепция:** "Один-к-одному". Сообщение, отправленное в очередь, может быть получено и обработано **только одним** получателем.
- **Сценарий:** Обработка заказов, выполнение задач. Каждое задание (сообщение) должно быть выполнено ровно один раз.
- **Поведение:** Если к одной очереди подключено несколько получателей, они будут "соревноваться" за сообщения. Брокер будет распределять сообщения между ними для параллельной обработки (балансировка нагрузки).

![alt text](https://i.imgur.com/GzQ5E6T.png)

- **Концепция:** "Один-ко-многим". Сообщение, опубликованное в топик, доставляется **каждому** активному подписчику (consumer).
- **Сценарий:** Рассылка уведомлений, оповещение о событиях. Новость о изменении цены на товар должна быть доставлена всем системам, которые на нее подписаны (системе аналитики, системе уведомления клиентов и т.д.).
- **Поведение:** Если на момент публикации сообщения у топика нет подписчиков, сообщение просто исчезает (если не используются "долговечные подписки" - durable subscriptions).

![alt text](https://i.imgur.com/5VlB6Qp.png)

Примеры кода

Важно: Для работы этих примеров на сервере приложений (например, WildFly) должны быть настроены `ConnectionFactory` и `Destination` (очередь или топик). Это делается через административную консоль или файлы конфигурации.

```
import jakarta.annotation.Resource;  
import jakarta.ejb.Stateless;  
import jakarta.jms.*;
```

```
@Stateless
```

```
public class MessageSenderService {
```

```
    // 1. Внедряем ресурсы, настроенные на сервере приложений
```

```
    @Resource(lookup = "java:/JmsXA") // Стандартное JNDI-имя для  
    ConnectionFactory в WildFly
```

```
    private ConnectionFactory connectionFactory;
```

`@Resource(lookup = "java:/jms/queue/MyOrderQueue")` // JNDI-имя для нашей очереди

```
private Queue myQueue;

public void sendOrderMessage(String orderDetails) {
    // 2. Используем try-with-resources для автоматического закрытия
    // соединений
    try (JMSContext context =
        connectionFactory.createContext(JMSContext.AUTO_ACKNOWLEDGE)) {
        // 3. Создаем текстовое сообщение и отправляем его
        // JMSContext объединяет в себе Connection и Session, что упрощает
        код
        context.createProducer().send(myQueue, orderDetails);
        System.out.println("Message sent to queue: " + orderDetails);
    } catch (JMSRuntimeException e) {
        // Обработка ошибок
        e.printStackTrace();
    }
}
```

- JMSContext — это современный, упрощенный API (появился в JMS 2.0), который значительно сокращает шаблонный код.
- lookup: JNDI-имя, по которому приложение находит сконфигурированный на сервере ресурс.

Message-Driven Bean (MDB) — это специальный тип EJB, который является асинхронным слушателем. Контейнер автоматически доставляет ему сообщения из указанного Destination. Это **самый простой и мощный** способ получать сообщения в Jakarta EE.

```
import jakarta.ejb.ActivationConfigProperty;
import jakarta.ejb.MessageDriven;
import jakarta.jms.JMSEException;
import jakarta.jms.Message;
import jakarta.jms.MessageListener;
import jakarta.jms.TextMessage;

// 1. Помечаем класс как MDB
@MessageDriven(
    activationConfig = {
        // 2. Указываем, откуда слушать сообщения
        @ActivationConfigProperty(propertyName = "destinationLookup",
            propertyValue = "java:/jms/queue/MyOrderQueue"),
        @ActivationConfigProperty(propertyName = "destinationType",
            propertyValue = "jakarta.jms.Queue")
    }
)
public class OrderProcessorMDB implements MessageListener {
    // 3. Реализуем интерфейс MessageListener
    // 4. Этот метод будет автоматически вызван контейнером для каждого
    // нового сообщения
    @Override
    public void onMessage(Message message) {
        try {
            if (message instanceof TextMessage) {
                TextMessage textMessage = (TextMessage) message;
                String orderDetails = textMessage.getText();
                System.out.println("Received message: " + orderDetails);
            }
        }
    }
}
```

```

        // ... здесь происходит сложная и долгая обработка заказа
        ...
    } else {
        System.out.println("Received a message of an unknown
        type.");
    }
} catch (JMSException e) {
    e.printStackTrace();
}
}
}

```

Что делает контейнер:

- Создает пул экземпляров OrderProcessorMDB.
- Устанавливает соединение с брокером и слушает очередь MyOrderQueue.
- Когда приходит сообщение, он берет свободный экземпляр MDB из пула и вызывает его метод onMessage, передавая туда сообщение.
- Управляет транзакциями. Если метод onMessage завершается с ошибкой, контейнер может настроить повторную доставку сообщения.

Этот подход полностью избавляет вас от необходимости вручную управлять подключениями, сессиями и потоками для прослушивания сообщений.

1.11.2. Модели обмена сообщениями в Jakarta Messaging (JMS)

JMS определяет два основных способа (модели) взаимодействия между отправителями и получателями:

4. **Point-to-Point (Точка-Точка)**, реализуемая через **Очереди (Queues)**.
5. **Publish/Subscribe (Публикация/Подписка)**, реализуемая через **Топики (Topics)**.

Выбор модели зависит от бизнес-требований и определяет, как сообщения будут доставляться и потребляться.

1.11.3. Модель Point-to-Point (РТР) с использованием Очерей (Queues)

Концепция: "Одно сообщение — один получатель".

В этой модели сообщение отправляется в специальный пункт назначения, называемый **очередью (Queue)**. Брокер хранит это сообщение до тех пор, пока какой-либо получатель его не заберет. Каждое сообщение может быть успешно обработано **только одним** получателем.

Ключевые характеристики:

- **Один потребитель:** Каждое сообщение имеет только одного потребителя.
- **Асинхронность по времени:** Отправитель и получатель не обязаны быть активны одновременно. Если получатель не работает, сообщения будут накапливаться в очереди.
- **Балансировка нагрузки:** Если к одной и той же очереди подключено несколько получателей, они будут "соревноваться" за сообщения. Брокер будет распределять сообщения между ними, что позволяет легко масштабировать обработку, просто добавляя больше экземпляров получателей.
- **Гарантированная доставка:** Сообщение остается в очереди, пока не будет успешно обработано и подтверждено (acknowledged) получателем.

Аналогия из жизни: Касса в супермаркете. Каждый покупатель (сообщение) встает в очередь и обслуживается ровно одной свободной кассой (получателем). Если открывается новая касса, она начинает обслуживать покупателей из той же очереди.

Сценарии использования:

- **Обработка заказов:** Каждый заказ должен быть обработан ровно один раз.
- **Выполнение фоновых задач:** Генерация отчетов, обработка загруженных файлов.
- **Транзакционные операции:** Когда необходимо гарантировать, что задача будет выполнена, даже если система-обработчик временно недоступна.

Пример кода

```
import jakarta.annotation.Resource;  
import jakarta.ejb.Stateless;  
import jakarta.jms.ConnectionFactory;  
import jakarta.jms.JMSContext;  
import jakarta.jms.Queue; // Используем интерфейс Queue
```

```

@Stateless
public class TaskProducer {
    @Resource(lookup = "java:/JmsXA")
    private ConnectionFactory connectionFactory;

    // Внедряем ресурс типа Queue
    @Resource(lookup = "java:/jms/queue/TaskQueue")
    private Queue taskQueue;

    public void submitTask(String taskDescription) {
        try (JMSContext context = connectionFactory.createContext()) {
            // Отправляем сообщение в конкретную очередь
            context.createProducer().send(taskQueue, "New task: " +
            taskDescription);
            System.out.println("Task submitted to the queue.");
        }
    }
}

```



```

import jakarta.ejb.ActivationConfigProperty;
import jakarta.ejb.MessageDriven;
import jakarta.jms.Message;
import jakarta.jms.MessageListener;
import jakarta.jms.TextMessage;

@MessageDriven(activationConfig = {
    // Указываем JNDI-имя нашей очереди
    @ActivationConfigProperty(propertyName = "destinationLookup",
propertyValue = "java:/jms/queue/TaskQueue"),
    // Явно указываем тип назначения
    @ActivationConfigProperty(propertyName = "destinationType", propertyValue
= "jakarta.jms.Queue")
})
public class TaskConsumerMDB implements MessageListener {
    @Override
    public void onMessage(Message message) {
        try {
            String task = ((TextMessage) message).getText();
            System.out.println("Processing task: " + task);
            // ... здесь происходит обработка задачи ...
            Thread.sleep(2000); // Имитация долгой работы
            System.out.println("Task completed: " + task);
        } catch (Exception e) {
            // Обработка ошибок
        }
    }
}

```

Если вы развернете несколько экземпляров этого MDB (например, в кластере), они будут параллельно разбирать задачи из TaskQueue.

1.11.4. Модель Publish/Subscribe (Pub/Sub) с использованием Топиков (Topics)

Концепция: "Одно сообщение — много получателей".

В этой модели сообщение **публикуется** в специальный пункт назначения, называемый **топиком (Topic)**. Брокер доставляет копию этого сообщения **каждому** активному клиенту, который на этот топик **подписан**.

Ключевые характеристики:

- **Множество потребителей:** Одно и то же сообщение получают все подписчики.
- **Временная зависимость:** По умолчанию, если подписчик не активен в момент публикации сообщения, он это сообщение **не получит**. (Эту проблему решают "долговечные подписки" - durable subscriptions).
- **Тематическое вещание:** Отлично подходит для распространения событий и уведомлений.

Аналогия из жизни: Подписка на журнал или газету. Издатель (producer) выпускает новый номер (сообщение), и каждый подписчик (consumer) получает свою собственную копию. Если вы не были подписаны в момент выхода номера, вы его не получите.

Сценарии использования:

- **Рассылка новостей или котировок акций:** Все заинтересованные клиенты должны получить информацию одновременно.
- **Оповещение о системных событиях:** Например, событие "пользователь зарегистрировался" может быть интересно нескольким системам: для отправки приветственного email, для аналитики, для службы безопасности.
- **Очистка кэша:** Сообщение в топик "cache-clear" заставит все узлы кластера сбросить свои локальные кэши.

Пример кода

```
import jakarta.annotation.Resource;
import jakarta.ejb.Stateless;
import jakarta.jms.ConnectionFactory;
import jakarta.jms.JMSContext;
import jakarta.jms.Topic; // Используем интерфейс Topic

@Stateless
public class NewsPublisher {
    @Resource(lookup = "java:/JmsXA")
    private ConnectionFactory connectionFactory;

    // Внедряем ресурс типа Topic
    @Resource(lookup = "java:/jms/topic/NewsTopic")
    private Topic newsTopic;

    public void publishNews(String newsHeadline) {
        try (JMSContext context = connectionFactory.createContext()) {
            // Публикуем сообщение в топик
            context.createProducer().send(newsTopic, "BREAKING NEWS: "
+ newsHeadline);
            System.out.println("News published to the topic.");
        }
    }
}
```

```

// Подписчик 1: Аналитическая система
@MessageDriven(activationConfig = {
    @ActivationConfigProperty(propertyName = "destinationLookup",
propertyValue = "java:/jms/topic/NewsTopic"),
    @ActivationConfigProperty(propertyName = "destinationType", propertyValue
= "jakarta.jms.Topic")
})

public class AnalyticsServiceMDB implements MessageListener {
    @Override
    public void onMessage(Message message) {
        // ... логика аналитики ...
        System.out.println("ANALYTICS: Received news update.");
    }
}

// Подписчик 2: Система рассылки уведомлений
@MessageDriven(activationConfig = {
    @ActivationConfigProperty(propertyName = "destinationLookup",
propertyValue = "java:/jms/topic/NewsTopic"),
    @ActivationConfigProperty(propertyName = "destinationType", propertyValue
= "jakarta.jms.Topic")
})

public class NotificationServiceMDB implements MessageListener {
    @Override
    public void onMessage(Message message) {
        // ... логика отправки push-уведомлений ...
        System.out.println("NOTIFICATIONS: Received news update. Sending
alerts.");
    }
}

```

Когда NewsPublisher опубликует одну новость, **оба** MDB (AnalyticsServiceMDB и NotificationServiceMDB) получают это сообщение и выполняют свою логику.

1.11.5. Сводная таблица различий

Характеристика	Point-to-Point (Queue)	Publish/Subscribe (Topic)
Основная цель	Обработка задач, гарантированное выполнение.	Распространение событий, вещание.
Доставка	Одно сообщение доставляется одному получателю.	Одно сообщение доставляется всем подписчикам.
Связь	1-к-1	1-ко-многим
Зависимость по времени	Получатель может быть офлайн, сообщения накопятся.	По умолчанию получатель должен быть онлайн (активным подписчиком).
Основной сценарий	"Сделай это для меня" (команда).	"Просто к сведению" (событие).
Интерфейс в JMS	jakarta.jms.Queue	jakarta.jms.Topic

1.11.6. Основные компоненты JMS API

Представьте себе процесс отправки почтового письма. Чтобы отправить письмо, вам нужно:

1. Знать адрес почтового отделения (**ConnectionFactory**).
2. Прийти в это отделение и установить с ним "контакт" (**Connection**).
3. Взять бланк и ручку для написания писем (**Session**).
4. Знать адрес получателя (**Destination**).
5. Написать письмо и опустить его в почтовый ящик (**MessageProducer**).
6. Почтальон, который забирает письма по этому адресу (**MessageConsumer**).
7. Сам человек, который читает доставленное письмо (**MessageListener**).

JMS API построен по очень похожей иерархической модели.

ConnectionFactory

- **Что это:** Объект, который инкапсулирует всю информацию, необходимую для подключения к конкретному JMS-брокеру. Он знает сетевой адрес брокера, порт, протокол и другие параметры.
- **Роль:** Точка входа в JMS. Это "фабрика" для создания подключений.
- **Как получается:** **Никогда не создается вручную** (`new ...`). `ConnectionFactory` конфигурируется администратором на сервере приложений (`WildFly`, `Payara` и т.д.) и получается в приложении через **внедрение зависимостей** (`@Resource`). Это позволяет менять брокер сообщений, не меняя код приложения.

```
import jakarta.annotation.Resource;
import jakarta.ejb.Stateless;
import jakarta.jms.ConnectionFactory;
```

```
@Stateless
```

```
public class MyService {
    // Внедряем ConnectionFactory, которую администратор
    // настроил на сервере и "привязал" к JNDI-имени "java:/JmsXA".
    @Resource(lookup = "java:/JmsXA")
    private ConnectionFactory connectionFactory;

    // ...
}
```

Connection

- **Что это:** Активное соединение с JMS-брокером. Представляет собой физическое сетевое подключение (например, TCP/IP сокет).
- **Роль:** Управляет соединением. Создание Connection — это относительно "дорогая" операция, поэтому их часто пулят (сервер приложений делает это автоматически).
- **Как получается:** Создается из ConnectionFactory вызовом `connectionFactory.createConnection()`.
- **Важно:** Один Connection может использоваться для создания нескольких Session.

Session

- **Что это:** Однопоточный контекст для отправки и получения сообщений. Это "рабочая лошадка" JMS.
- **Роль:** Предоставляет транзакционные границы и является фабрикой для MessageProducer, MessageConsumer и самих Message.
- **Как получается:** Создается из Connection вызовом `connection.createSession()`.
- **Важно:** Объект Session **не является потокобезопасным**. Нельзя использовать один и тот же экземпляр Session в нескольких потоках одновременно.

Упрощение с JMSContext (начиная с JMS 2.0)

В JMS 2.0 был введен интерфейс JMSContext, который объединяет в себе Connection и Session, значительно упрощая API и сокращая шаблонный код. Он создается прямо из ConnectionFactory и автоматически управляет ресурсами. **В современном коде следует предпочитать JMSContext.**

Пример (старый API vs новый API):

// --- Старый, многословный способ (до JMS 2.0) ---

```
public void sendWithOldApi(String text) {
    Connection connection = null;
    Session session = null;
    try {
        connection = connectionFactory.createConnection();
        // В сессии указывается, транзакционная ли она и какой режим
        // подтверждения
        session = connection.createSession(false,
        Session.AUTO_ACKNOWLEDGE);
        MessageProducer producer = session.createProducer(myQueue);
        TextMessage message = session.createTextMessage(text);
        producer.send(message);
    } catch (JMSEException e) {
        // ... обработка ошибки ...
    } finally {
        // Ручное закрытие ресурсов в правильном порядке
        if (session != null)
            try {
                session.close();
            }
            catch (JMSEException e) { }
        if (connection != null)
            try {
                connection.close();
            }
            catch (JMSEException e) { }
    }
}
```



```
// --- Новый, лаконичный способ (JMS 2.0+) ---
public void sendWithNewApi(String text) {
    // try-with-resources автоматически закроет JMSContext
    try (JMSContext context = connectionFactory.createContext()) {
        context.createProducer().send(myQueue, text);
    }
}
```

Как видите, JMSContext намного удобнее. Далее в примерах будем использовать его.

Destination

- **Что это:** Объект, представляющий пункт назначения сообщений. Это "виртуальный канал" в брокере.
- **Роль:** Указывает, куда отправлять или откуда получать сообщения.
- **Как получается:** Как и ConnectionFactory, Destination конфигурируется на сервере и получается в приложении через @Resource.
- **Основные реализации:**
 - jakarta.jms.Queue: для модели Point-to-Point.
 - jakarta.jms.Topic: для модели Publish/Subscribe.

```
@Resource(lookup = "java:/jms/queue/MyOrderQueue")
private Queue orderQueue;
```

```
@Resource(lookup = "java:/jms/topic/NewsTopic")
private Topic newsTopic;
```

MessageProducer

- **Что это:** Объект, используемый для отправки сообщений в Destination.
- **Роль:** Отправляет сообщения. Позволяет настраивать параметры для отправляемых сообщений, такие как приоритет, время жизни (TTL), режим доставки (постоянный/непостоянный).
- **Как получается:** Создается из JMSContext (или Session) вызовом context.createProducer().

```
try (JMSContext context = connectionFactory.createContext()) {  
    JMSProducer producer = context.createProducer();  
    // JMSProducer - это новый интерфейс для MessageProducer  
  
    // Настраиваем продюсер перед отправкой  
    producer.setPriority(9);  
    // Устанавливаем высокий приоритет (0-9, 9 - самый высокий)  
    producer.setTimeToLive(60000);  
    // Сообщение "протухнет" через 1 минуту  
  
    producer.send(myQueue, "This is an important and urgent message!");  
}
```

MessageConsumer

- **Что это:** Объект, используемый для получения сообщений из Destination.
- **Роль:** Получает сообщения. Может работать в двух режимах:
 1. **Синхронный:** Вы явно вызываете метод `receive()`, который блокирует поток и ждет поступления сообщения.
 2. **Асинхронный:** Вы регистрируете `MessageListener`, и брокер сам будет вызывать его при поступлении нового сообщения.
- **Как получается:** Создается из `JMSContext` (или `Session`) вызовом `context.createConsumer(destination)`.

Пример синхронного получения (используется редко, в основном для консольных утилит):

```
public void receiveMessageSync() {
    try (JMSContext context = connectionFactory.createContext()) {
        JMSConsumer consumer = context.createConsumer(myQueue);
        System.out.println("Waiting for a message...");
        // Блокируем поток на 5 секунд в ожидании сообщения
        Message message = consumer.receive(5000);
        if (message != null) {
            String text = message.getBody(String.class);
            // Удобный метод из JMS 2.0
            System.out.println("Received: " + text);
        } else {
            System.out.println("No message received in 5 seconds.");
        }
    } catch (JMSException e) {
        e.printStackTrace();
    }
}
```

MessageListener

- **Что это:** Интерфейс с одним методом `onMessage(Message message)`.
- **Роль:** Реализует логику асинхронной обработки сообщения. Это "сердце" получателя.
- **Как используется:** Вы создаете класс, реализующий этот интерфейс, и регистрируете его у `MessageConsumer`. Однако в Jakarta EE это почти никогда не делается вручную. Вместо этого используется **Message-Driven Bean (MDB)**, который является оберткой над `MessageListener`, управляемой контейнером.

Пример асинхронного получения с MDB (предпочтительный способ):

```
import jakarta.ejb.ActivationConfigProperty;
import jakarta.ejb.MessageDriven;
import jakarta.jms.Message;
import jakarta.jms.MessageListener;

// Контейнер создаст MessageConsumer "под капотом" и зарегистрирует
// этот класс в качестве его MessageListener'a.
@MessageDriven(activationConfig = {
    @ActivationConfigProperty(propertyName = "destinationLookup",
propertyValue = "java:/jms/queue/MyOrderQueue"),
    @ActivationConfigProperty(propertyName = "destinationType",
propertyValue = "jakarta.jms.Queue")
})
public class MyQueueListenerMDB implements MessageListener {
    // Этот метод будет вызван контейнером для каждого нового сообщения в
очереди.
    @Override
    public void onMessage(Message message) {
        try {
            // ... логика обработки сообщения ...
            System.out.println("Async processing of message with ID: " +
message.getJMSMessageID());
        } catch (Exception e) {
```

```

        // Обработка ошибок
    }
}
}

```

Этот подход полностью скрывает от разработчика сложность управления Connection, Session, MessageConsumer и потоками, позволяя сфокусироваться только на бизнес-логике обработки сообщения.

1.11.7. Типы сообщений в Jakarta Messaging (JMS)

JMS API предоставляет иерархию интерфейсов для сообщений. Все они наследуются от базового интерфейса `jakarta.jms.Message`. Этот базовый интерфейс позволяет работать с **заголовками (headers)** и **свойствами (properties)** сообщения, которые являются общими для всех типов.

- **Заголовки (Headers):** Это стандартные поля, устанавливаемые JMS-провайдером (например, `JMSMessageID`, `JMSDestination`, `JMSTimestamp`).
- **Свойства (Properties):** Это дополнительные метаданные в виде пар "ключ-значение", которые может добавлять разработчик. Они удобны для фильтрации сообщений на стороне получателя.

Помимо этого, каждый под-интерфейс определяет свой формат для **тела (body)** сообщения — то есть для основной полезной нагрузки.

Вот пять основных типов сообщений:

1. **TextMessage:** для передачи текста.
2. **ObjectMessage:** для передачи сериализуемых Java-объектов.
3. **BytesMessage:** для передачи потока неинтерпретируемых байтов.
4. **StreamMessage:** для передачи потока примитивных Java-типов.
5. **MapMessage:** для передачи набора пар "ключ-значение".

Рассмотрим каждый из них подробно.

1.11.8. TextMessage

- **Назначение:** Передача данных в виде строки (java.lang.String).
- **Когда использовать:** Это самый распространенный и рекомендуемый тип сообщений. Идеально подходит для передачи данных в легковесных, текстовых форматах, таких как **JSON** или **XML**.
- **Преимущества:**
 - **Интероперабельность:** Текстовые форматы легко могут быть прочитаны и поняты не только Java-приложениями, но и системами, написанными на других языках (Python, C#, JavaScript).
 - **Читаемость:** JSON и XML легко читать и отлаживать.
 - **Гибкость:** Формат данных не привязан к конкретным Java-классам.

Пример с TextMessage (используя JSON)

```
import jakarta.jms.JMSContext;
import jakarta.jms.Queue;

// Предполагается наличие библиотеки для работы с JSON, например, Jakarta
JSON Binding (JSON-B)
import jakarta.json.bind.Jsonb;
import jakarta.json.bind.JsonbBuilder;
// ...

public void sendOrderAsJson(Order order, JMSContext context, Queue
orderQueue) {
    // 1. Создаем JSON-B процессор
    try (Jsonb jsonb = JsonbBuilder.create()) {
        // 2. Сериализуем Java-объект Order в строку JSON
        String orderJson = jsonb.toJson(order);
        // 3. Отправляем строку как тело TextMessage
        context.createProducer().send(orderQueue, orderJson);
        System.out.println("Sent JSON: " + orderJson);
    } catch (Exception e) { // обработка ошибок }
}
```

```

import jakarta.jms.Message;
import jakarta.jms.MessageListener;
import jakarta.jms.TextMessage;
import jakarta.json.bind.Jsonb;
import jakarta.json.bind.JsonbBuilder;

// ...

@Override
public void onMessage(Message message) {
    if (message instanceof TextMessage) {
        try (Jsonb jsonb = JsonbBuilder.create()) {
            // 1. Получаем тело сообщения как строку
            String orderJson = ((TextMessage) message).getText();
            System.out.println("Received JSON: " + orderJson);
            // 2. Десериализуем JSON-строку обратно в Java-объект
            Order order = jsonb.fromJson(orderJson, Order.class);
            // 3. Обрабатываем объект
            processOrder(order);
        } catch (Exception e) {
            // обработка ошибок
        }
    }
}

```

1.11.9. ObjectMessage

- **Назначение:** Передача одного Java-объекта, который реализует интерфейс `java.io.Serializable`.
- **Когда использовать:** Когда и отправитель, и получатель являются Java-приложениями и имеют доступ к одному и тому же классу передаваемого объекта.
- **Недостатки:**
 - **Сильная связанность:** Создает жесткую зависимость между отправителем и получателем на уровне Java-классов. Если вы обновите класс на одной стороне, его нужно обновить и на другой.
 - **Плохая интероперабельность:** Системы на других языках не смогут прочесть такое сообщение.
 - **Безопасность:** Десериализация недоверенных объектов может быть небезопасной.
- **Рекомендация:** Используйте с осторожностью. В большинстве случаев `TextMessage` с JSON/XML является лучшей альтернативой.

Пример с ObjectMessage

```
public void sendOrderAsObject(Order serializableOrder, JMSContext context,
Queue orderQueue) {
    // Убедитесь, что класс Order реализует java.io.Serializable
    context.createProducer().send(orderQueue, serializableOrder);
    System.out.println("Sent serializable Order object.");
}
```

@Override

```
public void onMessage(Message message) {
    if (message instanceof ObjectMessage) {
        try {
            // 1. Кастуем к ObjectMessage
            ObjectMessage objectMessage = (ObjectMessage) message;
            // 2. Получаем объект. Возвращает java.io.Serializable.
            Order order = (Order) objectMessage.getObject();
            processOrder(order);
        }
    }
}
```



```

    } catch (JMSException e) {
        // обработка ошибок
    }
}
}

```

BytesMessage

- **Назначение:** Передача потока неинтерпретируемых байтов (byte[]).
- **Когда использовать:** Для передачи бинарных данных, таких как изображения, PDF-файлы, зашифрованные данные или данные в проприетарном бинарном формате (например, Protocol Buffers).
- **Преимущества:** Эффективно для передачи больших бинарных файлов.

Пример с BytesMessage

```

public void sendFileAsBytes(byte[] fileContent, JMSContext context, Queue
fileQueue) {
    // 1. Создаем BytesMessage
    BytesMessage bytesMessage = context.createBytesMessage();
    try {
        // 2. Записываем в него массив байт
        bytesMessage.writeBytes(fileContent);
        // 3. Отправляем
        context.createProducer().send(fileQueue, bytesMessage);
        System.out.println("Sent " + fileContent.length + " bytes.");
    } catch (JMSException e) {
        // обработка ошибок
    }
}
}

```

@Override

```

public void onMessage(Message message) {
    if (message instanceof BytesMessage) {
        try {

```

```

        BytesMessage bytesMessage = (BytesMessage) message;
        // 1. Создаем массив байт нужного размера
        byte[] fileContent = new byte[(int)
bytesMessage.getBodyLength()];
        // 2. Читаем байты из сообщения
        bytesMessage.readBytes(fileContent);
        System.out.println("Received " + fileContent.length + " bytes.");
        // ... логика сохранения файла или его обработки ...
    } catch (JMSEException e) {
        // обработка ошибок
    }
}
}

```

StreamMessage

- **Назначение:** Передача последовательности (потока) примитивных Java-типов.
- **Как работает:** Данные записываются в определенном порядке и должны быть прочитаны **ровно в том же порядке**.
- **Когда использовать:** Редко используется. Может быть полезен для интеграции с устаревшими системами, которые ожидают данные в виде строго упорядоченного потока примитивов.

Пример с StreamMessage

```
public void sendStream(JMSContext context, Queue dataQueue) {  
    try {  
        StreamMessage streamMessage = context.createStreamMessage();  
        streamMessage.writeLong(12345L); // Заказ ID  
        streamMessage.writeString("Product A"); // Название  
        streamMessage.writeInt(5); // Количество  
        streamMessage.writeDouble(19.99); // Цена  
  
        context.createProducer().send(dataQueue, streamMessage);  
    } catch (JMSException e) {  
        // ...  
    }  
}
```

```

@Override
public void onMessage(Message message) {
    if (message instanceof StreamMessage) {
        try {
            StreamMessage streamMessage = (StreamMessage) message;
            // Читаем в том же порядке, в котором записывали!
            long orderId = streamMessage.readLong();
            String productName = streamMessage.readString();
            int quantity = streamMessage.readInt();
            double price = streamMessage.readDouble();
            // ...
        } catch (JMSEException e) {
            // ...
        }
    }
}

```

1.11.10. MapMessage

- **Назначение:** Передача набора пар "ключ-значение". Ключ всегда является строкой (String), а значение — примитивным Java-типом.
- **Как работает:** Похоже на java.util.Map. Порядок чтения не важен, вы обращаетесь к значениям по ключу.
- **Когда использовать:** Когда нужно передать небольшой набор именованных, неструктурированных данных. Может быть альтернативой TextMessage с JSON для простых случаев, но менее гибок.

Пример с MapMessage

```
public void sendMap(JMSTContext context, Queue dataQueue) {  
    try {  
        MapMessage mapMessage = context.createMapMessage();  
        mapMessage.setLong("orderId", 12345L);  
        mapMessage.setString("productName", "Product A");  
        mapMessage.setInt("quantity", 5);  
        mapMessage.setDouble("price", 19.99);  
  
        context.createProducer().send(dataQueue, mapMessage);  
    } catch (JMSException e) {  
        // ...  
    }  
}
```

```

@Override
public void onMessage(Message message) {
    if (message instanceof MapMessage) {
        try {
            MapMessage mapMessage = (MapMessage) message;
            // Читаем в любом порядке по ключу
            long orderId = mapMessage.getLong("orderId");
            int quantity = mapMessage.getInt("quantity");
            // ...
        } catch (JMSException e) {
            // ...
        }
    }
}

```

1.11.11. Транзакции в JMS

Транзакционность в JMS гарантирует, что группа операций по отправке и/или получению сообщений будет выполнена как **единое атомарное действие (ACID)**.

Что это означает на практике?

- **Для отправки (Produce):** Все сообщения, отправленные в рамках одной транзакции, будут доставлены брокеру **только в момент фиксации (commit)** этой транзакции. Если произойдет откат (rollback), ни одно из этих сообщений не попадет в пункт назначения.
- **Для получения (Consume):** Сообщение считается "потребленным" (и удаляется из очереди) **только в момент фиксации (commit)** транзакции. Если произойдет откат, сообщение вернется в очередь и станет доступным для повторной обработки.

Это позволяет создавать надежные сценарии, такие как "**прочитать-обработать-записать**", где получение сообщения из одной очереди, обработка данных и отправка результата в другую очередь (или запись в БД) происходят в рамках одной атомарной операции.

Как JMS работает с транзакциями Jakarta EE (JTA)

В среде Jakarta EE, JMS-транзакции почти всегда интегрированы с глобальными транзакциями **JTA**. Это означает, что вы можете объединить операции с JMS и операции с базой данных (JPA) в одну общую, распределенную транзакцию.

Два основных способа управления транзакциями:

- 6. Управление контейнером (Container-Managed Transactions, CMT):**
Самый простой и рекомендуемый способ. Контейнер (EJB или CDI) сам управляет границами транзакции.
- 7. Управление сессией (Session-Transacted):** Более низкоуровневый способ, где транзакцией управляет сама JMS-сессия. Используется вне JTA-контекста.

1.11.12. Управление контейнером (CMT) с JTA

Это самый мощный подход. Вы просто выполняете JMS-операции внутри транзакционного метода (например, в EJB), и контейнер делает всю магию.

Сценарий: "Exactly-once processing" (Обработка ровно один раз)

Представим задачу: нужно получить сообщение с деталями платежа из очереди PaymentQueue, обработать его (сохранить в базу данных) и отправить уведомление в другую очередь NotificationQueue. Все это должно быть атомарно.

Generated java

```
import jakarta.annotation.Resource;
import jakarta.ejb.MessageDriven;
import jakarta.ejb.TransactionAttribute;
import jakarta.ejb.TransactionAttributeType;
import jakarta.jms.*;
import jakarta.persistence.EntityManager;
import jakarta.persistence.PersistenceContext;

@MessageDriven(activationConfig = {
    @ActivationConfigProperty(propertyName = "destinationLookup",
propertyValue = "java:/jms/queue/PaymentQueue"),
    @ActivationConfigProperty(propertyName = "destinationType", propertyValue
= "jakarta.jms.Queue")
})

public class PaymentProcessorMDB implements MessageListener {
```

```

@PersistenceContext
private EntityManager em; // Для работы с БД (JPA)

@Resource(lookup = "java:/JmsXA")
private ConnectionFactory connectionFactory;

@Resource(lookup = "java:/jms/queue/NotificationQueue")
private Queue notificationQueue;

// По умолчанию для MDB уже используется
TransactionAttributeType.REQUIRED
@Override
@Transactional(TransactionAttributeType.REQUIRED)
public void onMessage(Message message) {
    // --- НАЧАЛО ГЛОБАЛЬНОЙ JTA ТРАНЗАКЦИИ (неявно) ---
    // Получение сообщения из PaymentQueue уже является частью этой
    транзакции!
    try {
        String paymentDetails = message.getBody(String.class);
        System.out.println("Processing payment: " + paymentDetails);
        // 1. Работа с базой данных (JPA)
        Payment payment = new Payment(paymentDetails,
        "PROCESSED");
        em.persist(payment);
        // Симуляция ошибки при обработке "плохого" платежа
        if (paymentDetails.contains("ERROR")) {
            throw new RuntimeException("Simulating a processing
            error!");
        }
        // 2. Отправка сообщения в другую очередь (JMS)
        try (JMSContext context = connectionFactory.createContext()) {

```



```

        context.createProducer().send(notificationQueue, "Payment
        processed: " + payment.getId());
    }

    // --- КОНЕЦ ТРАНЗАКЦИИ ---
    // Если мы дошли до сюда без ошибок, контейнер вызовет COMMIT.
    // Что произойдет при COMMIT:
    // 1. Сообщение будет ОКОНЧАТЕЛЬНО удалено из PaymentQueue.
    // 2. Транзакция в БД будет зафиксирована (payment сохранится).
    // 3. Сообщение будет ОКОНЧАТЕЛЬНО отправлено в
    NotificationQueue.
    } catch (Exception e) {
        System.err.println(e.getMessage());
        // --- ОШИБКА! КОНТЕЙНЕР ДЕЛАЕТ ROLLBACK ---
        // Что произойдет при ROLLBACK:
        // 1. Сообщение ВЕРНЕТСЯ в PaymentQueue для повторной
        обработки.
        // 2. Транзакция в БД будет отменена (payment не сохранится).
        // 3. Сообщение НЕ будет отправлено в NotificationQueue.
        // Контейнер сам позаботится об откате, нам не нужно ничего
        делать.
        // Можно просто позволить исключению "всплыть".
    }
}
}

```

Ключевые моменты этого примера:

- **Атомарность:** Все три действия (получение, запись в БД, отправка) выполняются как единое целое.
- **Надежность:** Если во время обработки произойдет сбой, система вернется в исходное состояние, и сообщение не потеряется. Оно будет обработано позже.
- **Простота:** Весь сложный механизм координации скрыт за одной аннотацией `@Transactional` (или поведением MDB по умолчанию).

1.11.13. Управление сессией (Session-Transacted)

Этот способ используется, когда вы работаете с JMS напрямую, без EJB или JTA-контейнера, или когда вам нужен контроль над транзакцией, ограниченный только одной JMS сессией.

Как работает:

1. При создании сессии вы указываете, что она будет транзакционной.
2. Выполняете одну или несколько операций отправки/получения.
3. Явно вызываете `session.commit()` или `session.rollback()`.

Пример: Отправка нескольких сообщений как единый пакет.

```
public void sendBatchOfMessages() {  
    Connection connection = null;  
    Session session = null;  
    try {  
        connection = connectionFactory.createConnection();  
        connection.start();  
        // 1. Создаем транзакционную сессию  
        session = connection.createSession(true,  
        Session.SESSION_TRANSACTED);  
        MessageProducer producer = session.createProducer(myQueue);  
        // 2. Отправляем несколько сообщений. Они пока не доставлены  
        // брокеру,  
        // а буферизуются на стороне клиента в рамках сессии.  
        producer.send(session.createTextMessage("Message 1"));  
        producer.send(session.createTextMessage("Message 2"));  
        producer.send(session.createTextMessage("Message 3"));  
        System.out.println("Messages are ready to be committed.");  
        // 3. Фиксируем транзакцию. Только в этот момент все три  
        // сообщения будут атомарно отправлены брокеру.  
        session.commit();  
        System.out.println("Transaction committed. Messages sent.");  
    } catch (JMSException e) {  
        System.err.println("An error occurred. Rolling back transaction.");  
    }  
}
```

```

        if (session != null) {
            try {
                // 4. В случае ошибки откатываем транзакцию.
                // Ни одно из сообщений не будет отправлено.
                session.rollback();
            } catch (JMSEException rollbackEx) {
                // ...
            }
        }
    } finally {
        // Закрываем ресурсы
        if (session != null) try { session.close(); } catch (JMSEException e) {}
        if (connection != null) try { connection.close(); } catch (JMSEException e) {}
    }
}

```

Сравнение с JMSContext (JMS 2.0+):

Современный JMSContext API также поддерживает транзакции.

```

public void sendBatchWithJmsContext() {
    // try-with-resources для JMSContext, который будет управлять соединением
    // и сессией
    try (JMSContext context =
        connectionFactory.createContext(JMSContext.SESSION_TRANSACTED)) {
        try {
            JMSProducer producer = context.createProducer();
            producer.send(myQueue, "Message 1");
            producer.send(myQueue, "Message 2");
            producer.send(myQueue, "Message 3");
            // Явно вызываем commit на контексте
            context.commit();
            System.out.println("Transaction committed.");
        }
    }
}

```

```

    } catch (Exception e) {
        // Явно вызываем rollback
        context.rollback();
        System.err.println("Transaction rolled back.");
    }
}
}
}

```

1.11.14. Заключение

Подход	Когда использовать	Преимущества	Недостатки
Управление контейнером (CMT, JTA)	В 99% случаев в Jakarta EE. Особенно для сценариев "прочитать-обработать-записать".	Простота, надежность, поддержка распределенных транзакций (БД + JMS).	Требует EJB или CDI с поддержкой <code>@Transactional</code> .
Управление сессией (Session-Transacted)	Вне JTA-контекста (например, в standalone-приложениях). Когда нужна транзакция, затрагивающая только JMS-операции.	Гранулярный контроль над границами JMS-транзакции.	Сложность, ручное управление, не поддерживает распределенные транзакции.

1.11.15. Message-Driven Beans (MDB)

Что такое MDB?

Message-Driven Bean (MDB) — это специальный тип Enterprise JavaBean (EJB), который выступает в роли **асинхронного слушателя сообщений**. В отличие от других EJB (Session Beans), MDB не вызывается напрямую клиентом через интерфейс. Вместо этого он "приводится в действие" (driven) поступлением сообщения из пункта назначения (очереди или топика).

Ключевая идея: Вы пишете только **бизнес-логику** обработки сообщения. Вся инфраструктурную работу — установку соединения с брокером, прослушивание сообщений, управление пулом слушателей, управление транзакциями и обработку ошибок — берет на себя **EJB-контейнер**.

Асинхронная обработка сообщений с использованием MDB

Давайте рассмотрим, как MDB решает типичные задачи асинхронной обработки.

1. Жизненный цикл и пул экземпляров

- MDB является **stateless** (не хранит состояние).
- Контейнер создает **пул экземпляров** MDB для параллельной обработки сообщений.
- Когда в очередь поступает новое сообщение, контейнер берет свободный MDB из пула и вызывает его метод `onMessage`.
- После завершения обработки MDB возвращается в пул и готов к обработке следующего сообщения. Это очень эффективно и позволяет легко масштабировать обработку, просто увеличивая размер пула.

2. Интеграция с JMS

- MDB тесно интегрирован с JMS. Вы просто указываете в конфигурации, какой `Destination` (очередь или топик) нужно слушать.
- Контейнер "под капотом" создает `Connection`, `Session` и `MessageConsumer` и регистрирует MDB в качестве `MessageListener`.

3. Управление транзакциями

- По умолчанию, каждый вызов метода `onMessage` в MDB оборачивается в **JTA-транзакцию** (с атрибутом `REQUIRED`).
- Это обеспечивает надежную обработку: если в `onMessage` происходит ошибка (`RuntimeException`), транзакция откатывается, и сообщение **возвращается в очередь** для повторной попытки.
- Это позволяет легко реализовать сценарии "прочитать-обработать-записать" (например, прочитать из JMS, записать в JPA) с гарантиями атомарности.

4. Простота и декларативность

- Вам не нужно писать сложный код для управления потоками, подключениями или транзакциями.
- Вся конфигурация задается **декларативно**, с помощью аннотаций.

1.11.16. Пример: Система обработки заказов

Представим систему, где веб-фронтенд отправляет заказы в JMS очередь OrderQueue, а MDB их асинхронно обрабатывает: сохраняет в базу данных и отправляет подтверждение по email.

Шаг 1: Создание MDB

Класс MDB должен:

1. Быть помечен аннотацией @jakarta.ejb.MessageDriven.
2. Реализовывать интерфейс jakarta.jms.MessageListener.
3. Иметь activationConfig для указания источника сообщений.

```
import jakarta.ejb.ActivationConfigProperty;
import jakarta.ejb.MessageDriven;
import jakarta.inject.Inject;
import jakarta.jms.Message;
import jakarta.jms.MessageListener;
import jakarta.jms.TextMessage;
import jakarta.json.bind.Jsonb;
import jakarta.json.bind.JsonbBuilder;
import jakarta.persistence.EntityManager;
import jakarta.persistence.PersistenceContext;

// 1. Помечаем как MDB и конфигурируем
@MessageDriven(activationConfig = {
    // Указываем JNDI-имя очереди для прослушивания
    @ActivationConfigProperty(propertyName = "destinationLookup",
propertyValue = "java:/jms/queue/OrderQueue"),
    // Указываем тип назначения
    @ActivationConfigProperty(propertyName = "destinationType", propertyValue
= "jakarta.jms.Queue")
})

public class OrderProcessorMDB implements MessageListener { // 2. Реализуем
MessageListener

    // Внедряем другие Jakarta EE компоненты
```

```

@PersistenceContext
private EntityManager em;
@Inject
private EmailService emailService;

// 3. Реализуем главный метод onMessage
// Каждый вызов этого метода будет в своей JTA транзакции
@Override
public void onMessage(Message message) {
    try {
        if (message instanceof TextMessage) {
            String orderJson = ((TextMessage) message).getText();
            System.out.println("Received new order request: " +
                orderJson);
            // Десериализуем JSON в объект Order
            OrderData orderData;
            try (Jsonb jsonb = JsonbBuilder.create()) {
                orderData = jsonb.fromJson(orderJson,
                    OrderData.class);
            }
            // --- Начало бизнес-логики в рамках транзакции ---
            // 1. Сохраняем заказ в базу данных
            Order newOrder = new Order(orderData.getProductId(),
                orderData.getQuantity(), "PROCESSING");
            em.persist(newOrder);
            // 2. Отправляем email-подтверждение
            emailService.sendOrderConfirmation(orderData.getCustomerEmail(), newOrder.getId());
            System.out.println("Order " + newOrder.getId() + "
                processed successfully.");
            // --- Конец бизнес-логики ---
            // Если мы дошли сюда, контейнер сделает COMMIT.

```



```

        // Сообщение будет удалено из очереди, а заказ
        // сохранится в БД.
    }
} catch (Exception e) {
    // Если здесь произойдет любое исключение (ошибка парсинга
    // JSON,
    // сбой email-сервиса, ошибка БД), контейнер это перехватит.
    System.err.println("Failed to process order. Transaction will be
    rolled back.");
    // Контейнер сделает ROLLBACK транзакции.
    // Сообщение вернется в очередь для повторной обработки.
    // Это поведение можно настроить (например, через Dead Letter
    // Queue).
}
}
}

```

// Вспомогательные классы для примера

```
class OrderData {
```

```
    /* поля: customerEmail, productId, quantity, getters/setters */
```

```
}
```

@Entity

```
class Order {
```

```
    /* поля: id, productId, quantity, status, getters/setters */
```

```
    public Order(String productId, int quantity, String status) { /*...*/ }
```

```
}
```

@ApplicationScoped

```
class EmailService {
```

```
    public void sendOrderConfirmation(String email, Long orderId) {
```

```
        // ... логика отправки email ...
```

```
        System.out.println("Sending confirmation for order " + orderId + " to " +  
email);
```

```
    }
```

```
}
```

Шаг 2: Отправка сообщения в очередь (клиентский код)

Этот код может находиться в REST-контроллере, который принимает HTTP-запрос от фронтенда.

```
import jakarta.annotation.Resource;
import jakarta.ejb.Stateless;
import jakarta.jms.ConnectionFactory;
import jakarta.jms.JMSContext;
import jakarta.jms.Queue;
import jakarta.json.bind.Jsonb;
import jakarta.json.bind.JsonbBuilder;

@Stateless
public class OrderApi {
    @Resource(lookup = "java:/JmsXA")
    private ConnectionFactory connectionFactory;
    @Resource(lookup = "java:/jms/queue/OrderQueue")
    private Queue orderQueue;

    public void placeNewOrder(OrderData orderData) {
        try (JMSContext context = connectionFactory.createContext();
            Jsonb jsonb = JsonbBuilder.create()) {
            String orderJson = jsonb.toJson(orderData);
            context.createProducer().send(orderQueue, orderJson);
            System.out.println("Order request sent to the queue.");
        } catch (Exception e) { // Обработка ошибок отправки }
    }
}
```

1.11.17. Поток выполнения

1. Клиент вызывает `OrderApi.placeNewOrder()`.

2. OrderApi создает JSON-сообщение и отправляет его в OrderQueue. На этом его работа завершена, он не ждет обработки.
3. EJB-контейнер, который слушает OrderQueue, обнаруживает новое сообщение.
4. Контейнер берет свободный экземпляр OrderProcessorMDB из пула.
5. Контейнер **начинает JTA-транзакцию**.
6. Контейнер вызывает метод onMessage, передавая ему сообщение.
7. Внутри onMessage выполняются операции с JPA и EmailService.
8. **Если onMessage завершается успешно: контейнер фиксирует (commit) транзакцию.** Это атомарно удаляет сообщение из очереди и сохраняет изменения в БД.
9. **Если в onMessage возникает RuntimeException: контейнер откатывает (rollback) транзакцию.** Это атомарно возвращает сообщение в очередь и отменяет изменения в БД.

MDB — это идеальный пример принципа "Инверсии управления" (Inversion of Control), который является центральным для всей платформы Jakarta EE. Вы пишете только то, что нужно сделать, а контейнер решает, как это сделать надежно и эффективно.

1.11.18. Конфигурация MDB: @ActivationConfigProperty

activationConfig — это массив аннотаций @ActivationConfigProperty, каждая из которых представляет собой пару "ключ-значение" (propertyName и propertyValue). Эти свойства передаются **Resource Adapter**'у (адаптеру ресурсов), который отвечает за доставку сообщений. Для JMS это будет JMS Resource Adapter, который встроен в любой Jakarta EE сервер.

Давайте рассмотрим самые важные и часто используемые конфигурационные свойства.

1.11.19. Базовая конфигурация: указание источника сообщений

Это два самых главных свойства, которые обязательны для любого MDB, работающего с JMS.

- **destinationLookup**: Указывает **JNDI-имя** очереди (Queue) или топика (Topic), который MDB должен слушать. Этот ресурс должен быть предварительно создан и настроен на сервере приложений.
- **destinationType**: Указывает полное имя класса типа назначения. Это помогает адаптеру понять, с чем он работает.
 - "jakarta.jms.Queue" (для очередей)
 - "jakarta.jms.Topic" (для топиков)

Пример для Очереди (Queue)

```
import jakarta.ejb.ActivationConfigProperty;
import jakarta.ejb.MessageDriven;
import jakarta.jms.Message;
import jakarta.jms.MessageListener;

@MessageDriven(
    name = "MyQueueMdb", // Необязательное имя EJB
    activationConfig = {
        @ActivationConfigProperty(
            propertyName = "destinationLookup",
            propertyValue = "java:/jms/queue/MyApplicationQueue" // JNDI-имя
очереди
        ),
        @ActivationConfigProperty(
            propertyName = "destinationType",
            propertyValue = "jakarta.jms.Queue" // Тип – Очередь
        )
    }
)
public class MyQueueListener implements MessageListener {
    @Override
```

```

public void onMessage(Message message) {
    // ... логика обработки ...
}
}

```

Пример для Топика (Topic)

```

import jakarta.ejb.ActivationConfigProperty;
import jakarta.ejb.MessageDriven;
import jakarta.jms.Message;
import jakarta.jms.MessageListener;

@MessageDriven(
    activationConfig = {
        @ActivationConfigProperty(
            propertyName = "destinationLookup",
            propertyValue = "java:/jms/topic/MyApplicationTopic" // JNDI-имя
            топика
        ),
        @ActivationConfigProperty(
            propertyName = "destinationType",
            propertyValue = "jakarta.jms.Topic" // Тип – Топик
        )
    }
)
public class MyTopicListener implements MessageListener {
    @Override
    public void onMessage(Message message) { // ... логика обработки ... }
}

```

1.11.20. Конфигурация для работы с Топиками (Publish/Subscribe)

При работе с топиками часто требуются дополнительные настройки.

Долговечная подписка (Durable Subscription)

По умолчанию, если подписчик на топик неактивен, он пропускает все сообщения, опубликованные в это время. Долговечная подписка решает эту проблему: брокер будет накапливать сообщения для неактивного подписчика.

- **subscriptionDurability**: Устанавливается в "Durable".
- **subscriptionName**: Уникальное имя для этой подписки. Оно идентифицирует подписчика.
- **clientId**: Уникальный идентификатор клиента, к которому относится подписка. Важно, чтобы у каждого долговечного подписчика был свой clientId.

```
@MessageDriven(
    activationConfig = {
        @ActivationConfigProperty(propertyName = "destinationLookup",
            propertyValue = "java:/jms/topic/ImportantEventsTopic"),
        @ActivationConfigProperty(propertyName = "destinationType",
            propertyValue = "jakarta.jms.Topic"),
        // --- Настройки долговечной подписки ---
        @ActivationConfigProperty(propertyName = "subscriptionDurability",
            propertyValue = "Durable"),
        @ActivationConfigProperty(propertyName = "subscriptionName",
            propertyValue = "AuditServiceSubscription"),
        @ActivationConfigProperty(propertyName = "clientId", propertyValue =
            "AuditServiceClient")
    }
)
public class DurableTopicListener implements MessageListener { // ... }
```

Фильтрация сообщений (Message Selector)

MDB может быть настроен так, чтобы получать не все сообщения из Destination, а только те, которые удовлетворяют определенному условию. Это позволяет избежать лишней обработки на стороне приложения.

- **messageSelector**: Строка с SQL-подобным выражением (синтаксис основан на подмножестве SQL-92 WHERE), которое применяется к **свойствам (properties)** сообщения.

Сценарий: MDB должен обрабатывать только заказы с высоким приоритетом или из определенного региона.

1. При отправке сообщения добавляем свойство:

// ... в коде отправителя ...

```
JMSProducer producer = context.createProducer();
```

// Устанавливаем пользовательское свойство

```
producer.setProperty("region", "EMEA");
```

```
producer.setProperty("priority", "high");
```

```
producer.send(orderQueue, "Order details...");
```

2. В MDB настраиваем селектор:

```
@MessageDriven(
```

```
    activationConfig = {
```

```
        @ActivationConfigProperty(propertyName = "destinationLookup",  
        propertyValue = "java:/jms/queue/OrderQueue"),
```

```
        @ActivationConfigProperty(propertyName = "destinationType",  
        propertyValue = "jakarta.jms.Queue"),
```

```
        // --- Настройка селектора ---
```

```
        @ActivationConfigProperty(  
            propertyName = "messageSelector",
```

```
            // Этот MDB получит сообщение, только если оба условия истинны
```

```
            propertyValue = "region = 'EMEA' AND priority = 'high'"
```

```
        )
```

```
    }
```

```
)
```

```
public class HighPriorityEmeaOrderProcessor implements MessageListener {
```



```
// Этот MDB будет активирован только для сообщений, соответствующих селектору
}
```

1.11.21. Управление подтверждением (Acknowledgement)

По умолчанию MDB работает в транзакционном режиме, и подтверждение происходит автоматически при коммите транзакции. Но если вы отключите транзакции (`@TransactionManagement(TransactionManagementType.BEAN)` и не используете `UserTransaction`), вам может понадобиться управлять подтверждением вручную.

- **acknowledgeMode:**

- "Auto-acknowledge" (по умолчанию для не-транзакционных сессий): Сообщение считается доставленным, как только метод `onMessage` успешно завершается.
- "Dups-ok-acknowledge": Ленивое подтверждение. Менее надежно, но может дать прирост производительности. Возможна повторная доставка сообщений.

`@TransactionManagement(TransactionManagementType.BEAN)` // Отключаем CMT

```
@MessageDriven(
    activationConfig = {
        // ... destination properties ...
        @ActivationConfigProperty(
            propertyName = "acknowledgeMode",
            propertyValue = "Auto-acknowledge"
        )
    }
)

public class NonTransactionalMdb implements MessageListener { // ... }
```

Важно: В 99% случаев в Jakarta EE вы будете использовать транзакционное управление (CMT), поэтому свойство `acknowledgeMode` вам, скорее всего, не понадобится. Оно важнее для standalone JMS-клиентов.

1.11.22. Свойства, специфичные для брокера

Помимо стандартных свойств JMS, многие брокеры (и их адаптеры ресурсов) предоставляют свои собственные, специфичные свойства для тонкой настройки. Например, для ActiveMQ это могут быть настройки, связанные с prefetch-лимитом, Dead-Letter-Queue (DLQ) и политиками повторной доставки.

Их имена и значения зависят от конкретного сервера приложений и брокера. Например, в WildFly:

```
@MessageDriven(  
    activationConfig = {  
        // ... стандартные свойства ...  
        // --- Специфичные для WildFly/Artemis ---  
        @ActivationConfigProperty(propertyName = "maxSession",  
            propertyValue = "15"), // Максимальное количество сессий в пуле  
        @ActivationConfigProperty(propertyName = "reconnectAttempts",  
            propertyValue = "-1") // Бесконечные попытки переподключения  
    }  
)
```

Эти свойства нужно искать в документации к вашему серверу приложений.

1.11.23. Заключение

Конфигурация MDB — это мощный декларативный механизм.

- **Основа:** `destinationLookup` и `destinationType` для указания источника.
- **Для топиков:** `subscriptionDurability`, `subscriptionName`, `clientId` для надежной доставки.
- **Для фильтрации:** `messageSelector` для обработки только нужных сообщений.
- **Для экспертов:** Специфичные для брокера свойства для тонкой настройки производительности и отказоустойчивости.

Эта гибкость позволяет одному и тому же коду MDB быть переиспользованным в разных сценариях просто за счет изменения конфигурационных аннотаций.

1.11.24. Жизненный цикл Message-Driven Bean (MDB)

Жизненный цикл MDB состоит из двух основных состояний: "**Не существует**" (**Does Not Exist**) и "**Готов к работе**" (**Ready**). Весь жизненный цикл управляется EJB-контейнером.

Основная идея: Контейнер создает пул экземпляров MDB, чтобы они были готовы к немедленной обработке входящих сообщений. Вам, как разработчику, не нужно заботиться о создании или уничтожении экземпляров.

Давайте рассмотрим этот процесс по шагам.

Шаг 1: Создание экземпляра и внедрение зависимостей

Когда приложение развертывается, EJB-контейнер читает аннотацию `@MessageDriven` и понимает, что ему нужно подготовить пул слушателей для указанного `Destination`.

1. **Создание экземпляра:** Контейнер вызывает конструктор MDB без аргументов (`new MyMdb()`) для создания нового экземпляра.
2. **Внедрение зависимостей (Dependency Injection):** Сразу после создания экземпляра контейнер внедряет все ресурсы, помеченные аннотациями `@Inject`, `@EJB`, `@PersistenceContext`, `@Resource`. На этом этапе MDB еще не готов к работе.

Шаг 2: Метод обратного вызова `@PostConstruct`

- **Назначение:** Этот метод вызывается контейнером **один раз** в жизни экземпляра, сразу после внедрения всех зависимостей, но **до того**, как MDB будет помещен в пул и начнет принимать сообщения.
- **Идеальное место для:**
 - Инициализации ресурсов, которые требуют предварительной настройки.
 - Кэширования каких-либо данных.
 - Установки соединений с внешними системами (хотя это нужно делать осторожно).

```
import jakarta.annotation.PostConstruct;
import jakarta.ejb.MessageDriven;
import jakarta.inject.Inject;
import jakarta.jms.Message;
import jakarta.jms.MessageListener;
```

```
@MessageDriven(/*...activationConfig...*/)
public class MyMdb implements MessageListener {

    @Inject
    private ConfigurationService configService;
    private String someCachedValue;

    public MyMdb() {
        // В конструкторе зависимости еще НЕ внедрены!
        // configService здесь будет null.
        System.out.println("1. MDB instance created.");
    }

    @PostConstruct
    public void initialize() {
        // Здесь зависимости уже внедрены и доступны.
        // Этот метод вызывается один раз.
        System.out.println("2. PostConstruct: Initializing MDB instance.");
        this.someCachedValue = configService.getSomeImportantValue();
    }
}
```

```

    }

    @Override
    public void onMessage(Message message) {
        // ...
    }
}

```

Шаг 3: Состояние "Готов к работе" (Ready State) и метод onMessage

После выполнения `@PostConstruct` экземпляр MDB помещается в **пул готовых экземпляров**. Теперь он находится в состоянии "Готов к работе" и ждет сообщений.

- Когда в Destination (очередь или топик) поступает новое сообщение, контейнер:
 1. Берет свободный экземпляр MDB из пула.
 2. Начинает транзакцию (если настроено).
 3. Вызывает метод `onMessage(Message message)`, передавая ему пришедшее сообщение.
 4. Ждет завершения метода.
 5. Завершает транзакцию (`commit` или `rollback`).
 6. Возвращает экземпляр MDB обратно в пул.
- Метод `onMessage` может быть вызван **многократно** в течение жизни одного экземпляра MDB, но **никогда не параллельно для одного и того же экземпляра**. Если приходят два сообщения одновременно, контейнер использует два разных экземпляра MDB из пула для их обработки.

// ... продолжение класса MyMdb ...

```

@Override
public void onMessage(Message message) {
    // Этот метод может вызываться много раз для одного и того же экземпляра.
    System.out.println("3. onMessage: Processing a message using cached value: "
+ someCachedValue);
    // ... основная логика обработки ...
}

```

Шаг 4: Метод обратного вызова @PreDestroy

- **Назначение:** Этот метод вызывается контейнером непосредственно перед тем, как экземпляр MDB будет окончательно удален из пула и уничтожен.
- **Когда это происходит?**
 - При остановке или переразвертывании приложения.
 - Если контейнер решил сократить размер пула из-за низкой нагрузки.
- **Идеальное место для:**
 - Освобождения ресурсов, захваченных в `@PostConstruct`.
 - Корректного закрытия соединений.
 - Сохранения какого-либо состояния перед выключением.

```
import jakarta.annotation.PreDestroy;

// ... продолжение класса MyMdb ...

@PreDestroy
public void cleanup() {
    // Этот метод вызывается один раз перед уничтожением экземпляра.
    System.out.println("4. PreDestroy: Cleaning up resources for MDB instance.");
    this.someCachedValue = null;
    // ... например, закрыть соединения, если они были открыты ...
}
```

1.11.25. Схема жизненного цикла

```
graph TD
    A(Не существует) -->|1. new MyMdb()| B(Создан, зависимости НЕ внедрены);
    B -->|2. Внедрение зависимостей (@Inject, @Resource)| C(Зависимости внедрены);
    C -->|3. Вызов @PostConstruct| D[Состояние "Готов к работе" <br/> (в пуле)];
    D -->|4. Приходит сообщение| E{Вызов onMessage()};
    E -->|5. Обработка завершена| D;
    D -->|6. Контейнер решает <br/> уничтожить экземпляр| F(Вызов @PreDestroy);
    F -->|7. Освобождение ресурсов| A;
    style D fill:#c9ffc9,stroke:#333,stroke-width:2px
```

style A fill:#ffc9c9,stroke:#333,stroke-width:2px

1.11.26. Выводы и лучшие практики

1. **Не храните состояние, специфичное для сообщения, в полях MDB.** MDB является stateless, и один и тот же экземпляр обрабатывает разные сообщения. Все, что относится к конкретному сообщению, должно быть создано и использовано внутри метода onMessage.
2. **Используйте @PostConstruct для однократной инициализации.** Это эффективнее, чем выполнять тяжелую инициализацию в каждом вызове onMessage.
3. **Помните о пуле.** Ваш код в onMessage должен быть готов к тому, что он будет выполняться параллельно на разных экземплярах MDB.
4. **Обрабатывайте исключения.** Неперехваченное RuntimeException в onMessage приведет к откату транзакции и, возможно, к бесконечному циклу повторной доставки "отравленного" сообщения (poison message). Иногда лучше перехватить исключение, залогировать его и не дать ему "всплыть", чтобы транзакция закоммитилась и сообщение было удалено из очереди (например, после нескольких неудачных попыток). Это зависит от бизнес-логики.

1.11.27. Jakarta Connectors (JCA): Интеграция с EIS

Проблема: Хаос интеграции

Представьте себе крупную компанию. У нее есть:

- Современное Java-приложение (например, интернет-магазин).
- Устаревшая **ERP-система** (например, SAP) для управления складом.
- **Мейнфрейм** с системой биллинга, написанной на COBOL.
- Какая-то **CRM-система** со своим проприетарным протоколом.

Все эти системы называются **Корпоративными Информационными Системами (Enterprise Information Systems, EIS)**.

Чтобы ваше Java-приложение могло, например, проверить остаток товара на складе в SAP, вам нужно:

1. Найти и изучить специфический API или протокол для SAP.
2. Написать код для установки сетевого соединения.
3. Реализовать логику аутентификации.
4. Написать код для управления транзакциями с этой системой.
5. Сделать то же самое для мейнфрейма.
6. И еще раз для CRM...

В итоге ваше приложение превращается в клубок разнородного, сложного и нестандартного кода для интеграции. Каждая интеграция требует уникального подхода к управлению соединениями, транзакциями и безопасностью.

Решение: Стандартизация через Jakarta Connectors

Jakarta Connectors (JCA) — это стандартный фреймворк в Jakarta EE для интеграции сервера приложений с разнородными EIS.

Ключевая идея: Вместо того чтобы приложение напрямую "разговаривало" с EIS, оно взаимодействует со стандартным **адаптером ресурсов (Resource Adapter)**. А уже этот адаптер, специфичный для конкретной EIS (например, адаптер для SAP), берет на себя всю сложность низкоуровневой коммуникации.

Это очень похоже на то, как **JDBC** стандартизирует доступ к базам данных. Вы не пишете код для Oracle, PostgreSQL или MySQL. Вы пишете код для стандартного JDBC API, а **JDBC-драйвер** (который является формой адаптера ресурсов) делает всю работу. JCA — это как "JDBC для всего остального".

Архитектура Jakarta Connectors

JCA определяет набор стандартных **контрактов (System Contracts)** между сервером приложений и адаптером ресурсов.

- **Адаптер ресурсов (Resource Adapter):** Это компонент, поставляемый вендором EIS (например, SAP или IBM). Он упаковывается в специальный архив .rar (Resource Adapter Archive) и развертывается на сервере приложений.
- **Сервер приложений (Application Server):** WildFly, Payara и т.д.
- **Приложение (Application Component):** Ваш EJB или другой компонент.

Контракты между сервером и адаптером:

1. Connection Management (Управление соединениями):

- Адаптер предоставляет пул "дорогих" соединений к EIS.
- Сервер приложений управляет этим пулом, предоставляя соединения вашему приложению по запросу. Это гарантирует эффективность и масштабируемость.

2. Transaction Management (Управление транзакциями):

- Адаптер может участвовать в глобальных **JTA-транзакциях** (включая двухфазный коммит).
- Это позволяет выполнять атомарные операции, затрагивающие и вашу базу данных (через JPA), и внешнюю EIS (через JCA). Например, списать деньги в БД и подтвердить заказ в SAP в рамках одной транзакции.

3. Security Management (Управление безопасностью):

- Позволяет прозрачно передавать информацию о безопасности (кто выполняет операцию) из вашего приложения в EIS.

Контракт между приложением и адаптером:

- **Common Client Interface (CCI):** Это стандартный API, который ваше приложение **может** использовать для взаимодействия с адаптером. Он определяет общие интерфейсы, такие как ConnectionFactory, Connection, Interaction.

1.11.28. Пример: Как это выглядит на практике

Предположим, у нас есть адаптер для вымышленной системы управления запасами (Inventory System), который позволяет проверить наличие товара.

Шаг 1: Развертывание и настройка адаптера

4. Администратор берет файл `inventory-adapter.rar` (от вендора EIS) и развертывает его на сервере приложений (например, WildFly).
5. Затем администратор настраивает **фабрику соединений (ConnectionFactory)** для этого адаптера и привязывает ее к JNDI-имени, например, `java:/eis/InventoryCF`. В настройках указывается IP-адрес EIS, порт и т.д.

Шаг 2: Написание кода в приложении (используя CCI)

Ваш бизнес-компонент (например, EJB) теперь может использовать этот адаптер через стандартный API.

```
import jakarta.annotation.Resource;
```

```
import jakarta.ejb.Stateless;
```

```
import jakarta.resource.cci.Connection;
```

```
import jakarta.resource.cci.ConnectionFactory;
```

```
import jakarta.resource.cci.Interaction;
```

```
import jakarta.resource.cci.MappedRecord;
```

```
import jakarta.resource.cci.RecordFactory;
```

```
@Stateless
```

```
public class StockCheckService {
```

```
    // 1. Внедряем ConnectionFactory, настроенную администратором.
```

```
    // Тип - стандартный интерфейс из Jakarta Connectors API.
```

```
    @Resource(lookup = "java:/eis/InventoryCF")
```

```
    private ConnectionFactory connectionFactory;
```

```
    public int getStockLevel(String productId) {
```

```
        Connection connection = null;
```

```
        Interaction interaction = null;
```

```
        try {
```

```
            // 2. Получаем соединение с EIS из пула, управляемого сервером.
```

```

        connection = connectionFactory.getConnection();
        // 3. Создаем объект "взаимодействия" (Interaction).
        interaction = connection.createInteraction();
        // 4. Создаем запись для отправки данных (входные
        параметры).
        // MappedRecord - это как java.util.Мар.
        RecordFactory          recordFactory          =
connectionFactory.getRecordFactory();
        MappedRecord          input                    =
recordFactory.createMappedRecord("CheckStockInput");
        input.put("productId", productId);
        // 5. Выполняем "функцию" на EIS. Имя функции "checkStock"
        // определяется самим адаптером.
        // Возвращается другая запись с выходными данными.
        MappedRecord output = (MappedRecord) interaction.execute(null,
input);
        // 6. Извлекаем результат.
        return (Integer) output.get("quantity");
    } catch (Exception e) {
        // Обработка ошибок
        throw new RuntimeException("Failed to check stock in EIS", e);
    } finally {
        // 7. Корректно закрываем ресурсы. Соединение вернется в пул.
        if (interaction != null) {
            try { interaction.close(); } catch (Exception e) {}
        }
        if (connection != null) {
            try { connection.close(); } catch (Exception e) {}
        }
    }
}

```

```
}
```

Шаг 3 (Самое интересное): Интеграция с транзакциями

Теперь представим, что мы хотим уменьшить остаток на складе в EIS и сохранить заказ в нашей БД в рамках одной атомарной операции.

```
@Stateless
```

```
public class OrderPlacementService {
```

```
    @Resource(lookup = "java:/eis/InventoryCF")
```

```
    private ConnectionFactory inventoryConnectionFactory;
```

```
    @PersistenceContext
```

```
    private EntityManager em;
```

```
    // Этот метод будет обернут в глобальную JTA-транзакцию
```

```
    @Transactional
```

```
    public void placeOrder(Order order, String productId) throws Exception {
```

```
        // --- НАЧАЛО РАСПРЕДЕЛЕННОЙ ТРАНЗАКЦИИ ---
```

```
        // 1. Работа с EIS через JCA
```

```
        try (Connection conn = inventoryConnectionFactory.getConnection();
```

```
            Interaction interaction = conn.createInteraction()) {
```

```
            RecordFactory rf = inventoryConnectionFactory.getRecordFactory();
```

```
            MappedRecord input = rf.createMappedRecord("DecreaseStockInput");
```

```
            input.put("productId", productId);
```

```
            input.put("quantity", order.getQuantity());
```

```
            interaction.execute(null, input); // Уменьшаем остаток в EIS
```

```
        }
```

```
        // 2. Работа с нашей БД через JPA
```

```
        em.persist(order); // Сохраняем заказ в нашей базе
```

```
        // --- КОНЕЦ ТРАНЗАКЦИИ ---
```

```
        // JTA-менеджер сервера приложений проведет двухфазный коммит (2PC)
```

```
        // с участием и адаптера ресурсов (EIS), и менеджера персистентности (БД).
```

```
        // Либо ОБА действия будут зафиксированы, либо ОБА будут отменены.
```

```
}  
}
```

1.11.29. Выводы

- **JCA — это "JDBC для корпоративных систем"**. Он предоставляет стандартный способ интеграции, скрывая детали реализации.
- **Ключевая роль — Адаптер Ресурсов (.rar файл)**, который поставляется вендором EIS.
- Сервер приложений управляет **пулами соединений, транзакциями и безопасностью**, что снимает эту головную боль с разработчика.
- **Главное преимущество — участие в распределенных JTA-транзакциях**, что позволяет обеспечить целостность данных при работе с несколькими системами.
- Прикладной код (используя CCI) становится стандартным и переносимым. Вы не привязаны к проприетарному API конкретной EIS.

Хотя JCA может показаться сложным, в средах, где необходима надежная интеграция с унаследованными системами, он является незаменимым и мощным инструментом. В современных системах его роль часто выполняют веб-сервисы (REST/SOAP), но для старых, не-веб EIS, JCA остается актуальным.

1.12. Безопасность и валидация данных

1.12.1. Jakarta Security: Основные концепции

Jakarta Security (ранее Java EE Security) — это спецификация, которая определяет стандартный API и модель для защиты приложений, работающих на сервере Jakarta EE.

Главная цель: Отделить логику безопасности от бизнес-логики. Разработчик должен фокусироваться на том, **что** приложение делает, а не на том, **как** оно проверяет пароли или права доступа. Эту работу на себя берет сервер приложений, а разработчик лишь **декларативно** указывает требования к безопасности.

Давайте разберем четыре кита, на которых держится безопасность.

Аутентификация (Authentication)

- **Что это?** Процесс проверки личности пользователя. Ответ на вопрос: "Кто вы?".

- **Как работает:** Пользователь предоставляет свои **учетные данные (credentials)** — например, логин и пароль, токен, сертификат. Система проверяет их подлинность по некоторому хранилищу пользователей (база данных, LDAP, и т.д.).
- **Результат:** Если аутентификация успешна, система создает объект, представляющий этого пользователя, и связывает его с текущим контекстом (например, с HTTP-сессией).

Принципал (Principal)

- **Что это?** Объект, который представляет **аутентифицированную** сущность (пользователя). Это результат успешной аутентификации.
- **Как представлен в Java:** Обычно это объект, реализующий интерфейс `java.security.Principal`. Самая простая реализация — `GenericPrincipal`.
- **Роль:** Принципал содержит имя пользователя и, что более важно, является носителем информации о его ролях.

Роли (Roles)

- **Что это?** Роль — это **группа разрешений**. Вместо того чтобы назначать права доступа каждому пользователю индивидуально (что очень сложно в управлении), пользователей группируют по ролям.
- **Примеры:** ADMIN, MANAGER, USER, GUEST.
- **Как работает:** Пользователю "John" назначается роль ADMIN. В коде или конфигурации мы говорим: "Только пользователи с ролью ADMIN могут вызывать этот метод". Нам не важно, что это "John", нам важна его роль. Это делает систему гибкой: если появится новый администратор "Jane", мы просто дадим ей роль ADMIN, не меняя код приложения.

Авторизация (Authorization)

- **Что это?** Процесс проверки прав доступа аутентифицированного пользователя (принципала) к определенному ресурсу. Ответ на вопрос: "**Что вам разрешено делать?**".
- **Как работает:** После успешной аутентификации, когда пользователь пытается выполнить какое-либо действие (вызвать EJB-метод, зайти на веб-страницу), система безопасности проверяет, есть ли у его принципала необходимая для этого действия роль.
- **Результат:** Если у пользователя есть нужная роль, доступ разрешается. Если нет — доступ запрещается (обычно возвращается ошибка 403 Forbidden для веба или бросается исключение EJBAccessException).

1.12.2. Примеры реализации в Jakarta EE

Jakarta Security позволяет применять эти концепции на разных уровнях приложения.

Шаг 1: Настройка механизма аутентификации и хранилища пользователей

Это делается **на уровне сервера приложений**, а не в коде. Вы должны сообщить серверу, как проверять пользователей.

- **Механизм аутентификации (Authentication Mechanism):** Как приложение будет получать учетные данные от пользователя. Стандартные механизмы:
 - **FORM:** Через HTML-форму логина.
 - **BASIC:** Через стандартное диалоговое окно браузера (небезопасно без HTTPS).
 - **CLIENT-CERT:** Через SSL-сертификаты.

- **Хранилище пользователей (Identity Store):** Где хранятся пользователи и их роли. Это может быть база данных, LDAP-каталог, или даже простой файл. В Jakarta EE это настраивается через SecurityDomain (в WildFly) или Realm (в Payara/GlassFish).

Пример конфигурации в web.xml (для веб-приложений):

```
<!-- src/main/webapp/WEB-INF/web.xml -->
<web-app ...>
  <!-- Описываем, как будет проходить аутентификация -->
  <login-config>
    <!-- Используем HTML-форму -->
    <auth-method>FORM</auth-method>
    <!-- Имя security-домена/реалма, настроенного на сервере -->
    <realm-name>my-app-realm</realm-name>
    <form-login-config>
      <!-- Страница, где находится форма логина -->
      <form-login-page>/login.xhtml</form-login-page>
      <!-- Страница, на которую перенаправят в случае ошибки -->
      <form-error-page>/login-error.xhtml</form-error-page>
    </form-login-config>
  </login-config>
  <!-- Декларируем роли, которые использует наше приложение -->
  <security-role>
    <role-name>ADMIN</role-name>
  </security-role>
  <security-role>
    <role-name>USER</role-name>
  </security-role>
</web-app>
```

Форма логина (login.xhtml):

Важно, что action у формы должен быть j_security_check, а имена полей — j_username и j_password. Это стандарт, который понимает сервер.


```

<form method="post" action="j_security_check">
  <input type="text" name="j_username" />
  <input type="password" name="j_password" />
  <button type="submit">Login</button>
</form>

```

Шаг 2: Применение авторизации (Декларативный способ)

Это самый простой и мощный способ. Вы просто аннотируете свои методы или классы.

Используются аннотации из пакета `jakarta.annotation.security`.

```

import jakarta.annotation.security.DeclareRoles;
import jakarta.annotation.security.DenyAll;
import jakarta.annotation.security.PermitAll;
import jakarta.annotation.security.RolesAllowed;
import jakarta.ejb.Stateless;

```

```
@Stateless
```

```
@DeclareRoles({"ADMIN", "MANAGER"}) // Объявляем роли,
используемые в этом EJB
```

```
public class AdminService {
```

```
    // Этот метод могут вызывать только пользователи с ролью "ADMIN"
```

```
    @RolesAllowed("ADMIN")
```

```
    public void performCriticalAdminTask() {
```

```
        System.out.println("Critical task performed.");
```

```
    }
```

```
    // Этот метод могут вызывать и ADMIN, и MANAGER
```

```
    @RolesAllowed({"ADMIN", "MANAGER"})
```

```
    public void generateReport() {
```

```
        System.out.println("Report generated.");
```

```
    }
```

```
    // Этот метод может вызывать любой аутентифицированный пользователь,
```

```
    // независимо от его роли.
```

```

@PermitAll
public String getSystemStatus() {
    return "System is OK.";
}
// Этот метод не может вызвать НИКТО, даже администратор.
@DenyAll
public void shutdownSystemImmediately() {
    // ...
}
}

```

Если пользователь без роли ADMIN попытается вызвать `performCriticalAdminTask()`, EJB-контейнер перехватит вызов и бросит `jakarta.ejb.EJBAccessException`.

Это делается в `web.xml`.

```

<!-- src/main/webapp/WEB-INF/web.xml -->
<web-app ...>
    <!-- ... login-config и security-role ... -->
    <!-- Защищаем URL-паттерн -->
    <security-constraint>
        <web-resource-collection>
            <!-- Название для группы защищаемых ресурсов -->
            <web-resource-name>Admin Area</web-resource-name>
            <!-- Все URL, начинающиеся с /admin/ -->
            <url-pattern>/admin/*</url-pattern>
        </web-resource-collection>
        <!-- Указываем, какие роли имеют доступ -->
        <auth-constraint>
            <role-name>ADMIN</role-name>
            <role-name>MANAGER</role-name>
        </auth-constraint>
    </security-constraint>

```

```

<security-constraint>
  <web-resource-collection>
    <web-resource-name>User Area</web-resource-name>
    <url-pattern>/user/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <!-- Доступ есть у всех трех ролей -->
    <role-name>USER</role-name>
    <role-name>ADMIN</role-name>
    <role-name>MANAGER</role-name>
  </auth-constraint>
</security-constraint>
</web-app>

```

Если неаутентифицированный пользователь попытается зайти на /admin/dashboard.xhtml, сервер перенаправит его на страницу логина. Если пользователь с ролью USER попытается зайти на ту же страницу, он получит ошибку 403 Forbidden.

Шаг 3: Получение информации о пользователе (Программный способ)

Иногда нужно в коде узнать, кто текущий пользователь или проверить его роль.

```
import jakarta.ejb.Stateless;
import jakarta.ejb.SessionContext;
import jakarta.annotation.Resource;

@Stateless
public class UserInfoService {
    // Внедряем SessionContext, который предоставляет информацию
    // о текущем вызове.
    @Resource
    private SessionContext sessionContext;

    public void showUserInfo() {
        // 1. Получаем Принципала
        java.security.Principal principal = sessionContext.getCallerPrincipal();
        if (principal != null) {
            System.out.println("Current user name: " + principal.getName());
        } else {
            System.out.println("No authenticated user.");
        }
        // 2. Проверяем роль программно
        if (sessionContext.isCallerInRole("ADMIN")) {
            System.out.println("User has ADMIN role. Granting special
            access.");
            // ... какая-то особая логика ...
        } else if (sessionContext.isCallerInRole("USER")) {
            System.out.println("User has USER role.");
        }
    }
}
```

}

Для веб-а аналогичную информацию можно получить из `HttpServletRequest`:
`request.getUserPrincipal()` и `request.isUserInRole("ADMIN")`.

Вывод: Jakarta Security предоставляет мощный, многоуровневый и декларативный подход к защите приложений. Основная работа по настройке хранилища пользователей и механизма аутентификации выносится на уровень сервера, а в коде приложения вы лишь указываете, каким ролям что разрешено, делая бизнес-логику чистой и не загроможденной проверками безопасности.

1.12.3. Современные механизмы аутентификации в Jakarta Security

Основной интерфейс в новой спецификации — это **HttpAuthenticationMechanism**. Это CDI-бин, который отвечает на три вопроса:

1. Как получить учетные данные из HTTP-запроса? (`validateRequest`)
2. Как поступить, если пользователь не аутентифицирован, но пытается получить доступ к защищенному ресурсу? (`challenge`)
3. Как очистить информацию об аутентификации (например, при выходе)? (`cleanSubject`)

К счастью, нам не нужно реализовывать этот интерфейс вручную для стандартных сценариев. Спецификация предоставляет готовые, простые в использовании аннотации, которые активируют встроенные реализации `HttpAuthenticationMechanism`.

Все эти аннотации применяются к одному классу-активатору, обычно простому сервлету или любому другому CDI-бину.

```

import jakarta.enterprise.context.ApplicationScoped;
import
jakarta.security.enterprise.authentication.mechanism.http.FormAuthenticationMechanism;

import
jakarta.security.enterprise.authentication.mechanism.http.LoginToContinue;
import jakarta.servlet.http.HttpServlet;

// Этот класс - просто точка для активации механизма аутентификации.
// Он может быть пустым.
@FormAuthenticationMechanism(
    loginToContinue = @LoginToContinue(
        loginPage = "/login.xhtml",
        errorPage = "/login-error.xhtml"
    )
)
@ApplicationScoped
public class ApplicationConfig {
    // В Jakarta EE 10+ этот класс даже не обязан быть сервлетом.
    // Достаточно быть CDI-бином.
}

```

Давайте рассмотрим основные встроенные механизмы.

1.12.4. @BasicAuthenticationMechanism

- **Что это:** Реализует стандартную HTTP Basic-аутентификацию.
- **Как работает:** Браузер показывает стандартное модальное окно с полями для логина и пароля. Учетные данные передаются в заголовке Authorization в формате Base64.
- **Когда использовать:** Для защиты REST API, веб-сервисов или для очень простых внутренних приложений.
- **Безопасность:** Крайне не рекомендуется использовать без HTTPS, так как пароль передается практически в открытом виде (Base64 — это кодирование, а не шифрование).

Пример кода

```
import jakarta.enterprise.context.ApplicationScoped;

import
jakarta.security.enterprise.authentication.mechanism.http.BasicAuthenticationMechanism;

@BasicAuthenticationMechanism(
    realmName = "my-app-realm" // Имя, которое отобразится в диалоговом окне
браузера
)
@ApplicationScoped
public class ApplicationConfig {
    // Пустой класс-активатор
}
```

Этого достаточно! Теперь при попытке доступа к защищенному ресурсу браузер покажет стандартное окно для ввода логина/пароля. Никакого web.xml не нужно.

1.12.5. @FormAuthenticationMechanism

- **Что это:** Реализует аутентификацию через HTML-форму. Это самый распространенный механизм для веб-приложений с пользовательским интерфейсом.
- **Как работает:** Пользователь вводит логин и пароль в кастомную HTML-форму. Сервер перехватывает POST-запрос с этой формы, извлекает учетные данные и проверяет их.
- **Ключевой параметр:** loginToContinue. Он определяет поведение "перенаправления на логин". Если неаутентифицированный пользователь пытается зайти на защищенную страницу /admin/dashboard, механизм перенаправит его на страницу логина, а после успешного входа вернет обратно на /admin/dashboard.

Пример кода

```
import jakarta.enterprise.context.ApplicationScoped;
```

```
import
jakarta.security.enterprise.authentication.mechanism.http.FormAuthenticationMechanism;

import
jakarta.security.enterprise.authentication.mechanism.http.LoginToContinue;
```

```
@FormAuthenticationMechanism(
    // Настраиваем поведение перенаправления
    loginToContinue = @LoginToContinue(
        loginPage = "/login.xhtml", // Страница с формой входа
        errorPage = "/login-error.xhtml", // Страница для отображения ошибок
        входа
        useForwardToLogin = false // Использовать redirect (302), а не forward.
        Рекомендуется.
    )
)
@ApplicationScoped
public class ApplicationConfig {
    // Пустой класс-активатор
}
```

HTML-форма (login.xhtml):

В отличие от старого подхода с `j_security_check`, здесь **нет никаких магических имен**. Вы можете использовать любые имена полей и любой action.


```
<!-- login.xhtml -->
<form method="post" action="#{request.contextPath}/login">
  <!-- action может быть любым -->
  <p>
    <label>Username:</label>
    <input type="text" name="username" /> <!-- имя поля может быть
любым -->
  </p>
  <p>
    <label>Password:</label>
    <input type="password" name="password" /> <!-- имя поля может быть
любым -->
  </p>
  <button type="submit">Log In</button>
</form>
```

Как это работает?

Механизм `@FormAuthenticationMechanism` не знает, какой URL или имена полей вы используете. Для фактической аутентификации вам нужно создать обработчик. Это делается с помощью `SecurityContext.authenticate()`.

Пример обработчика (например, в JSF backing bean):

```
import jakarta.enterprise.context.RequestScoped;
import jakarta.faces.application.FacesMessage;
import jakarta.faces.context.ExternalContext;
import jakarta.faces.context.FacesContext;
import jakarta.inject.Inject;
import jakarta.inject.Named;
import jakarta.security.enterprise.AuthenticationStatus;
import jakarta.security.enterprise.SecurityContext;
import jakarta.security.enterprise.credential.Credential;
import jakarta.security.enterprise.credential.Password;
import jakarta.security.enterprise.credential.UsernamePasswordCredential;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
```

```
@RequestScoped
```

```
@Named
```

```
public class LoginBean {
```

```
    @Inject
```

```
    private SecurityContext securityContext;
```

```
    @Inject
```

```
    private FacesContext facesContext;
```

```
    private String username;
```

```
    private String password;
```

```
    // getters/setters
```

```
    public void login() {
```

```
        Credential credential = new UsernamePasswordCredential(username, new
        Password(password));
```

```

// Вызываем аутентификацию программно
AuthenticationStatus status = securityContext.authenticate(
    (HttpServletRequest) facesContext.getExternalContext().getRequest(),
    (HttpServletResponse) facesContext.getExternalContext().getResponse(),
    Credential
);
if (status.equals(AuthenticationStatus.SEND_CONTINUE)) {
    // Успех! Механизм FormAuthenticationMechanism сам
    позаботится
    // о перенаправлении на исходную страницу.
    facesContext.responseComplete();
} else if (status.equals(AuthenticationStatus.SEND_FAILURE)) {
    // Неудача
    facesContext.addMessage(null, new
    FacesMessage(FacesMessage.SEVERITY_ERROR, "Invalid
    credentials", null));
}
}
}

```

Этот подход дает гораздо больше гибкости, чем старый `j_security_check`.

1.12.6. @CustomFormAuthenticationMechanism

- **Что это:** Вариант @FormAuthenticationMechanism, который позволяет использовать стандартное поведение j_security_check из старого web.xml, но в виде аннотации.
- **Когда использовать:** Для быстрой миграции старых приложений или если вы хотите избежать написания программного обработчика login(), как в примере выше.
- **Как работает:** Полностью имитирует старый механизм. Вы должны использовать action="j_security_check" и имена полей j_username, j_password.

Пример кода

```
import jakarta.enterprise.context.ApplicationScoped;

import
jakarta.security.enterprise.authentication.mechanism.http.CustomFormAuthentication
Mechanism;

import
jakarta.security.enterprise.authentication.mechanism.http.LoginToContinue;
```

```
@CustomFormAuthenticationMechanism(
    loginToContinue = @LoginToContinue(
        loginPage = "/login.xhtml",
        errorPage = "/login-error.xhtml"
    )
)
```

```
@ApplicationScoped
public class ApplicationConfig {
    // Пустой класс-активатор
}
```

HTML-форма (login.xhtml): Теперь имена полей и action должны быть строго определенными.

```
<form method="post" action="j_security_check">
    <input type="text" name="j_username" />
```

```

<input type="password" name="j_password" />
<button type="submit">Login</button>
</form>

```

Этот способ проще, так как не требует написания LoginBean, но он менее гибкий.

1.12.7. Создание своего собственного механизма

Если ни один из стандартных механизмов не подходит (например, вам нужна аутентификация по токену в заголовке X-API-Key), вы можете реализовать интерфейс HttpAuthenticationMechanism самостоятельно.

Пример (концептуальный):

```

import jakarta.enterprise.context.ApplicationScoped;
import jakarta.security.enterprise.AuthenticationStatus;
import
jakarta.security.enterprise.authentication.mechanism.http.HttpAuthenticationMechanism;
import
jakarta.security.enterprise.authentication.mechanism.http.HttpMessageContext;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;

@ApplicationScoped
public class MyTokenAuthMechanism implements
HttpAuthenticationMechanism {
    @Inject
    private MyTokenIdentityStore identityStore; // Ваше хранилище для проверки
ТОКЕНОВ

    @Override
    public AuthenticationStatus validateRequest(HttpServletRequest request,
HttpServletResponse response, HttpMessageContext httpMessageContext) {
        String token = request.getHeader("X-API-Key");

```

```

        if (token != null) {
            // Создаем кастомный Credential
            TokenCredential credential = new TokenCredential(token);

            // Проверяем учетные данные через IdentityStore
            // и уведомляем контейнер о результате
            return
            httpMessageContext.notifyContainerAboutLogin(identityStore.validate(c
            redential));
        }

        // Если токена нет, просто продолжаем, возможно, сработает другой
        механизм
        return httpMessageContext.doNothing();
    }
}

```

Чтобы этот механизм заработал, его нужно просто объявить как CDI-бин (`@ApplicationScoped`). Контейнер сам его найдет и будет использовать.

1.12.8. Заключение

Современный Jakarta Security API предоставляет гибкие и мощные механизмы аутентификации:

- **@BasicAuthenticationMechanism**: для API и простых сценариев.
- **@CustomFormAuthenticationMechanism**: простой, но негибкий способ для аутентификации по форме, совместимый со старыми приложениями.
- **@FormAuthenticationMechanism**: гибкий и мощный способ для аутентификации по форме, требующий программного вызова `SecurityContext.authenticate()`.
- **Собственная реализация HttpAuthenticationMechanism**: для любых нестандартных сценариев (токены, OAuth и т.д.).

1.12.9. Хранилища идентификационных данных (IdentityStore)

Что такое IdentityStore?

IdentityStore (Хранилище идентификационных данных) — это интерфейс, который представляет собой абстракцию над хранилищем пользователей (база данных, LDAP-каталог, файл, и т.д.). Его основная задача — ответить на два вопроса:

1. **Аутентификация:** Существует ли пользователь с такими учетными данными (Credential)? Если да, вернуть CredentialValidationResult с информацией о пользователе (имя, группы/роли).
2. **Авторизация:** Какие роли/группы есть у указанного пользователя?

Ключевая идея: Вы отделяете логику проверки паролей и получения ролей от остального приложения. Сервер приложений через HttpAuthenticationMechanism получает учетные данные и передает их в IdentityStore для проверки.

Jakarta Security предоставляет два удобных способа определения IdentityStore:

1. **Декларативный (с помощью аннотаций):** Для стандартных случаев (хранение в БД или LDAP) можно использовать готовые аннотации. Это самый простой способ.
2. **Программный (реализация интерфейса):** Для нестандартных случаев (например, хранение паролей в другом формате, обращение к внешнему веб-сервису для аутентификации) вы можете реализовать интерфейс IdentityStore самостоятельно.

1.12.10. Декларативные хранилища: @DataBaseIdentityStoreDefinition

Это самый распространенный способ. Он позволяет настроить аутентификацию по базе данных с помощью всего одной аннотации.

- **Как работает:** Вы указываете JNDI-имя вашего DataSource, SQL-запросы для получения пароля и ролей пользователя, а также алгоритм хеширования паролей.

Пример

Предположим, у нас есть две таблицы в базе данных:

Таблица app_users:

username (PK)	password_hash
john	PBKDF2WithHmacSHA256:2048:AbCd...==:XyZ...== (хеш)
admin	PBKDF2WithHmacSHA256:2048:EfGh...==:WvX...== (хеш)

Таблица user_roles:

username (FK)	role_name
john	USER
admin	USER
admin	ADMIN

Код с аннотацией:

Эту аннотацию можно разместить на любом классе-активаторе (как и механизмы аутентификации).

```
import jakarta.enterprise.context.ApplicationScoped;
import jakarta.security.enterprise.identitystore.DatabaseIdentityStoreDefinition;
import jakarta.security.enterprise.identitystore.Pbkdf2PasswordHash;

@DatabaseIdentityStoreDefinition(
    // 1. Указываем JNDI-имя источника данных, настроенного на сервере
    dataSourceLookup = "java:jboss/datasources/MyWebAppDS",
    // 2. Запрос для получения хеша пароля пользователя по его имени
    callerQuery = "SELECT password_hash FROM app_users WHERE username =
?",
    // 3. Запрос для получения списка ролей пользователя по его имени
    groupsQuery = "SELECT role_name FROM user_roles WHERE username = ?",
    // 4. Указываем алгоритм хеширования, который используется для
хранения паролей
    hashAlgorithm = Pbkdf2PasswordHash.class,
    // 5. Параметры для алгоритма хеширования (должны совпадать с теми, что
использовались при создании хеша)
    hashAlgorithmParameters = {
        "Pbkdf2PasswordHash.Iterations=2048",
        "Pbkdf2PasswordHash.Algorithm=PBKDF2WithHmacSHA256",
        "Pbkdf2PasswordHash.SaltSizeBytes=32"
    },
    priority = 30 // Приоритет хранилища, если их несколько
)
@ApplicationScoped
public class ApplicationConfig {
    // Пустой класс-активатор
}
```

Что здесь происходит?

1. Когда пользователь вводит логин "john" и пароль "password123", механизм аутентификации передает их в это хранилище.
2. DataBaseIdentityStoreDefinition выполняет callerQuery, подставляя "john". Он получает из БД хеш пароля для "john".
3. Затем он берет пароль "password123", хеширует его с теми же параметрами (Pbkdf2PasswordHash, 2048 итераций) и сравнивает с хешем из БД.
4. Если хеши совпадают, аутентификация успешна.
5. Далее он выполняет groupsQuery, подставляя "john", и получает его роль (USER).
6. Вся эта информация (имя пользователя "john" и роль "USER") упаковывается и передается обратно в контекст безопасности.

1.12.11. Декларативные хранилища: @LdapIdentityStoreDefinition

Этот способ используется для интеграции с корпоративными LDAP-каталогами (например, Microsoft Active Directory).

- **Как работает:** Вы указываете данные для подключения к LDAP-серверу и шаблоны для поиска пользователя и его групп.

Пример

Generated java

```
import jakarta.enterprise.context.ApplicationScoped;
```

```
import jakarta.security.enterprise.identitystore.LdapIdentityStoreDefinition;
```

```
@LdapIdentityStoreDefinition(
```

```
    // 1. URL LDAP-сервера
```

```
    url = "ldap://localhost:10389/",
```

```
    // 2. Учетные данные для подключения к LDAP (если не анонимный доступ)
```

```
    binderDn = "uid=admin,ou=system",
```

```
    binderDnPassword = "secret",
```

```
    // 3. Шаблон для поиска пользователя (DN - Distinguished Name)
```

```
    // {uid} будет заменено на имя пользователя, введенное в форме
```

```
    callerDn = "uid={uid},ou=users,ou=system",
```

```
    // 4. База для поиска групп
```

```
    groupSearchBase = "ou=groups,ou=system",
```

```
    // 5. Атрибут, в котором хранится имя группы
```

```
    groupNameAttribute = "cn",
```

```
    // 6. Атрибут в записи группы, который содержит членов группы
```

```
    groupMemberAttribute = "member"
```

```
)
```

```
@ApplicationScoped
```

```
public class ApplicationConfig {
```

```
    // Пустой класс-активатор
```

```
}
```

Эта аннотация позволяет интегрироваться с LDAP без написания сложного кода для работы с JNDI и LDAP API.

1.12.12. Программное хранилище: реализация IdentityStore

Если стандартных declarative-хранилищ недостаточно, вы можете реализовать интерфейс IdentityStore самостоятельно. Это дает максимальную гибкость.

Сценарии:

- Хранение пользователей в NoSQL базе данных.
- Аутентификация через внешний OAuth/OpenID Connect провайдер.
- Использование кастомного алгоритма хеширования.
- Комбинирование нескольких источников данных для получения ролей.

Пример: Простое хранилище в памяти (для тестов)

```
import jakarta.enterprise.context.ApplicationScoped;
import jakarta.security.enterprise.credential.Credential;
import jakarta.security.enterprise.credential.UsernamePasswordCredential;
import jakarta.security.enterprise.identitystore.CredentialValidationResult;
import jakarta.security.enterprise.identitystore.IdentityStore;
```

```
import java.util.Arrays;
import java.util.Collections;
import java.util.HashSet;
```

```
@ApplicationScoped // Объявляем как CDI-бин, чтобы контейнер его нашел
public class InMemoryIdentityStore implements IdentityStore {
    @Override
    public CredentialValidationResult validate(Credential credential) {
        if (credential instanceof UsernamePasswordCredential) {
            UsernamePasswordCredential login =
                (UsernamePasswordCredential) credential;
            String username = login.getCaller();
            String password = login.getPassword().getValue();

```

```

        // Простая проверка в памяти (НИКОГДА не делайте так в
        продакшене!)
        if ("john".equals(username) && "password123".equals(password))
        {
            // Успех! Возвращаем результат с именем пользователя и
            его ролями.
            return new CredentialValidationResult("john", new
            HashSet<>(Arrays.asList("USER")));
        }

        if ("admin".equals(username) && "supersecret".equals(password))
        {
            return new CredentialValidationResult("admin", new
            HashSet<>(Arrays.asList("USER", "ADMIN")));
        }
    }
    // Если аутентификация не удалась
    return CredentialValidationResult.INVALID_RESULT;
}
}

```

Важно: Метод `getGroups` тоже является частью интерфейса `IdentityStore` и может быть реализован для получения ролей отдельно от процесса аутентификации.

1.12.13. Комбинирование нескольких хранилищ

Вы можете определить **несколько** `IdentityStore` в одном приложении. Контейнер будет опрашивать их в порядке, заданном атрибутом `priority` (меньшее число — выше приоритет). Это позволяет, например, сначала проверить пользователя в базе данных, а если его там нет, то попробовать найти в LDAP.

Вывод:

`IdentityStore` — это мощная абстракция, которая отделяет логику проверки учетных данных от остальной части приложения.

- Используйте **@DataBaseIdentityStoreDefinition** для аутентификации по БД — это самый частый и простой случай.

- Используйте **@LdapIdentityStoreDefinition** для интеграции с корпоративными каталогами.
- Реализуйте **IdentityStore** вручную для любых нестандартных сценариев, требующих полной гибкости.

1.12.14. Декларативная безопасность в Jakarta EE

Основная идея — вы описываете **политику безопасности** с помощью аннотаций, а EJB-контейнер или веб-контейнер обеспечивает ее соблюдение во время выполнения.

Стандартные аннотации для декларативной безопасности находятся в пакете `jakarta.annotation.security`.

1.12.15. @DeclareRoles

- **Назначение:** Объявляет список ролей, которые используются **программно** внутри компонента (например, в вызове `sessionContext.isCallerInRole(...)`).
- **Зачем это нужно:** Это своего рода "подсказка" для контейнера и для инструментов развертывания. Аннотация говорит: "Этот компонент в своем коде будет проверять вот эти роли. Убедитесь, что они существуют в приложении".
- **Где применяется:** На уровне класса (EJB или сервлета).
- **Важно:** Эта аннотация **не предоставляет никаких прав доступа**. Она просто объявляет о намерениях. Фактическое назначение ролей пользователям происходит на уровне сервера.

Пример:

```
import jakarta.annotation.Resource;
import jakarta.annotation.security.DeclareRoles;
import jakarta.ejb.SessionContext;
import jakarta.ejb.Stateless;

// Объявляем, что в коде этого бина мы будем проверять роли
// "SUPERVISOR" и "AUDITOR"
@Stateless
@DeclareRoles({"SUPERVISOR", "AUDITOR"})
public class ReportingService {
    @Resource
    private SessionContext ctx;

    public Report generateComplexReport() {
        Report report = new Report();
        // ... основная логика генерации отчета ...
        // Проверяем роль программно, чтобы добавить дополнительную
        // информацию
        if (ctx.isCallerInRole("SUPERVISOR")) {
            report.addSupervisoryNotes("Generated by a supervisor.");
        }
    }
}
```

```

    }
    if (ctx.isCallerInRole("AUDITOR")) {
        report.setForAudit(true);
    }
    return report;
}
}

```

Без `@DeclareRoles` вызов `isCallerInRole` может работать некорректно в некоторых окружениях, так как контейнер не будет "знать" об этих ролях заранее.

1.12.16. `@RolesAllowed`

- **Назначение:** Это основная аннотация для **авторизации**. Она указывает, какие роли необходимы для вызова метода или всех методов класса.
- **Как работает:** Перед вызовом метода контейнер проверяет, имеет ли текущий аутентифицированный пользователь хотя бы одну из перечисленных в аннотации ролей.
 - Если **да** — вызов метода разрешается.
 - Если **нет** — вызов блокируется, и бросается исключение `jakarta.ejb.EJBAccessException` (для EJB) или возвращается HTTP-статус 403 Forbidden (для веба).
- **Где применяется:** На уровне метода или класса. Если аннотация стоит на классе, она применяется ко всем его публичным методам. Аннотация на методе переопределяет аннотацию на классе.

Пример для EJB:

```
import jakarta.annotation.security.RolesAllowed;
import jakarta.ejb.Stateless;
```

```
@Stateless
```

```
@RolesAllowed("MANAGER") // По умолчанию, все методы доступны
только для MANAGER
```

```
public class EmployeeManagementService {
    // Этот метод унаследует @RolesAllowed("MANAGER") от класса
    public Employee findEmployee(long id) {
        // ...
    }
}
```

```
// Здесь мы переопределяем права доступа.
```

```
// Этот метод доступен и для MANAGER, и для HR_SPECIALIST
```

```
@RolesAllowed({"MANAGER", "HR_SPECIALIST"})
```

```
public void hireEmployee(EmployeeData data) {
    // ...
}
```

```
// А этот метод доступен только для роли, которой нет ни у кого из обычных
```

```
// менеджеров. Только для супер-администратора.
```

```
@RolesAllowed("SYSTEM_ADMIN")
```

```
public void deleteAllEmployees() {
    // ... очень опасная операция ...
}
}
```

Пример для Веб-компонентов (JAX-RS)

Аннотации безопасности прекрасно работают и с REST-контроллерами.

```
import jakarta.annotation.security.RolesAllowed;
import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.Produces;
import jakarta.ws.rs.core.MediaType;
```

```
@Path("/reports")
```

```
@RolesAllowed("VIEWER") // Все методы для просмотра отчетов требуют
роль VIEWER
```

```
public class ReportResource {
    @GET
    @Path("/public")
    @Produces(MediaType.TEXT_PLAIN)
    public String getPublicReport() {
        // Этот метод унаследует @RolesAllowed("VIEWER")
        return "This is a report for all viewers.";
    }
}
```

```
@GET
@Path("/financial")
@Produces(MediaType.TEXT_PLAIN)
@RolesAllowed("FINANCE_MANAGER") // Только финансовые
менеджеры могут видеть этот отчет
public String getFinancialReport() {
    return "This is a confidential financial report.";
}
}
```

1.12.17. @PermitAll

- **Назначение:** Явно разрешает доступ к ресурсу **всем**, включая **неаутентифицированных** (анонимных) пользователей.
- **Как работает:** Отключает проверку безопасности для конкретного метода или класса.
- **Где применяется:** На уровне метода или класса. Часто используется для "открытия" отдельных методов в защищенном по умолчанию классе.

Пример:

```
import jakarta.annotation.security.PermitAll;
import jakarta.annotation.security.RolesAllowed;
import jakarta.ejb.Stateless;

@Stateless
// По умолчанию весь сервис требует аутентификации с ролью USER
@RolesAllowed("USER")
public class ProductService {
    // Этот метод доступен только для пользователей с ролью USER
    public ProductDetails getProductDetails(long id) {
        // ...
    }
    // А этот метод мы делаем публичным. Его может вызвать любой,
    // например, для отображения списка товаров на главной странице.
    @PermitAll
    public List<ProductSummary> listAllProducts() {
        // ...
    }
}
```

1.12.18. @DenyAll

- **Назначение:** Явно **запрещает доступ всем**, включая администраторов и пользователей с любыми ролями.
- **Как работает:** Гарантирует, что метод никогда не будет вызван через внешний интерфейс компонента.

- **Где применяется:** На уровне метода или класса.
- **Сценарий использования:** Для методов, которые находятся в процессе разработки и еще не готовы к использованию, или для устаревших (deprecated) методов, которые должны быть отключены, но еще не удалены из кода. Также для "внутренних" public методов, которые не должны быть доступны извне, а вызываются только другими методами этого же класса.

Пример:

```
import jakarta.annotation.security.DenyAll;
import jakarta.ejb.Stateless;
```

```
@Stateless
```

```
public class LegacyPaymentService {
    public void processNewPayment(PaymentData data) {
        // ... логика нового, работающего метода ...
    }
    // Этот метод устарел и содержит уязвимость.
    // Мы запрещаем его вызов, чтобы никто случайно его не использовал.
```

```
@DenyAll
```

```
public void processOldUnsafePayment(String rawPaymentData) {
    // Этот код никогда не будет выполнен при вызове извне.
    // Будет брошено EJBAccessException.
}
}
```

1.12.19. Иерархия и приоритеты

1. Аннотация на **методе** всегда имеет приоритет над аннотацией на **классе**.
2. @DenyAll — самый сильный. Если он есть, доступ запрещен.
3. @PermitAll разрешает доступ, если нет @DenyAll.
4. @RolesAllowed разрешает доступ, если нет ни @DenyAll, ни @PermitAll.

5. Если нет ни одной из этих аннотаций, применяется политика по умолчанию для контейнера, которая обычно означает, что доступ разрешен (если не настроено иначе). Поэтому **хорошей практикой является установка `@RolesAllowed` или `@DenyAll` на уровне класса, а затем выборочное открытие доступа к методам.**

Декларативная безопасность — это мощный инструмент, который делает код чище, безопаснее и проще в поддержке, перенося логику авторизации из императивного кода в декларативные метаданные.

1.12.20. Программная безопасность: `SecurityContext`

Программная безопасность — это способ взаимодействия с контекстом безопасности непосредственно из вашего Java-кода. Вы можете получить информацию о текущем пользователе, проверить его роли и на основе этого принять решение о доступе.

Основным инструментом для этого в современной Jakarta Security является интерфейс `jakarta.security.enterprise.SecurityContext`.

Что такое `SecurityContext`?

`SecurityContext` — это CDI-бин, который предоставляет унифицированный доступ к информации о безопасности текущего вызова, независимо от того, где он выполняется (в сервлете, EJB, JAX-RS ресурсе и т.д.).

Основные методы `SecurityContext`:

- `getCallerPrincipal()`: Возвращает объект `java.security.Principal`, представляющий текущего аутентифицированного пользователя. Если пользователь не аутентифицирован, возвращает `null`.
- `isCallerInRole(String roleName)`: Проверяет, имеет ли текущий пользователь указанную роль. Возвращает `true` или `false`.
- `hasAccessToWebResource(String resource, String... methods)`: Проверяет, имеет ли текущий пользователь доступ к указанному веб-ресурсу (URL) с указанными HTTP-методами, согласно правилам, определенным в `web.xml` или других конфигурациях.
- `authenticate(...)`: Программно инициирует процесс аутентификации (как мы видели в теме про `FormAuthenticationMechanism`).

Как получить `SecurityContext`?

Поскольку это CDI-бин, вы можете просто внедрить его с помощью `@Inject` в любой другой компонент, управляемый CDI (EJB, сервлет, JAX-RS ресурс и т.д.).

```
import jakarta.enterprise.context.ApplicationScoped;
```

```
import jakarta.inject.Inject;
import jakarta.security.enterprise.SecurityContext;
```

```
@ApplicationScoped
public class MyService {
    @Inject
    private SecurityContext securityContext;
    // ...
}
```

1.12.21. Сценарии использования программной безопасности

Сценарий 1: Проверка "владельца" объекта

Декларативная безопасность отлично проверяет роли, но она не может проверить, является ли текущий пользователь владельцем запрашиваемого объекта.

Задача: Пользователь с ролью USER может редактировать свой собственный профиль, но не профиль другого пользователя. Администратор с ролью ADMIN может редактировать любой профиль.

```
import jakarta.ejb.Stateless;
import jakarta.inject.Inject;
import jakarta.persistence.EntityManager;
import jakarta.persistence.PersistenceContext;
import jakarta.security.enterprise.SecurityContext;
import java.security.Principal;
```

```
@Stateless
public class UserProfileService {
    @PersistenceContext
    private EntityManager em;
```

```
@Inject
    private SecurityContext securityContext;
```

```

public void updateProfile(long profileId, UserProfileData data) {
    // Получаем имя текущего аутентифицированного пользователя
    Principal caller = securityContext.getCallerPrincipal();
    if (caller == null) {
        throw new SecurityException("Authentication required.");
    }
    String callerUsername = caller.getName();
    // Загружаем профиль, который пытаются изменить
    UserProfile profileToUpdate = em.find(UserProfile.class, profileId);
    if (profileToUpdate == null) {
        throw new NotFoundException("Profile not found.");
    }
    // --- Логика программной безопасности ---
    // Проверяем, является ли текущий пользователь владельцем
    // профиля
    boolean isOwner =
    callerUsername.equals(profileToUpdate.getUsername());
    // Проверяем, имеет ли пользователь роль администратора
    boolean isAdmin = securityContext.isCallerInRole("ADMIN");
    // Разрешаем доступ, только если пользователь - владелец ИЛИ
    // администратор
    if (!isOwner && !isAdmin) {
        throw new SecurityException("User " + callerUsername + " is not
        allowed to modify profile " + profileId);
    }
    // --- Конец логики безопасности ---
    // Если проверки пройдены, выполняем обновление
    profileToUpdate.setFullName(data.getFullName());
    profileToUpdate.setBio(data.getBio());
    em.merge(profileToUpdate);
}

```

}

В этом примере мы комбинируем декларативную безопасность (проверка роли ADMIN) и программную (проверка имени владельца), чтобы реализовать сложную бизнес-логику авторизации.

Сценарий 2: Динамическое отображение UI

В зависимости от роли пользователя, мы можем захотеть показать или скрыть определенные элементы на веб-странице (например, кнопку "Удалить").

Пример с использованием JSF (Jakarta Server Faces): SecurityContext можно внедрить прямо в backing-бин для JSF-страницы.

Backing-бин (DocumentBean.java):

```
import jakarta.enterprise.context.RequestScoped;
import jakarta.inject.Inject;
import jakarta.inject.Named;
import jakarta.security.enterprise.SecurityContext;

@RequestScoped
@Named("docBean")
public class DocumentBean {
    @Inject
    private SecurityContext securityContext;

    public boolean canDeleteDocument() {
        // Логика может быть сложной. Например, удалять может
        // только автор документа ИЛИ пользователь с ролью
        // 'MODERATOR'.
        String currentUser = securityContext.getCallerPrincipal().getName();
        String documentAuthor = getCurrentDocument().getAuthor(); //
        // Получаем автора текущего документа
        boolean isAuthor = currentUser.equals(documentAuthor);
        boolean isModerator =
            securityContext.isCallerInRole("MODERATOR");
        return isAuthor || isModerator;
    }

    // ...
}
```

JSF-страница (document.xhtml):

Мы используем Expression Language (EL) для вызова метода из нашего бина.

```
<h:body>
  <h1>Document: #{docBean.currentDocument.title}</h1>
  <p>Author: #{docBean.currentDocument.author}</p>
  <!-- Кнопка "Удалить" будет отображена только если
docBean.canDeleteDocument() вернет true -->
  <h:commandButton value="Delete Document"
    action="#{docBean.delete}"
    rendered="#{docBean.canDeleteDocument()}" />
</h:body>
```

Старый API: `SessionContext` и `HttpServletRequest`

До появления `SecurityContext` в Jakarta EE 8, для программной безопасности использовались другие интерфейсы:

- **В EJB:** `jakarta.ejb.SessionContext`
 - `sessionContext.getCallerPrincipal()`
 - `sessionContext.isCallerInRole(roleName)`
- **В сервлетах и JAX-RS:** `jakarta.servlet.http.HttpServletRequest`
 - `request.getUserPrincipal()`
 - `request.isUserInRole(roleName)`

Эти методы все еще работают и являются валидными, но `SecurityContext` предоставляет единый, унифицированный API для всех типов компонентов, что делает его предпочтительным в современных приложениях.

1.12.22. Заключение

Подход	Когда использовать	Преимущества	Недостатки
Декларативная безопасность	В 90% случаев. Когда авторизация основана только на статических ролях пользователя.	Простота, чистота кода, надежность (обеспечивается контейнером), переносимость.	Негибкая, не может обрабатывать динамические условия, зависящие от данных.
Программная безопасность	Когда авторизация зависит от состояния приложения (например, "кто владелец объекта?"). Для динамического рендеринга UI.	Максимальная гибкость, позволяет реализовать любую сложную логику авторизации.	Усложняет код, смешивает бизнес-логику с логикой безопасности, выше риск ошибки.

Лучшая практика — использовать оба подхода вместе. Применяйте декларативную безопасность для грубой, ролевой фильтрации на уровне методов, а программную безопасность — для тонкой, специфичной для данных проверки внутри этих методов.

1.12.23. OWASP Top 10 и защита в Jakarta EE

OWASP Top 10 — это стандартный документ, повышающий осведомленность о наиболее критических рисках безопасности веб-приложений. Список периодически обновляется. Мы рассмотрим несколько классических и всегда актуальных уязвимостей.

1. A01:2021 – Broken Access Control (Нарушенный контроль доступа)

- **Что это:** Уязвимость, при которой пользователи могут получить доступ к данным или функциям, на которые у них нет прав. Например, обычный пользователь может получить доступ к панели администратора, просто угадав URL.
- **Как Jakarta EE помогает:**

- **Декларативная безопасность:** Это основной инструмент. С помощью `@RolesAllowed` или `security-constraint` в `web.xml` вы можете жестко привязать роли к EJB-методам и URL-адресам. Контейнер берет на себя всю работу по проверке.
- **Программная безопасность (SecurityContext):** Для более сложных случаев (например, "пользователь может редактировать только свой заказ") вы можете использовать `securityContext.getCallerPrincipal()` для проверки владельца объекта.

Пример (Декларативная защита):

```
// JAX-RS ресурс
@Path("/admin")
@RolesAllowed("ADMIN") // Весь этот ресурс доступен ТОЛЬКО для роли
ADMIN

public class AdminResource {
    @GET
    @Path("/dashboard")
    public String getAdminDashboard() {
        // Если пользователь без роли ADMIN попытается вызвать этот метод
        // (например, через GET /api/admin/dashboard),
        // контейнер вернет 403 Forbidden.
        return "Welcome to the admin dashboard!";
    }
}
```

Без этой аннотации разработчику пришлось бы вручную писать `if (request.isUserInRole("ADMIN")) { ... }`, и если он забудет это сделать, появится уязвимость. Jakarta EE делает безопасность неотъемлемой частью компонента.

2. A03:2021 – Injection (Внедрение), в частности SQL-инъекции

- **Что это:** Злоумышленник внедряет вредоносный код (чаще всего SQL) в поля ввода, чтобы обмануть приложение и заставить его выполнить непредусмотренные команды.
- **Как Jakarta EE помогает:**
 - **Jakarta Persistence (JPA):** Это главный защитник от SQL-инъекций. При использовании **параметризованных запросов** (JPQL, Criteria API, или `Query.setParameter()`) JPA никогда не конкатенирует (не "склеивает") строки. Она передает данные в базу данных отдельно от самого SQL-запроса, что делает инъекцию невозможной.

Плохой, уязвимый код (так делать НЕЛЬЗЯ):

// НЕ ДЕЛАЙТЕ ТАК!

```
String username = request.getParameter("username");
```

```
String jpql = "SELECT u FROM User u WHERE u.name = '" + username + "'"; //
```

Уязвимость!

```
List<User> users = em.createQuery(jpql, User.class).getResultList();
```

Если злоумышленник введет `John' OR '1'='1`, запрос превратится в ... `WHERE u.name = 'John' OR '1'='1'`, что вернет всех пользователей.

Хороший, безопасный код с JPA:

```
String username = request.getParameter("username");
```

```
String jpql = "SELECT u FROM User u WHERE u.name = :uname";
```

// Используем именованный параметр

```
List<User> users = em.createQuery(jpql, User.class)
```

```
.setParameter("uname", username) // Передаем данные отдельно
```

```
.getResultList();
```

В этом случае JPA передаст в БД запрос и параметр отдельно. База данных никогда не будет исполнять данные из параметра как часть SQL-кода.

3. A02:2021 – Cryptographic Failures (Криптографические сбои), в частности хранение паролей

- **Что это:** Неправильное использование криптографии, например, хранение паролей в открытом виде или с использованием слабых, устаревших алгоритмов хеширования (таких как MD5 или SHA1).
- **Как Jakarta EE помогает:**
 - **Jakarta Security (IdentityStore):** Спецификация активно продвигает использование современных, сильных и "соленых" алгоритмов хеширования. @DatabaseIdentityStoreDefinition имеет встроенную поддержку **PBKDF2**, который является индустриальным стандартом для хеширования паролей.
 - **Пароль никогда не хранится в открытом виде.** Хранится только его необратимый хеш.

Пример (@DatabaseIdentityStoreDefinition):

```
@DatabaseIdentityStoreDefinition(  
    // ...  
    // Явно указываем, что пароли в БД хешированы с помощью PBKDF2  
    hashAlgorithm = Pbkdf2PasswordHash.class,  
    hashAlgorithmParameters = {  
        "Pbkdf2PasswordHash.Iterations=310000", // Большое количество  
        итераций  
        "Pbkdf2PasswordHash.Algorithm=PBKDF2WithHmacSHA512", //  
        Сильный алгоритм  
        "Pbkdf2PasswordHash.SaltSizeBytes=64" // Длинная "соль"  
    }  
)
```

Это заставляет разработчиков использовать правильные криптографические практики по умолчанию.

4. A07:2021 – Identification and Authentication Failures (Сбои идентификации и аутентификации)

- **Что это:** Уязвимости, связанные с неправильной реализацией механизма входа/выхода, управления сессиями. Например, предсказуемые ID сессий, отсутствие защиты от перебора паролей (brute-force).
- **Как Jakarta EE помогает:**
 - **Управление сессиями:** Веб-контейнер (например, Tomcat, Undertow) сам управляет жизненным циклом сессий (HttpSession). Он генерирует длинные, криптографически случайные идентификаторы сессий, что защищает от их подбора.
 - **Стандартные механизмы аутентификации:** Используя @FormAuthenticationMechanism и SecurityContext, вы полагаетесь на проверенную временем реализацию от вендора сервера приложений, а не изобретаете свой собственный, потенциально уязвимый механизм.
 - **Защита от brute-force:** Хотя сама Jakarta EE не предоставляет встроенной защиты от перебора паролей, IdentityStore можно легко расширить для ее реализации. Например, в кастомном IdentityStore можно считать количество неудачных попыток входа для пользователя и временно блокировать его.

5. A05:2021 – Security Misconfiguration (Неправильная конфигурация безопасности)

- **Что это:** Ошибки, связанные с небезопасными настройками по умолчанию, открытыми портами, подробными сообщениями об ошибках, которые раскрывают внутреннюю структуру приложения.
- **Как Jakarta EE помогает:**
 - **Разделение ответственности:** Конфигурация сервера (порты, security-домены, datasource) отделена от кода приложения. Этим занимается администратор, а не разработчик.
 - **Обработка ошибок:** Фреймворки вроде Jakarta Faces (JSF) позволяют легко настроить кастомные страницы ошибок, чтобы не показывать пользователю стектрейсы Java.

Пример в web.xml:

```
<error-page>
  <!-- Для всех исключений типа java.lang.Throwable -->
  <exception-type>java.lang.Throwable</exception-type>
  <!-- Показывать эту страницу -->
  <location>/error.xhtml</location>
</error-page>
<error-page>
  <!-- Для ошибки 404 (Not Found) -->
  <error-code>404</error-code>
  <location>/not-found.xhtml</location>
</error-page>
```

6. Уязвимости, связанные с Cross-Site Scripting (XSS)

- **Что это:** Злоумышленник внедряет вредоносный JavaScript-код на страницу, который затем выполняется в браузере другого пользователя.
- **Как Jakarta EE помогает:**
 - **Jakarta Server Faces (JSF):** Этот UI-фреймворк по умолчанию **экранирует (escapes)** все данные, которые выводятся на страницу через его компоненты (например, `<h:outputText>`). Это означает, что если в переменной `user.name` будет строка `<script>alert('XSS')</script>`, JSF выведет ее на страницу как обычный текст `<script>alert('XSS')</script>`, и браузер не будет исполнять его как скрипт.
 - **Jakarta Expression Language (EL) 3.0+:** Также имеет встроенные механизмы экранирования.

Пример JSF (безопасно по умолчанию):

```
<!-- Предположим, aBean.unsafeData содержит "<script>alert('XSS')</script>" -->

<!-- JSF автоматически экранирует вывод. Вредоносный скрипт не выполнится. -->

<h:outputText value="#{aBean.unsafeData}" />
```

7. Уязвимости, связанные с Cross-Site Request Forgery (CSRF)

- **Что это:** Злоумышленник заставляет браузер аутентифицированного пользователя выполнить нежелательное действие на другом сайте. Например, вы залогинены в своем банке, а на другом сайте есть картинка ``. Браузер попытается загрузить эту "картинку", отправив запрос на перевод денег вместе с вашими cookies.
- **Как Jakarta EE помогает:**
 - **Jakarta Faces (JSF):** JSF автоматически включает в каждую форму скрытое поле с уникальным, случайным токеном (CSRF-токен). При отправке формы сервер проверяет, что этот токен совпадает с тем, который он ожидает для данной сессии. Если злоумышленник попытается подделать форму на своем сайте, у него не будет этого токена, и запрос будет отклонен.
 - **Jakarta Security API:** Позволяет программно генерировать и проверять CSRF-токены для защиты, например, JAX-RS эндпоинтов, которые изменяют состояние.

1.12.24. Заключение

Jakarta EE не является "серебряной пулей", которая магически защищает от всех уязвимостей. Однако, благодаря своей многоуровневой архитектуре и следованию принципам "безопасность по умолчанию" (secure by default), она предоставляет разработчикам мощный набор инструментов и практик:

- **JPA** защищает от SQL-инъекций.
- **Jakarta Security** стандартизирует аутентификацию, авторизацию и криптографию.
- **JSF** защищает от XSS и CSRF по умолчанию.
- **Контейнер** управляет сессиями и обеспечивает соблюдение декларативных правил безопасности.

Используя эти инструменты правильно, можно значительно повысить уровень защищенности приложения и избежать многих распространенных ошибок.

1.12.25. Jakarta Bean Validation: Декларативная валидация

Что это такое?

Jakarta Bean Validation (ранее Bean Validation) — это спецификация, которая предоставляет стандартный способ для валидации (проверки корректности) объектов Java (JavaBeans) с помощью **декларативных аннотаций**.

Основная идея: Вместо того чтобы писать громоздкие и повторяющиеся проверки `if (user.getEmail() == null || !user.getEmail().contains("@")) { ... }` в разных частях вашего кода, вы один раз описываете **ограничения (constraints)** прямо на полях вашего класса-модели. А затем на разных уровнях приложения (веб-контроллер, бизнес-сервис) вы можете инициировать проверку объекта на соответствие этим ограничениям.

Это позволяет:

- **Следовать принципу DRY (Don't Repeat Yourself):** Правила валидации определены в одном месте — рядом с данными.
- **Сделать код чище:** Бизнес-логика не загромождена кодом валидации.
- **Обеспечить целостность данных:** Одни и те же правила применяются везде, где используется объект.

1.12.26. Стандартные аннотации ограничений

Спецификация предоставляет богатый набор встроенных аннотаций, которые находятся в пакете `jakarta.validation.constraints`.

Самые распространенные аннотации:

Для проверки на null/пустоту:

- **@NotNull:** Поле не должно быть null.
- **@NotEmpty:** Для строк, коллекций, массивов. Не должно быть null и не должно быть пустым (например, `size() > 0`).
- **@NotBlank:** Только для строк. Не должно быть null и не должно состоять только из пробельных символов.

Для проверки размера/длины:

- **@Size(min = ..., max = ...):** Проверяет размер для строк, коллекций, массивов.

Для числовых значений:

- **@Min(value = ...):** Минимальное значение (включительно).
- **@Max(value = ...):** Максимальное значение (включительно).
- **@Positive, @PositiveOrZero:** Число должно быть положительным / положительным или нулем.
- **@Negative, @NegativeOrZero:** Аналогично для отрицательных.

Для строковых шаблонов:

- `@Email`: Проверяет, что строка имеет формат email-адреса.
- `@Pattern(regex = "...")`: Проверяет соответствие строки регулярному выражению.

Для дат:

- `@Past`, `@PastOrPresent`: Дата должна быть в прошлом / в прошлом или настоящем.
- `@Future`, `@FutureOrPresent`: Дата должна быть в будущем / в будущем или настоящем.

Для булевых значений:

- `@AssertTrue`, `@AssertFalse`: Поле должно быть true / false.

Пример: Модель данных с аннотациями валидации

Давайте создадим класс `RegistrationForm`, который представляет данные, приходящие с формы регистрации пользователя.

```
import jakarta.validation.constraints.*;
import java.time.LocalDate;

public class RegistrationForm {
    @NotBlank(message = "Username cannot be empty.")
    @Size(min = 3, max = 20, message = "Username must be between 3 and 20 characters.")
    private String username;

    @NotBlank(message = "Email is required.")
    @Email(message = "Please provide a valid email address.")
    private String email;

    @NotBlank(message = "Password is required.")
    @Size(min = 8, message = "Password must be at least 8 characters long.")
    private String password;

    // Поле для подтверждения пароля. Сама валидация будет на уровне класса.
    private String confirmPassword;
```

```

@NotNull(message = "Date of birth is required.")
@Past(message = "Date of birth must be in the past.")
private LocalDate dateOfBirth;

@AssertTrue(message = "You must agree to the terms and conditions.")
private boolean termsAccepted;
// Геттеры и сеттеры...
}

```

Что мы здесь сделали?

- Мы описали все бизнес-правила для данных регистрации прямо в классе-модели.
- Каждая аннотация имеет атрибут `message`, который позволяет задать кастомное, человекочитаемое сообщение об ошибке. Если его не указать, будет использовано сообщение по умолчанию.

1.12.27. Интеграция с другими спецификациями Jakarta EE

Магия Bean Validation в том, что она **автоматически интегрирована** с другими частями платформы. Вам редко нужно вызывать валидатор вручную.

Интеграция с Jakarta Persistence (JPA)

JPA-провайдер (например, Hibernate) **автоматически** запускает валидацию перед сохранением (`persist`) или обновлением (`merge`) сущности. Если хотя бы одно ограничение нарушено, будет брошено исключение `jakarta.validation.ConstraintViolationException`, и транзакция откатится.

Это гарантирует, что в вашу базу данных не попадут некорректные данные.

```

@Entity
public class UserAccount {
    @Id
    @GeneratedValue
    private Long id;

    // Ограничения Bean Validation работают вместе с аннотациями JPA
    @Column(unique = true, nullable = false)
    @NotBlank
    @Email
    private String email;

    @NotNull
    @Size(min = 2, max = 50)
    private String fullName;
    // ...
}

// ... в EJB-сервисе ...
@Stateless
public class UserService {
    @PersistenceContext
    private EntityManager em;

    public void createUser(UserAccount user) {
        // Перед вызовом em.persist(user), Hibernate Validator
        // автоматически проверит все аннотации на объекте user.
        // Если, например, email некорректен, будет брошено исключение,
        // и транзакция откатится.
        em.persist(user);
    }
}

```

```
}  
}
```

Интеграция с Jakarta REST (JAX-RS)

JAX-RS автоматически валидирует параметры методов и тела запросов, если они помечены аннотацией `@Valid`.

```
import jakarta.validation.Valid;  
import jakarta.ws.rs.Consumes;  
import jakarta.ws.rs.POST;  
import jakarta.ws.rs.Path;  
import jakarta.ws.rs.core.MediaType;  
import jakarta.ws.rs.core.Response;
```

```
@Path("/users")
```

```
public class UserResource {
```

```
    @POST
```

```
    @Path("/register")
```

```
    @Consumes(MediaType.APPLICATION_JSON)
```

```
    public Response registerUser(@Valid RegistrationForm form) {
```

```
        // 1. JAX-RS получает JSON и десериализует его в объект  
        RegistrationForm.
```

```
        // 2. Благодаря аннотации @Valid, он автоматически запускает  
        валидатор
```

```
        // для этого объекта.
```

```
        // 3. Если есть ошибки валидации, JAX-RS прерывает выполнение  
        метода
```

```
        // и автоматически возвращает клиенту HTTP-статус 400 (Bad Request)
```

```
        // со списком ошибок.
```

```
        // Если мы дошли сюда, значит, все данные в 'form' корректны.
```

```
        // ... логика регистрации пользователя ...
```

```
        return Response.status(Response.Status.CREATED).build();
```

```
    }
```

```
}
```

Аннотация `@Valid` — это ключ к каскадной валидации. Она говорит валидатору: "Зайди внутрь этого объекта и проверь все ограничения, которые есть в нем".

Интеграция с Jakarta Faces (JSF)

JSF также автоматически использует Bean Validation. Когда пользователь отправляет форму, JSF:

1. Заполняет модель (backing bean) данными из формы.
2. Запускает валидацию для этой модели.
3. Если есть ошибки, JSF **не вызывает** метод-действие (action method), а перерисовывает ту же страницу, отображая сообщения об ошибках рядом с полями ввода, для которых они произошли (если вы используете компонент `<h:message>`).

```
<!-- registration.xhtml -->
```

```
<h:form>
```

```
  <h:panelGrid columns="3">
```

```
    Username:
```

```
    <h:inputText id="username"
value="#{registrationBean.form.username}" />
```

```
    <h:message for="username" style="color: red;" />
```

```
    Email:
```

```
    <h:inputText id="email" value="#{registrationBean.form.email}" />
```

```
    <h:message for="email" style="color: red;" />
```

```
    <!-- ... другие поля ... -->
```

```
  </h:panelGrid>
```

```
  <h:commandButton value="Register" action="#{registrationBean.register}" />
```

```
</h:form>
```

1.12.28. Программная валидация

Иногда нужно запустить валидацию вручную. Для этого используется интерфейс `Validator`.

```
import jakarta.validation.ConstraintViolation;
```

```
import jakarta.validation.Validation;
```

```

import jakarta.validation.Validator;
import jakarta.validation.ValidatorFactory;
import java.util.Set;

public class ManualValidatorExample {
    public void validateManually() {
        // 1. Получаем фабрику и сам валидатор
        ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
        Validator validator = factory.getValidator();
        // 2. Создаем объект с некорректными данными
        RegistrationForm form = new RegistrationForm();
        form.setUsername("a"); // Слишком короткое
        form.setEmail("not-an-email"); // Неверный формат
        form.setTermsAccepted(false); // Не отмечено
        // 3. Вызываем валидацию
        Set<ConstraintViolation<RegistrationForm>> violations =
validator.validate(form);
        if (violations.isEmpty()) {
            System.out.println("Object is valid!");
        } else {
            System.out.println("Validation errors found: " + violations.size());
            // 4. Выводим сообщения об ошибках
            for (ConstraintViolation<RegistrationForm> violation : violations)
            {
                System.out.println(
                    "Field: " + violation.getPropertyPath() +
                    ", Error: " + violation.getMessage()
                );
            }
        }
    }
}

```


}

Вывод консоли:

Validation errors found: 3

Field: username, Error: Username must be between 3 and 20 characters.

Field: email, Error: Please provide a valid email address.

Field: termsAccepted, Error: You must agree to the terms and conditions.

Это показывает, как работает механизм под капотом у JPA, JAX-RS и JSF.

1.12.29. Стандартные ограничения (Constraints)

1. Ограничения для проверки на null и пустоту

Эти аннотации используются для обеспечения того, что поле имеет значение.

- **@NotNull**

- **Применяется к:** Любому типу объекта.
- **Проверяет:** Что значение поля не является null.
- **Пример:**

```
@NotNull(message = "User must be assigned to a department.")
```

```
private Department department;
```

```
@NotNull(message = "Creation date cannot be null.")
```

```
private LocalDate creationDate;
```

- **@NotEmpty** (появилась в Bean Validation 2.0)

- **Применяется к:** CharSequence (например, String), Collection, Map, массивам.
- **Проверяет:** Что поле не null и его размер/длина больше нуля. Для строки это означает, что она не пустая ("").
- **Пример:**

```
@NotEmpty(message = "The list of tags cannot be empty.")
```

```
private List<String> tags;
```

```
@NotEmpty(message = "Product name is required.")
```

```
private String productName;
```

- **@NotBlank** (появилась в Bean Validation 2.0)

- **Применяется к:** CharSequence (например, String).
- **Проверяет:** Что поле не null и после удаления начальных и конечных пробелов его длина больше нуля. Это более строгая проверка, чем @NotEmpty, так как строка, состоящая только из пробелов (" "), не пройдет валидацию.
- **Пример:**

```
// Идеально для полей, где пользователь может ввести только пробелы
```

```
@NotBlank(message = "Username cannot be blank.")
```

```
private String username;
```

Сравнение:

Значение	@NotNull	@NotEmpty	@NotBlank
Null	FAIL	FAIL	FAIL
"" (пустая строка)	PASS	FAIL	FAIL
" " (пробелы)	PASS	PASS	FAIL
"text"	PASS	PASS	PASS

2. Ограничения на размер и диапазон

- **@Size**

- **Применяется к:** CharSequence, Collection, Map, массивам.
- **Проверяет:** Что размер/длина поля находится в указанном диапазоне (включительно).
- **Пример:**

@Size(min = 8, max = 100, message = "Password must be between 8 and 100 characters.")

```
private String password;
```

@Size(max = 5, message = "A user can have at most 5 phone numbers.")

```
private List<String> phoneNumbers;
```

- **@Min / @Max**

- **Применяется к:** Числовым типам (byte, short, int, long, BigDecimal, BigInteger и их оберткам).
- **Проверяет:** Что числовое значение больше или равно (@Min) или меньше или равно (@Max) указанному значению.
- **Пример:**

@Min(value = 1, message = "Quantity must be at least 1.")

```
private int quantity;
```

@Max(value = 150, message = "Age cannot be greater than 150.")

```
private int age;
```

- **@Positive / @PositiveOrZero / @Negative / @NegativeOrZero**

- **Применяется к:** Числовым типам.
- **Проверяет:** Что значение положительное (> 0), положительное или ноль (≥ 0), отрицательное (< 0) или отрицательное или ноль (≤ 0) соответственно.
- **Пример:**

```
@Positive(message = "Price must be a positive number.")
```

```
private BigDecimal price;
```

```
@NegativeOrZero(message = "Discount cannot be positive.")
```

```
private int discount;
```

3. Ограничения на формат и шаблон

- **@Email**

- **Применяется к:** CharSequence.
- **Проверяет:** Что строка соответствует общепринятому формату email-адреса. Проверка не гарантирует, что такой email существует, она проверяет только синтаксис.
- **Пример:**

```
@NotBlank
```

```
@Email(message = "Invalid email format.")
```

```
private String email;
```

- **@Pattern**

- **Применяется к:** CharSequence.
- **Проверяет:** Что строка соответствует указанному **регулярному выражению**. Это очень мощная аннотация для создания сложных правил.
- **Пример:**

```
// Проверяем, что номер телефона в формате США (например, 123-456-7890)
```

```
@Pattern(regex = "^\\d{3}-\\d{3}-\\d{4}$", message = "Phone number must be in format XXX-XXX-XXXX.")
```

```
private String phoneNumber;
```

```
// Проверяем, что почтовый индекс состоит из 5 цифр
```

```
@Pattern(regex = "^\\d{5}$", message = "Zip code must be 5 digits.")
```

```
private String zipCode;
```

4. Ограничения для дат и времени

Эти аннотации особенно полезны при работе с типами из `java.time` (например, `LocalDate`, `LocalDateTime`).

- **@Past / @PastOrPresent**

- **Применяется к:** Типам дат/времени.
- **Проверяет:** Что дата находится в прошлом (`@Past`) или в прошлом/настоящем (`@PastOrPresent`) относительно текущего момента времени.
- **Пример:**

`@NotNull`

`@Past(message = "Date of birth must be a past date.")`

`private LocalDate dateOfBirth;`

- **@Future / @FutureOrPresent**

- **Применяется к:** Типам дат/времени.
- **Проверяет:** Что дата находится в будущем (`@Future`) или в будущем/настоящем (`@FutureOrPresent`).
- **Пример:**

`@NotNull`

`@Future(message = "Expiration date must be in the future.")`

`private LocalDate creditCardExpiration;`

5. Ограничения для булевых значений

- **@AssertTrue / @AssertFalse**

- **Применяется к:** `boolean` или `Boolean`.
- **Проверяет:** Что значение поля равно `true` или `false` соответственно.
- **Пример:**

`@AssertTrue(message = "You must accept the terms of service to register.")`

`private boolean termsAccepted;`

1.12.30. Полный пример класса с использованием разных ограничений

```
import jakarta.validation.constraints.*;
```

```
import java.math.BigDecimal;
```

```
import java.time.LocalDate;
```

```
import java.util.List;
```

```

public class ProductDto {
    @NotBlank(message = "Product SKU cannot be blank.")
    @Pattern(regexp = "^[A-Z0-9]{5,10}$", message = "SKU must be 5-10
uppercase letters or digits.")
    private String sku;

    @NotEmpty(message = "Product name is required.")
    @Size(max = 255)
    private String name;

    @NotNull(message = "Price is required.")
    @Positive(message = "Price must be positive.")
    private BigDecimal price;

    @Min(value = 0, message = "Stock quantity cannot be negative.")
    private int stockQuantity;

    @NotNull
    @PastOrPresent(message = "Release date cannot be in the future.")
    private LocalDate releaseDate;

    @Size(max = 10, message = "Cannot have more than 10 tags.")
    private List<String> tags;

    // Геттеры и сеттеры
}

```

Этот пример показывает, как можно комбинировать разные аннотации для точного описания правил валидации для объекта ProductDto. При передаче этого объекта в JAX-RS метод, помеченный @Valid, или при сохранении связанной с ним сущности в JPA, все эти правила будут проверены автоматически.

1.12.31. Создание собственных ограничений (Custom Constraints)

Процесс создания собственного ограничения состоит из трех шагов:

1. **Создание аннотации ограничения:** Это сама аннотация, лично! Хотя стандартные ограничения покрывают большинство случаев, часто возникают уникальные бизнес-правила, которую вы будете использовать в своем коде (например, `@PasswordMatches`).
2. **Создание класса валидатора, для которых нужно создавать собственные правила валидации. Jakarta Bean Validation предоставляет элегантный и мощный механизм для этого:** Это класс, который содержит реальную логику проверки для вашей аннотации.
3. **Связывание аннотации и валидатора:** В аннотации нужно указать, какой класс-валидатор используется.
4. **Создание аннотации ограничения:** Это сама аннотация, которую вы будете использовать в своем коде (тогда ее реализует).

Давайте рассмотрим этот процесс на двух практических примерах: простом (валидация поля) и более сложном (валидация на уровне класса).

Пример 1: Валидация поля (например, `@PasswordComplexity`).

1. **Создание класса валидатора:** Это класс, который содержит фактическую логику проверки.

2. **(Опционально) Настройка сообщений об ошибках:** До — `@ValidCase`

Задача: Создать аннотацию `@ValidCase`, которая проверяет, что строка написана в соответствии с требованиями кастомных сообщений в файл `ValidationMessages.properties`.

Пример: Создание валидатора в определенном регистре (например, только в верхнем или только в нижнем).

- Шаг 1: Создание аннотации (`@ValidCase`)

```
import jakarta.validation.Constraint;
import jakarta.validation.Payload;
import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
```

// 1. Указываем, к чему можно применять аннотацию (поля, методы, пометки, заглавную букву).

```
@Target({ElementType.FIELD, ElementType.METHOD,
          ElementType.PARAMETER, ElementType.ANNOTATION_TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface ValidCase {
    String message() default "";
    Class[] payload() default {};
}
```

// 2. Указываем, что аннотация будет доступна во время выполнения.

```
```java
```

```
import jakarta.validation.Constraint;
import jakarta.validation.Payload;
import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
```

// 3. Связываем аннотацию с ее валидатором (`.lang.annotation.Documented`;

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
```



```

@Constraint(validatedBy = ValidCaseValidator.class)
@Documented
annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Targetpublic @interface ValidCase {

 // 4. Обязательные атрибуты для любой аннотации-;

 // 1. @Constraint: Самая важная аннотация. Указывает, какой класс-
валиограничения

 // Сообщение об ошибке по умолчанию
 String message() default "Invalid caseдатор будет
 // обрабатывать это ограничение.
 @Constraint(validatedBy = PasswordValidator.class)

 ";

 // Группы валидации (для продвинутого использования)
 Class<?>[] groups// 2. @Target: Указывает, к чему можно применять нашу
аннотацию
 // (по() default {});

 // Полезная нагрузка (для продвинутого использования)
 Class<? extends Payloadдля, методы, параметры методов и т.д.).
 @Target({
 ElementType.METHOD,
 ElementType.FIELD,
 ElementType.ANNOTATION_TYPE, ElementType.PARAMETER })

 // 3. @Retention: Указывает, что>[] payload() default {});

 // 5. Добавляем собственный атрибут для нашей аннотации

```

аннотация должна быть доступна во время выполнения.

`@Retention(RetentionPolicy.RUNTIME)`

`@Documented`

```
public CaseMode value();
}
```

- `@Constraint(validatedBy = ...)`: Эта аннотация `@interface ValidPassword {`  
    // 4. `message()`: Стандартный атрибут для сообщения об ошибке. Аннотация "включает" механизм Bean Validation и связывает `@ValidCase` с `ValidCaseValidator`.
- `// {...}` - это ключ для поиска сообщения в resource bundle (`ValidationMessages.properties`).  
    `String message(), groups(), payload()`: Эти три атрибута **обязательны** по умолчанию.  
    `"{com.example.validation.ValidPassword.message}";`  
    // 5. `groups()`: Стандартный атрибут.
- `CaseMode value()`: Мы добавили свой собственный параметр, чтобы сделать аннотацию гибким атрибутом для группировки ограничений.  
    // Позволяет валидировать только определенные группы.

Создадим enum `CaseMode`:

```
public enum CaseMode {
 UP правил.
 Class<?>[] groups() default {};
 // 6. payload(): Стандартный атрибут для передачи PER,
 LOWER;
}
```

- Шаг 2: Создание класса-валидатора (с дополнительной мета-информацией.

```
Class<? extends Payload>[] payload() default {};
}
```

`ValidCaseValidator`)

Валидатор должен реализовывать интерфейс

```
`jakarta.validation.ConstraintValidator<A **Разбор мета-аннотаций.**`
```

\* ``@Constraint(validatedBy = ...)``: Свя, T>`, где `A` — это ваша аннотация, а `T` — тип поля, к которому она применяется аннотацией с ее логикой.

\* `@Target`: Определяет, где можно использовать аннотацию применяется.

```
```java
```

```
import jakarta.validation.ConstraintValidator;
```

```
import jakarta.validation.ConstraintValidatorContext;
```

```
public.
```

* `@Retention(RetentionPolicy.RUNTIME)`: Обязательно, чтобы фреймворк
class ValidCaseValidator implements ConstraintValidator<ValidCase, String> {

```
private CaseMode caseMode;
```

валидации мог прочесть аннотацию во время выполнения.

* `message()`, `groups()`, `payload()` `@Override`

```
public void initialize(ValidCase constraintAnnotation) {
```

// Этот метод вызывается один раз перед вали: Это три обязательных атрибута для любой аннотации-ограничения.

// Здесь мы получаем значение, переданное в аннотацию.

```
this.caseMode = constraint.класс-валидатора `PasswordValidator`
```

Этот класс должен реализовывать интерфейс
`jakarta.validation.ConstraintAnnotation.value()`;

```
}
```

```
@Override
```

```
public boolean isValid(String value, ConstraintValidatorContext context)  
{Validator<A, T>`, где:
```

* `A`` — это ваша аннотация ограничения (`ValidPassword``).

// Здесь находится сама логика валидации.

// Считаем, что null является вали

* `T` — это тип данных, к которому можно применять аннотацию (`String`).

```
```java
```

```
import jakarta.validation.ConstraintValidator;
import jakarta.validation.ConstraintValidatorContext;
```

```
public class PasswordValidator
```

```
 // Если поле не должно быть null, используйте @NotNull дополнительно.
```

```
 if (value == null || value.isEmpty()) {
 return true;
 }
```

```
 if (caseMode == CaseMode.UP) implements
 ConstraintValidator<ValidPassword, String> {
```

```
 // Метод initialize() вызывается один раз перед началом валидации.
```

```
 PER) {
 return value.equals(value.toUpperCase());
 } else {
```

```
 return value. // Здесь можно получить доступ к атрибутам аннотации, если
они есть.
```

```
 @Override
```

```
 public equals(value.toLowerCase());
 }
```

```
 }
```

```
 }
```

- Шаг 3: Использование void initialize(ValidPassword constraintAnnotation) {  
Generated code

// В нашем случае у @ValidPassword нет атрибутов, поэтому нового  
ограничения

Теперь мы можем использовать нашу аннотацию в любом классе-модели.

```

public class Product {

 @ValidCase(value = CaseMode.UPPER, message = "Product code must be in
здесь пусто.
 }

 // isValid() - это главный метод, содержащий логику валидации.
 @Override
 public boolean isValid(String password, ConstraintValidatorContext context) {
 // Если пароль null, мы upper case.")
 @Size(min = 5, max = 5)
 private String productCode; // считаем его валидным.
 // За проверку на null должна отвечать аннотация @NotNull/@ например,
 "PROD1" - валидно, "prod1" - невалидно

 @ValidCase(value = CaseMode.LOWER, message = "Category tag must be in
lower case.")
 private String categoryNotBlank.
 // Это позволяет комбинировать ограничения.
 if (password == null) {
 return true;
 Tag; // например, "books" - валидно, "Books" - невалидно

 // getters and setters
 }

 // Проверка длины
 if (password.length() < 8) {
 }
 Готово! Мы создали полностью переиспользуемое и настраиваемое
ограничение.

```

```
 return false;
}
// Проверки на наличие разных типов символов
boolean hasUpper = false;
```

## Пример 2: Валидация на уровне класса — @PasswordMatches

Часто правила валидации зави boolean hasLower = false;  
boolean hasDigit = false;

for (char ch : password.toCharArray) {  
Классический пример — проверка совпадения пароля и его подтверждения. Ta()  
{

if (Character.isUpperCase(ch)) {

hasUpper = true;

} кую проверку нельзя повесить на одно поле, ее нужно применять ко всему  
объекту.

**Задача:** Создать аннотацию else if (Character.isLowerCase(ch)) {  
hasLower = true;  
}  
else if (Character.isDigit(ch)) {  
hasDigit = true;  
}  
}

return hasUpper && hasLower && hasDigit;

}

}

##### Шаг 3: (Опциона `@PasswordMatches`, которая проверяет, что два поля  
с паролями в объекте совпадают.

##### Шаг 1льно) Настройка сообщения об ошибке

Мы указали ключ `{com.example.validation.ValidPassword.message}` в:  
Создание аннотации (`@PasswordMatches`)

Аннотация для валидации на уровне класса должна иметь  
`@Target(ElementType.TYPE)`.

```
```java
```

```
import jakarta.validation.Constraint;
```

```
import jakarta.validation.Payload;
```

```
import java. нашей аннотации. Теперь создадим файл
`ValidationMessages.properties` в classpath (обычно в
`src/main/resources/lang.annotation.ElementType;
```

```
import java.lang.annotation.Retention;
```

```
import java.lang.annotation.Retention`) и добавим туда наше сообщение.
```

```
**`src/main/resources/ValidationMessages.properties`**
```

```
``propertiesPolicy;
```

```
import java.lang.annotation.Target;
```

```
// Указываем, что аннотация применяется к T
```

```
com.example.validation.ValidPassword.message=Password must be at least 8
characters long and contain atИПУ (классу)
```

```
@Target(ElementType.TYPE)
```

```
@Retention(RetentionPolicy.RUNTIME)
```

```
@Constraint least one uppercase letter, one lowercase letter, and one digit.
```

```
Теперь, если валидация не пройдет, будет(validatedBy =
PasswordMatchesValidator.class)
```

```
public @interface PasswordMatches {
```

```
String message() default " использовано это понятное сообщение.
```

```
Шаг 4: Использование нашего нового ограничения
```

```
Теперь мы можем использовать `@ValidPasswordPasswords do not match";
```

```
Class<?>[] groups() default {};
```

```
Class<? extends Payload>[] payload` так же, как и любое стандартное
ограничение.
```

```
import jakarta.validation.constraints.NotBlank;
```

```
() default {};
```

```
}
```

```
Шаг 2: Создание класса-валидатора (PasswordMatchesValidator)
```

```
public class UserRegistrationDto {
```



```
@NotBlank(message = "Username is required.")
```

private String username;Валидатор для класса будет принимать в `isValid` весь объект, а не одно поле.

```
content_copydownload
```

Use code with caution.

Generated java

```
import jakarta
```

```
@NotBlank(message = "Password is required.")
```

```
@ValidPassword // <-- Наше собственное  
ограничение.validation.ConstraintValidator;
```

```
import jakarta.validation.ConstraintValidatorContext;
```

```
// В качестве второго generic-пара!
```

```
private String password;
```

```
// getters and setters...
```

```
}
```

Когда валидатор будет проверметра указываем тип объекта, который валидируем.

```
public class PasswordMatchesValidator implements  
ConstraintValidator<PasswordMatches есть объект UserRegistrationDto, он увидит  
аннотацию @ValidPassword на поле password, найдет связан, Object> {
```

```
@Override
```

```
public void initialize(PasswordMatches constraintAnnotation) {
```

```
// Здесьный с ней `PasswordValidator`, создаст его экземпляр и вызовет  
метод `isValid()`, передав можно получить атрибуты из аннотации, если они есть.
```

```
}
```

```
@Override
```

```
public boolean isValid(Object ему значение поля.
```

Валидаторы на уровне класса (Cross-Field Validation)

Иногда нужно проверить obj, ConstraintValidatorContext context) {
// Здесь мы получаем весь объект для валидации.

// Предполагаем, что объект имеет методы getPassword() и
getConfirmPassword()

// В реальном приложении взаимосвязь между несколькими полями.
Например, "пароль и подтверждение пароля должны совпадать". лучше
использовать рефлексии или передавать

// имена полей через атрибуты аннотации для большей гибкости.

Такую проверку нельзя сделать на уровне одного поля. Для этого создается
валидатор на уровне класса.

```
if (obj instanceof RegistrationForm) {
```

```
RegistrationForm form = (RegistrationForm) obj;
```

```
String Пример: `@PasswordsMatch`
```

1. Аннотация:

```
@Constraint(validatedBy = Pass password = form.getPassword());
```

```
String confirmPassword = form.getConfirmPassword();
```

```
// PasswordsMatchValidator.class)
```

```
// @Target теперь указывает на TYPE, т.е. на класс.
```

```
@Target({верям, что оба поля не null и равны друг другу
```

```
return password != null && password.equals(confirm ElementType.TYPE })
```

```
@Retention(RetentionPolicy.RUNTIME)
```

```
public @interface PasswordsMatch {
```

```
StringPassword);
```

```
}
```

```
// Если аннотация применена к другому классу, считаем валид message()  
default "Passwords do not match.";
```

```
Class<?>[] groups() default {};
```

```
Class<? extends Payload>[] payloadным.
```

```
return true;
```

```

    }
    }
    Улучшение:      Этот      валидатор      жест()      default      {}
}

```

****2. Валидатор:****

Теперь валидатор будет работать с типом всего что привязан к классу `RegistrationForm`. Чтобы сделать его более универсальным, можно передавать имена полей через объект (`UserRegistrationDto`), а не со `String`.

```

``java
public      class      PasswordsMatchValidator      implements
ConstraintValidator<Passwords аннотацию.
    `@PasswordMatches(field = "password", fieldMatch = "confirmPassword")`
    Match, UserRegistrationDto> {

    @Override
    public      boolean      isValid(UserRegistrationDto      userDto,
ConstraintValidatorContextA в `initialize` сохранять эти имена, а в `isValid`
получать значения полей с помощью рефлекс context) {
        if (userDto.getPassword() == null || userDto.getConfirmPassword() == null)
        {сии.

```

Шаг 3: Использование нового ограничения

Применяем аннотацию на уровне класса.

```

// Делегируем проверку на null другим аннотациям
return true;
}

```

```

// Главная логика: сравниваем два поля
return userDto.getPassword().equals(userDto

```

```

```java
// Применяем аннотацию к классу
@PasswordMatches
public class RegistrationForm {

 @.getConfirmPassword());
}
}

```

### 3. Применение:

Аннотация `NotBlank`

```
private String username;
```

```
@NotBlank
```

```
@Size(min = 8)
```

```
private String password;
```

```
@NotBlank
```

```
private String confirmPassword;
```

```
// getters and setters...
```

вводится на сам класс.

```
@PasswordsMatch // <-- Валидатор на уровне класса
```

```
public}
```

Теперь, когда `RegistrationForm` будет валидироваться, `PasswordMatchesValidator` проверит, что значения `class UserRegistrationDto` {

```
@NotBlank
```

```
@ValidPassword
```

```
private String password;
```

@ полей `'password'` и `'confirmPassword'` совпадают.

### 1.12.32. Выводы

Создание собственных ограничений — этоNotBlank  
private String confirmPassword;

// getters and setters

}

Создание собственных ограничений — это мощный механизм, который позволяет:

- Инкапсулировать сложную, повторяющуюся бизнес-логику валидаций в переиспользуемые, декларативные аннотации, делая код чистым и надежным.

### 1.12.33. Группы валидации (Validation Groups)

#### Проблема:

Представьте себе объект User, который используется в двух разных сценариях:

1. **При регистрации:** Нам нужно валидировать имя, email, пароль и согласие с условиями.
2. **При обновлении профиля:** Пользователь уже залогинен. Нам нужно валидировать только имя и email, но **не** пароль (так как для смены пароля обычно есть отдельная форма) и **не** согласие с условиями (он уже согласился).

Если мы повесим все аннотации на класс User, то при обновлении профиля валидация пароля не пройдет (так как поле будет пустым), и мы не сможем обновить профиль.

#### Решение: Группы валидации.

Группы позволяют вам разделить ограничения на подмножества и затем при валидации указать, какую именно группу (или группы) правил нужно применить.

#### Как это работает:

1. **Создание интерфейсов-маркеров:** Группа — это просто пустой интерфейс.
2. **Привязка ограничений к группам:** В аннотации ограничения вы указываете, к какой группе оно относится, через атрибут groups.
3. **Запуск валидации для конкретной группы:** При вызове валидатора вы передаете ему класс-маркер группы.

#### Пример

Шаг 1: Создаем интерфейсы-маркеры для групп

// Группа для правил, применяемых при регистрации

```
public interface OnRegistration {}
```

// Группа для правил, применяемых при обновлении

```
public interface OnProfileUpdate {}
```

Шаг 2: Привязываем ограничения к группам в нашей модели

```
import jakarta.validation.constraints.Email;
import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.NotNull;
import jakarta.validation.constraints.Size;
import jakarta.validation.groups.Default; // Стандартная группа

public class User {
 // Это поле валидируется всегда (потому что оно в группе Default)
 // и при обновлении профиля.
 @NotBlank(groups = {Default.class, OnProfileUpdate.class})
 @Size(min = 2, max = 50, groups = {Default.class, OnProfileUpdate.class})
 private String fullName;

 // Email тоже валидируется в обоих случаях.
 @NotBlank(groups = {Default.class, OnProfileUpdate.class})
 @Email(groups = {Default.class, OnProfileUpdate.class})
 private String email;

 // Пароль валидируется ТОЛЬКО при регистрации.
 // Группа Default здесь не указана!
 @NotBlank(groups = OnRegistration.class)
 @Size(min = 8, groups = OnRegistration.class)
 private String password;

 // getters and setters...
}
```

- **Default group:** Если группа не указана в аннотации, ограничение автоматически принадлежит к стандартной группе `jakarta.validation.groups.Default`. Когда вы вызываете валидацию без указания групп, проверяется только группа `Default`.

Шаг 3: Запускаем валидацию для конкретных групп

**В JAX-RS ресурсе:**

Аннотация `@Valid` не позволяет указать группу. Но для параметров методов в JAX-RS и других спецификациях есть решение. Вместо `@Valid` используется специальная аннотация, которая позволяет это сделать (или программная валидация).

**Программная валидация (самый наглядный способ):**

```
import jakarta.inject.Inject;
import jakarta.validation.ConstraintViolation;
import jakarta.validation.Validator;
import java.util.Set;

public class UserService {
 @Inject
 private Validator validator; // Внедряем стандартный валидатор

 public void registerNewUser(User user) {
 // Запускаем валидацию, указывая группы Default И OnRegistration
 Set<ConstraintViolation<User>> violations = validator.validate(user,
Default.class, OnRegistration.class);
 if (!violations.isEmpty()) {
 throw new ValidationException(violations);
 }
 // ... логика регистрации ...
 }

 public void updateUserProfile(User user) {
 // Запускаем валидацию, указывая ТОЛЬКО группу OnProfileUpdate
 Set<ConstraintViolation<User>> violations = validator.validate(user,
OnProfileUpdate.class);
 if (!violations.isEmpty()) {
 throw new ValidationException(violations);
 }
 }
}
```



```
 // ... логика обновления ...
 }
}
```

Теперь, когда будет вызван `updateUserProfile`, поле `password` **не будет** проверяться, и валидация пройдет успешно, даже если оно `null`.

#### 1.12.34. Валидация методов и конструкторов (Executable Validation)

Bean Validation позволяет применять ограничения не только к полям объектов, но и к **параметрам и возвращаемым значениям** методов и конструкторов.

Это очень мощный инструмент для реализации **"Программирования по контракту" (Design by Contract)**. Вы можете определить:

- Предусловия (Preconditions): Какие условия должны выполняться для параметров, передаваемых в метод.
- Постусловия (Postconditions): Какие условия должны выполняться для значения, возвращаемого из метода.

##### **Как это работает:**

Эта функция обычно реализуется с помощью перехватчиков (interceptors), которые предоставляет среда выполнения (например, CDI или EJB-контейнер). Перехватчик "оборачивает" вызов вашего метода, проверяет параметры перед его выполнением и возвращаемое значение — после.

## Пример

### Шаг 1: Активация валидации методов

Чтобы эта функция работала, ее нужно активировать. Обычно это делается в конфигурационном файле META-INF/validation.xml.

**src/main/resources/META-INF/validation.xml**

```
<validation-config
 xmlns="https://jakarta.ee/xml/ns/validation/configuration"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="https://jakarta.ee/xml/ns/validation/configuration
https://jakarta.ee/xml/ns/validation/configuration/validation-configuration-3.0.xsd"
 version="3.0">
 <!-- Указываем, какие типы исполняемых элементов валидировать -->
 <executable-validation enabled="true">
 <default-validated-executable-types>
 <executable-type>CONSTRUCTORS</executable-type>
 <executable-type>NON_GETTER_METHODS</executable-
type>
 </default-validated-executable-types>
 </executable-validation>
</validation-config>
```

- **NON\_GETTER\_METHODS:** Валидировать все методы, кроме геттеров (методы, начинающиеся с `get` или `is`). Это делается для производительности.

## Шаг 2: Применение аннотаций к методам и конструкторам

Теперь мы можем применять аннотации прямо в коде сервиса.

```
import jakarta.validation.constraints.*;
import jakarta.validation.Valid;

// Предполагаем, что этот класс - CDI-бин или EJB,
// чтобы перехватчик валидации мог сработать.
@ApplicationScoped
public class ReservationService {
 // --- Валидация параметров метода (предусловия) ---
 public Reservation createReservation(
 @NotNull @FutureOrPresent LocalDate checkInDate,
 @Min(1) int durationInDays,
 @NotNull @Valid Customer customer // Каскадная валидация для
 объекта Customer
) {
 // Если мы дошли сюда, все параметры уже проверены контейнером
 // и они гарантированно корректны.
 // ... логика создания бронирования ...
 Reservation reservation = new Reservation(/* ... */);
 return reservation;
 }

 // --- Валидация возвращаемого значения (постусловия) ---
 @NotNull // Гарантируем, что метод никогда не вернет null
 public Reservation findReservationById(@Positive long reservationId) {
 Reservation found = // ... логика поиска в БД ...
 // Если мы вернем отсюда null, контейнер бросит
 ConstraintViolationException
 // после выполнения метода.
 return found;
 }
}
```

```

 }

 // --- Валидация конструктора ---
 public ReservationService(
 @NotNull(message = "Configuration cannot be null") Configuration
 config) {
 // ...
 }
}

```

### Что происходит при вызове createReservation?

1. Клиент вызывает reservationService.createReservation(null, 0, invalidCustomer).
2. Перехватчик Bean Validation перехватывает этот вызов **до** того, как выполнится тело метода.
3. Он проверяет все аннотации на параметрах:
  - checkInDate не null? Нет, null. Ошибка.
  - durationInDays больше 1? Нет, 0. Ошибка.
  - customer не null и валиден? Нет, invalidCustomer. Ошибка.
4. Поскольку есть ошибки, перехватчик **не вызывает** тело метода, а сразу бросает jakarta.validation.ConstraintViolationException со списком всех нарушений.

Это чрезвычайно мощный способ гарантировать, что ваши бизнес-методы всегда вызываются с корректными данными, и они всегда возвращают корректные результаты, тем самым повышая надежность всей системы.

### 1.12.35. Интеграция Bean Validation с JPA, JAX-RS и CDI

Давайте рассмотрим, как Bean Validation работает в связке с каждой из этих спецификаций.

#### 1. Интеграция с Jakarta Persistence (JPA)

Это, пожалуй, самая важная интеграция. Она обеспечивает **целостность данных на уровне базы данных**.

##### Как работает:

Любой JPA-провайдер (например, Hibernate), совместимый с Jakarta EE, **автоматически** запускает процесс валидации перед выполнением следующих операций с EntityManager:

- persist(entity): перед вставкой новой записи.
- merge(entity): перед обновлением существующей записи.
- (Реже) remove(entity): если на сущности есть специфичные для удаления ограничения.

Если валидация не проходит, JPA-провайдер бросает исключение jakarta.validation.ConstraintViolationException, и текущая транзакция **автоматически помечается для отката (rollback)**.

**Результат:** В вашу базу данных **никогда** не попадут данные, нарушающие бизнес-правила, определенные с помощью аннотаций Bean Validation.

##### Пример

Представим сущность Product, которая должна храниться в БД.

```
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.Id;
import jakarta.validation.constraints.Min;
import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.NotNull;
import jakarta.validation.constraints.Size;
```

```
@Entity // JPA аннотация
```

```
public class Product {
```

```
 @Id
```

```
 @GeneratedValue
```

```
private Long id;
```

```
// Комбинация аннотаций JPA и Bean Validation
```

```
@NotNull // Bean Validation
```

```
@Size(min = 5, max = 10) // Bean Validation
```

```
private String sku; // Stock Keeping Unit
```

```
@NotBlank(message = "Product name must not be empty.") // Bean Validation
```

```
private String name;
```

```
@Min(value = 0, message = "Stock cannot be negative.") // Bean Validation
```

```
private int stockCount;
```

```
// геттеры и сеттеры
```

```
}
```

Теперь рассмотрим сервис, который пытается сохранить некорректный продукт.

```
import jakarta.ejb.Stateless;
```

```
import jakarta.persistence.EntityManager;
```

```
import jakarta.persistence.PersistenceContext;
```

```
import jakarta.validation.ConstraintViolationException;
```

```
@Stateless
```

```
public class ProductService {
```

```
 @PersistenceContext
```

```
 private EntityManager em;
```

```
 public void createProduct() {
```

```
 Product invalidProduct = new Product();
```

```
 invalidProduct.setSku("ABC"); // Нарушает @Size(min=5)
```

```
 invalidProduct.setName(" "); // Нарушает @NotBlank
```

```

invalidProduct.setStockCount(-10); // Нарушает @Min(0)
try {
 // Этот EJB-метод по умолчанию транзакционный (CMT)
 // ПЕРЕД тем, как выполнить SQL INSERT, Hibernate (JPA-
 провайдер)
 // запустит валидацию для объекта invalidProduct.
 em.persist(invalidProduct);
 // Эта строка никогда не будет достигнута
 System.out.println("Product persisted successfully?");
} catch (ConstraintViolationException e) {
 // EJB-контейнер перехватит это исключение, откатит
 транзакцию
 // и может "обернуть" его в EJBException.
 System.err.println("Validation failed! Transaction will be rolled
 back.");
 e.getConstraintViolations().forEach(v ->
 System.err.println(" - " + v.getPropertyPath() + ": " +
 v.getMessage())
);
}
}
}

```

### Вывод в консоли:

Validation failed! Transaction will be rolled back.

- stockCount: Stock cannot be negative.
- name: Product name must not be empty.
- sku: size must be between 5 and 10

Никаких записей в базу данных добавлено не будет.

## 2. Интеграция с Jakarta REST (JAX-RS)

Эта интеграция обеспечивает валидацию входящих данных на границе приложения (на уровне API).

### Как работает:

JAX-RS автоматически валидирует параметры методов REST-ресурсов, если они помечены аннотацией `@jakarta.validation.Valid`. Это относится к:

- Телу запроса (объекты, десериализованные из JSON/XML).
- Параметрам пути (`@PathParam`), запроса (`@QueryParam`), заголовка (`@HeaderParam`) и т.д.

Если валидация не проходит, JAX-RS **не вызывает** тело метода вашего ресурса. Вместо этого он автоматически формирует и отправляет клиенту ответ с HTTP-статусом **400 (Bad Request)**, часто включая в тело ответа детализированный список ошибок валидации.

### Пример

Создадим REST-контроллер для создания продуктов.

```
import jakarta.inject.Inject;
import jakarta.validation.Valid; // Ключевая аннотация для интеграции
import jakarta.validation.constraints.NotBlank;
import jakarta.ws.rs.*;
import jakarta.ws.rs.core.MediaType;
import jakarta.ws.rs.core.Response;

@Path("/products")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class ProductResource {
```



@Inject

private ProductCreationService service;

@POST

```
public Response createProduct(@Valid ProductDto productDto) {
 // 1. JAX-RS получает JSON, десериализует его в ProductDto.
 // 2. Аннотация @Valid говорит: "Проверь этот объект!".
 // 3. JAX-RS запускает валидатор для productDto.
 // 4. Если есть ошибки -> прерывает выполнение и возвращает 400
 Bad Request.
 // Этот код выполнится, только если productDto полностью валиден.
 service.createFromDto(productDto);
 return Response.status(Response.Status.CREATED).build();
}
```

@GET

@Path("/search")

```
public Response searchProducts(
 // Валидация применяется и к параметрам запроса
 @QueryParam("query") @NotBlank @Size(min = 3) String searchQuery
) {
 // Если параметр query пустой или короче 3 символов,
 // JAX-RS вернет 400 Bad Request, и этот метод не будет вызван.
 // ... логика поиска ...
 return Response.ok().build();
}
}
```

// DTO для входящих данных

```
class ProductDto {
```

```

@NotBlank
@Size(min = 5, max = 10)
private String sku;
// ... другие поля с аннотациями ...
}

```

### Пример HTTP-взаимодействия:

Клиент отправляет: POST /api/products

Тело: {"sku": "A", "name": ""}

Сервер автоматически ответит (формат ответа может немного отличаться у разных реализаций): HTTP/1.1 400 Bad Request

Тело:

```

{
 "parameterViolations": [
 {
 "path": "createProduct.productDto.sku",
 "message": "size must be between 5 and 10"
 },
 {
 "path": "createProduct.productDto.name",
 "message": "must not be blank"
 }
]
}

```

## 3. Интеграция с CDI (Contexts and Dependency Injection)

CDI предоставляет "клей", который делает возможной большую часть этой магии.

### Как работает:

1. **Внедрение Validator:** CDI предоставляет стандартный Validator как готовый к внедрению бин. Это позволяет вам запускать валидацию программно в любом CDI-бине, где это необходимо.

2. **Валидация методов (Executable Validation):** CDI (через механизм перехватчиков/interceptors) является той средой, которая обеспечивает автоматическую валидацию параметров и возвращаемых значений методов, помеченных аннотациями ограничений.

**Пример: Программная валидация в CDI-бине**

Иногда вам нужно проверить объект, который не связан напрямую с JPA или JAX-RS.

```
import jakarta.enterprise.context.ApplicationScoped;
import jakarta.inject.Inject;
import jakarta.validation.ConstraintViolation;
import jakarta.validation.Validator;
import java.util.Set;
```

```
@ApplicationScoped
```

```
public class ReportGenerator {
```

```
 // 1. Внедряем стандартный валидатор
```

```
 @Inject
```

```
 private Validator validator;
```

```
 public Report generate(ReportParameters params) {
```

```
 // 2. Запускаем валидацию программно
```

```
 Set<ConstraintViolation<ReportParameters>> violations =
 validator.validate(params);
```

```
 if (!violations.isEmpty()) {
```

```
 // Если параметры отчета некорректны, мы можем
```

```
 // бросить кастомное исключение или вернуть ошибку.
```

```
 throw new InvalidReportParametersException(violations);
```

```
 }
```

```
 // Если мы здесь, параметры валидны.
```

```
 // ... логика генерации отчета ...
```

```
 return new Report();
```

```
 }
```

```
}
```

```
// Класс с параметрами отчета
```

```
class ReportParameters {
```

```
 @NotNull
```

```
 private LocalDate fromDate;
```

```
 @NotNull
```

```
 private LocalDate toDate;
```

```
 // ...
```

```
}
```

### 1.12.36. Сводка по интеграции

| Спецификация  | Как происходит интеграция                                                                                                                    | Цель                                                                                          |
|---------------|----------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------|
| <b>JPA</b>    | <b>Автоматически</b> перед <code>persist()</code> и <code>merge()</code> .                                                                   | Защита целостности данных в базе данных. Не допустить попадания "грязных" данных в хранилище. |
| <b>JAX-RS</b> | <b>Автоматически</b> для параметров, помеченных <code>@Valid</code> .                                                                        | Валидация входящих данных на границе API. Отклонить некорректные запросы как можно раньше.    |
| <b>CDI</b>    | <b>Внедрение</b> <code>Validator</code> через <code>@Inject</code> . <b>Обеспечение</b> валидации методов через перехватчики (interceptors). | Предоставление механизма для <b>программной валидации</b> и <b>валидации бизнес-методов</b> . |

Эта тесная интеграция — одна из причин, почему Jakarta EE является такой продуктивной платформой для создания надежных корпоративных приложений.

## 1.13. Тестирование JEE приложений

### 1.13.1. Тестирование Jakarta EE приложений: Юнит-тестирование

#### 1. Что такое Юнит-тестирование?

**Юнит (Unit)** — это наименьшая тестируемая часть приложения. В объектно-ориентированном программировании это, как правило, один класс или даже один метод.

**Юнит-тестирование** — это процесс проверки корректности работы этого юнита **в полной изоляции** от его внешних зависимостей (других классов, базы данных, файловой системы, сетевых сервисов и т.д.).

**Главная цель:** Убедиться, что **бизнес-логика**, инкапсулированная внутри конкретного класса, работает правильно при различных входных данных.

#### Почему важна изоляция?

- **Скорость:** Юнит-тесты должны выполняться очень быстро (миллисекунды). Они не должны зависеть от медленных операций, таких как доступ к БД или сети.

- **Надежность:** Результат теста не должен зависеть от состояния внешней среды. Если тест падает, мы должны быть на 100% уверены, что проблема именно в тестируемом классе, а не в том, что "база данных легла".
- **Локализация ошибок:** Быстрое определение точного места, где произошла ошибка.

## 2. Проблема юнит-тестирования в Jakarta EE

Компоненты Jakarta EE (CDI-бины, EJB, JAX-RS ресурсы) сильно зависят от **контейнера**, который предоставляет им сервисы через **внедрение зависимостей (Dependency Injection)**.

```
@ApplicationScoped
```

```
public class OrderService {
```

```
 @Inject // Зависимость от другого сервиса
```

```
 private PricingService pricingService;
```

```
 @Inject // Зависимость от репозитория
```

```
 private OrderRepository orderRepository;
```

```
 public Order createOrder(Cart cart) {
```

```
 // ... бизнес-логика ...
```

```
 BigDecimal finalPrice = pricingService.calculateFinalPrice(cart);
```

```
 // ...
```

```
 Order newOrder = new Order(finalPrice);
```

```
 orderRepository.save(newOrder);
```

```
 return newOrder;
```

```
 }
```

```
}
```

Если мы попробуем протестировать этот класс "в лоб":

```
@Test
void testCreateOrder() {
 OrderService orderService = new OrderService(); // Создаем вручную
 //...
 orderService.createOrder(cart); // УПАДЕТ с NullPointerException!
}
```

Тест упадет, потому что поля `pricingService` и `orderRepository` будут `null`. В обычном юнит-тесте нет контейнера, который бы их внедрил.

### 3. Решение: Мок-объекты (Mock Objects)

Чтобы изолировать тестируемый класс, мы заменяем его реальные зависимости на **моки (mocks)** или **стабы (stubs)**.

- **Мок-объект** — это "поддельный" объект, который имитирует поведение реальной зависимости. Мы можем настроить его так, чтобы при вызове определенного метода он возвращал заранее заданное значение или проверял, что метод был вызван с правильными параметрами.

#### Популярные библиотеки для мокирования в Java:

- **Mockito:** Самая популярная и простая в использовании.
- **EasyMock:** Более старая альтернатива.
- **JMockit:** Очень мощная, но более сложная.

Мы будем использовать **Mockito** в сочетании с **JUnit 5** (самый популярный фреймворк для тестирования в Java).

### 4. Пример юнит-тестирования сервисного слоя

Давайте протестируем наш `OrderService`.

#### Зависимости в `pom.xml` (Maven):

```
<dependency>
 <groupId>org.junit.jupiter</groupId>
 <artifactId>junit-jupiter-api</artifactId>
 <version>5.9.2</version>
 <scope>test</scope>
</dependency>
<dependency>
 <groupId>org.mockito</groupId>
```

```

 <artifactId>mockito-core</artifactId>
 <version>5.2.0</version>
 <scope>test</scope>
</dependency>
<dependency>
 <!-- Для интеграции Mockito с JUnit 5 -->
 <groupId>org.mockito</groupId>
 <artifactId>mockito-junit-jupiter</artifactId>
 <version>5.2.0</version>
 <scope>test</scope>
</dependency>

```

### Тестируемый класс OrderService:

Generated java

```
// src/main/java/com/example/OrderService.java
```

```
@ApplicationScoped
```

```
public class OrderService {
```

```
 @Inject
```

```
 private PricingService pricingService;
```

```
 @Inject
```

```
 private OrderRepository orderRepository;
```

```
 public Order createOrder(Cart cart, User user) {
```

```
 if (cart == null || cart.isEmpty()) {
```

```
 throw new IllegalArgumentException("Cart cannot be empty");
```

```
 }
```

```
 BigDecimal finalPrice = pricingService.calculateFinalPrice(cart,
 user.getDiscountRate());
```

```
 Order newOrder = new Order(finalPrice, user);
```

```
 // Мы хотим проверить, что именно этот метод был вызван
 }
```



```

 orderRepository.save(newOrder);
 return newOrder;
 }
}

```

### **Класс с тестами (OrderServiceTest.java):**

Он должен находиться в src/test/java/...

```

// src/test/java/com/example/OrderServiceTest.java
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoExtension;

import java.math.BigDecimal;

import static org.junit.jupiter.api.Assertions.*;
import static org.mockito.Mockito.*;

// 1. Активируем расширение Mockito для JUnit 5
@ExtendWith(MockitoExtension.class)
class OrderServiceTest {
 // 2. Создаем моки для всех зависимостей нашего сервиса
 @Mock
 private PricingService pricingServiceMock;
 @Mock
 private OrderRepository orderRepositoryMock;
 // 3. Создаем экземпляр тестируемого класса
 // и просим Mockito внедрить в него моки, созданные выше.
 @InjectMocks
 private OrderService orderService;

```

```

@Test
void createOrder_shouldSaveOrderWithCorrectPrice_whenCartIsNotEmpty() {
 // --- ARRANGE (Подготовка) ---
 // Создаем входные данные
 Cart cart = new Cart(/* ... добавляем товары ... */);
 User user = new User("john", 0.1); // 10% скидка
 // Настраиваем поведение моков:
 // "Когда будет вызван метод calculateFinalPrice с ЛЮБЫМ объектом
 // Cart и скидкой 0.1,
 // тогда ВЕРНУТЬ значение 90.00"
 when(pricingServiceMock.calculateFinalPrice(any(Cart.class), eq(0.1)))
 .thenReturn(new BigDecimal("90.00"));
 // --- АКТ (Действие) ---
 // Вызываем тестируемый метод
 Order resultOrder = orderService.createOrder(cart, user);
 // --- ASSERT (Проверка) ---
 // Проверяем, что результат не null и цена правильная
 assertNotNull(resultOrder);
 assertEquals(new BigDecimal("90.00"), resultOrder.getTotalPrice());
 assertEquals(user, resultOrder.getOwner());
 // Проверяем, что метод save у репозитория был вызван РОВНО 1 раз
 // и что ему был передан именно тот объект, который мы ожидаем.
 verify(orderRepositoryMock, times(1)).save(resultOrder);
 // (Опционально) Проверяем, что другие методы мока не вызывались
 verifyNoMoreInteractions(orderRepositoryMock);
}

@Test
void createOrder_shouldThrowException_whenCartIsEmpty() {
 // --- ARRANGE ---
 Cart emptyCart = new Cart();

```

```
User user = new User("test", 0);
// --- ACT & ASSERT ---
// Проверяем, что при вызове метода с пустыми данными
// будет брошено исключение IllegalArgumentException.
assertThrows(IllegalArgumentException.class, () -> {
 orderService.createOrder(emptyCart, user);
});
// Убеждаемся, что в этом случае никакие методы зависимостей не
// вызывались
verifyNoInteractions(pricingServiceMock, orderRepositoryMock);
}
}
```

### Разбор аннотаций Mockito:

- `@ExtendWith(MockitoExtension.class)`: Связывает JUnit 5 с Mockito.
- `@Mock`: Создает мок-объект для поля.
- `@InjectMocks`: Создает реальный экземпляр тестируемого класса и пытается внедрить в него все поля, помеченные `@Mock`.

### Разбор методов Mockito:

- `when(...).thenReturn(...)`: Задает поведение мока ("когда ... тогда вернуть ...").
- `any(Class)`: Матчер, который означает "любой объект данного класса".
- `eq(value)`: Матчер, который означает "объект, равный этому значению".
- `verify(...)`: Проверяет, был ли вызван метод мока.
- `times(n)`: Уточняет, сколько раз должен был быть вызван метод.
- `assertThrows(...)`: Стандартный метод JUnit 5 для проверки исключений.

### 1.13.2. Выводы по юнит-тестированию

- **Цель:** Тестировать логику одного класса в изоляции.
- **Инструменты:** JUnit 5 для структуры тестов и Mockito для создания моков внешних зависимостей.
- **Процесс:**
  1. Используйте `@Mock` для всех зависимостей, которые внедряются в ваш сервис.
  2. Используйте `@InjectMocks` для создания экземпляра тестируемого класса.
  3. В каждом тесте следуйте структуре **Arrange-Act-Assert**.
  4. В секции **Arrange** настраивайте поведение моков с помощью `when().thenReturn()`.
  5. В секции **Assert** проверяйте не только результат, но и **взаимодействия** с моками с помощью `verify()`.

Этот подход позволяет создавать быстрые, надежные и легко поддерживаемые тесты для бизнес-логики ваших Jakarta EE компонентов, не требуя для этого запуска тяжелого сервера приложений.

### 1.13.3. Интеграционное тестирование с Arquillian

#### 1. Проблема: Почему простого юнит-теста недостаточно?

Юнит-тесты прекрасны, но они не могут проверить:

- **Правильность внедрения зависимостей:** Действительно ли CDI-контейнер правильно внедрит PricingService в OrderService?
- **Работу EJB-сервисов:** Корректно ли контейнер управляет транзакциями для EJB? Работает ли декларативная безопасность (@RolesAllowed)?
- **Корректность JPA-маппинга:** Правильно ли наши @Entity классы отображаются на таблицы БД и генерируются ли SQL-запросы без ошибок?
- **Работу JAX-RS ресурсов:** Правильно ли наш REST-контроллер десериализует JSON и вызывает нужные сервисы?

Чтобы проверить все это, нам нужно запустить наш код **внутри реального или встраиваемого сервера приложений (контейнера)**. Делать это вручную — долго и сложно.

#### 2. Решение: Arquillian

**Arquillian** — это инновационный фреймворк для тестирования, который автоматизирует процесс интеграционного тестирования.

**Ключевая идея:** Arquillian позволяет вам написать тест, который выглядит как обычный JUnit-тест, но выполняется он **внутри контейнера**.

**Как это работает (упрощенно):**

1. Вы пишете тест и с помощью специальной аннотации @Deployment создаете **микро-развертывание (micro-deployment)**. Это миниатюрное приложение (например, .war файл), содержащее **только те классы, которые нужны для вашего теста**.
2. Перед запуском теста Arquillian:
  - Запускает сервер приложений (например, встраиваемый WildFly или Payara Micro).
  - Развертывает на нем ваше микро-приложение.
  - Выполняет ваш тест **внутри этого сервера**.
  - После теста останавливает сервер.

Это дает вам возможность тестировать компоненты так, как они будут работать в "боевой" среде, со всеми сервисами контейнера (CDI, EJB, JPA, JAX-RS и т.д.).

### 3. Настройка проекта с Arquillian

Настройка немного сложнее, чем для юнит-тестов. Вам нужно добавить несколько зависимостей в pom.xml.

```
<!-- pom.xml -->

<properties>
 <version.arquillian>1.7.0.Final</version.arquillian>
</properties>

<dependencyManagement>
 <dependencies>
 <!-- "Bill of Materials" для управления версиями всех Arquillian
артефактов -->
 <dependency>
 <groupId>org.jboss.arquillian</groupId>
 <artifactId>arquillian-bom</artifactId>
 <version>${version.arquillian}</version>
 <scope>import</scope>
 <type>pom</type>
 </dependency>
 </dependencies>
</dependencyManagement>

<dependencies>
 <!-- JUnit 5 -->
 <dependency>
 <groupId>org.junit.jupiter</groupId>
 <artifactId>junit-jupiter-api</artifactId>
 <scope>test</scope>
 </dependency>
 <!-- Связка Arquillian с JUnit 5 -->
 <dependency>
 <groupId>org.jboss.arquillian.junit5</groupId>
 <artifactId>arquillian-junit5-container</artifactId>
```

```

 <scope>test</scope>
 </dependency>
 <!-- Контейнерный адаптер для WildFly (управляемый) -->
 <dependency>
 <groupId>org.wildfly.arquillian</groupId>
 <artifactId>wildfly-arquillian-container-managed</artifactId>
 <version>5.0.0.Final</version>
 <scope>test</scope>
 </dependency>
 <!-- (Опционально) Клиент для REST-Assured для тестирования JAX-RS --
 >
 <dependency>
 <groupId>io.rest-assured</groupId>
 <artifactId>rest-assured</artifactId>
 <version>5.3.0</version>
 <scope>test</scope>
 </dependency>
 <!-- ... ваши Jakarta EE API зависимости ... -->
 <dependency>
 <groupId>jakarta.platform</groupId>
 <artifactId>jakarta.jakartaee-api</artifactId>
 <version>10.0.0</version>
 <scope>provided</scope>
 </dependency>
</dependencies>

```

Также нужен файл `src/test/resources/arquillian.xml` для конфигурации контейнера.

```
<!-- arquillian.xml -->
<arquillian xmlns="http://www.jboss.org/schema/arquillian"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.jboss.org/schema/arquillian
http://www.jboss.org/schema/arquillian/arquillian_1_0.xsd">
 <container qualifier="wildfly-managed" default="true">
 <configuration>
 <!-- Путь к вашему установленному серверу WildFly -->
 <property name="jbossHome">
 /path/to/your/wildfly
 </property>
 </configuration>
 </container>
</arquillian>
```

#### 4. Пример: Интеграционное тестирование EJB и JPA

**Задача:** Проверить, что наш `OrderService` (EJB) правильно использует `EntityManager` (JPA) для сохранения заказа в реальной (встроенной) базе данных H2.

##### Тестируемые компоненты:

- `Order.java` (`@Entity`)
- `OrderService.java` (`@Stateless EJB`)



### Тестовый класс:

```
import jakarta.ejb.EJB;
import jakarta.inject.Inject;
import jakarta.persistence.EntityManager;
import jakarta.persistence.PersistenceContext;
import org.jboss.arquillian.container.test.api.Deployment;
import org.jboss.arquillian.junit5.ArquillianExtension;
import org.jboss.shrinkwrap.api.ShrinkWrap;
import org.jboss.shrinkwrap.api.asset.EmptyAsset;
import org.jboss.shrinkwrap.api.spec.WebArchive;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;

import static org.junit.jupiter.api.Assertions.*;

// 1. Используем ArquillianExtension вместо MockitoExtension
@ExtendWith(ArquillianExtension.class)
class OrderServiceIT { // IT - конвенция для Integration Test
 // 2. Создаем "микро-развертывание"
 @Deployment
 public static WebArchive createDeployment() {
 return ShrinkWrap.create(WebArchive.class, "test.war")
 // Добавляем классы, которые нужны для теста
 .addClasses(Order.class, OrderService.class)
 // Добавляем persistence.xml для настройки JPA и H2
 .addAsResource("test-persistence.xml", "META-INF/persistence.xml")
 // Добавляем beans.xml для активации CDI
 .addAsWebInfResource(EmptyAsset.INSTANCE, "beans.xml");
 }
}
```

```

// 3. Внедряем РЕАЛЬНЫЕ компоненты, управляемые контейнером.
// Это не моки! Arquillian внедрит сюда настоящие EJB и EntityManager.
@EJB
private OrderService orderService;

@PersistenceContext
private EntityManager em;

@Test
void shouldCreateAndPersistOrder() {
 // --- ARRANGE ---
 // Ничего не нужно мокать!

 // --- ACT ---
 Order newOrder = orderService.createOrder("Test Product", 5);

 // --- ASSERT ---
 // Проверяем, что EJB вернул объект с сгенерированным ID
 assertNotNull(newOrder);
 assertNotNull(newOrder.getId());
 assertEquals("Test Product", newOrder.getProductName());
 // Проверяем напрямую в базе данных, что запись действительно
 // существует
 // em.clear() очищает кэш первого уровня, чтобы гарантировать
 // чтение из БД
 em.clear();
 Order persistedOrder = em.find(Order.class, newOrder.getId());
 assertNotNull(persistedOrder);
 assertEquals("Test Product", persistedOrder.getProductName());
}
}

```

**test-persistence.xml:**

```
<!-- src/test/resources/test-persistence.xml -->
<persistence version="3.0" ...>
 <persistence-unit name="test" transaction-type="JTA">
 <!-- Используем встроенную H2 базу данных для тестов -->
 <jta-data-source>java:jboss/datasources/ExampleDS</jta-data-source>
 <properties>
 <!-- Hibernate будет автоматически создавать/удалять таблицы -->
 <property name="jakarta.persistence.schema-
generation.database.action" value="drop-and-create"/>
 </properties>
 </persistence-unit>
</persistence>
```

### Разбор:

- **@Deployment:** Этот метод будет вызван Arquillian для создания .war файла "на лету".
- **ShrinkWrap:** Это API для программного создания Java-архивов (.jar, .war). Мы добавляем в него только то, что нужно.
- **@EJB, @Inject, @PersistenceContext:** Эти аннотации теперь работают "по-настоящему". Arquillian перехватывает их и просит контейнер внедрить реальные, управляемые им экземпляры.
- **Тест проверяет всю цепочку:** вызов EJB -> управление транзакцией -> работа JPA -> сохранение в БД.

### 5. Пример: Тестирование JAX-RS ресурса

Мы можем расширить наш тест, чтобы проверить и REST-слой.

// ... в том же тестовом классе ...

// Добавляем JAX-RS ресурс в наше развертывание

@Deployment

public static WebArchive createDeployment() {

return ShrinkWrap.create(...)

.addClasses(Order.class, OrderService.class, OrderResource.class,  
 RestActivator.class)

// ...

}

// @ArquillianResource используется для внедрения URL нашего развернутого приложения

@ArquillianResource

private URL deploymentUrl;

@Test

void shouldCreateOrderViaRestEndpoint() {

// Используем REST-Assured для отправки HTTP-запросов

given()

.contentType(ContentType.JSON)

```
.body("{\"productName\": \"REST Product\", \"quantity\": 10}")
.when()
.post(deploymentUrl + "api/orders")
.then()
.statusCode(201) // Проверяем, что статус ответа - 201 Created
.body("productName", equalTo("REST Product"));
}
```

Этот тест проверяет **полный сквозной сценарий (end-to-end)**:  
HTTP-запрос -> JAX-RS ресурс -> CDI -> EJB -> JPA -> База данных -> HTTP-ответ.

#### 1.13.4. Выводы

| Тип теста                   | Цель                                                                    | Инструменты                   | Среда выполнения  | Скорость      |
|-----------------------------|-------------------------------------------------------------------------|-------------------------------|-------------------|---------------|
| Юнит-тестирование           | Проверить логику одного класса в изоляции.                              | JUnit, Mockito                | Обычная JVM       | Очень быстрая |
| Интеграционное тестирование | Проверить, как несколько компонентов работают вместе внутри контейнера. | JUnit, Arquillian, ShrinkWrap | Сервер приложений | Медленная     |

#### Лучшая практика:

- Пишите **много** быстрых **юнит-тестов** для покрытия всей бизнес-логики, ветвлений, граничных случаев.
- Пишите **несколько** ключевых **интеграционных тестов** для проверки "счастливых путей" (happy paths) и того, что компоненты правильно "склеены" между собой.

Arquillian — это сложный, но невероятно мощный инструмент, который позволяет писать настоящие, надежные интеграционные и end-to-end тесты для Jakarta EE приложений.

#### 1.13.5. Тестирование REST API

Тестирование REST API обычно включает проверку следующих аспектов:

- **Коды состояния HTTP:** Возвращает ли API правильные статусы (200 OK, 201 Created, 400 Bad Request, 404 Not Found, 403 Forbidden и т.д.)?
- **Тело ответа (Response Body):** Соответствует ли структура и содержимое JSON/XML ответа ожиданиям?
- **Заголовки (Headers):** Присутствуют ли в ответе необходимые заголовки (например, Content-Type)?

- **Обработка ошибок:** Корректно ли API обрабатывает невалидные входные данные?
- **Безопасность:** Правильно ли API реагирует на запросы с правильными и неправильными правами доступа?

Существует два основных подхода к тестированию REST API, созданных с помощью JAX-RS.

## 1. Юнит-тестирование JAX-RS Ресурса

Этот подход аналогичен юнит-тестированию любого другого класса. Мы тестируем **логику самого класса-ресурса в изоляции**, используя моки для всех его зависимостей.

**Цель:** Проверить, что ресурс правильно вызывает нижележащие сервисы и корректно формирует объект Response на основе их ответов. **Мы не тестируем HTTP-слой, сериализацию JSON или работу сервера.**

**Инструменты:** JUnit 5, Mockito.

### Пример

Тестируемый JAX-RS ресурс:

```
// src/main/java/com/example/ProductResource.java
import jakarta.inject.Inject;
import jakarta.ws.rs.*;
import jakarta.ws.rs.core.MediaType;
import jakarta.ws.rs.core.Response;
import java.util.Optional;

@Path("/products")
@Produces(MediaType.APPLICATION_JSON)
public class ProductResource {
 @Inject
 private ProductService productService;

 @GET
 @Path("/{id}")
 public Response getProductById(@PathParam("id") long id) {
 // Логика: вызываем сервис для поиска продукта
 Optional<Product> productOptional = productService.findById(id);
 // В зависимости от ответа сервиса, формируем разный HTTP-ответ
```



```

 return productOptional
 .map(product -> Response.ok(product).build()) // Если найден ->
 200 ОК с продуктом в теле
 .orElse(Response.status(Response.Status.NOT_FOUND).build());
 // Если не найден -> 404 Not Found
 }
}

```

### **Юнит-тест для ProductResource:**

```

// src/test/java/com/example/ProductResourceTest.java
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoExtension;
import jakarta.ws.rs.core.Response;
import java.util.Optional;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.mockito.Mockito.when;

@ExtendWith(MockitoExtension.class)
class ProductResourceTest {

 @Mock
 private ProductService productServiceMock; // Мокаем зависимость

 @InjectMocks
 private ProductResource productResource; // Создаем экземпляр ресурса

 @Test
 void getProductById_shouldReturn200Ok_whenProductExists() {
 // --- ARRANGE ---
 }
}

```

```

 long productId = 1L;
 Product product = new Product(productId, "Test Product");
 // Настраиваем мок: "когда сервис ищет продукт с id=1, вернуть
Optional с продуктом"
 when(productServiceMock.findById(productId)).thenReturn(Optional.of
(product));
 // --- АКТ ---
 Response response = productResource.getProductById(productId);
 // --- ASSERT ---
 // Проверяем, что статус ответа - 200 OK
 assertEquals(200, response.getStatus());
 // Проверяем, что в теле ответа находится наш объект продукта
 assertEquals(product, response.getEntity());
 }

```

```

@Test
void getProductById_shouldReturn404NotFound_whenProductDoesNotExist()
{
 // --- ARRANGE ---
 long nonExistentId = 99L;
 // Настраиваем мок: "когда сервис ищет продукт с id=99, вернуть
пустой Optional"
 when(productServiceMock.findById(nonExistentId)).thenReturn(Option
al.empty());
 // --- АКТ ---
 Response response = productResource.getProductById(nonExistentId);
 // --- ASSERT ---
 // Проверяем, что статус ответа - 404 Not Found
 assertEquals(404, response.getStatus());
}
}

```

**Преимущества:**

- **Быстрота:** Тесты выполняются мгновенно.
- **Изоляция:** Мы уверены, что тестируем только логику `ProductResource`, а не `ProductService`.

#### **Недостатки:**

- **Неполное покрытие:** Этот тест не проверяет, правильно ли JAX-RS обработает HTTP-запрос, десериализует параметры и сериализует ответ в JSON.

## **2. Интеграционное (End-to-End) тестирование REST API**

Этот подход проверяет **всю цепочку**, от HTTP-запроса до ответа. Мы отправляем реальный HTTP-запрос на запущенное приложение и анализируем полученный HTTP-ответ.

**Цель:** Проверить, что API работает так, как ожидает клиент, включая работу сети, сериализацию/десериализацию, интеграцию всех компонентов (JAX-RS, CDI, EJB, JPA) и корректность HTTP-ответов.

#### **Инструменты:**

- **Arquillian:** Для запуска приложения в контейнере.
- **REST-Assured:** Очень популярная и удобная Java-библиотека для написания тестов для REST API. Она предоставляет простой, читаемый DSL (Domain-Specific Language) для создания запросов и проверки ответов.

#### **Пример с Arquillian и REST-Assured**

**Задача:** Проверить полный сценарий получения продукта через REST API.

**Настройка:** (см. предыдущий ответ про Arquillian, нужно добавить зависимость на `rest-assured`).

#### **Тестовый класс:**

```
import io.restassured.http.ContentType;
import org.jboss.arquillian.container.test.api.Deployment;
import org.jboss.arquillian.container.test.api.RunAsClient;
import org.jboss.arquillian.junit5.ArquillianExtension;
import org.jboss.arquillian.test.api.ArquillianResource;
import org.jboss.shrinkwrap.api.ShrinkWrap;
import org.jboss.shrinkwrap.api.spec.WebArchive;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
```

```

import org.junit.jupiter.api.extension.ExtendWith;

import java.net.URL;

import static io.restassured.RestAssured.given;
import static org.hamcrest.Matchers.equalTo;

@ExtendWith(ArquillianExtension.class)
class ProductResourceIT {
 // Создаем микро-развертывание со всеми необходимыми классами
 @Deployment
 public static WebArchive createDeployment() {
 return ShrinkWrap.create(WebArchive.class, "test.war")
 .addClasses(Product.class, ProductService.class,
 ProductResource.class, RestActivator.class)
 .addAsResource("test-persistence.xml", "META-INF/persistence.xml")
 .addAsWebInfResource("beans.xml");
 }

 // Arquillian внедрит сюда базовый URL нашего развернутого приложения
 @ArquillianResource
 private URL deploymentUrl;

 // Вспомогательный метод для подготовки данных в БД перед каждым тестом
 @BeforeEach
 void setup(@ArquillianResource URL url) {
 // ... здесь может быть код, который через отдельный API
 // или напрямую в БД добавляет тестовые данные, например, продукт
 // с ID=1
 }
}

```

```
}
```

```
@Test
```

```
@RunAsClient // Эта аннотация говорит Arquillian, что тест нужно
запускать НЕ внутри контейнера, а как внешний клиент
```

```
void shouldGetProductAndReturn200() {
```

```
 // --- ARRANGE ---
```

```
 // Данные подготовлены в setup()
```

```
 // --- ACT & ASSERT ---
```

```
 // Используем REST-Assured DSL
```

```
 given()
```

```
 // Ничего не передаем в теле, просто делаем GET
```

```
 .when()
```

```
 // Отправляем GET-запрос на URL нашего API
```

```
 .get(deploymentUrl + "api/products/1")
```

```
 .then()
```

```
 // Проверяем ответ
```

```
 .statusCode(200) // Ожидаем статус 200 OK
```

```
 .contentType(ContentType.JSON) // Ожидаем, что Content-Type
будет application/json
```

```
 .body("id", equalTo(1)) // Проверяем, что в JSON-теле поле "id"
равно 1
```

```
 .body("name", equalTo("Test Product From DB")); // И поле
"name" имеет нужное значение
```

```
}
```

```
@Test
```

```
@RunAsClient
```

```
void shouldReturn404ForNonExistentProduct() {
```

```
 given()
```

```

 .when()
 .get(deploymentUrl + "api/products/999")
 .then()
 .statusCode(404); // Ожидаем статус 404 Not Found
 }
}

```

### Разбор:

- **@RunAsClient:** Это важная аннотация. Она говорит, что код теста должен выполняться в отдельной JVM, как обычный HTTP-клиент, а не внутри сервера. Это позволяет нам использовать клиентские библиотеки, такие как REST-Assured.
- **@ArquillianResource URL:** Arquillian внедряет сюда URL, по которому доступно наше тестовое приложение (например, `http://localhost:8080/test/`).
- **REST-Assured DSL:**
  - `given():` Секция для настройки запроса (заголовки, тело, параметры).
  - `when():` Секция для выполнения действия (GET, POST, PUT, DELETE).
  - `then():` Секция для проверки ответа (статус, тело, заголовки).
- `equalTo():` Это матчер из библиотеки Hamcrest, который часто используется с REST-Assured для проверок.

### 1.13.6. Заключение и рекомендации

| Подход                             | Когда использовать                                                                                                           | Плюсы                                                                                      | Минусы                                                                                    |
|------------------------------------|------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------|
| <b>Юнит-тестирование</b>           | <b>Всегда.</b> Для проверки логики ветвлений, граничных случаев, маппинга данных в Response внутри класса-ресурса.           | <b>Очень быстро,</b> просто в настройке, не требует сервера.                               | <b>Не проверяет</b> HTTP-слой, сериализацию JSON, интеграцию с другими компонентами.      |
| <b>Интеграционное тестирование</b> | <b>Для ключевых сценариев.</b> Для проверки "счастливых путей" (happy paths), контракта API, безопасности, обработки ошибок. | <b>Проверяет все приложение целиком,</b> дает максимальную уверенность в корректности API. | <b>Медленно,</b> требует сложной настройки с Arquillian, тесты могут быть более хрупкими. |

**Лучшая практика:** Пишите много быстрых юнит-тестов для вашего JAX-RS ресурса, чтобы покрыть всю его внутреннюю логику. Затем напишите несколько более медленных, но всеобъемлющих интеграционных тестов с Arquillian и REST-Assured для проверки основных сценариев использования вашего API.

## 1.14. JEE и микросервисная архитектура

### 1.14.1. Eclipse MicroProfile: Обзор спецификаций

#### Что такое Eclipse MicroProfile?

**Eclipse MicroProfile** — это набор спецификаций с открытым исходным кодом, который определяет API и функциональность для создания **облачно-ориентированных (cloud-native) микросервисов** на Java.

**Ключевая идея:** Jakarta EE предоставляет фундаментальную базу для создания приложений (REST, CDI, JSON-B), а MicroProfile добавляет к ней спецификации, которые решают типичные проблемы распределенных систем:

- Внешняя конфигурация
- Проверка состояния сервиса
- Сбор метрик
- Отказоустойчивость
- Защита API
- Документирование API
- Распределенная трассировка

MicroProfile не заменяет Jakarta EE, а **дополняет** ее. Большинство современных серверов приложений, таких как Open Liberty, WildFly, Payara, Helidon, поддерживают и Jakarta EE, и MicroProfile.

Давайте рассмотрим основные спецификации MicroProfile.

### 1.14.2. MicroProfile Config (Конфигурация)

- **Проблема:** В микросервисной архитектуре конфигурация (адреса баз данных, ключи API, feature-флаги) должна быть **внешней** по отношению к приложению. Жестко прошивать ее в коде или в .war файле — плохая практика.
- **Решение:** MP Config предоставляет стандартный способ получения конфигурационных свойств из различных источников с четко определенным приоритетом.
  - **Источники (по убыванию приоритета):**
    1. Системные свойства Java (-Dmy.prop=value)
    2. Переменные окружения (export my\_prop=value)
    3. Файл META-INF/microprofile-config.properties внутри вашего приложения.
- **API:** Простое внедрение значений с помощью аннотации @Inject и @ConfigProperty.

**Пример кода:**



## **src/main/resources/META-INF/microprofile-config.properties**

```
greeting.message=Hello from properties file
payment.service.url=http://default-payment-service.com
payment.service.timeout=500
```

### **Java-код:**

```
import jakarta.enterprise.context.ApplicationScoped;
import jakarta.inject.Inject;
import org.eclipse.microprofile.config.inject.ConfigProperty;
import java.util.Optional;
```

```
@ApplicationScoped
```

```
public class GreetingService {
```

```
 // 1. Внедрение простого значения. Если свойство не найдено, будет ошибка
 при запуске.
```

```
 @Inject
```

```
 @ConfigProperty(name = "greeting.message")
```

```
 private String message;
```

```
 // 2. Внедрение с указанием значения по умолчанию.
```

```
 @Inject
```

```
 @ConfigProperty(name = "payment.service.url", defaultValue =
"http://fallback-url.com")
```

```
 private String paymentServiceUrl;
```

```
 // 3. Внедрение опционального значения.
```

```
 @Inject
```

```
 @ConfigProperty(name = "optional.feature.flag")
```

```
 private Optional<Boolean> featureFlag;
```

```
 public String greet() {
```

```

 // Если запустить с -Dgreeting.message="Hello from System Prop",
 // то будет использовано это значение, а не из файла.
 return message;
 }
}

```

### 1.14.3. MicroProfile Health (Проверка состояния)

- **Проблема:** В облачных средах (например, Kubernetes) оркестратор должен знать, "жив" ли сервис (liveness) и "готов" ли он принимать трафик (readiness).
- **Решение:** MP Health предоставляет стандартный способ создания HTTP-эндпоинтов (/health/live и /health/ready), которые возвращают статус сервиса в формате JSON.
- **API:** Создание CDI-бинов, реализующих интерфейс HealthCheck и помеченных аннотациями @Liveness или @Readiness.

#### Пример кода:

```

import jakarta.enterprise.context.ApplicationScoped;
import org.eclipse.microprofile.health.HealthCheck;
import org.eclipse.microprofile.health.HealthCheckResponse;
import org.eclipse.microprofile.health.Liveness;
import org.eclipse.microprofile.health.Readiness;

// Проверка "живучести". Если она не проходит, Kubernetes перезапустит
под.
@Liveness
@ApplicationScoped
public class ServiceLivenessCheck implements HealthCheck {
 @Override
 public HealthCheckResponse call() {
 // Проверяем, что сервис в принципе запущен и не завис
 return HealthCheckResponse.up("Service is alive");
 }
}

```

```
}
```

// Проверка "готовности". Если она не проходит, Kubernetes перестанет слать на под трафик.

```
@Readiness
```

```
@ApplicationScoped
```

```
public class DatabaseReadinessCheck implements HealthCheck {
```

```
 // @Inject private DataSource dataSource;
```

```
@Override
```

```
public HealthCheckResponse call() {
```

```
 if (isDatabaseConnected()) {
```

```
 return HealthCheckResponse.up("Database connection is ready");
```

```
 } else {
```

```
 return HealthCheckResponse.down("Cannot connect to the
database");
```

```
 }
```

```
}
```

```
private boolean isDatabaseConnected() {
```

```
 // ... логика проверки соединения с БД ...
```

```
 return true;
```

```
}
```

```
}
```

При запросе GET /health вы получите сводный ответ:

```
{
```

```
 "status": "UP",
```

```
 "checks": [
```

```
 {"name": "Service is alive", "status": "UP"},
```

```
 {"name": "Database connection is ready", "status": "UP"}]
```

```
]
```

```
}
```

#### 1.14.4. MicroProfile Metrics (Метрики)

- **Проблема:** Чтобы понимать, как работает сервис в продакшене, нужно собирать метрики: количество запросов, время ответа, использование памяти и т.д.
- **Решение:** MP Metrics предоставляет стандартный API для сбора и экспорта метрик в форматах, понятных системам мониторинга (например, Prometheus).
- **API:** Аннотации для декларативного сбора метрик (@Counted, @Timed, @Gauge) или программный API через MetricRegistry.

##### Пример кода:

```
import jakarta.enterprise.context.ApplicationScoped;
import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import org.eclipse.microprofile.metrics.annotation.Counted;
import org.eclipse.microprofile.metrics.annotation.Timed;

@Path("/inventory")
@ApplicationScoped
public class InventoryResource {
 @GET
 @Path("/{id}")
 @Counted(name = "products_searched_total", description = "Total number of
products searched.")
 @Timed(name = "product_search_duration", description = "Time taken to
search for a product.")
 public Product getProduct(@PathParam("id") String id) {
 // ... логика поиска ...
 return new Product(id, "My Product");
 }
}
```

Сервер автоматически создаст эндпоинт /metrics, который вернет метрики в формате Prometheus:

```
HELP products_searched_total Total number of products searched.
TYPE products_searched_total counter
products_searched_total 5.0
HELP product_search_duration_seconds Time taken to search for a product.
TYPE product_search_duration_seconds summary
product_search_duration_seconds_count 5.0
product_search_duration_seconds_sum 0.123
...
```

#### 1.14.5. MicroProfile Fault Tolerance (Отказоустойчивость)

- **Проблема:** В распределенной системе вызовы удаленных сервисов могут завершаться неудачно (сетевые ошибки, медленные ответы). Приложение должно уметь корректно обрабатывать такие сбои.
- **Решение:** MP Fault Tolerance предоставляет набор аннотаций для реализации популярных паттернов отказоустойчивости.
- **API (аннотации):**
  - @Timeout: Прерывает выполнение, если оно длится слишком долго.
  - @Retry: Повторяет вызов в случае сбоя.
  - @CircuitBreaker: "Размыкает цепь" после нескольких последовательных сбоев, чтобы не "добивать" неработающий сервис.
  - @Bulkhead: Ограничивает количество одновременных вызовов.
  - @Fallback: Предоставляет альтернативный путь выполнения в случае сбоя.

#### Пример кода:

```
import jakarta.enterprise.context.ApplicationScoped;
import org.eclipse.microprofile.faulttolerance.Fallback;
import org.eclipse.microprofile.faulttolerance.Retry;
import org.eclipse.microprofile.faulttolerance.Timeout;
import java.util.Collections;
import java.util.List;
```

```

@ApplicationScoped
public class RecommendationService {
 @Timeout(1000) // Таймаут в 1 секунду
 @Retry(maxRetries = 3, delay = 200) // 3 попытки с задержкой 200 мс
 @Fallback(fallbackMethod = "getFallbackRecommendations") // Если все
попытки провалились
 public List<Product> getRecommendations(String userId) {
 // ... вызов внешнего, потенциально нестабильного, ML-сервиса ...
 // Этот вызов может упасть по таймауту или бросить исключение
 return callRemoteMLService(userId);
 }

 // Метод-фолбэк должен иметь ту же сигнатуру
 public List<Product> getFallbackRecommendations(String userId) {
 System.out.println("Falling back to default recommendations.");
 // Возвращаем пустой список или какие-то стандартные рекомендации
 return Collections.emptyList();
 }
}

```

#### 1.14.6. MicroProfile JWT Authentication (Аутентификация по JWT)

- **Проблема:** В микросервисах классическая аутентификация по сессиям не подходит. Нужен stateless-механизм. **JSON Web Token (JWT)** является стандартом для этого.
- **Решение:** MP JWT Auth определяет стандартный способ защиты JAX-RS эндпоинтов с помощью JWT.
- **Как работает:** Вы настраиваете ожидаемого издателя токена (issuer) и публичный ключ для проверки его подписи. Сервер автоматически проверяет входящие токены в заголовке Authorization: Bearer ... и предоставляет информацию из токена (claims) через стандартные API (JsonWebToken, Principal).

**Пример кода:**

**microprofile-config.properties**

```
Ожидаемый издатель токена
mp.jwt.verify.issuer=https://my-auth-server.com
Путь к публичному ключу для проверки подписи
mp.jwt.verify.publickey.location=/META-INF/publicKey.pem
```

### **Защищенный ресурс:**

```
import jakarta.annotation.security.RolesAllowed;
import jakarta.enterprise.context.RequestScoped;
import jakarta.inject.Inject;
import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import org.eclipse.microprofile.jwt.JsonWebToken;
```

```
@Path("/profile")
```

```
@RequestScoped
```

```
public class UserProfileResource {
```

```
 // Внедряем распарсенный и проверенный JWT
```

```
 @Inject
```

```
 private JsonWebToken jwt;
```

```
 @GET
```

```
 @RolesAllowed("USER") // Проверяет claim "groups" в токене
```

```
 public String getProfile() {
```

```
 // Получаем имя пользователя из claim "upn" (User Principal Name)
```

```
 String username = jwt.getName();
```

```
 // Получаем кастомный claim
```

```
 Long userId = jwt.getClaim("user_id");
```

```
 return "Profile for user " + username + " (ID: " + userId + ")";
```

```
 }
```

```
}
```

### **1.14.7. MicroProfile OpenAPI (Документирование API)**

- **Проблема:** API должен быть документирован. Писать документацию вручную — долго и подвержено ошибкам.
- **Решение:** MP OpenAPI автоматически генерирует документацию в формате **OpenAPI 3.0** на основе вашего JAX-RS кода и специальных аннотаций.
- **API:** Аннотации из пакета `org.eclipse.microprofile.openapi.annotations`.



### Пример кода:

```
import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.Produces;
import jakarta.ws.rs.core.MediaType;
import org.eclipse.microprofile.openapi.annotations.Operation;
import org.eclipse.microprofile.openapi.annotations.media.Content;
import org.eclipse.microprofile.openapi.annotations.media.Schema;
import org.eclipse.microprofile.openapi.annotations.responses.APIResponse;

@Path("/greetings")
public class GreetingResource {

 @GET
 @Produces(MediaType.APPLICATION_JSON)
 @Operation(summary = "Returns a friendly greeting", description = "Generates
a greeting object.")
 @APIResponse(
 responseCode = "200",
 description = "A successful greeting.",
 content = @Content(mediaType = "application/json",
 schema = @Schema(implementation = Greeting.class))
)
 public Greeting greet() {
 return new Greeting("Hello, OpenAPI!");
 }
}
```

Сервер создаст эндпоинт `/openapi`, который вернет `openapi.yaml` или `.json` файл, на основе которого можно сгенерировать красивый UI (например, Swagger UI) и клиентский код.

### 1.14.8. MicroProfile OpenTracing (Распределенная трассировка)

- **Проблема:** В микросервисной архитектуре один запрос может проходить через несколько сервисов. Если что-то пошло не так, как найти, в каком именно сервисе произошла ошибка или задержка?
- **Решение:** МР OpenTracing предоставляет стандарт для **распределенной трассировки**. Он автоматически создает и передает между сервисами идентификаторы трассировки и спанов (span), позволяя системам вроде **Jaeger** или **Zipkin** построить визуальный граф вызовов.
- **Как работает:** В большинстве случаев работает "из коробки". Адаптер автоматически перехватывает входящие и исходящие JAX-RS вызовы и добавляет в них нужные заголовки.

#### Пример кода:

```
import jakarta.inject.Inject;
import jakarta.ws.rs.client.Client;
import jakarta.ws.rs.client.ClientBuilder;
import org.eclipse.microprofile.opentracing.Traced;
```

```
@ApplicationScoped
```

```
public class MyService {
```

```
 @Inject
```

```
 private Tracer tracer; // Можно внедрить Tracer для создания кастомных спанов
```

```
 // Аннотация @Traced автоматически создаст span для этого метода
```

```
 @Traced(operationName = "call-downstream-service")
```

```
 public String callAnotherService() {
```

```
 // Создаем JAX-RS клиент. МР автоматически добавит в него
```

```
 // interceptor, который пробросит заголовки трассировки.
```

```
 Client client = ClientBuilder.newClient();
```

```
 return client.target("http://another-
service/api/data").request().get(String.class);
```

```
 }
```

```
}
```

Этот набор спецификаций делает разработку микросервисов на Java стандартизированной, надежной и готовой к работе в облачных средах.

#### 1.14.9. Разработка легковесных Jakarta EE-приложений для микросервисов

Основная идея — отказаться от "всего и сразу" (Full Platform) в пользу минималистичного, но достаточного набора технологий.

### 1. Выбор правильного профиля и сервера приложений

Это первый и самый важный шаг.

- **Профили:**

- **Jakarta EE Web Profile:** Это отличная отправная точка. Он содержит все необходимое для создания REST API (JAX-RS, CDI, JSON-B, JPA, Bean Validation), но исключает более тяжелые корпоративные спецификации.
- **Eclipse MicroProfile:** Как мы обсуждали ранее, он добавляет критически важные для микросервисов возможности (конфигурация, health checks, метрики, отказоустойчивость).
- **Идеальная комбинация: Jakarta EE Web Profile + MicroProfile.**

- **Серверы приложений, оптимизированные для микросервисов:**

- **Open Liberty:** Один из лидеров в этой области. Модульный, динамичный (изменения конфигурации на лету), очень быстрый старт и низкое потребление памяти.
- **Payara Micro / Helidon / Quarkus / WildFly:** Также предоставляют легковесные версии или специальные режимы для запуска микросервисов.
  - **Quarkus** заслуживает отдельного упоминания. Он использует компиляцию в нативный код (GraalVM) для достижения феноменально быстрого старта и минимального потребления памяти, при этом поддерживая API Jakarta EE и MicroProfile.

### 2. Принципы проектирования микросервиса на Jakarta EE

#### 1. Ставка на CDI, а не на EJB:

- Для большинства задач достаточно CDI-бинов (@ApplicationScoped, @RequestScoped). Они легковесны и просты.
- Используйте **Stateless EJB (@Stateless)** только тогда, когда вам действительно нужен его сервис, в первую очередь — **декларативное управление транзакциями JTA (@Transactional)**. Это гораздо проще и надежнее, чем управлять транзакциями вручную.

#### 2. Stateless-архитектура:

- Микросервисы должны быть stateless (не хранить состояние). Это позволяет легко их масштабировать горизонтально (просто запуская больше экземпляров).
- Избегайте @SessionScoped CDI-бинов или @Stateful EJB. Состояние пользователя (например, сессия) должно храниться либо на клиенте (в JWT-токене), либо во внешнем хранилище (например, Redis).

### 3. Внешняя конфигурация (MicroProfile Config):

- Никогда не прошивайте конфигурацию в код. Используйте @ConfigProperty для получения URL-адресов других сервисов, ключей API, данных для подключения к БД и т.д. Это позволяет одному и тому же артефакту (.jar или .war) работать в разных окружениях (dev, test, prod) без пересборки.

### 4. Отказоустойчивость (MicroProfile Fault Tolerance):

- Любой сетевой вызов к другому микросервису или базе данных должен быть обернут в паттерны отказоустойчивости. Как минимум, используйте @Timeout и @Retry. Для критически важных вызовов — @CircuitBreaker и @Fallback.

## 5. Наблюдаемость (Observability):

- Каждый микросервис должен быть "прозрачным". Сразу же добавляйте:
  - **Health Checks (@Liveness, @Readiness):** Чтобы оркестраторы (Kubernetes) могли им управлять.
  - **Метрики (@Counted, @Timed):** Чтобы вы могли отслеживать его производительность (с помощью Prometheus, Grafana).
  - **Трассировку (@Traced):** Чтобы можно было отслеживать запросы между сервисами (с помощью Jaeger, Zipkin).

## 3. Пример: создание простого микросервиса "Каталог товаров"

Давайте создадим микросервис, который предоставляет информацию о товарах.

### Технологический стек:

- Jakarta EE 10 (Web Profile)
- MicroProfile 6.0
- Сервер: Open Liberty
- Сборка: Maven

### Структура проекта (pom.xml):

```
<dependencies>
 <!-- Jakarta EE Web Profile -->
 <dependency>
 <groupId>jakarta.platform</groupId>
 <artifactId>jakarta.jakartaee-web-api</artifactId>
 <version>10.0.0</version>
 <scope>provided</scope>
 </dependency>
 <!-- Eclipse MicroProfile -->
 <dependency>
 <groupId>org.eclipse.microprofile</groupId>
 <artifactId>microprofile</artifactId>
 <version>6.0</version>
 <type>pom</type>
 <scope>provided</scope>
 </dependency>
```

</dependencies>

**src/main/resources/META-INF/microprofile-config.properties**

// Сущность Product (JPA)

@Entity

public class Product { /\* id, name, price \*/ }

// Репозиторий для доступа к данным (CDI бин)

@ApplicationScoped

public class ProductRepository {

@PersistenceContext

private EntityManager em;

public Optional<Product> findById(long id) {

return Optional.ofNullable(em.find(Product.class, id));

}

}

@ApplicationScoped

public class CatalogService {

@Inject

private ProductRepository repository;

@Inject

@ConfigProperty(name = "catalog.default.currency", defaultValue = "EUR")

private String defaultCurrency;

// Декларативная отказоустойчивость для потенциально медленного запроса к БД

@Timeout(500) // Таймаут 500 мс

@Retry(maxRetries = 2)

@Fallback(fallbackMethod = "findFallback")

public Optional<ProductView> findProduct(long id) {

return repository.findById(id)

```

 .map(p -> new ProductView(p.getName(), p.getPrice(),
defaultCurrency));
 }

 // Метод-фолбэк
 public Optional<ProductView> findFallback(long id) {
 // Можно вернуть кэшированное значение или пустоту
 return Optional.empty();
 }
}

// DTO для ответа
record ProductView(String name, BigDecimal price, String currency) {}

@Path("/products")
@ApplicationScoped
public class ProductResource {
 @Inject
 private CatalogService service;

 @GET
 @Path("/{id}")
 @Produces(MediaType.APPLICATION_JSON)
 // Метрики
 @Timed(name = "getProductById_timer", description = "Time to get a product
by ID.")
 @Counted(name = "getProductById_counter", description = "Count of
getProductById calls.")
 // Документация
 @Operation(summary = "Get a product by its ID")
 @ApiResponse(responseCode = "200", description = "Product found")

```

```

 @ApiResponse(responseCode = "404", description = "Product not found")
 public Response getProduct(@PathParam("id") long id) {
 return service.findProduct(id)
 .map(productView -> Response.ok(productView).build())
 .orElse(Response.status(Response.Status.NOT_FOUND).build());
 }
}

```

// Проверка готовности к работе

```

@Readiness
@ApplicationScoped
public class DatabaseHealthCheck implements HealthCheck {
 @Inject
 private ProductRepository repository;

 @Override
 public HealthCheckResponse call() {
 // Пробуем получить доступ к БД
 try {
 repository.findById(1L); // Просто тестовый вызов
 return HealthCheckResponse.up("Database Connection");
 } catch (Exception e) {
 return HealthCheckResponse.down("Database Connection");
 }
 }
}

```

#### 4. Упаковка и запуск

Приложения для микросервисов часто упаковываются как **"толстые" или "исполняемые" JAR-файлы (fat/executable JAR)**, которые содержат в себе и код приложения, и сам сервер. Это упрощает развертывание, особенно в контейнерах Docker.



Плагины для Maven, такие как `open-liberty-maven-plugin` или `quarkus-maven-plugin`, позволяют сделать это одной командой.

**Пример команды для Open Liberty:**

```
mvn package liberty:run
```

Эта команда:

1. Соберет ваше приложение.
2. Скачает сервер Open Liberty.
3. Установит только те фичи Jakarta EE и MicroProfile, которые вы используете.
4. Запустит сервер с вашим приложением.

### Dockerfile:

Создание Docker-образа становится тривиальным:

```
FROM open-liberty:kernel-java17
```

```
COPY --chown=1001:0 target/my-microservice.war /config/dropins/
```

```
COPY --chown=1001:0 src/main/liberty/config/server.xml /config/
```

### 1.14.10. Заключение

Чтобы создавать легковесные Jakarta EE микросервисы:

1. **Выбирайте правильный сервер и профиль:** Open Liberty, Payara Micro, Quarkus + Web Profile + MicroProfile.
2. **Проектируйте stateless-сервисы:** Используйте CDI как основу, EJB — для транзакций.
3. **Используйте MicroProfile для решения облачных задач:** Внешняя конфигурация, отказоустойчивость и наблюдаемость (health, metrics, tracing) — это не опции, а необходимость.
4. **Упаковывайте как исполняемый JAR:** Это упрощает контейнеризацию и развертывание.

Следуя этим принципам, вы можете использовать всю мощь и стандартизацию Jakarta EE для создания современных, эффективных и надежных микросервисов.

## 2. ПРАКТИЧЕСКИЙ РАЗДЕЛ

### 2.1. Программа лабораторных занятий

1. Запуск JSP/Servlet/JSP. (Hello world!)
  2. Подключение Log4J2.
  3. Передача числа от клиента на сервер, возвращение этого числа, умноженного на 2.
  4. Добавление артефакта предметной области системы. Занесение в базу данных информации, например, телефонной книги (номер и фамилия).
  5. Получение в ответ (в случае успешного добавления) списка всех абонентов\пользователей.
  6. Использование страниц об ошибках (404, 500 etc).
  7. Авторизация (sign in) и выход (sign out) в/из системы.
- Регистрационная форма MVC (использовать слои). Использовать шифрование пароля.
8. Валидация данных на клиенте и на сервере.
  9. Регистрация пользователя (sign up).
  10. Регистрация с подтверждением по почте.
  11. Модификация информации (например: редактирование профиля, создание и редактирование товара, заказа и т.д.)
  12. Удаление информации (например: отмена заказа, записи из телефонной книги и т.д.)
  13. Загрузка\выгрузка медиафайлов на сервер, в том числе и в/из БД.
  14. Пул соединений с БД. Применение Proxy.
  15. Фильтры. Модификация запроса. Проверка прав пользователя при вызове страницы. Установка кодировки запроса\ответа.
  16. События. Конфигурирование пула соединений. Отслеживание действий пользователя. Управление информацией в сессии.
  17. Локализация приложения. Изменение всех заголовков, имен и сообщений по нажатию кнопки локализации.
  18. Cookie. Хранение промежуточной информации между и во время сессии клиента с приложением.
  19. Шифрование паролей.
  20. AJAX. Асинхронные запросы.
  21. Постраничный вывод данных.
  22. Post-Redirect-Get. (Защита от F5)
  23. Custom tags library.
  24. Разработка unit-тестов, в том числе mock.
  25. Telegram bot.

## 2.2. Разработка информационной системы

Целями задания являются:

- закрепление магистрантами теоретических основ проектирования, создания и использования информационных систем;
- получение практических навыков создания проектов с помощью IDE;
- знакомство с технологиями использования проектирования и разработки информационных систем;
- оформление отчёта о проделанной работе.

Поставленные цели достигаются магистрантами путем проектирования и информационных систем, позволяющей автоматизировать отдельные внутренние и внешние бизнес-процессы некоторой организации (согласно выбранной магистрантами тематике).

Выполнение задания предполагает несколько последовательных шагов, соответствующих основным этапам жизненного цикла разработки информационных систем:

1) проектирование — на данном этапе определяется предметная область (часть реального мира, которая подлежит изучению) и на ее основе строится информационно-логическая модель (таблицы вариантов использования);

2) создание — на основе информационно-логической модели с помощью любого выбранного магистрантом средства разработки;

Результаты первого, второго и второго этапов заносятся в письменный отчет к заданию, который должен содержать:

Титульный лист с указанием названия проекта, фамилии, имени и учебной группы автора, а также года выполнения.

Постановку задачи.

Отчёт должен быть подготовлен в MS Word или Google Docs с использованием стилей, созданием автоматически обновляемого оглавления. В конце отчёта привести список использованных источников (не менее двух печатных изданий и двух электронных изданий),

К заданию, кроме отчёта прикрепить файл с кодом приложения.

При защите проекта обязательным является:

знание и правильное употребление основных терминов, используемых при выполнении задания;

знание потенциальных возможностей;

умение объяснять взаимосвязь всех сущностей и отношений, присутствующих в информационно-логической модели, и объектов, созданных на их основе.

Примерный перечень предметных областей для разработки информационной системы:

- Построить веб-систему, поддерживающую заданную функциональность: — на основе сущностей предметной области создать классы, их описывающие; — классы и методы должны иметь названия отражающие их функциональность, и быть грамотно структурированы по пакетам;
- оформление кода должно соответствовать Java Code Convention;
  - информацию о предметной области хранить в БД, для доступа использовать API JDBC с использованием пула соединений, стандартного или разработанного самостоятельно. В качестве СУБД рекомендуется MySQL или Derby;
  - приложение должно поддерживать работу с кириллицей (быть многоязычной), в том числе и при хранении информации в БД;
  - архитектура приложения должна соответствовать шаблону Model-View-Controller;
  - при реализации алгоритмов бизнес-логики использовать шаблоны GoF: Factory Method, Command, Builder, Strategy, State, Observer etc.
  - используя сервлеты и JSP, реализовать функции, предложенные в постановке конкретной задачи;
    - в страницах JSP применять библиотеку JSTL;
    - при разработке бизнес-логики использовать сессии и фильтры;
  - использовать Log4j;
    - код должен содержать комментарии.
1. Система **Факультатив**. Существует перечень **Курсов**, за каждым из которых закреплен один **Преподаватель**. **Студент** записывается на один или несколько **Курсов**. По окончании обучения **Преподаватель** выставляет **Студенту** и добавляет отзыв.
  2. Система **Платежи**. **Клиент** имеет одну или несколько **Кредитных Карт**, каждая из которых соответствует некоторому **Счету** в системе платежей. **Клиент** может при помощи **Счета** сделать **Платеж**, заблокировать **Счет** и пополнить **Счет**. **Администратор** снимает блокировку.
  3. Система **Приемная комиссия**. **Абитуриент** регистрируется на один из **Факультетов** с фиксированным планом набора, вводит баллы по соответствующим **Предметам** и **аттестату**. Результаты **Администратором** регистрируются в **Ведомости**. Система подсчитывает сумму баллов и определяет **Абитуриентов**, зачисленных в учебное заведение.
  4. Система **Библиотека**. **Читатель** имеет возможность осуществлять поиск и заказ **Книг** в **Каталоге**. **Библиотекарь** выдает **Читателю Книгу** на абонемент или в читальный зал. **Книга** может присутствовать в **Библиотеке** в одном или нескольких экземплярах.

5. Система **Больница**. **Врач** определяет диагноз, делает назначение **Пациенту** (процедуры, лекарства, операции). Назначение может выполнить **Медсестра** (процедуры, лекарства) или **Врач** (любое назначение). **Пациент** может быть выписан из **Больницы**, при этом фиксируется окончательный диагноз.
6. Система **Турагентство**. **Заказчик** выбирает и оплачивает **Тур** (отдых, экскурсия, шоппинг). **Турагент** определяет тур как «горящий», размеры ски-док постоянным клиентам.
7. Система **Телефонная станция**. **Администратор** осуществляет подключение **Абонентов**. **Абонент** может выбрать одну или несколько из предоставляемых **Услуг**. **Абонент** оплачивает **Счет** за разговоры и **Услуги**. **Администратор** может просмотреть список неоплаченных **Счетов** и заблокировать **Абонента**.
8. Система **Автобаза**. **Диспетчер** распределяет **Заявки** на **Рейсы** между **Водителями**, за каждым из которых закреплен свой **Автомобиль**. На **Рейс** может быть назначен **Автомобиль**, находящийся в исправном состоянии и характеристики которого соответствуют **Заявке**. **Водитель** делает отметку о выполнении **Рейса** и состоянии **Автомобиля**.
9. Система **Интернет-магазин**. **Администратор** осуществляет ведение каталога **Товаров**. **Клиент** делает и оплачивает **Заказ** на **Товары**. **Администратор** может занести неплательщиков в «черный список».
10. Система **Авиакомпания**. **Авиакомпания** имеет список рейсов. **Диспетчер** формирует летную **Бригаду** (пилоты, штурман, радист, стюардессы) на **Рейс**. **Администратор** управляет списком рейсов.
11. Система **LowCost-Авиакомпания**. **Клиент** заказывает и оплачивает **Билет** на **Рейс** с учетом наличия/отсутствия багажа и права первоочередной регистрации и посадки (**Цена Билета** может быть ниже стоимости провоза багажа). С приближением даты **Рейса** или наполнением самолета цена на **Билет** может повышаться.
12. Система **Периодические издания**. **Администратор** осуществляет ведение каталога периодических **Изданий**. **Читатель** может оформить **Подписку**, предварительно выбрав периодические **Издания** из списка. Система подсчитывает стоимость и регистрирует **Платеж**.
13. Система **Заказ гостиницы**. **Клиент** заполняет **Заявку**, указывая количество мест в номере, класс апартаментов и время пребывания. **Администратор** просматривает поступившую **Заявку**, выделяет наиболее подходящий из доступных **Номеров**, после чего система выставляет **Счет Клиенту**.
14. Система **Жилищно-коммунальные услуги**. **Квартиросъемщик** отправляет **Заявку**, в которой указывает род работ, масштаб и желаемое время выполнения. **Диспетчер** формирует соответствующую **Бригаду** и регистрирует её в **Плане работ**.

15. Система **Прокат автомобилей**. Клиент выбирает **Автомобиль** из списка доступных. Заполняет форму **Заказа**, указывая паспортные данные, срок аренды. Клиент оплачивает **Заказ**. Администратор регистрирует возврат автомобиля. В случае повреждения **Автомобиля** Администратор вносит информацию и выставляет счет за ремонт. Администратор может отклонить **Заявку**, указав причины отказа.
16. Система **Скачки**. Клиент делает **Ставки** разных видов на **Забег**. Букмекер устанавливает уровень выигрыша. Администратор фиксирует результаты **Забегов**.
17. Система **Тестирование**. Тьютор создает **Тест** из нескольких **Вопросов** закрытого типа (выбор одного или более вариантов из N предложенных) по определенному **Предмету**. Студент просматривает список доступных **Тестов**, отвечает на **Вопросы**.
18. Система **Ресторан**. Клиент осуществляет заказ из **Меню**. Администратор подтверждает **Заказ** и отправляет его на кухню для исполнения. Администратор выставляет **Счет**. Клиент производит его оплату.
19. Система **Кофе-машина**. Пользователь обладает **Счетом**. Кофе-машина содержит набор **Напитков**, с заданным числом порций и дополнительных **Ингредиентов**. Пользователь может купить один или несколько **Напитков**. Администратор Кофе-машины осуществляет ее наполнение.
20. Система **Парк**. Владелец парка дает указания **Леснику** о высадке (лечении, художественной обработке, уничтожении) **Растений**. Лесник отчитывается о выполнении. Владелец просматривает результаты и подтверждает исполнение.
21. Система **Команда разработчиков**. Заказчик представляет **Техническое Задание (ТЗ)**, в котором перечислен перечень **Работ** с указанием квалификации и количества требуемых специалистов. Менеджер рассматривает **ТЗ** и оформляет **Проект**, назначая на него незанятых **Разработчиков** требуемой квалификации, после чего рассчитывается стоимость **Проекта** и **Заказчику** выставляется **Счет**. Разработчик имеет возможность отменить количество часов, затраченных на работу над проектом.
22. Система **Железнодорожная касса\***. Пассажир делает **Заявку** на билет до необходимой ему станции назначения, время и дату поездки. Система осуществляет поиск подходящего **Поезда**. Пассажир делает выбор **Поезда** и получает **Счет** на оплату. Администратор управляет списком зарегистрированных пассажиров.
23. Система **Городской транспорт\***. На **Маршрут** назначаются **Автобус**, **Троллейбус** или **Трамвай**. Транспортные средства должны двигаться с определенным для каждого **Маршрута** интервалом или расписанием.

### 3. РАЗДЕЛ КОНТРОЛЯ ЗНАНИЙ

#### 3.1. Перечень рекомендуемых средств диагностики

Диагностика результатов учебной деятельности по дисциплине «Разработка Веб-приложений» проводится, как правило, во время аудиторных занятий. Для диагностики используются:

- устный опрос;
- отчет по лабораторным и домашним заданиям.

Контроль УСП проводится преподавателем с использованием ИКТ в форме опроса и проверки результатов выполнения работы.

Формой промежуточной аттестации по дисциплине «Разработка Веб-приложений» учебным планом предусмотрен зачет.

#### 3.2. Примерный перечень заданий для управляемой самостоятельной работы магистрантов

##### Установка и конфигурирование IDE. (2ч)

##### Задание 1. Загрузка и установка IDE

Форма контроля – опрос, отчет по лабораторным / домашним заданиям.

#### 3.3. Описание инновационных подходов и методов к преподаванию учебной дисциплины

При организации образовательного процесса используются

- **методы и приемы развития критического мышления**, которые представляют собой систему, формирующую навыки работы с информацией в процессе чтения и письма; понимании информации как отправного, а не конечного пункта критического мышления;

- **практико-ориентированный подход**, который предполагает освоение содержание образования через решения практических задач.

#### 3.4. Методические рекомендации по организации самостоятельной работы обучающихся

При изучении учебной дисциплины рекомендуется использовать следующие формы самостоятельной работы:

- изучение литературы и материалов электронных источников по проблемам дисциплины;
- выполнение домашнего задания;
- подготовка к лабораторным занятиям;



- курсовые, дипломные и научно-исследовательские работы, связанные с тематикой дисциплины;

- подготовка к участию в конференциях с докладами по проблемам дисциплины.

Для организации самостоятельной работы магистрантов по учебной дисциплине рекомендуется использовать современные информационные ресурсы, размещенные на образовательном портале смешанного и дистанционного обучения БГУ, содержащие учебные материалы (курс лекций, задания к лабораторным и домашним работам, перечень вопросов к зачёту и т. д.)

### **3.5. Примерный перечень вопросов к зачету**

1. Что такое Java-сервлеты и как они связаны с веб-приложениями?
2. Как создать Java-сервлет и какие методы жизненного цикла они имеют?
3. Что такое контейнер сервлетов и как он обрабатывает запросы и ответы?
4. Как получить параметры запроса в Java-сервлете?
5. Как передать данные между сервлетами?
6. Что такое JSP (JavaServer Pages) и как они используются в веб-приложениях?
7. Как объявить и использовать переменные в JSP?
8. Что такое скриптлеты в JSP и как они используются?
9. Как использовать выражения JSP для вывода данных на веб-страницу?
10. Какие директивы JSP вы знаете и для чего они нужны?
11. Что такое JSP-теги и как они используются?
12. Какие стандартные теги JSTL (JSP Standard Tag Library) вы знаете и для чего они предназначены?
13. Как использовать циклы и условные операторы в JSP с помощью JSTL?
14. Как использовать JSTL для работы с коллекциями данных?
15. Как обрабатывать исключения в сервлетах и JSP?
16. Как реализовать сессии в Java-сервлетах?
17. Какие методы HTTP запросов вы знаете и как их использовать в сервлетах?
18. Как отправить перенаправление (redirect) в сервлете?
19. Как работать с файлами в сервлетах?
20. Как обрабатывать формы HTML в сервлетах?
21. Как работать с куками (cookies) в сервлетах?
22. Как обеспечить безопасность веб-приложения с помощью сервлетов и JSP?
23. Как использовать базы данных в сервлетах и JSP?
24. Какие средства тестирования и отладки вы знаете для сервлетов и JSP?

25. Как работать с сеансами (sessions) в JSP?
26. Как ограничить доступ к определенным страницам или функциональности в веб-приложении?
27. Как обрабатывать загрузку файлов в сервлетах?
28. Как использовать фильтры (filters) в сервлетах?
29. Как обрабатывать AJAX-запросы в сервлетах и JSP?
30. Как развернуть веб-приложение, содержащее сервлеты и JSP, на сервере приложений?
31. Что делает `RequestDispatcher.include()`?
32. В какой последовательности выполняются сервлет фильтры?
33. Как обрабатывается тег с телом?
34. Что нужно написать в строке браузера, чтобы обратиться к хосту, на котором установлен tomcat, развернуто приложение, в котором есть несколько сервлетов? Как обратиться к конкретному сервлету? Что такое `www`? Где нужно указывать порт?
35. Можно ли в `web.xml` определить сервлет без указания url паттерна и как к нему обратиться?
36. Что такое HTTP? Отличия HTTP 1.0 и HTTP 2.
37. Сохраняет ли http протокол своё состояние.
38. Как сервер понимает, что для пользователя создана сессия и не нужно её создавать?
39. Отличие авторизации от аутентификации.

## 4. ВСПОМОГАТЕЛЬНЫЙ РАЗДЕЛ

### 4.1. Рекомендуемая литература

#### Основная литература

1. Эккель, Брюс. Философия Java = Thinking in Java / Брюс Эккель ; [пер. с англ. Е. Матвеев]. - 4-е полное изд. - Санкт-Петербург [и др.] : Питер, 2023. - 1168 с.
2. Блинов, И. Н. Java from EPAM : учебно-методическое пособие [для студентов высших учебных заведений, обучающихся по специальностям 1-31 03 08 "Математика и информационные технологии (по направлениям)", 1-31 03 01 "Математика (по направлениям)"] / И. Н. Блинов, В. С. Романчик. - Минск : Четыре четверти : EPAM Training Center, 2020. - 559 с.
3. Белугина, С. В. Разработка программных модулей программного обеспечения для компьютерных систем. Прикладное программирование : учебное пособие / С. В. Белугина. — Санкт-Петербург : Лань, 2020. — 312 с. — Текст : электронный // Лань : электронно-библиотечная система. — [Электронный ресурс]. – Режим доступа: <https://e.lanbook.com/book/133920>.
4. Гонсалвес, Э. Изучаем Java EE 7. - Санкт-Петербург [и др.] : Питер, 2018. - 640 с.

#### Дополнительная литература

1. Блинов И.Н., Романчик В.С. Методы программирования Java. – Четыре четверти, 2013. – 896 с.
2. Хорстманн К. С., Корнелл Г. Библиотека профессионала. Java 2. Том 1. Основы. – М.: “Вильямс”, 2019. – 866 с.
3. Блинов И.Н., Романчик В.С. Java. Промышленное программирование. – Универсалпресс, 2007. – 704 с.

## 4.2. Учебно-методическая карта учебной дисциплины

Дневная форма получения образования

| Номер<br>раздела,<br>темы | Название раздела,<br>темы | Количество<br>аудиторных часов |              |              |      | Ко<br>личество<br>о часов<br>УСР | Формы<br>контроля<br>знаний |
|---------------------------|---------------------------|--------------------------------|--------------|--------------|------|----------------------------------|-----------------------------|
|                           |                           | Лекции                         | Практические | Лабораторные | Иное |                                  |                             |
|                           | 2                         |                                |              | 5            | 6    | 7                                | 8                           |

|  |                                                                                                                             |  |  |   |  |  |                                                 |
|--|-----------------------------------------------------------------------------------------------------------------------------|--|--|---|--|--|-------------------------------------------------|
|  | Архитектура платформы Jakarta EE: Контейнерная модель Серверы приложений Jakarta EE: Обзор, установка, базовая конфигурация |  |  | 2 |  |  | опрос                                           |
|  | Профили, архитектура, паттерны и инструменты Jakarta EE                                                                     |  |  | 4 |  |  | Опрос, отчет по лабораторным / домашним работам |
|  | Сервлеты                                                                                                                    |  |  | 4 |  |  | Опрос                                           |
|  | Java Server Page                                                                                                            |  |  | 4 |  |  | Опрос, отчет по лабораторным / домашним работам |
|  | Сессии, события и фильтры JSP Standard Tag Library                                                                          |  |  | 4 |  |  | Опрос                                           |
|  | JSP и JSF                                                                                                                   |  |  | 4 |  |  | Опрос, отчет по лабораторным / домашним работам |
|  | Ядро корпоративных компонентов Управление данными и транзакциями                                                            |  |  | 4 |  |  | Опрос                                           |

|  |                                                                                                    |          |          |           |  |  |                                                 |
|--|----------------------------------------------------------------------------------------------------|----------|----------|-----------|--|--|-------------------------------------------------|
|  | Безопасность и валидация данных<br>Тестирование JEE приложений<br>JEE и микросервисная архитектура |          |          | 4         |  |  | Опрос, отчет по лабораторным / домашним работам |
|  | <b>ВСЕГО ЧАСОВ</b>                                                                                 | <b>4</b> | <b>-</b> | <b>30</b> |  |  | <b>Зачет</b>                                    |

### Заочная форма получения образования

| Номер раздела, темы | Название раздела, темы | Количество аудиторных часов |                         |                      |      | Количество часов УСР | Формы контроля знаний |
|---------------------|------------------------|-----------------------------|-------------------------|----------------------|------|----------------------|-----------------------|
|                     |                        | Лекции                      | Практические (семинары) | Лабораторные занятия | Иное |                      |                       |
|                     | 2                      |                             | 4                       | 5                    | 6    | 7                    | 8                     |

|        |                                                                                                                             |        |  |         |  |  |                                                 |
|--------|-----------------------------------------------------------------------------------------------------------------------------|--------|--|---------|--|--|-------------------------------------------------|
| -<br>2 | Архитектура платформы Jakarta EE: Контейнерная модель Серверы приложений Jakarta EE: Обзор, установка, базовая конфигурация |        |  | 0<br>,5 |  |  | Опрос                                           |
|        | Профили, архитектура, паттерны и инструменты Jakarta EE                                                                     |        |  | 0<br>,5 |  |  | Опрос                                           |
|        | Сервлеты                                                                                                                    | ,<br>5 |  | 1       |  |  | Опрос                                           |
|        | Java Server Page                                                                                                            | ,<br>5 |  | 1       |  |  | Опрос                                           |
|        | Сессии, события и фильтры JSP Standard Tag Library                                                                          |        |  | 1       |  |  | Опрос                                           |
|        | JSP и JSF                                                                                                                   |        |  | 1       |  |  | Опрос                                           |
|        | Ядро корпоративных компонентов Управление данными и транзакциями                                                            |        |  | 1       |  |  | Опрос, отчет по лабораторным / домашним работам |

|  |                         |  |          |          |  |  |              |
|--|-------------------------|--|----------|----------|--|--|--------------|
|  | <b>ВСЕГ<br/>О ЧАСОВ</b> |  | <b>-</b> | <b>6</b> |  |  | <b>Зачет</b> |
|--|-------------------------|--|----------|----------|--|--|--------------|

### 4.3. Электронные ресурсы

1. Электронная библиотека БГУ [Электронный ресурс]. – Режим доступа: <https://library.bsu.by/MegaPro/Web/SearchResult/ToPage/1>. – Дата доступа : 29.01.2025.

2. Электронная библиотека БГУ [Электронный ресурс]. – Режим доступа: <https://library.bsu.by/MegaPro/Web/SearchResult/ToPage/1>. – Дата доступа : 29.01.2025.

3. Электронная библиотека БГУ [Электронный ресурс]. – Режим доступа: <https://library.bsu.by/MegaPro/Web/SearchResult/ToPage/1>. – Дата доступа : 29.01.2025.