Технологии программирования на Java

Содержание

Лекция 1. Вводная лекция: Java и JVM **8**

О чём курс . . . 8
Историческая справка: предпосылки появления Java . . . 9
Эволюция языков программирования . . . 9
Рождение Java . . . 9
Managed vs Unmanaged Code . . . 10
Байт-код и JVM . . . 10
Три кита Java-платформы . . . 10
Три части JVM . . . 10
Пример: Java-код и его байт-код . . . 11
Основы синтаксиса Java . . . 11
Типы данных, операции, управляющие конструкции . . . 11
Синтаксис классов . . . 12
Исключения . . . 12
JavaDoc . . . 12
Что такое сборка мусора? . . . 14
Проблема утечек памяти в unmanaged-языках . . . 14
Почему утечки памяти — серьёзная проблема . . . 14
Автоматическое управление памятью . . . 14
Подходы к обнаружению мусора . . . 15
1. Reference Counting (подсчёт ссылок) . . . 15
2. Tracing (отслеживание) . . . 15
Типы GC Root в Java 8 . . . 15
Scope жизни GC Root . . . 16
Алгоритмы очистки памяти . . . 16
1. Копирующая сборка . . . 16
Stop-the-World . . . 16
2. Mark-and-Sweep . . . 16
3. Mark-and-Sweep Compact . . . 16
Сборка мусора на поколениях . . . 17
«Слабая гипотеза о поколениях» . . . 17
Характеристики сборщиков мусора . . . 17
Структура памяти JVM . . . 18
Поколения в Heap . . . 18
Метаданные . . . 18
Типы сборок . . . 18

Minor сборка . . . 18
Major сборка . . . 19
Full сборка . . . 19
Реализации GC в HotSpot VM . . . 19
Краткие характеристики . . . 20
Заключение . . . 20

Лекция 3. Collection Framework и Stream API **21**
Packages — пакеты в Java . . . 21
Java Editions . . . 21
Java Collection Framework . . . 21
Иерархия Collection Framework . . . 21
List — списки . . . 22
ArrayList vs LinkedList . . . 22
Vector — предшественник ArrayList . . . 22
Потокобезопасность коллекций . . . 23
Queue — очереди . . . 23
Deque — двусторонняя очередь . . . 23
Блокирующие очереди . . . 23
Set — множества . . . 23
Map — карты . . . 24
Основные методы . . . 24
Сторонние библиотеки коллекций . . . 24
Stream API — «LINQ в Java» . . . 24
Получение стрима . . . 24
Операторы Stream API . . . 25
Важные моменты . . . 25
Пример использования . . . 25
Сравнение с LINQ (C#) . . . 25

Лекция 4. Системы сборки **27**
Зачем нужны системы сборки . . . 27
Что такое автоматизация сборки? . . . 27
Эволюция инструментов сборки в Java . . . 27
Apache Ant + Ivy . . . 27
Ant . . . 27
Ivy . . . 28
Пример конфигурации Ant с Ivy . . . 28
Структура Ant build.xml . . . 28
Apache Maven . . . 28
Maven Central . . . 29
POM — объектная модель проекта . . . 29
Структура pom.xml . . . 29
Артефакт в Maven . . . 29
Зависимости . . . 30
Архетипы . . . 30
Жизненный цикл Maven . . . 30
Maven-плагины . . . 31

Список популярных Maven-плагинов . . . 31
Gradle . . . 31
Преимущества Gradle . . . 31
Gradle и репозитории . . . 32
gradlew — Gradle Wrapper . . . 32

Лекция 5. Работа с базами данных (JDBC и JPA) 33
Архитектура Java EE приложений . . . 33
Технологии уровня доступа к данным . . . 33
JDBC — Java Database Connectivity . . . 33
JDBC Driver Manager . . . 33
Connection String . . . 33
Установка JDBC Driver . . . 34
Statements . . . 34
Callable Statements . . . 34
DataSource и ConnectionPoolDataSource . . . 34
JTA — Java Transaction API . . . 35
JPA — Java Persistence API . . . 35
JDO vs JPA . . . 35
Hibernate . . . 35
JPA Entity . . . 35
Жизненный цикл Entity . . . 35
JPA EntityManager . . . 36
Группы операций EntityManager . . . 36

Лекция 6. Spring Framework 37
Что такое Spring Framework . . . 37
Spring и Java EE . . . 37
Inversion of Control (IoC) — инверсия управления . . . 37
Два способа реализации IoC . . . 37
Способы внедрения зависимостей . . . 38
Spring изнутри: жизненный цикл бина . . . 38
1. Парсинг конфигурации и создание BeanDefinition . . . 38
XML-конфигурация . . . 38
Конфигурация через аннотации и JavaConfig . . . 38
2. Настройка созданных BeanDefinition . . . 39
3. FactoryBean . . . 39
4. Создание экземпляров бинов . . . 39
5. Настройка созданных бинов через BeanPostProcessor . . . 39
Scope бинов . . . 39

Лекция 7. Spring AOP 41
Аспектно-ориентированное программирование . . . 41
Ключевые понятия Spring AOP . . . 41
Join Point — точка соединения . . . 41
Pointcut — срез . . . 41
Advice — совет . . . 41
Aspect — аспект . . . 42

Производительность Spring AOP . . . 42
Реализация прокси в Spring AOP . . . 42

Лекция 8. Spring MVC 43
Что такое Spring MVC . . . 43
DispatcherServlet — сердце Spring MVC . . . 43
Базовый алгоритм работы DispatcherServlet . . . 43
Аннотация @Controller . . . 44
Специальные бины Spring MVC . . . 44
Реализации HandlerMapping . . . 44
Реализации HandlerAdapter . . . 45
Полный путь HTTP-запроса в Spring MVC . . . 45
Шаг 1. Поиск обработчика . . . 45
Шаг 2. Выбор HandlerAdapter . . . 45
Шаг 3. Предобработка (preHandle) . . . 45
Шаг 4. Вызов handle . . . 45
Шаг 5. Возврат ModelAndView . . . 46
Шаг 6. Постобработка (postHandle) . . . 46
Шаг 7. Обработка исключений . . . 46
Шаг 8. Рендеринг View . . . 46
HttpMessageConverter . . . 46

Лекция 9. Spring Boot 48
Что такое Spring Boot . . . 48
Зачем нужен Spring Boot . . . 48
Типовые шаги старта без Spring Boot . . . 48
Основные цели Spring Boot . . . 48
За счёт чего достигаются цели Spring Boot . . . 48
Starter Packages . . . 49
Примеры стартеров . . . 49
Application Server (Apache Tomcat) . . . 49
AutoConfiguration — автоматическая конфигурация . . . 49
Как работает AutoConfiguration . . . 49
Конфигурация в @SpringBootApplication . . . 50
Постепенная замена автоконфигурации . . . 50
Отключение автоконфигурации . . . 50
Spring Initializr . . . 50
Spring Beans и внедрение зависимостей . . . 51
Инструменты разработчика (DevTools) . . . 51
Автоматический перезапуск (Hot Module Reload) . . . 51
Технология перезапуска . . . 51
Цикл жизни Spring Application . . . 51
Дополнительные события . . . 52
Тестирование Spring приложений . . . 52
Что внутри spring-boot-starter-test . . . 53
Преимущества DI для тестирования . . . 53
Аннотация @SpringBootTest . . . 53
MockMvc . . . 53

Лекция 10. Spring Data JPA **54**
Что такое Spring Data . . . 54
Spring Repository . . . 54
Базовый интерфейс — CrudRepository . . . 54
Другие абстракции репозиториев . . . 54
Расширение базового интерфейса . . . 54
Spring Data и сканирование сущностей . . . 55
Подключение Spring Data JPA . . . 55
Application properties . . . 55
Включение Spring Data . . . 55
Naming Conventions для методов репозитория . . . 55
Аннотация @Query . . . 55
Явная реализация репозиториев . . . 56
Что делать с DAO-слоем? . . . 56
Полезные инструменты . . . 56

Лекция 11. Spring Security **57**
Что такое Spring Security . . . 57
Архитектура: цепочка фильтров . . . 57
Основные фильтры Spring Security . . . 57
WebAsyncManagerIntegrationFilter . . . 57
SecurityContextPersistenceFilter . . . 57
HeaderWriterFilter . . . 57
LogoutFilter . . . 57
BasicAuthenticationFilter . . . 58
RequestCacheAwareFilter . . . 58
AnonymousAuthenticationFilter . . . 58
SessionManagementFilter . . . 58
ExceptionTranslationFilter . . . 59
FilterSecurityInterceptor . . . 59
AuthenticationManager . . . 59
ProviderManager . . . 59
AuthenticationProvider . . . 59

Лекция 12. Современная аутентификация и авторизация: Keycloak **61**
План лекции . . . 61
Что такое Keycloak . . . 61
Ключевые возможности . . . 61
Быстрый старт с Docker . . . 61
Realm — изолированный домен безопасности . . . 62
Ключевые свойства Realm . . . 62
Сущности Realm: Клиенты (Clients) . . . 63
Типы клиентов . . . 63
Сущности Realm: Пользователи и Роли . . . 63
 Users (Пользователи) . . . 63
 Roles (Роли) . . . 63
Назначение ролей и экспорт Realm . . . 64
OAuth 2.0 и OpenID Connect . . . 64

Роли в архитектуре OAuth 2.0 / OIDC . . . 64
JWT — JSON Web Token . . . 65
Три ключевых свойства . . . 65
Формат токена . . . 65
Структура JWT . . . 65
JWT Claims в Keycloak . . . 65
Преимущества Stateless JWT-подхода . . . 66
Миграция от сессий к JWT . . . 66
Старый код (Stateful) . . . 66
Новый код (Stateless JWT) . . . 67
Три ключевых изменения . . . 67
Два подхода интеграции Spring Boot + Keycloak . . . 67
Настройка Stateless-клиента . . . 68
application.yml . . . 68
SecurityFilterChain: настройка доступа . . . 69
JwtAuthenticationConverter — извлечение ролей из JWT . . . 69
Получение данных пользователя в контроллере . . . 70
Способ 1: @AuthenticationPrincipal (предпочтительно) . . . 70
Способ 2: SecurityContextHolder . . . 70
Как Spring проверяет подпись JWT: JWKS . . . 70
Stateful-клиент: oauth2Login . . . 71
SecurityFilterChain . . . 71
application.yml . . . 71
Важные нюансы oauth2Login . . . 72
Когда что использовать . . . 72
 oauth2ResourceServer (Stateless) . . . 72
 oauth2Login (Stateful) . . . 72
Ошибка: смешивание двух подходов . . . 72
 Так делать нельзя . . . 72
 Правильное решение . . . 73
AutoConfiguration в Spring Boot . . . 73
Какие стартеры нужны . . . 73
Бины oauth2-resource-server . . . 74
Бины oauth2-client (Stateful) . . . 74
NimbusJwtDecoder: цепочка проверок . . . 74
Кастомизация JwtDecoder . . . 74
Полная конфигурация: oauth2ResourceServer . . . 75
application.yml . . . 75
SecurityConfig.java . . . 75
Лучшие практики и частые ошибки . . . 76
 Делайте так . . . 76
 Не делайте так . . . 76
Cheat Sheet . . . 76

Лекция 13. Введение в межсервисное взаимодействие 78
Микросервисная архитектура . . . 78
Монолит vs Микросервисы: взаимодействие . . . 78
Сложность перехода на микросервисы . . . 78

Типы связи: синхронная vs асинхронная . . . 78
Брокеры сообщений: ActiveMQ и Kafka . . . 79
Система обмена сообщениями . . . 79
Роль посредника . . . 79
Модель Point-to-Point (точка-точка) . . . 79
Надёжность vs Персистентность . . . 80
Когда использовать Point-to-Point . . . 80
Модель Publisher-Subscriber (Pub/Sub) . . . 80
Когда использовать Pub/Sub . . . 81
Гибридные модели . . . 81
ActiveMQ . . . 81
ActiveMQ и JMS . . . 81
Протоколы . . . 82
AMQP (Advanced Message Queue Protocol) . . . 82
MQTT (Message Queuing Telemetry Transport) . . . 82
JMS API . . . 82
ConnectionFactory . . . 82
Connection . . . 83
Session . . . 83
MessageProducer и MessageConsumer . . . 83
Message . . . 83
Модель обмена сообщениями в ActiveMQ . . . 84
Процесс отправки сообщения . . . 84
Publisher Confirmations . . . 84
Buffering . . . 84
Процесс получения сообщений . . . 85
Курсорный механизм . . . 85

Лекция 14. Apache Kafka **86**
Что такое Kafka . . . 86
Назначение . . . 86
Ключевые свойства Kafka . . . 86
Немного истории . . . 86
Задача, которую решает Kafka . . . 86
Основные сущности Kafka . . . 87
Kafka Broker (Kafka Server / Kafka Node) . . . 87
Функции брокера . . . 87
Kafka-кластер . . . 87
Zookeeper . . . 87
Controller . . . 88
Kafka Message (Record / Event) . . . 88
Kafka Topic — поток данных (stream of data) . . . 88
Свойства Topic . . . 88
Partitions — разделение топика . . . 89
Topic placement by Brokers (в кластере) . . . 89
Storage for Kafka Topic — где хранятся данные . . . 89
Segments — сегменты файлов . . . 89
Data removing from Kafka Topic — удаление данных . . . 90

Data Replication — надёжность данных и отказоустойчивость 90
Решение: replication-factor 90
Master-Slave — гарантия согласованности данных 90
Leader и Followers 90
Несбалансированность Leaders 90
Data sync between Leader and Followers 91
Followers must fetch data from Leader 91
Проблема: что если Leader упал? 91
Решение — In-Sync Replicas (ISR) 91
Kafka Producer — отправка сообщений 91
acks — гарантии доставки 91
Delivery semantic support 92
Конвейер обработки сообщения Producer’ом 92
Kafka Consumer — получение сообщений 93
Принцип работы 93
Подключение Consumer 93
Consumer Group 93
Kafka Consumer Offset 93
Что хранится 93
Зачем offsets 94
Kafka Consumer Offset commit 94
Типы коммитов 94
А как же exactly once? 94
Kafka Consumer Offset missing 95
Параметры 95
После активации группы 95
Kafka performance — почему так быстро? 95
Итоги: Apache Kafka 95

Лекция 1. Вводная лекция: Java и JVM

О чём курс

Курс охватывает следующие темы:

- Устройство виртуальной машины Java
- Типы данных и ООП на Java
- Обработка ошибок, Generics
- Java коллекции и функциональное программирование
- Работа с базами данных
- Средства сборки и тестирование
- Spring и микросервисы

Историческая справка: предпосылки появления Java

Эволюция языков программирования

40-е годы — перфокарты. Суперкомпьютер того времени мог принимать на ввод около 125 карт/минуту и выдавать 100 карт/минуту на вывод.

60-е годы. Появляются FORTRAN, BASIC, COBOL, Pascal, Ассемблер. Зарождаются основные парадигмы программирования. Разбиение на множество концепций (ПП, СП, ООП, ФП). Главный недостаток того времени — ни один язык не удовлетворял одновременно критериям:

- простота использования
- предоставляемые возможности
- безопасность
- эффективность
- устойчивость
- расширяемость

70-е годы. Появляется **Си** — современный этап развития языков. Удовлетворял всем критериям и был создан программистами для программистов (в отличие от академических или бюрократических предшественников).

80-е годы — период консолидации. Появляется **C++**, объединивший черты объектно-ориентированного и системного программирования. ООП решает проблему размера программ, разбивая их на классы и модули.

90-е годы. Массовая домашняя компьютеризация. Появляются требования к переносимости — возможности запуска одного и того же кода на различных платформах.

Рождение Java

Из «хотелки» переносимости родилась Java — язык, который теоретически мог бы запуститься на любом холодильнике, микроволновке или мобильном телефоне. Одновременно росла всемирная паутина, и вопрос платформонезависимости стал ещё острее.

В результате появился ЯП, который позволял запускать **единожды скомпилированную программу на любой системной архитектуре** и соответствовал высоким стандартам C++ (хотя в некоторых аспектах уступал ему).

Managed vs Unmanaged Code

Высокоуровневые языки обычно компилируют код не в машинный, а в промежуточный язык. Этот код потом подаётся в финальный компилятор, который создаёт объектный модуль или машинный код. Делается это для облегчения оптимизации и увеличения портируемости.

Байт-код и JVM

Байт-код Java — набор инструкций, исполняемых виртуальной машиной Java. Каждый код операции — один байт; из 256 возможных значений используются не все (51 зарезервирован для будущего).

Знание байт-кода не обязательно для программирования на Java, но понимание его генерации помогает Java-программисту так же, как знание ассемблера помогает C/C++-программисту.

Три кита Java-платформы

Аббревиатура	Расшифровка	Назначение
JVM	Java Virtual Machine	Виртуальная машина, исполняющая байт-код
JRE	Java Runtime Environment	Среда выполнения: библиотеки классов, загрузчик, JVM
JDK	Java Development Kit	Средства разработки. Основной — OpenJDK

Три части JVM

1. **Спецификация** — набор правил, описывающий как должна быть реализована JVM. По сути сводится к «JVM должна правильно запускать программы, написанные на Java».
2. **Реализация** — реальная программа, которая запускает Java-код. Существует много реализаций (как коммерческих, так и open-source).
3. **Экземпляр** — оболочка над вашим кодом, которая его исполняет. Например, Minecraft — это программа на Java, но запускается она в экземпляре JVM со своими ресурсами.

Пример: Java-код и его байт-код

Java-код:

```java
package ru.butenko.springdatatest;
import java.time.LocalDate;

public class Dog {
    public String name;
    private LocalDate birthdate;

    public int calculateAge() {
        return LocalDate.now().getYear() - this.birthdate.getYear();
    }
}
```

Соответствующий байт-код:

```
public class ru/butenko/springdatatest/Dog {
  public Ljava/lang/String; name
  private Ljava/time/LocalDate; birthdate

  public calculateAge()I
    INVOKESTATIC java/time/LocalDate.now ()Ljava/time/LocalDate;
    INVOKEVIRTUAL java/time/LocalDate.getYear ()I
    ALOAD 0
    GETFIELD ru/butenko/springdatatest/Dog.birthdate
    INVOKEVIRTUAL java/time/LocalDate.getYear ()I
    ISUB
    IRETURN
}
```

Основы синтаксиса Java

Типы данных, операции, управляющие конструкции

- **Целочисленные:** `byte`, `short`, `int`, `long`
- **С плавающей точкой:** `float`, `double`
- **Логический:** `boolean`
- **Символьный:** `char`

- **Объекты и массивы:** `Object`, `int[] x`

Поддерживаются: арифметические и логические операции, операции отношения, `if`/`switch`, `while`/`do-while`/`for`.

Синтаксис классов

Наследование:

```
public class Dog extends Animal {
    // код класса
}
```

Имплементация интерфейса:

```
public class Cat implements Eatable {
    // код класса
}
```

Исключения

Важно понимать ключевое различие:

- Проверка на **checked-исключения** происходит в момент **компиляции** (compile-time checking).
- **Перехват** исключений (`catch`) происходит в момент **выполнения** (runtime checking).

```
public class App {
    // "пугаем" компилятор тем, что можем выбросить Exception
    public static void main(String[] args) throws Exception {
        throw new Exception();
    }
}
```

JavaDoc

JavaDoc — встроенный инструмент документирования. Поддерживает теги `@param`, `@return`, `@throws` и другие.

```
/**
 * Класс, описывающий объект "Собака".
```

```java
 */
public class Dog {
    public String name;
    private LocalDate birthdate;

    /**
     * Метод, который вычисляет возраст собаки на основе её даты рождения.
     *
     * @return int возраст собаки.
     */
    public int calculateAge() {
        return LocalDate.now().getYear() - this.birthdate.getYear();
    }

    /**
     * Метод, который задаёт кличку собаки.
     *
     * @param name - кличка, которая будет дана собаке.
     */
    public void setName(String name) {
        this.name = name;
    }

    /**
     * Метод, который выбрасывает исключение.
     *
     * @throws Exception - базовое исключение для демонстрации тега.
     */
    public void doException() throws Exception {
        throw new Exception("Exception!");
    }
}
```

Что такое сборка мусора?

Сборка мусора — это процесс восстановления заполненной памяти среды выполнения путём уничтожения неиспользуемых объектов.

Проблема утечек памяти в unmanaged-языках

В таких языках, как C и C++, программист отвечает и за создание, и за уничтожение объектов. Иногда программист может забыть уничтожить бесполезный объект, и память не освобождается. В результате приложение страдает от **утечек памяти**, и в конечном итоге падает с `OutOfMemoryError`.

В C++ для освобождения памяти используется `delete`, в C — `free()`. В managed-языках сборка мусора происходит **автоматически**.

Почему утечки памяти — серьёзная проблема

Не допускать утечек памяти на первый взгляд несложно — нужно «класть на место всё, что взяли». Но на практике это сильно осложняется:

- хитрой архитектурой
- нелинейным порядком выполнения (из-за исключений)
- человеческим фактором

Помимо утечек важна и противоположная проблема — **обращение к уже высвобожденной памяти**. Если указатель остался, а память освобождена, обращение по нему ведёт к `Segmentation fault`.

Автоматическое управление памятью

Java-код транслируется в байт-код, который исполняет JVM. Таким образом, **JVM играет ключевую роль** и предоставляет важнейшую возможность — автоматическое управление памятью.

Инструмент, освобождающий неиспользуемую память, называется **Garbage Collector (GC)**. У GC две задачи:

1. **Обнаружение** мусора
2. **Очистка** мусора

   «Мусор» — это структура данных (объект) в памяти, к которому из программного кода больше невозможно получить доступ.

Подходы к обнаружению мусора

1. Reference Counting (подсчёт ссылок)

Каждый объект имеет счётчик ссылок. Когда ссылка уничтожается — счётчик уменьшается. Если счётчик равен нулю — объект считается мусором.

Используется в: Python и других языках.

Плюсы: - Простой - Не требует долгих пауз для сборки

Минусы: - Плохо сочетается с многопоточностью - Сложно выявлять **циклические зависимости** — когда два объекта ссылаются друг на друга, но никто живой на них не ссылается; в итоге утечка - Влияет на производительность — каждое чтение/запись ссылки требует обновления счётчика

2. Tracing (отслеживание)

Подход вводит понятие **GC Root**.

> Живые объекты — это те, до которых мы можем добраться от корня (GC Root). Всё остальное — мусор. Всё, что доступно с живого объекта — также живое.

Если представить все объекты и ссылки как дерево, нужно пройти от корневых узлов по всем достижимым. До чего не дошли — мусор. **Проблема циклических зависимостей решается автоматически.**

Типы GC Root в Java 8

- Локальные переменные и параметры методов
- Потоки Java
- Статические переменные
- Ссылки из JNI (Java Native Interface — механизм для запуска кода из unmanaged-языков)

Даже самое простое Java-приложение имеет GC Root:

- Локальные переменные внутри `main`
- Статические переменные класса `main`
- Параметры `main` (`args`)
- Поток, выполняющий `main`

Scope жизни GC Root

Компилятор вычисляет для каждой переменной **live range** — путь от определения переменной до последнего использования. Если переменная вне своего live range, она перестаёт быть корнем для GC.

Алгоритмы очистки памяти

1. Копирующая сборка

Память делится на области **from-space** и **to-space**. Все объекты создаются в from-space. Когда место заканчивается, происходит **stop-the-world** — все «живые» объекты копируются в to-space, from-space очищается полностью, и области меняются местами.

Stop-the-World

Простые сборщики полностью останавливают выполнение программы во время сборки. Это гарантирует, что новые объекты не выделяются и старые не становятся внезапно недоступными.

Преимущества паузы:

- Проще определять достижимость (граф объектов заморожен)
- Проще перемещать объекты в куче

Для задач с критичными к паузам системами существует **инкрементальная сборка** — много кратковременных пауз вместо одной длинной.

2. Mark-and-Sweep

Похож на копирующий, но с улучшениями:

1. Приложение полностью останавливается
2. Проходим по всем объектам и помечаем (**mark**) «живые»
3. Делаем **sweep** — чистим всё непомеченное

Минус: память становится фрагментированной (остаются «дыры»).

3. Mark-and-Sweep Compact

Улучшение Mark-and-Sweep:

1. Ищем «мёртвые» объекты и помечаем их для переноса **параллельно работе приложения**
2. Останавливаем приложение для очистки
3. Делаем **compact** — дефрагментируем память, объекты сдвигаются к более близким адресам

Плюсы: - Нет фрагментации памяти - Эффективно при большом количестве «живых» объектов

Минусы: - Плохо работает при большом количестве «мёртвых» объектов - Compact — дорогостоящая операция

Сборка мусора на поколениях

«Слабая гипотеза о поколениях»

Анализ работающих систем выявил две закономерности:

1. **Большинство объектов живут либо очень долго, либо очень недолго.** Долгоживущих очень мало.
2. **Между «старыми» и «новыми» объектами мало связей.**

Отсюда — идея разделения объектов на **поколения**:

- **Young Generation** (молодое поколение)
- **Old Generation** (старое поколение)

Сборки тоже делятся:

- **Minor GC** — затрагивает только младшее поколение, выполняется часто
- **Full GC** — затрагивает оба поколения, выполняется редко

Характеристики сборщиков мусора

При оценке GC учитываются три фактора:

1. **Максимальная задержка** (STW pause) — максимальное время, на которое сборщик приостанавливает выполнение программы.
2. **Пропускная способность** — отношение общего времени работы к времени простоя из-за сборки.
3. **Потребляемые ресурсы** — объём CPU и дополнительной памяти, потребляемых сборщиком.

Достичь идеала по всем трём показателям одновременно — невозможно. При выборе реализации сосредотачиваются на двух конкретных характеристиках.

Структура памяти JVM

Поколения в Heap

- **Young Generation** — здесь создаются новые объекты. Делится на три части:
  - **Eden** — область, где изначально создаются объекты через `new Object()`
  - **Survivor S0 / From Space** — куда попадают пережившие «изгнание из Эдема»
  - **Survivor S1 / To Space** — последнее место перед переходом в Tenured
- **Old Generation (Tenured)** — здесь хранятся давно живущие объекты

Метаданные

До Java 8 существовал **PermGen** — раздел для метаданных классов, интернированных строк и т.д. Был сложен в настройке размера, часто давал `OutOfMemoryError`.

Начиная с Java 8, эта область убрана. Информация переехала:

- Интернированные строки — в heap
- Остальные метаданные — в область **Metaspace** в native memory

Максимальный размер Metaspace по умолчанию ограничен только объёмом нативной памяти.

Типы сборок

Minor сборка

- Очищает только Young Generation (Eden + Survivor)
- «Живые» молодые объекты, пережившие достаточное число циклов, перемещаются в Tenured
- Остальные «живые» отправляются в пустую Survivor-область
- Eden и очищенная Survivor могут быть переиспользованы
- **Происходит часто, быстро, уничтожает много мусора**

Major сборка

- Очищает Old Generation
- Тесно связана с minor, поэтому обычно рассматривается в составе full

Full сборка

- Очищается и Young, и Old Generation
- Запускается, когда minor не может перенести объект (недостаточно места)
- **Происходит редко, но занимает много времени**

Сборщики, работающие с поколениями, называются **Generational Garbage Collection**.

Реализации GC в HotSpot VM

Сборщик	Версия Java	JVM-аргумент	Назначение
Serial GC	Все	`-XX:+UseSerialGC`	Небольшие приложения, однопоточные среды
Parallel GC	По умолчанию	`-XX:+UseParallelGC`	Средние/большие данные, многопоточное оборудование
CMS GC	До Java 9	`-XX:+UseConcMarkSweepGC`	Веб-приложения, минимальные паузы
G1 GC	Java 9+ default	`-XX:+UseG1GC`	Крупный размер кучи (>4 ГБ)
Epsilon GC	Java 11+	`-XX:+UseEpsilonGC`	Эксперименты, сверхнизкая задержка
Shenandoah GC	Java 15+	`-XX:+UseShenandoahGC`	Параллельная сборка, минимальные паузы

Сборщик	Версия Java	JVM-аргумент	Назначение
ZGC	Java 15+	`-XX:+UseZGC`	Низкая задержка (<10 мс), терабайтные кучи

Краткие характеристики

- **Serial GC** — все события сборки в одном потоке, всегда вызывает stop-the-world.
- **Parallel GC** — несколько потоков для minor GC, один для major. Также вызывает stop-the-world. Подходит для пакетных задач.
- **CMS** — Concurrent Mark and Sweep. Работает одновременно с приложением, минимизируя паузы. Потребляет больше CPU. В CMS не выполняется уплотнение.
- **G1** — Garbage First. Разбивает кучу на области (1–32 МБ), сканирует их параллельно. Имеет дополнительные области: **Humongous** (для объектов >50% размера кучи) и **Available** (неиспользуемое пространство).
- **Epsilon** — не реализует никакого реального механизма сборки. Как только куча исчерпана — JVM завершает работу. Используется для приложений, чувствительных к сверхвысокой задержке, или для тестирования.
- **Shenandoah** — большая часть цикла выполняется параллельно с приложением, объекты перемещаются на лету. Сильнее нагружает процессор.
- **ZGC** — паузы менее 10 мс даже на терабайтных кучах. Только для больших серверных решений.

Заключение

> **Отдавайте предпочтение настройкам по умолчанию.** Если у вас небольшое автономное Java-приложение, скорее всего, настраивать GC не нужно. Это не то, на что должен ежедневно обращать внимание разработчик.

Лекция 3. Collection Framework и Stream API

Packages — пакеты в Java

Классы в Java объединяются в **пакеты**. Пакеты позволяют логически организовать классы и избежать конфликтов имён. Java имеет встроенные пакеты: `java.lang`, `java.util`, `java.io` и т.д.

Если для класса пакет не определён, он находится в пакете «по умолчанию». Чтобы использовать классы из других пакетов, нужно их подключить через `import`. **Исключение** — `java.lang` (`String`, `Object` и др.) подключается автоматически.

Java Editions

Edition	Описание
Java SE	Standard Edition. Стандартная редакция для консольных, GUI, апплет-приложений.
Java EE	Enterprise Edition. Для распределённых приложений масштаба предприятий. Включает EJB, JPA, Servlets, JMS.
Java ME	Micro Edition. Для микрокомпьютеров и мобильных платформ.

Спецификации формируются через Java Community Process.

Java Collection Framework

Collection Framework — иерархия интерфейсов и реализаций, которая является частью JDK и предоставляет «из коробки» большое количество структур данных.

Все коллекции наследуются от интерфейса `java.util.Collection`. Он же наследуется от `Iterable` — поэтому всё, что является коллекцией, может использоваться в циклах `for-each`.

Иерархия Collection Framework

```
Iterable
  └── Collection
```

```
        ├── List
        │     ├── ArrayList
        │     ├── LinkedList
        │     └── Vector → Stack
        ├── Queue
        │     └── Deque
        │           └── ArrayDeque
        └── Set
              ├── HashSet
              └── TreeSet

Map (отдельная иерархия)
  ├── HashMap
  └── TreeMap
```

List — списки

ArrayList vs LinkedList

Характеристика	ArrayList	LinkedList
Основа	Массив	Связанный список (Entry хранит data + next + previous)
Тип доступа	Произвольный (Random Access) — по индексу	Последовательный (Sequential)
Рост	В 1.5 раза при заполнении	По одной Entry

> Массив сам по себе не может изменять размер. Когда в `ArrayList` место заканчивается, создаётся новый массив **в 1.5 раза больше** и старые элементы копируются.

Vector — предшественник ArrayList

`java.util.Vector` отличается от `ArrayList` тем, что методы для работы синхронизированы — один поток работает с коллекцией, остальные ждут.

- Растёт **в 2 раза** (не в 1.5, как ArrayList)
- В JavaDoc прямым текстом рекомендуется использовать `ArrayList`, если потокобезопасность не нужна
- Vector имеет наследника — `java.util.Stack` (LIFO-структура с методами `push`, `pop`, `peek`)

Потокобезопасность коллекций

- **Synchronized** — потокобезопасные. Только один поток одновременно работает с коллекцией. Гарантируют целостность, но снижают производительность.
- **Non-Synchronized** — не потокобезопасные. Множество потоков могут работать одновременно.

Потокобезопасные версии коллекций лежат в пакете `java.util.concurrent`.

Queue — очереди

Очередь (`java.util.Queue`) — FIFO-структура (first in — first out).

Deque — двусторонняя очередь

`java.util.Deque` — двусторонняя очередь, позволяет использовать структуру с обоих концов. Реализация — `ArrayDeque`.

Блокирующие очереди

- **BlockingQueue** реализуют: `PriorityBlockingQueue`, `SynchronousQueue`, `ArrayBlockingQueue`, `DelayQueue`, `LinkedBlockingQueue`
- **BlockingDeque** реализует: `LinkedBlockingDeque`

Каждая блокирующая очередь предназначена для определённых задач многопоточного программирования.

Set — множества

Set отличается от очереди и списка большей абстракцией — «мешок с предметами», где порядок не определён.

- Сам интерфейс `Set` не добавляет новых методов к `Collection`, а лишь **уточняет требования** (отсутствие дубликатов)

- Получение элементов — только через `Iterator`
- Основные реализации:
  - **HashSet** — на основе хэш-кода
  - **TreeSet** — на основе дерева

Map — карты

`java.util.Map` — структура «ключ — значение». Аналог словарей из C#.

Основные методы

- `keySet()` — возвращает набор ключей (Set, т.к. ключи уникальны)
- `values()` — возвращает коллекцию значений (Collection, т.к. значения могут повторяться)
- `entrySet()` — возвращает набор пар «ключ — значение»

Сторонние библиотеки коллекций

При необходимости можно создать собственную реализацию или использовать готовые сторонние решения:

- **Google Guava**
- **Apache Commons Collections**

Эти библиотеки выступают как boost для Java-коллекций (аналог Boost для C++).

Stream API — «LINQ в Java»

Стримы и коллекции **похожи, но разные по назначению:**

- **Коллекции** — для эффективного доступа к одиночным объектам
- **Стримы** — для параллельных и последовательных агрегаций через цепочки методов

Получение стрима

```
list.stream()                    // из коллекции
Stream.empty()                   // пустой
Stream.of("1", "2", "3")         // из указанных элементов
```

Операторы Stream API

Промежуточные (intermediate, lazy) — обрабатывают элементы и возвращают новый стрим. Их в цепочке может быть много.

Терминальные (terminal, eager) — обрабатывают элементы и завершают работу стрима. В цепочке только один.

Важные моменты

1. **Обработка не начнётся до вызова терминального оператора.**
2. **Экземпляр стрима нельзя использовать более одного раза** (в отличие от `IEnumerable` в C#).
3. Каждый раз для работы с коллекцией нужно открывать **новый поток**.

Пример использования

```
list.stream()
    .filter(x -> x.toString().length() == 3)
    .forEach(System.out::println);

list.stream().forEach(x -> System.out.println(x));
```

Сравнение с LINQ (C#)

C# (LINQ)	Java (Stream API)
`.Where(predicate)`	`.filter(predicate)`
`.Select(...)`	`.map(...)`
`.SelectMany(...)`	`.flatMap(...)`
`Enumerable.Range`	`IntStream.range`
`.Take()`	`.limit()`
`.Skip()`	`.skip()`
`.Distinct()`	`.distinct()`
`.OrderBy(...)`	`.sorted(comparator)`
`.Count()`	`.count()`
`.Aggregate()`	`.reduce()`
`.First()`	`.findFirst()`
`.Any()`	`.anyMatch()`

C# (LINQ)	Java (Stream API)
`.Any()` с отрицанием	`.noneMatch()`
`.ToList()`	`.toList()`
`String.Join()`	`.joining(delimiter, prefix, suffix)`
`.FirstOrDefault(..., 5)`	`.map(...).orElse(5)`
`.Single(...)`	`.filter(...).orElseThrow()`

Лекция 4. Системы сборки

Зачем нужны системы сборки

Технический подход в разработке заключается в том, что программы и библиотеки разбивают на части — пакеты и модули. В сообществе Java популярно написание библиотек на все случаи жизни и выкладывание их в общий доступ.

Дополнительный стимул — Java получила большую популярность на бэкенде. У серверного ПО более высокие требования к надёжности, и использование проверенных временем библиотек предпочтительнее написания своего кода.

Легко встретить бэкенд-проект из нескольких десятков модулей и сотни библиотек. **Описывать (и изменять!) сценарии сборки таких проектов вручную — чрезвычайно трудно.**

Что такое автоматизация сборки?

Автоматизация сборки — это процесс автоматизации задач, необходимых для создания, выполнения и тестирования программ.

Исторически такие задачи решались с помощью **make-файлов**. Сегодня — с помощью средств автоматизации сборки или серверов сборки. Термины «автоматизация сборки» и «система сборки» используются как синонимы.

Эволюция инструментов сборки в Java

В начале был **Make**, потом **GNU Make**. Сегодня в экосистеме JVM доминируют три инструмента:

1. **Apache Ant + Apache Ivy** — «Муравей с плющом»
2. **Apache Maven** — «Знаток/Эксперт»
3. **Google Gradle** — «Доктор Грейдл»

Apache Ant + Ivy

Ant

Ant был выпущен в 2000 году и быстро стал популярным. Имеет низкую кривую обучения, основан на **процедурном программировании**.

Недостатки:

- XML как формат скриптов сборки — иерархический по природе, плохо подходит для процедурного подхода
- XML становится неуправляемо большим во всех проектах, кроме очень маленьких
- Изначально не имел управления зависимостями — позже принял Apache Ivy

Ivy

Apache Ivy — менеджер зависимостей для Java-проектов с поддержкой **транзитивных зависимостей** (возможность использовать зависимости других зависимостей).

Для работы Ivy нужны метаданные о ваших модулях — обычно их определяют в **Ivy-файлах**. Артефакты зависимостей (.jar) ищутся в настраиваемых репозиториях.

Пример конфигурации Ant с Ivy

```
<ivy-module version="2.0">
    <info organisation="org.apache" module="hello-ivy"/>
    <dependencies>
        <dependency org="commons-lang" name="commons-lang" rev="2.0"/>
        <dependency org="commons-cli" name="commons-cli" rev="1.0"/>
    </dependencies>
</ivy-module>
```

Структура Ant build.xml

В Ant фазы сборки называются **целями** (`<target>`). Типичный набор:

1. **clean** — отчистка артефактов предыдущей сборки
2. **compile** — запуск `javac`
3. **jar** — упаковка class-файлов в Java Archive
4. **run** — запуск архива в JVM

Apache Maven

Maven выпущен в 2004 году как усовершенствование Ant. Это инструмент сборки и **менеджер проектов** на основе XML.

Ключевое отличие от Ivy: Apache Maven — это не только менеджер зависимостей, но и инструмент управления проектом и его сборкой. Ivy же — только инструмент управления зависимостями.

Maven Central

Репозиторий по умолчанию — **Maven Central**. Де-факто репозиторий для всех Java-библиотек (аналог nuget.org из мира .NET).

POM — объектная модель проекта

Проекты Maven описываются файлами **POM** (Project Object Model) — `pom.xml`. До Maven у каждой IDE был свой формат project-файла (зачастую бинарный). Maven предложил универсальный открытый стандарт.

В `pom.xml` хранятся:

- Информация о версии стандарта Maven-проекта
- Информация о самом проекте
- Информация об используемых библиотеках (зависимостях)

Структура pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                             http://maven.apache.org/xsd/maven-4.0.0.xsd">
    ...
</project>
```

Артефакт в Maven

В стандарте Maven всё (программа, проект, модуль, библиотека) называется **артефактом**. Артефакт описывается тремя параметрами:

- **groupId** — пакет, к которому принадлежит приложение, обычно с именем домена
- **artifactId** — уникальный строковый ключ (id проекта)
- **version** — версия проекта

Эти три параметра однозначно описывают любой артефакт. Например: `org.junit.jupiter:junit-jupiter-api:5.9.2`.

Зависимости

Зависимость в терминологии Maven — это какой-либо артефакт, который необходим проекту для корректной работы. Артефактом может быть обычный Java-проект, собранный и распространённый с помощью Maven.

Архетипы

Архетипы — стандартизированные шаблоны проектов. Стартовая структура папок (`src`, `java`, `test` и т.д.) задаётся архетипом. Список архетипов можно получить командой:

```
mvn archetype:generate
```

Жизненный цикл Maven

Основное преимущество Maven — его жизненный цикл. Пока проект следует определённым стандартам, через него можно легко пройти полный цикл.

№	Фаза	Описание
1	**validate**	Проверяет корректность метаинформации о проекте
2	**compile**	Компилирует исходники
3	**test**	Прогоняет тесты
4	**package**	Упаковывает классы в артефакт: jar, war, zip
5	**verify**	Проверяет корректность артефакта и соответствие требованиям качества
6	**install**	Кладёт артефакт в локальный репозиторий
7	**deploy**	Заливает артефакт на production-сервер или удалённый репозиторий

Maven-плагины

Жизненный цикл можно расширить с помощью **плагинов**. Плагины — самая обычная вещь в Maven: чтобы задать нюансы сборки, нужно подключить плагин.

Плагины описываются почти так же, как зависимости:

- `dependencies` → `plugins`
- `dependency` → `plugin`
- `repositories` → `pluginRepositories`

Список популярных Maven-плагинов

Плагин	Назначение
`maven-compiler-plugin`	Управляет Java-компиляцией
`maven-resources-plugin`	Управляет включением ресурсов в сборку
`maven-source-plugin`	Управляет включением исходного кода
`maven-dependency-plugin`	Управляет копированием библиотек зависимостей
`maven-jar-plugin`	Создание итогового jar-файла
`maven-war-plugin`	Создание итогового war-файла
`maven-surefire-plugin`	Управляет запуском тестов
`buildnumber-maven-plugin`	Генерирует номер сборки

Gradle

Gradle выпущен в 2008 году как преемник Maven. Главное отличие — вместо XML использует **DSL** (Domain Specific Language) на основе **Groovy** (один из языков JVM).

Преимущества Gradle

- Скрипты сборки **намного короче и понятнее**, чем Ant/Maven
- Меньше boilerplate-кода
- DSL предназначен для решения конкретной задачи — продвижение ПО через жизненный цикл

Gradle и репозитории

Gradle **не имеет собственных репозиториев** — использует Maven и Ivy репозитории как источники зависимостей. Интерфейс работы с ними на базовом уровне одинаков.

gradlew — Gradle Wrapper

Существует два варианта работы с Gradle:

1. **Глобально** через `gradle`, установленный в системе
2. **Через `gradlew`** — Wrapper (обёртка)

Преимущество Wrapper:

- Версия Gradle одинакова у всей команды
- Никому не нужно вручную качать дистрибутив
- Wrapper установит нужную версию инструментов **локально для проекта**

Лекция 5. Работа с базами данных (JDBC и JPA)

Архитектура Java EE приложений

Java EE приложения разделены по функциональному принципу на изолированные модули. Обычно делятся на **три уровня** (как у Фаулера):

1. **Клиентский уровень**
2. **Промежуточный уровень (BLL)** — Business Logic Layer
3. **Уровень доступа к данным (DAL)**

Технологии уровня доступа к данным

- **JDBC** — Java Database Connectivity API
- **JPA** — Java Persistence API
- **Java EE Connector Architecture**
- **JTA** — Java Transaction API

JDBC — Java Database Connectivity

Низкоуровневое API для доступа к данным в хранилищах. Типичное использование — написание SQL-запросов к конкретной базе.

JDBC Driver Manager

Java-приложение общается с БД через **JDBC Driver** — набор классов, реализующих JDBC API для конкретной СУБД.

Драйвер выбирается через **DriverManager**. Можно использовать In-Memory-DB, NoSQL-DB или базу, встроенную в Android-приложение — Java-разработчика эти нюансы не касаются, DriverManager сам выберет подходящий драйвер.

Connection String

Для подключения протокола JDBC API:

`jdbc:mysql://localhost:3306/db_scheme`

- `mysql` — протокол работы с сервером
- `localhost` — имя хоста в сети
- `3306` — порт
- `db_scheme` — имя схемы (БД)

Установка JDBC Driver

Драйвер для конкретной БД нужно установить через систему сборки:

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.29</version>
</dependency>
```

Statements

SQL-запросы делятся на две группы:

Группа	Операторы	Метод
Получение данных	`SELECT`	`executeQuery()` — возвращает `ResultSet`
Изменение данных	`INSERT`, `UPDATE`, `DELETE`	`executeUpdate()` — возвращает число изменённых строк

Callable Statements

CallableStatement используется для вызова хранимых процедур в БД. Хранимая процедура похожа на функцию класса, но находится в базе данных. Тяжёлые операции с БД могут выиграть в производительности при выполнении в том же пространстве памяти, что и сервер БД.

DataSource и ConnectionPoolDataSource

Интерфейсы из пакета `javax.sql`, реализуемые поставщиками JDBC-классов.

Основное назначение: предоставить возможность получения соединения с БД **абстрагируясь от местоположения сервера СУБД и типа драйвера**.

Объекты `DataSource` используются для получения **физического соединения** с БД и являются **альтернативой DriverManager** — нет необходимости регистрировать драйвер. Достаточно установить параметры соединения и вызвать `getConnection()`.

JTA — Java Transaction API

API для определения и управления транзакциями, включая распределённые транзакции, а также транзакции, затрагивающие множество хранилищ данных.

JPA — Java Persistence API

JPA — спецификация Java EE и Java SE, описывающая систему управления **сохранением Java-объектов в таблицы реляционных БД**. Гораздо более высокоуровневое API по сравнению с JDBC — скрывает всю его сложность.

JDO vs JPA

JDO (Java Data Objects) — более общая спецификация ORM для любых баз и хранилищ. **JPA** можно рассматривать как часть JDO, специализированную на реляционных базах.

Hibernate

Hibernate — самая популярная open-source реализация JPA (де-факто стандарт). JPA только описывает правила и API, а Hibernate реализует эти описания. Hibernate имеет дополнительные возможности, не описанные в JPA (и не переносимые на другие реализации).

JPA Entity

Entity — это легковесный хранимый объект бизнес-логики (persistent domain object). Основная программная сущность — entity-класс. Может использовать дополнительные вспомогательные классы.

Жизненный цикл Entity

У Entity четыре статуса:

1. **new** — объект создан, ещё нет сгенерированных первичных ключей, не сохранён в БД
2. **managed** — объект создан, управляется JPA, имеет первичные ключи
3. **detached** — объект был создан, но не управляется (или больше не управляется) JPA

4. **removed** — объект создан, управляется JPA, но будет удалён после коммита транзакции

JPA EntityManager

EntityManager — главный интерфейс для работы с JPA. Описывает API для всех основных операций.

Группы операций EntityManager

1. Операции над Entity:

- `persist` — сохранение нового объекта
- `merge` — слияние с существующим
- `remove` — удаление
- `refresh` — обновление из БД
- `detach` — отсоединение от управления
- `lock` — блокировка

2. Получение данных:

- `find` — поиск по ключу
- `createQuery`, `createNamedQuery`, `createNativeQuery` — создание запросов
- `contains` — проверка наличия
- `createNamedStoredProcedureQuery`, `createStoredProcedureQuery` — вызов процедур

3. Получение других сущностей JPA:

- `getTransaction` — транзакция
- `getEntityManagerFactory` — фабрика
- `getCriteriaBuilder` — построитель критериев
- `getMetamodel` — метамодель
- `getDelegate` — делегат

4. Работа с EntityGraph:

- `createEntityGraph`, `getEntityGraph`

5. Общие операции:

- `close`, `isOpen`, `getProperties`, `setProperty`, `clear`

Лекция 6. Spring Framework

Что такое Spring Framework

Spring Framework — open-source фреймворк (платформа) для языков семейства JVM:

- Java
- Kotlin
- Groovy
- Scala
- C# (форк)

Spring и Java EE

Spring является **альтернативой Java EE (Jakarta EE)** и по сути его расширением:

- Spring совместим с Java EE (но не обратное)
- Имеет множество расширений (MVC, Data и т.п.)
- Активно поддерживается сообществом

Inversion of Control (IoC) — инверсия управления

IoC-контейнер — центральная часть фреймворка. Предоставляет средства конфигурирования и управления Java-объектами с помощью **рефлексии**.

IoC-контейнер отвечает за:

- Управление **жизненным циклом** объекта
- Создание объектов
- Вызов методов инициализации
- Связывание объектов между собой

Объекты, создаваемые контейнером, называются **бинами (Beans)**.

Два способа реализации IoC

1. **Поиск зависимостей (Dependency Lookup)** — вызывающий объект запрашивает у контейнера экземпляр объекта с определённым именем или типом
2. **Внедрение зависимостей (Dependency Injection)** — контейнер передаёт экземпляры объектов другим объектам по их имени

Способы внедрения зависимостей

- **Через конструктор** (Constructor Injection)
- **Через set-метод** (Setter Injection)
- **Через свойства** (Field Injection)

Spring изнутри: жизненный цикл бина

1. Парсинг конфигурации и создание BeanDefinition

Существует три способа конфигурации Spring:

Способ	Класс контекста
XML-конфигурация	`ClassPathXmlApplicationContext("context.xml")`
Аннотации со сканированием пакета	`AnnotationConfigApplicationContext("package.name")`
JavaConfig (через `@Configuration`)	`AnnotationConfigApplicationContext(JavaConfig.class)`

XML-конфигурация

Использует класс `XmlBeanDefinitionReader`, реализующий интерфейс `BeanDefinitionReader`. Каждый элемент XML обрабатывается, и если он является бином — создаётся `BeanDefinition`. Все `BeanDefinition` помещаются в `Map`, хранящийся в `DefaultListableBeanFactory`.

Конфигурация через аннотации и JavaConfig

Используется класс `AnnotationConfigApplicationContext`. Внутри два важных поля:

- **ClassPathBeanDefinitionScanner** — сканирует указанный пакет на наличие классов с аннотацией `@Component`. Найденные классы парсируются, для них создаются `BeanDefinition`.
- **AnnotatedBeanDefinitionReader** — работает в несколько этапов:
  1. Регистрирует все `@Configuration` для дальнейшего парсинга
  2. Регистрирует специальный `BeanFactoryPostProcessor` (а именно `BeanDefinitionRegistryPostProcessor`), который парсит JavaConfig через `ConfigurationClassParser` и создаёт `BeanDefinition`

2. Настройка созданных BeanDefinition

После первого этапа `BeanDefinition` хранятся в `Map`. Архитектура Spring позволяет повлиять на бины **ещё до их фактического создания** — у нас есть доступ к метаданным класса.

Для этого существует интерфейс **`BeanFactoryPostProcessor`**. Реализовав его, мы получаем доступ к созданным `BeanDefinition` и можем их изменять.

Метод `postProcessBeanFactory` принимает `ConfigurableListableBeanFactory`. Через `getBeanDefinitionNames` можем получить все имена, а затем по конкретному имени — конкретный `BeanDefinition` для обработки метаданных.

3. FactoryBean

FactoryBean — generic-интерфейс, которому можно делегировать процесс создания бинов определённого типа. Был особенно нужен в эпоху XML-конфигурации, когда требовался механизм управления процессом создания бинов.

4. Создание экземпляров бинов

На этом этапе Spring создаёт сами экземпляры бинов согласно `BeanDefinition`.

5. Настройка созданных бинов через BeanPostProcessor

Интерфейс **`BeanPostProcessor`** позволяет вклиниться в процесс настройки бинов **до того, как они попадут в контейнер**. Содержит два метода:

- `postProcessBeforeInitialization` — вызывается **до** init-метода
- `postProcessAfterInitialization` — вызывается **после** init-метода

  Важно: на этом этапе экземпляр бина уже создан и идёт его донастройка. Если хотите сделать прокси над объектом — делайте это в `postProcessAfterInitialization`.

Scope бинов

Spring поддерживает несколько scope для бинов:

- **`SCOPE_SINGLETON`** — инициализация **один раз** на этапе поднятия контекста (по умолчанию)
- **`SCOPE_PROTOTYPE`** — инициализация **каждый раз** по запросу

(Также существуют request, session, application и websocket для веб-приложений.)

Лекция 7. Spring AOP

Аспектно-ориентированное программирование

АОП — парадигма программирования, являющаяся дальнейшим развитием процедурного и объектно-ориентированного программирования. Идея АОП заключается в выделении так называемой **сквозной функциональности** — кода, который повторяется в разных местах приложения (логирование, безопасность, транзакции, кэширование).

Ключевые понятия Spring AOP

Join Point — точка соединения

Join point — точки наблюдения, присоединения к коду, где планируется введение функциональности. Это конкретные места в программе, где может быть применён аспект (вызовы методов, обращения к полям и т.д.).

Pointcut — срез

Pointcut — срез, запрос точек присоединения. Может содержать одну или несколько точек.

Правила запросов очень разнообразные: запрос по аннотации на методе, по конкретному методу и т.д. Правила можно объединять через &&, ||, !.

Advice — совет

Advice — набор инструкций, выполняемых на точках среза (Pointcut). Инструкции можно выполнять по событиям разных типов:

Тип Advice	Когда выполняется
Before	Перед вызовом метода
After	После вызова метода
After returning	После возврата значения из функции
After throwing	В случае exception
After finally	В случае выполнения блока finally
Around	Можно сделать пред- и пост-обработку, а также вообще обойти вызов метода

На один `Pointcut` можно «повесить» несколько `Advice` разного типа.

Aspect — аспект

Aspect — модуль, в котором собраны описания Pointcut и Advice. По сути, это класс-контейнер для всей логики, связанной с одной сквозной функциональностью.

> *«If you want your code to be easy to write, make it easy to read.» — Robert C. Martin, Clean Code*

Производительность Spring AOP

Всё происходит в **рантайме**, если не используются AspectJ-компиляторы, которые могут сгенерировать бины заранее (это называется **Weaving**, ткачество).

Реализация прокси в Spring AOP

Spring AOP использует два механизма прокси:

Механизм	Когда используется
JDK Dynamic Proxy	По умолчанию. Используется, если целевой объект реализует хотя бы один **интерфейс**. Позволяет проксировать любой интерфейс (или их набор).
CGLIB Proxy	По умолчанию используется, если бизнес-объект **не реализует ни одного интерфейса**.

Лекция 8. Spring MVC

Что такое Spring MVC

Spring MVC — модуль, обеспечивающий архитектуру паттерна **Model — View — Controller** при помощи слабосвязанных готовых компонентов.

Паттерн MVC разделяет аспекты приложения и обеспечивает свободную связь между ними:

Компонент	Назначение
Model (Модель)	Инкапсулирует данные приложения. Обычно — POJO («Plain Old Java Objects» или бины)
View (Отображение)	Отвечает за отображение данных Модели, обычно генерируя HTML
Controller (Контроллер)	Обрабатывает запрос пользователя, создаёт Модель и передаёт её в View

DispatcherServlet — сердце Spring MVC

Spring MVC построен вокруг центрального сервлета — **DispatcherServlet**, который:

- Распределяет запросы по контроллерам
- Полностью интегрирован в Spring IoC-контейнер
- Имеет доступ ко всем возможностям Spring

Базовый алгоритм работы DispatcherServlet

1. Получает HTTP-запрос
2. Обращается к **HandlerMapping** для определения, какой контроллер вызвать
3. Отправляет запрос в нужный контроллер
4. Контроллер вызывает соответствующий метод (на основе GET/POST), формирует Модель и возвращает имя View
5. **ViewResolver** определяет, какой View использовать на основе полученного имени
6. После создания View, DispatcherServlet отправляет данные Модели в виде атрибутов в View, который отображается в браузере

Аннотация @Controller

`DispatcherServlet` отправляет запрос **контроллерам**. Аннотация **`@Controller`** указывает, что класс является контроллером. **`@RequestMapping`** связывает класс или метод с URL.

Специальные бины Spring MVC

`DispatcherServlet` делегирует специальные компоненты для обработки запросов:

Бин	Назначение
HandlerMapping	Сопоставляет запрос с обработчиком (handler-объектом) и списком интерсепторов
HandlerAdapter	Помогает DispatcherServlet вызвать обработчик независимо от того, как он реализован
HandlerExceptionResolver	Стратегия разрешения исключений: сопоставление с обработчиками, представлениями ошибок
ViewResolver	Преобразует логические имена View в фактические View
LocaleResolver / LocaleContextResolver	Определяет «локаль» клиента и часовой пояс для интернационализированных представлений
ThemeResolver	Определяет темы веб-приложения
MultipartResolver	Абстракция для разбора multipart-запросов (загрузка файлов)
FlashMapManager	Хранит и извлекает «входную» и «выходную» FlashMap для передачи атрибутов между запросами (через redirect)

Реализации HandlerMapping

Основные реализации:

- **RequestMappingHandlerMapping** — поддерживает аннотированные методы `@RequestMapping`
- **SimpleUrlHandlerMapping** — поддерживает явную регистрацию шаблонов пути URI к обработчикам
- **BeanNameUrlHandlerMapping**

Реализации HandlerAdapter

- **HttpRequestHandlerAdapter** — поддерживает классы, реализующие `HttpRequestHandler`
- **SimpleControllerHandlerAdapter** — поддерживает классы, реализующие `Controller`
- **RequestMappingHandlerAdapter** — поддерживает контроллеры с `@RequestMapping`

Полный путь HTTP-запроса в Spring MVC

Шаг 1. Поиск обработчика

После получения HTTP-запроса `DispatcherServlet` перебирает доступные `HandlerMapping`. Один из них определит, какой метод какого контроллера должен быть вызван.

- HandlerMapping по `HttpServletRequest` находит **handler-объект** (например, `HandlerMethod`)
- Каждый HandlerMapping может иметь несколько реализаций `HandlerInterceptor` — для кастомизации пред- и пост-обработки запроса
- Список `HandlerInterceptor` + handler-объект → формируют `HandlerExecutionChain`

Шаг 2. Выбор HandlerAdapter

Для выбранного обработчика определяется соответствующий `HandlerAdapter`.

Шаг 3. Предобработка (preHandle)

Вызывается `applyPreHandle` на `HandlerExecutionChain`:

- Если вернул `true` — все интерсепторы выполнили предобработку, переходим к основному обработчику
- Если вернул `false` — один из интерсепторов взял обработку ответа на себя

Шаг 4. Вызов handle

HandlerAdapter извлекается из `HandlerExecutionChain`. Через метод `handle` принимаются объекты запроса и ответа, а также метод-обработчик.

Шаг 5. Возврат ModelAndView

Метод-обработчик возвращает в `DispatcherServlet` объект **ModelAndView**. Через `ViewResolver` определяется, какой View использовать.

Для REST-контроллеров:

- Вместо `ModelAndView` возвращается `null`
- `ViewResolver` не используется
- Ответ полностью содержится в теле `HttpServletResponse`
- Контроллер аннотируется `@RestController` или методы — `@ResponseBody`

Шаг 6. Постобработка (postHandle)

Перед завершением вызывается `applyPostHandle` для постобработки с помощью интерсепторов.

Шаг 7. Обработка исключений

Если в процессе обработки выбрасывается исключение, обрабатывается через одну из реализаций `HandlerExceptionResolver`:

- **ExceptionHandlerExceptionResolver** — обрабатывает исключения из методов с `@ExceptionHandler`
- **ResponseStatusExceptionResolver** — отображает исключения с `@ResponseStatus` в HTTP-статусы
- **DefaultHandlerExceptionResolver** — отображает стандартные исключения Spring MVC в HTTP-статусы

Шаг 8. Рендеринг View

Для классических контроллеров — `DispatcherServlet` отправляет данные в виде атрибутов в View, который записывается в `HttpServletResponse`. Для REST — данная логика не вызывается, ответ уже в `HttpServletResponse`.

HttpMessageConverter

Когда HTTP-запрос приходит с заголовком `Accept`, Spring MVC перебирает доступные `HttpMessageConverter`, пока не найдёт того, кто сможет конвертировать из POJO в указанный тип `Accept`.

HttpMessageConverter работает в обоих направлениях:

- Тела входящих запросов конвертируются в Java-объекты
- Java-объекты конвертируются в тела HTTP-ответов

Spring Boot определяет довольно обширный набор реализаций `HttpMessageConverter`. Можно добавлять собственные реализации или переопределять существующие.

Лекция 9. Spring Boot

Что такое Spring Boot

Spring Boot — дополнение к Spring, которое облегчает и ускоряет работу с ним. Представляет собой набор утилит, автоматизирующих настройки фреймворка.

Зачем нужен Spring Boot

Сам по себе Spring имеет:

- Большое количество зависимостей
- Множество сложных конфигураций (часто в XML)

У рядового разработчика это вызывает проблемы при знакомстве с фреймворком. Spring Boot — самый быстрый и популярный способ запуска Spring-проектов.

Типовые шаги старта без Spring Boot

Создавая каждое Spring-приложение, приходилось:

1. Импортировать Spring-модули в зависимости от типа приложения
2. Импортировать библиотеку web-контейнеров (для web)
3. Импортировать сторонние библиотеки совместимых версий (Hibernate, Jackson)
4. Конфигурировать DAO-компоненты: DataSource, transaction management
5. Определить класс для загрузки всех конфигураций

Основные цели Spring Boot

- Обеспечить **быстрый старт** для любых разработок на Spring
- Возможность кастомизировать стандартное поведение
- Предоставлять **нефункциональные возможности**: встроенные серверы, безопасность, метрики, проверка работоспособности, тестирование и конфигурация
- Уйти от старых подходов с **XML-конфигурациями**

За счёт чего достигаются цели Spring Boot

Согласно spring.io:

- **‘starter’ зависимости**, облегчающие конфигурацию

- **Автоматическая конфигурация** различных библиотек
- **Преднастроенный Application Server** (Apache Tomcat)
- **Готовые рецепты** для широко используемых подходов

Starter Packages

Starter-пакеты — набор удобных дескрипторов зависимостей. Включают в проект универсальное решение для конкретной технологии, избавляя от поиска совместимых версий.

Примеры стартеров

- **`spring-boot-starter-data-jpa`** — подтянет всё для работы с БД (драйверы, Hibernate)
- **`spring-boot-starter-web`** — подтянет всё для веб-приложений: `spring-webmvc`, `jackson-json`, `validation-api`, Tomcat
- **`spring-boot-starter-test`** — набор тестовых библиотек

  Spring Boot собирает все общие зависимости и определяет их в одном месте, что позволяет разработчикам **просто использовать их**, а не «изобретать колесо» при создании каждого нового приложения.

Application Server (Apache Tomcat)

Apache Tomcat — комплект серверных программ от Apache Software Foundation для тестирования, отладки и исполнения веб-приложений на Java. Обычно называется **контейнером сервлетов**.

`DispatcherServlet` из Spring крутится **внутри Tomcat**. Spring Boot включает Tomcat «из коробки».

AutoConfiguration — автоматическая конфигурация

Spring Boot пытается автоматически настроить приложение **на основе добавленных jar-зависимостей**.

Как работает AutoConfiguration

- Краеугольным камнем являются **AutoConfiguration-классы**, которые Spring Boot находит при запуске

- Аннотация **`@EnableAutoConfiguration`** сообщает Spring Boot, что нужно «угадать» как настроить Spring
- Если в classpath найдены `spring-boot-starter-web` (с Tomcat и Spring MVC) — настраивается веб-приложение
- Если используется `spring-boot-starter-jdbc` — автоматически регистрируются `DataSource`, `EntityManagerFactory`, `TransactionManager`, читается информация из `application.properties`
- Если БД не указана — настраивается резидентная база в памяти (если есть H2 или HSQL Driver)

Конфигурация в @SpringBootApplication

Аннотация `@SpringBootApplication` объединяет несколько других:

- `@EnableAutoConfiguration` — включает автоконфигурацию
- `@ComponentScan` — сканирует компоненты
- `@SpringBootConfiguration` — помечает класс как конфигурацию

Постепенная замена автоконфигурации

Автоконфигурация работает **неагрессивно** — в любой момент можно начать определять свою конфигурацию, чтобы заменить определённые её части.

Если добавить свой `DataSource` — поддержка встроенной БД отключается.

Для отладки автоконфигурации запустите приложение с параметром:

```
--debug
```

Это активирует отладочные журналы и выведет отчёт об условиях на консоль.

Отключение автоконфигурации

Если определённые классы автоконфигурации не нужны, используйте атрибут `exclude` в `@SpringBootApplication`.

Spring Initializr

Spring Initializr — сервис, предоставляющий интерфейс для генерации заготовки проекта Spring со стандартными зависимостями.

- Можно конфигурировать зависимости

- Можно выбрать предпочитаемый сборщик
- IntelliJ IDEA имеет встроенную интеграцию

Spring Beans и внедрение зависимостей

При использовании `@ComponentScan` (или `@SpringBootApplication`, которая её включает) все компоненты (`@Component`, `@Service`, `@Repository`, `@Controller` и др.) **автоматически регистрируются как бины Spring**.

Инструменты разработчика (DevTools)

`spring-boot-devtools` предоставляет инструменты для разработки. **Автоматически деактивируются** при запуске «упакованного» приложения.

Если приложение запускается через `java -jar` или специальный загрузчик — оно считается **Production-сборкой**. Управлять можно через `spring.devtools.restart.enabled`.

> Использовать DevTools в production **нельзя** — это угроза безопасности.

Автоматический перезапуск (Hot Module Reload)

Приложения с `spring-boot-devtools` **автоматически перезапускаются** при изменении файлов в classpath. Очень полезная функция при работе в IDE — обеспечивает быструю обратную связь.

Технология перезапуска

Spring Boot использует **два загрузчика классов**:

- **Основной загрузчик** — для неизменяемых классов (сторонние jar)
- **Перезапускающий загрузчик** — для активно разрабатываемых классов

При перезапуске единовременно используется перезапускающий, после чего создаётся новый. Поэтому перезапуск **гораздо быстрее холодного запуска** — основной загрузчик уже доступен и заполнен.

Цикл жизни Spring Application

События приложения отправляются в следующем порядке:

№	Событие	Описание
1	`ApplicationStartingEvent`	В начале выполнения, перед обработкой (только регистрация слушателей)
2	`ApplicationEnvironmentPreparedEvent`	Когда `Environment` известно, но контекст ещё не создан
3	`ApplicationContextInitializedEvent`	`ApplicationContext` подготовлен, инициализаторы вызваны, но определения бинов ещё не загружены
4	`ApplicationPreparedEvent`	Перед обновлением, после загрузки определений бинов
5	`ApplicationStartedEvent`	После обновления контекста, перед вызовом командных средств
6	`AvailabilityChangeEvent (LivenessState.CORRECT)`	Приложение считается работающим
7	`ApplicationReadyEvent`	После вызова всех средств выполнения
8	`AvailabilityChangeEvent (ReadinessState.ACCEPTING_TRAFFIC)`	Приложение готово к обработке запросов
□	`ApplicationFailedEvent`	Если при запуске возникло исключение

Дополнительные события

Между `ApplicationPreparedEvent` и `ApplicationStartedEvent` также публикуются:

- **WebServerInitializedEvent** — после готовности WebServer (`ServletWebServerInitialized` или `ReactiveWebServerInitializedEvent`)
- **ContextRefreshedEvent** — при обновлении `ApplicationContext`

Тестирование Spring приложений

Spring Boot предусматривает утилиты и аннотации, упрощающие тестирование. Поддержка обеспечивается двумя модулями:

- **`spring-boot-test`** — основные элементы
- **`spring-boot-test-autoconfigure`** — автоконфигурация тестов

Большинство разработчиков используют стартер **`spring-boot-starter-test`**, импортирующий оба модуля + полезные библиотеки.

Что внутри spring-boot-starter-test

Библиотека	Назначение
JUnit 5	Стандарт де-факто для модульного тестирования
Spring Test, Spring Boot Test	Утилиты для тестирования Spring Boot
AssertJ	Библиотека текучих утверждений
Hamcrest	Библиотека объектов-сопоставителей (matchers)
Mockito	Java-фреймворк для мокирования объектов
JSONassert	Библиотека утверждений для JSON
JsonPath	XPath для JSON

Преимущества DI для тестирования

Одно из главных преимуществ DI — облегчает модульное тестирование. Можно создавать объекты через `new`, не вовлекая Spring. Можно использовать mock-объекты вместо реальных зависимостей.

Аннотация @SpringBootTest

Spring Boot предоставляет `@SpringBootTest` — альтернативу стандартной `@ContextConfiguration`.

Поиск конфигурации:

- Аннотации `@*Test` ищут вашу первичную конфигурацию автоматически
- Поиск начинается с пакета, содержащего тест
- Идёт до класса с `@SpringBootApplication` или `@SpringBootConfiguration`

MockMvc

По умолчанию `@SpringBootTest` **не запускает сервер**, а создаёт имитационное окружение для тестирования конечных веб-точек.

В Spring MVC можно запрашивать веб-точки с помощью **MockMvc**.

Лекция 10. Spring Data JPA

Что такое Spring Data

Spring Data — дополнительный удобный механизм для взаимодействия с сущностями БД, их организации в репозитории, извлечения и изменения данных. В некоторых случаях достаточно объявить **интерфейс с методом без реализации** — Spring сгенерирует её сам.

Spring Repository

Основное понятие в Spring Data — **репозиторий**. Это интерфейсы, использующие JPA Entity для взаимодействия с базой.

Базовый интерфейс — CrudRepository

```
public interface CrudRepository<T, ID extends Serializable>
    extends Repository<T, ID>
```

Обеспечивает основные **CRUD-операции:**

Буква	Операция
C	Create — создание
R	Read — чтение
U	Update — обновление
D	Delete — удаление

Другие абстракции репозиториев

- **PagingAndSortingRepository** — добавляет пагинацию и сортировку
- **JpaRepository** — JPA-специфичный, добавляет flush и batch-операции

Расширение базового интерфейса

Если предоставленных методов достаточно — расширяете базовый интерфейс для своей сущности и добавляете свои методы запросов.

Spring Data и сканирование сущностей

Традиционно сущности JPA задаются в `persistence.xml`. **В Spring Boot этот файл не требуется** — используется **Entity Scanning**. По умолчанию поиск выполняется во всех пакетах ниже основного конфигурационного класса (с `@EnableAutoConfiguration` или `@SpringBootApplication`).

Подключение Spring Data JPA

Через стартер:

```
implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
```

Application properties

DataSource по умолчанию использует H2 — встроенную in-memory БД:

```
spring.datasource.url=jdbc:h2:mem:db;DB_CLOSE_DELAY=-1
spring.datasource.username=sa
spring.datasource.password=sa
```

Включение Spring Data

```
@Configuration
@EnableJpaRepositories(basePackages = "org.example.data")
class JpaConfig {}
```

Naming Conventions для методов репозитория

Spring Data JPA генерирует SQL-запросы **на основе имён методов**. Например:

- `findByNameAndAge(String name, int age)` — `SELECT ... WHERE name = ? AND age = ?`
- `findByLastNameOrderByFirstNameAsc(String lastName)` — `SELECT ... WHERE last_name = ? ORDER BY first_name ASC`

Полная документация: Spring Data JPA Query Methods

Аннотация @Query

Если name conventions не подходят — можно явно указать запрос через `@Query`.

```
@Query("SELECT u FROM User u WHERE u.email = ?1")
User findByEmail(String email);
```

Также поддерживаются native SQL-запросы.

Явная реализация репозиториев

Если нужно явно реализовать репозитории — это тоже возможно.

> **По умолчанию Spring Data будет «генерировать» реализацию только тех методов, которые не переопределены в классах с постфиксом `Impl`.**

То есть, если есть интерфейс `UserRepository` — создаём класс `UserRepositoryImpl`, и Spring Data будет использовать его реализации для пересекающихся методов.

Что делать с DAO-слоем?

Spring Data JPA менеджит большую часть рутины:

- Генерирует SQL-запросы
- Производит маппинг
- Управляет транзакциями (JTA)
- … и многое другое

  Старые явные реализации DAO как будто бы и не нужны…

Полезные инструменты

JPA-Buddy plugin для IntelliJ IDEA — упрощает создание Entity, репозиториев и других классов. Но не все фичи доступны в бесплатной версии.

Полезные ссылки для изучения:

- Baeldung: Spring Data
- Baeldung: Persistence with Spring

Лекция 11. Spring Security

Что такое Spring Security

Spring Security — Java/Java EE фреймворк, предоставляющий механизмы построения систем **аутентификации и авторизации**, а также другие возможности обеспечения безопасности для корпоративных приложений на Spring Framework.

Архитектура: цепочка фильтров

Spring Security использует паттерн **«Цепочка обязанностей» (Chain of Responsibility)**. При выполнении HTTP-запроса происходит **поэтапная обработка фильтрами**, которые производят валидацию запроса.

Основные фильтры Spring Security

WebAsyncManagerIntegrationFilter

Интегрирует `SecurityContext` с `WebAsyncManager`, который ответственен за асинхронные запросы.

SecurityContextPersistenceFilter

- Ищет `SecurityContext` в сессии
- Заполняет `SecurityContextHolder`, если находит
- По умолчанию используется `ThreadLocalSecurityContextHolderStrategy`, хранящий `SecurityContext` в `ThreadLocal`-переменной

HeaderWriterFilter

Просто добавляет заголовки в response.

LogoutFilter

Проверяет совпадает ли URL с паттерном (`/logout` POST по умолчанию) и запускает процедуру логаута:

1. Удаляется CSRF-токен
2. Завершается сессия
3. Чистится `SecurityContextHolder`

BasicAuthenticationFilter

- Проверяет, есть ли заголовок `Authorization: Basic ...`
- Если находит — извлекает логин/пароль и передаёт их в `AuthenticationManager`

RequestCacheAwareFilter

Восстанавливает оригинальный запрос пользователя после логина:

1. Пользователь заходит на защищённый URL
2. Его перекидывает на страницу логина
3. После успешной авторизации — возвращает на исходную страницу

Внутри проверяет, есть ли сохранённый запрос, и подменяет им текущий. Запрос сохраняется в сессии.

AnonymousAuthenticationFilter

Если к моменту выполнения этого фильтра `SecurityContextHolder` пуст (т.е. аутентификации не произошло), фильтр заполняет его **анонимной аутентификацией** — `AnonymousAuthenticationToken` с ролью `ROLE_ANONYMOUS`.

Это гарантирует, что в `SecurityContextHolder` всегда будет объект — можно не бояться `NullPointerException` и более гибко настраивать доступ для неавторизованных пользователей.

SessionManagementFilter

На этом этапе производятся действия, связанные с сессией:

- Смена идентификатора сессии
- Ограничение количества одновременных сессий
- Сохранение `SecurityContext` в `securityContextRepository`

В обычном случае:

1. `HttpSessionSecurityContextRepository` сохраняет `SecurityContext` в сессию
2. Вызывается `sessionAuthenticationStrategy.onAuthentication`
3. По умолчанию включена защита от **session fixation attack** — после аутентификации меняется ID сессии
4. Если был передан CSRF-токен — генерируется новый

ExceptionTranslationFilter

К этому моменту `SecurityContext` должен содержать анонимную или нормальную аутентификацию. Этот фильтр прокидывает запрос/ответ по filter chain и обрабатывает возможные ошибки авторизации.

FilterSecurityInterceptor

На **последнем этапе** происходит **авторизация** на основе URL запроса.

- Наследуется от `AbstractSecurityInterceptor`
- Решает, имеет ли текущий пользователь доступ к URL
- Существует другая реализация — `MethodSecurityInterceptor` — для допуска к методам (при использовании `@Secured` / `@PreAuthorize`)
- Внутри вызывается `AccessDecisionManager`
- Несколько стратегий принятия решения, по умолчанию: **AffirmativeBased**

AuthenticationManager

AuthenticationManager — интерфейс, который принимает `Authentication` и возвращает `Authentication`. Это центральная точка процесса аутентификации.

Типичная реализация `Authentication` — **`UsernamePasswordAuthenticationToken`**.

ProviderManager

ProviderManager — стандартная реализация `AuthenticationManager`. Содержит список `AuthenticationProvider`.

AuthenticationProvider

Когда мы передаём объект `Authentication` в `ProviderManager`, он:

1. **Перебирает** существующие `AuthenticationProvider`-ы
2. **Проверяет**, поддерживает ли `AuthenticationProvider` эту имплементацию `Authentication`
3. Внутри `AuthenticationProvider.authenticate` мы приводим переданный `Authentication` к нужной реализации
4. Извлекаем credentials
5. **Если аутентификация не удалась** — выбрасываем исключение
6. **ProviderManager** ловит исключение и пробует следующий провайдер

7. Если **ни один** провайдер не вернул успешную аутентификацию — ProviderManager пробрасывает последнее пойманное исключение

После успешной аутентификации `BasicAuthenticationFilter` сохраняет полученный `Authentication`:

```
SecurityContextHolder.getContext().setAuthentication(authResult);
```

Если выбрасывается `AuthenticationException` — контекст сбрасывается, вызывается `AuthenticationEntryPoint`.

Лекция 12. Современная аутентификация и авторизация: Keycloak

План лекции

1. **Keycloak** — что это и зачем
2. **OAuth 2.0 и OpenID Connect** — теория, JWT
3. **Интеграция со Spring Boot** — `oauth2Login` vs `oauth2ResourceServer`
4. **Магия AutoConfiguration** — какие бины создаются автоматически
5. **Практика и лучшие практики** — реальные репозитории и шпаргалка

Что такое Keycloak

Keycloak — open-source решение для управления идентификацией и доступом (**IAM** — Identity and Access Management), поддерживаемое Red Hat. Берёт на себя весь комплекс задач по аутентификации, оставляя вашему приложению только авторизацию.

Ключевые возможности

Возможность	Описание
SSO и Sign-Out	Бесшовный переход между приложениями без повторного ввода пароля
Identity Brokering	Аутентификация через Google, GitHub, LDAP, Active Directory
Открытые стандарты	OpenID Connect, OAuth 2.0, SAML 2.0 — всё из коробки
2FA и Login Flows	Двухфакторная аутентификация, кастомизация тем и гибкие потоки

Быстрый старт с Docker

```
services:
  keycloak:
    container_name: keycloak.openid-provider
```

```
    image: quay.io/keycloak/keycloak:26.0
    command:
      - start-dev
      - --import-realm        # Импортируем готовый realm при старте
    ports:
      - 8080:8080
    volumes:
      - ./keycloak/:/opt/keycloak/data/import/
    environment:
      KEYCLOAK_ADMIN: admin
      KEYCLOAK_ADMIN_PASSWORD: ${KEYCLOAK_ADMIN_PASSWORD}
```

> **Совет:** Флаг `--import-realm` позволяет версионировать конфигурацию безопасности вместе с кодом — Realm-файл хранится в Git наравне с исходниками.

Realm — изолированный домен безопасности

Realm — изолированный домен безопасности внутри Keycloak. Это контейнер верхнего уровня, инкапсулирующий уникальный набор пользователей, клиентов, ролей и конфигураций.

> **Аналогия:** Keycloak — многоквартирный дом. Каждый Realm — отдельная квартира со своим замком, жильцами и правилами.

Ключевые свойства Realm

Свойство	Описание
Изоляция	Пользователи и настройки одного Realm недоступны в другом. Можно разделить dev и prod
Управление	Realm управляет пользователями, их учётными данными, ролями и группами
Наследование	Мастер-администратор может создавать несколько Realms и управлять ими из единой точки

Сущности Realm: Клиенты (Clients)

Client — приложение или сервис, взаимодействующий с Keycloak для аутентификации и авторизации. Каждый клиент имеет уникальный **Client ID** и тип доступа.

Типы клиентов

Тип	Описание	Поток
Confidential	Серверные приложения (бэкенд), могут безопасно хранить client-secret. Пример: Spring Boot API	Authorization Code Flow
Public	Клиентские приложения (SPA, мобильные), не могут безопасно хранить секреты	Authorization Code Flow with PKCE

Создание клиента: `Clients` → `Create client` → указать Client ID → выбрать тип → `Save`.

Сущности Realm: Пользователи и Роли

Users (Пользователи)

Учётная запись конечного пользователя, аутентифицирующегося в системе.

1. `Users` → `Add user`
2. Заполнить Username, Email, First/Last Name
3. Вкладка `Credentials` → задать пароль, отключить флаг Temporary

Roles (Роли)

Именованный набор прав (**RBAC** — Role-Based Access Control). Keycloak поддерживает два типа ролей:

Тип ролей	Описание
Realm Roles	Глобальные роли, доступные всем клиентам в Realm. Подходят для `admin`, `user`, `manager`

Тип ролей	Описание
Client Roles	Роли, специфичные для конкретного клиента. Когда разрешения уникальны для одного приложения

Назначение ролей и экспорт Realm

Назначение роли: 1. `Users` → выбрать пользователя 2. Вкладка `Role mapping` → `Assign role` 3. Выбрать нужные Realm- или Client-роли → `Assign`

Экспорт Realm: 1. `Realm Settings` → `Action` → `Partial Export` → JSON-файл 2. Сохранить JSON в репозиторий 3. Импорт: Docker `--import-realm` + смонтированная директория

OAuth 2.0 и OpenID Connect

OAuth 2.0 — протокол **авторизации**. Позволяет приложению получить доступ к ресурсам пользователя без передачи пароля.

OIDC (OpenID Connect) — слой **аутентификации** поверх OAuth 2.0, добавляющий **ID Token** — ответ на вопрос «кто этот пользователь?».

Роли в архитектуре OAuth 2.0 / OIDC

Роль	Кто это	Что делает
Authorization Server	Keycloak	Выдаёт токены после успешной аутентификации
Resource Server	Spring Boot API	Принимает и проверяет Access Token
Access Token	JWT	Доступ к защищённым эндпоинтам API
ID Token	JWT	Данные о пользователе (OIDC-специфичный)
Refresh Token	Долгоживущий токен	Обновление Access Token без участия пользователя

JWT — JSON Web Token

JWT (RFC 7519) — открытый стандарт, компактный и самодостаточный способ безопасной передачи информации между сторонами в виде JSON-объекта.

Три ключевых свойства

1. **Компактный** — передаётся в URL, POST-параметрах или HTTP-заголовке
2. **Самодостаточный** — содержит всю информацию о пользователе и его правах
3. **Подписанный** — подпись гарантирует целостность данных

Формат токена

JWT состоит из **трёх частей**, разделённых точками:

```
xxxxx.yyyyy.zzzzz
Header.Payload.Signature
```

Каждая часть кодируется в Base64Url. Итоговый токен передаётся в заголовке:

```
Authorization: Bearer eyJhbGci...
```

Структура JWT

Часть	Содержимое
Header	`typ: "JWT"`, алгоритм подписи `alg: "RS256"`. Кодируется в Base64Url
Payload (Claims)	Зарегистрированные (`iss`, `exp`, `sub`) и кастомные (`realm_access.roles`, `resource_access`). Кодируется в Base64Url
Signature	`HMAC(Base64Url(header) + "." + Base64Url(payload), privateKey)`. Проверяет целостность и подлинность

JWT Claims в Keycloak

Keycloak добавляет специфические claims, на которых строится авторизация в Spring Security:

realm_access — Realm Roles:

```
{
  "realm_access": {
    "roles": ["user", "admin"]
  }
}
```

resource_access — Client Roles:

```
{
  "resource_access": {
    "my-backend-api": {
      "roles": ["client_user"]
    }
  }
}
```

> **Важно:** Именно эти claims являются основой для авторизации в Spring Security. Без кастомного конвертера Spring «не увидит» роли в JWT от Keycloak.

Преимущества Stateless JWT-подхода

Преимущество	Описание
Горизонтальное масштабирование	Любой экземпляр сервера обработает запрос — состояние не хранится на сервере
Упрощение архитектуры	Отпадает необходимость в Spring Session
Универсальный токен	Один JWT используется для доступа к нескольким микросервисам одновременно

Миграция от сессий к JWT

Старый код (Stateful)

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
```

```
    http
        .sessionManagement(session -> session
            .sessionCreationPolicy(SessionCreationPolicy.IF_REQUIRED))
        .formLogin(Customizer.withDefaults())
        .authorizeHttpRequests(auth -> auth.anyRequest().authenticated());
    return http.build();
}
```

Новый код (Stateless JWT)

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http
        .sessionManagement(session -> session
            .sessionCreationPolicy(SessionCreationPolicy.STATELESS))  // □
        .csrf(csrf -> csrf.disable())                                  // □
        .authorizeHttpRequests(auth -> auth
            .requestMatchers("/api/public/**").permitAll()
            .anyRequest().authenticated())
        .oauth2ResourceServer(oauth2 ->
            oauth2.jwt(Customizer.withDefaults()));                    // □
    return http.build();
}
```

Три ключевых изменения

1. **STATELESS** — `SessionCreationPolicy.STATELESS` полностью отключает создание HTTP-сессий. Каждый запрос аутентифицируется заново по токену.
2. **`csrf().disable()`** — для stateless API защита от CSRF не требуется: браузер не отправляет cookies с токеном автоматически.
3. **`oauth2ResourceServer().jwt()`** — настраиваем Spring на приём и проверку JWT. Spring Boot автоматически создаст `JwtDecoder` на основе `issuer-uri`.

Два подхода интеграции Spring Boot + Keycloak

Характеристика	`oauth2Login` (Stateful)	`oauth2ResourceServer` (Stateless)
Назначение	Серверные веб-приложения с UI (MVC, Thymeleaf)	REST API, микросервисы
Состояние	Есть (HTTP-сессии)	Нет (Bearer-токен в каждом запросе)
Защита от CSRF	Требуется	Не требуется
Тип объекта аутентификации	`OAuth2AuthenticationToken`	`JwtAuthenticationToken`
Масштабирование	Нужна репликация сессий (Redis)	Легко, stateless
Аутентификация	Редирект на страницу логина	Bearer-токен в заголовке

> **Правило:** Никогда не смешивайте `oauth2Login` и `oauth2ResourceServer` в одном `SecurityFilterChain`!

Настройка Stateless-клиента

application.yml

Минимальная конфигурация — **одна строка**:

```
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: http://localhost:8080/realms/baeldung-keycloak
```

Указав `issuer-uri`, Spring Boot **автоматически:**

1. Запросит конфигурацию OIDC-провайдера
2. Получит публичные ключи (JWKS)
3. Создаст `JwtDecoder` — без единой строки Java-кода

SecurityFilterChain: настройка доступа

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http
        .authorizeHttpRequests(auth -> auth
            .requestMatchers("/api/public/**").permitAll()      // открыто всем
            .requestMatchers("/api/admin/**").hasRole("ADMIN")  // только ADMIN
            .anyRequest().authenticated())                       // остальное — авториз
        .oauth2ResourceServer(oauth2 ->
            oauth2.jwt(Customizer.withDefaults()));
    return http.build();
}
```

> **Важно:** Метод `.hasRole("ADMIN")` автоматически ищет `ROLE_ADMIN` в `GrantedAuthorities`. Необходим кастомный конвертер!

JwtAuthenticationConverter — извлечение ролей из JWT

По умолчанию Spring Security не знает, где в JWT от Keycloak лежат роли. Нужен кастомный конвертер:

```
@Bean
public JwtAuthenticationConverter jwtAuthenticationConverter() {
    JwtGrantedAuthoritiesConverter grantedAuthoritiesConverter =
        new JwtGrantedAuthoritiesConverter();
    // Префикс, который Spring Security ожидает для hasRole()
    grantedAuthoritiesConverter.setAuthorityPrefix("ROLE_");
    // Путь к ролям в JWT-структуре Keycloak
    grantedAuthoritiesConverter.setAuthoritiesClaimName("realm_access.roles");

    JwtAuthenticationConverter jwtAuthenticationConverter =
        new JwtAuthenticationConverter();
    jwtAuthenticationConverter.setJwtGrantedAuthoritiesConverter(grantedAuthoritiesConv
    return jwtAuthenticationConverter;
}
```

Регистрация в SecurityFilterChain:

```
.oauth2ResourceServer(oauth2 ->
    oauth2.jwt(jwt ->
        jwt.jwtAuthenticationConverter(jwtAuthenticationConverter())))
```

Получение данных пользователя в контроллере

Способ 1: @AuthenticationPrincipal (предпочтительно)

```
@GetMapping("/api/user")
public Map getUserInfo(@AuthenticationPrincipal Jwt jwt) {
    return Map.of(
        "username", jwt.getClaimAsString("preferred_username"),
        "email", jwt.getClaimAsString("email"),
        "roles", jwt.getClaimAsStringList("realm_access.roles")
    );
}
```

Способ 2: SecurityContextHolder

Используется вне контроллера — в сервисах и компонентах, где нет доступа к параметрам запроса.

```
Authentication auth = SecurityContextHolder.getContext().getAuthentication();
Jwt jwt = (Jwt) auth.getPrincipal();
String username = jwt.getClaimAsString("preferred_username");
```

Как Spring проверяет подпись JWT: JWKS

Для верификации каждого входящего токена бэкенду нужны публичные ключи Keycloak. Spring Boot получает их **автоматически и кэширует**:

1. **Конфигурация** — Spring читает `issuer-uri` из `application.yml`
2. **Discovery** — `GET /.well-known/openid-configuration`
3. **Извлечение** — берёт `jwks_uri` из JSON-ответа
4. **Загрузка** — скачивает JWKS с certs endpoint

   Keycloak периодически ротирует ключи. Spring Boot автоматически обновляет кэш при получении JWT с неизвестным `kid` (Key ID), обеспечивая бесперебойную работу.

Stateful-клиент: oauth2Login

Если бэкенд рендерит страницы (Thymeleaf, JSP) и управляет сессиями — используйте `oauth2Login`. Spring Security полностью берёт на себя редирект и обработку callback.

SecurityFilterChain

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http
        .authorizeHttpRequests(auth -> auth
            .requestMatchers("/public/**").permitAll()
            .anyRequest().authenticated())
        .oauth2Login(oauth2 -> oauth2.defaultSuccessUrl("/home", true))
        .logout(logout -> logout
            .logoutSuccessHandler(oidcLogoutSuccessHandler()));
    return http.build();
}
```

application.yml

```
spring:
  security:
    oauth2:
      client:
        registration:
          keycloak:
            client-id: ${KC_CLIENT_ID}
            client-secret: ${KC_SECRET}
            authorization-grant-type: authorization_code
            scope: openid, profile, email
        provider:
          keycloak:
            issuer-uri: ${KC_ISSUER_URI}
            user-name-attribute: preferred_username
```

Важные нюансы oauth2Login

- **RP-Initiated Logout** — используйте `OidcClientInitiatedLogoutSuccessHandler` для корректного выхода. Без него пользователь останется залогиненным в Keycloak.
- **Back-Channel Logout** — Keycloak может инициировать logout на всех клиентах. Зарегистрируйте специальный URL — приложение получит POST от Keycloak.
- **User Info и OAuth2UserService** — для получения дополнительной информации настройте `OAuth2UserService` или `OidcUserService`.

Когда что использовать

✅ oauth2ResourceServer (Stateless)

- Разрабатываете REST API или микросервис
- Бэкенд не управляет сессиями
- Планируете горизонтальное масштабирование
- Клиент — SPA, мобильное приложение или другой сервис

🏆 Выбор по умолчанию для большинства современных бэкенд-сервисов

✅ oauth2Login (Stateful)

- Разрабатываете веб-приложение с серверным рендерингом (MVC)
- Нужны сессии на сервере
- Хотите, чтобы Spring полностью управлял процессом логина/логаута
- Нет нужды в масштабировании без состояния

Ошибка: смешивание двух подходов

❌ Так делать нельзя

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http
        .oauth2Login(Customizer.withDefaults())
        .oauth2ResourceServer(oauth2 ->
            oauth2.jwt(Customizer.withDefaults()));
```

```
    return http.build();
}
```

Почему нельзя? Один компонент использует сессии, другой — stateless JWT. Они **конфликтуют** в обработке аутентификации.

✅ Правильное решение

Создайте **два отдельных бина** `SecurityFilterChain`, разграничив их по URL с помощью `securityMatcher`:

```
@Bean
@Order(1)
public SecurityFilterChain uiChain(HttpSecurity http) throws Exception {
    http.securityMatcher("/ui/**").oauth2Login(...);
    return http.build();
}

@Bean
@Order(2)
public SecurityFilterChain apiChain(HttpSecurity http) throws Exception {
    http.securityMatcher("/api/**").oauth2ResourceServer(...);
    return http.build();
}
```

AutoConfiguration в Spring Boot

Какие стартеры нужны

Стартер	Назначение
spring-boot-starter-oauth2-resource-server	Для REST API. Поддержка JWT и Opaque токенов, библиотека Nimbus JOSE+JWT
spring-boot-starter-oauth2-client	Для серверных веб-приложений. Authorization Code Flow и управление сессиями
spring-boot-starter-security	Базовый стартер, обязателен в обоих случаях

 Устаревший **Keycloak Adapter больше не используется** — только стандартные стартеры Spring Security OAuth2.

Бины oauth2-resource-server

Когда обнаружены стартер и `issuer-uri`, Spring Boot создаёт три ключевых бина:

1. **JwtDecoder** — реализация `NimbusJwtDecoder`. Декодирует и верифицирует подпись, проверяет `exp`, `nbf`, `iss`.
2. **JwtAuthenticationConverter** — стандартный извлекает `scope`. Почти всегда нужно переопределять для маппинга ролей Keycloak.
3. **BearerTokenAuthenticationFilter** — перехватывает запрос, извлекает токен из `Authorization: Bearer ...`, делегирует проверку `JwtDecoder` и `JwtAuthenticationProvider`.

Бины oauth2-client (Stateful)

1. **ClientRegistrationRepository** — хранилище конфигураций OAuth2-клиентов из `application.yml`
2. **OAuth2AuthorizedClientService** — управляет токенами доступа. По умолчанию — in-memory
3. **OAuth2AuthorizationRequestRedirectFilter** — перенаправляет на страницу логина Keycloak
4. **OAuth2LoginAuthenticationFilter** — обрабатывает callback после успешного логина

NimbusJwtDecoder: цепочка проверок

1. Получает публичный ключ (JWK) из кэша или JWKS-эндпоинта Keycloak
2. Проверяет подпись с помощью публичного ключа
3. Проверяет `exp` (не просрочен) и `nbf` (already valid)
4. Проверяет, что `iss` совпадает с `issuer-uri`
5. Возвращает объект `Jwt` или бросает исключение

Кастомизация JwtDecoder

```
@Bean
public JwtDecoder jwtDecoder() {
    NimbusJwtDecoder decoder = NimbusJwtDecoder.withJwkSetUri(jwksUri).build();
```

```
    decoder.setJwtValidator(
        new DelegatingOAuth2TokenValidator<>(
            JwtValidators.createDefaultWithIssuer(issuer),
            new CustomBlacklistValidator()
        )
    );
    return decoder;
}
```

Полная конфигурация: oauth2ResourceServer

application.yml

```
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: http://localhost:8080/realms/baeldung-keycloak
```

SecurityConfig.java

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http
            .sessionManagement(s -> s.sessionCreationPolicy(STATELESS))
            .csrf(AbstractHttpConfigurer::disable)
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/api/public/**").permitAll()
                .anyRequest().authenticated())
            .oauth2ResourceServer(oauth2 -> oauth2
                .jwt(jwt -> jwt.jwtAuthenticationConverter(jwtAuthenticationConverter()
        return http.build();
```

```
    }

    @Bean
    public JwtAuthenticationConverter jwtAuthenticationConverter() {
        JwtGrantedAuthoritiesConverter conv = new JwtGrantedAuthoritiesConverter();
        conv.setAuthorityPrefix("ROLE_");
        conv.setAuthoritiesClaimName("realm_access.roles");

        JwtAuthenticationConverter converter = new JwtAuthenticationConverter();
        converter.setJwtGrantedAuthoritiesConverter(conv);
        return converter;
    }
}
```

Это **полный минимум** для production-ready Resource Server с поддержкой Realm Roles из Keycloak.

Лучшие практики и частые ошибки

✅ Делайте так

- Всегда указывайте `issuer-uri` — это источник правды для автоконфигурации
- Используйте `preferred_username` как `user-name-attribute`
- Храните секреты в переменных окружения или Vault (HashiCorp, AWS Secrets Manager)
- Версионируйте Realm-конфигурацию в Git через `--import-realm`

❌ Не делайте так

- Не используйте устаревший Keycloak Adapter — только стандартный Spring Security OAuth2
- Не смешивайте `oauth2Login` и `oauth2ResourceServer` в одном `SecurityFilterChain`
- Не забывайте про префикс `ROLE_`: `.hasRole("ADMIN")` ищет `ROLE_ADMIN`
- Не используйте `.hasAuthority("ROLE_ADMIN")` и `.hasRole("ADMIN")` вперемешку без понимания разницы

Cheat Sheet

Тема	Главное
Keycloak иерархия	`Realm` → `Client` → `Users` → `Roles`. Экспортируйте Realm в JSON и храните в Git
OAuth2 / OIDC	Keycloak = Authorization Server, выдаёт Access Token (JWT). Spring Boot API = Resource Server, проверяет токен
JWT структура	`Header.Payload.Signature`. Роли Keycloak в `realm_access.roles`. Используйте кастомный `JwtAuthenticationConverter`
Stateless vs Stateful	`oauth2ResourceServer` + STATELESS для REST API. `oauth2Login` + сессии для MVC. Не смешивать!
Ключевые бины	`JwtDecoder` (подпись и claims), `JwtAuthenticationConverter` (маппит роли), `BearerTokenAuthenticationFilter` (перехватывает запросы)
Минимальный старт	Стартер `spring-boot-starter-oauth2-resource-server` + одна строка `issuer-uri` + кастомный конвертер ролей

Лекция 13. Введение в межсервисное взаимодействие

Микросервисная архитектура

Микросервисная архитектура позволяет:

- Разделять сервис на отдельные функции
- Независимо масштабировать отдельные части
- Обеспечивать повышенную устойчивость к сбоям
- Использовать разные технологии под разные задачи

Но переход от монолита к микросервисам — сложный процесс. Самый трудный этап — **изменение механизма взаимодействия внутренних компонентов**.

Монолит vs Микросервисы: взаимодействие

Архитектура	Взаимодействие
Монолитная	Компоненты обращаются друг к другу на уровне кода или функции в рамках одного процесса
Микросервисная	Модули взаимодействуют через **сеть** по протоколонезависимой технологии

Прямое преобразование внутрипроцессных вызовов в удалённые **не подойдёт** распределённым средам.

Сложность перехода на микросервисы

Унифицированного решения нет — для каждой ситуации нужно своё. Варианты:

1. **Изоляция бизнес-значимых микросервисов.** Внутренние микросервисы взаимодействуют асинхронно. Вызовы группируются, данные агрегируются и только потом отправляются клиенту.
2. **Архитектура с частично зависимыми, но согласованными микросервисами.** Каждый микросервис имеет свои данные и логику.

Типы связи: синхронная vs асинхронная

Тип	Описание	Пример
Асинхронный	Отправитель не ждёт ответ. Сообщения передаются в очередь брокера	**AMQP** (Advanced Message Queue Protocol)
Синхронный	При отправке клиент **ждёт ответ**. Задачи выполняются только после ответа сервера	**HTTP**

Брокеры сообщений: ActiveMQ и Kafka

Книги в этой области сравнивают и противопоставляют две популярные технологии:

- **Apache ActiveMQ**
- **Apache Kafka**

Система обмена сообщениями

Для общения двух приложений нужно:

1. **Определить интерфейс**:
   - Выбрать транспорт/протокол
   - Согласовать форматы сообщений
2. **Стандартизировать** через схему (XML, JSON) или неформальное соглашение

Пока **формат сообщений и порядок их отправки согласованы** — внутренности систем могут меняться. Эти системы **расцеплены** интерфейсом.

Роль посредника

Системы обмена сообщениями обычно используют **посредника (брокера)** между взаимодействующими системами для дальнейшего расцепления отправителя от получателя. Отправитель отправляет сообщение, **не зная**, где получатель, активен ли он, сколько его экземпляров.

Модель Point-to-Point (точка-точка)

Аналогия: Александра идёт на почту, отправляет посылку Адаму. Сотрудник забирает посылку и выдаёт квитанцию. Адаму не нужно быть дома. Александра

уверена, что посылка будет доставлена, и продолжает дела.

Эта модель реализуется через **очереди**:

- Очередь — буфер **FIFO**
- На неё может подписаться один или несколько потребителей
- **Каждое сообщение доставляется только одному** из подписанных потребителей
- Сообщения распределяются справедливо между потребителями

Надёжность vs Персистентность

- **Надёжность (durability)** — система сохраняет сообщения **при отсутствии подписчиков** до тех пор, пока потребитель не подпишется
- **Персистентность (persistence)** — система **записывает сообщение в хранилище** между получением и отправкой потребителю

Эти термины часто путают, но они выполняют разные функции.

Когда использовать Point-to-Point

Когда требуется **однократное действие**: внесение средств на счёт, выполнение заказа. Очереди обеспечивают в лучшем случае **доставку хотя бы один раз** (at-least-once).

Модель Publisher-Subscriber (Pub/Sub)

Аналогия: Габриэлла набирает номер конференции. Пока она подключена — слышит всё, что говорит спикер. Когда отключается — пропускает. При повторном подключении продолжает слышать.

Эта модель реализуется через **топики**:

- Сообщение, отправленное в топик, распределяется **по всем подписанным** пользователям
- Топики обычно **ненадёжные (nondurable)** — подписчики пропускают сообщения, отправленные пока они оффлайн
- Гарантия: **доставка не более одного раза** (at-most-once) для каждого потребителя

Когда использовать Pub/Sub

Когда сообщения **информационные**, и потеря одного — не критична. Пример: температурные датчики передают показания раз в секунду. Если одно сообщение пропущено — следующее придёт скоро.

Гибридные модели

Сценарии часто требуют **совмещения** моделей:

- Несколько систем нуждаются в копии сообщения (как Pub/Sub)
- Но требуется надёжность и персистентность (как Point-to-Point)

Решение: адресат (общий термин для очередей и топиков) распределяет сообщения **как топик**, но каждая система может иметь несколько потребителей, как в очереди. Гарантия — **один раз на каждую заинтересованную сторону**.

Гибридные модели применяются:

- В **ActiveMQ** — через виртуальные или составные адресаты
- В **Kafka** — неявно, как фундаментальное свойство дизайна адресата

ActiveMQ

ActiveMQ — классическая система обмена сообщениями, написанная в 2004 году. Восполняла потребность в open-source брокере сообщений.

ActiveMQ и JMS

ActiveMQ разработана как реализация спецификации **JMS** (Java Message Service):

- JMS описывает абстракции для отправки/получения сообщений в **асинхронном режиме**
- JMS включает чёткие указания по обязанностям клиента и брокера
- Сама **связь клиент-брокер исключена** из спецификации — чтобы существующие брокеры могли стать JMS-совместимыми

ActiveMQ свободна в определении своего протокола — **OpenWire**. Используется в реализациях JMS, .NET (NMS) и C++ (CMS).

Протоколы

AMQP (Advanced Message Queue Protocol)

ISO/IEC 19464:2014. Не путать с предшественником 0.x, который реализован в **RabbitMQ** (0.9.1).

AMQP 1.0:

- Двоичный протокол общего назначения
- Нет понятий клиентов или брокеров
- Поддерживает управление потоками, транзакции, QoS:
  - Не более одного раза (at-most-once)
  - Не менее одного раза (at-least-once)
  - Точно один раз (exactly-once)

MQTT (Message Queuing Telemetry Transport)

ISO/IEC 20922:2016. Лёгкий Pub/Sub-протокол. Используется для:

- Приложений «Машина-Машина» (M2M)
- Интернет вещей (IoT)

ActiveMQ поддерживает оба протокола.

JMS API

JMS определяет набор программных интерфейсов:

```
ConnectionFactory → Connection → Session
                                    │
                                    ├── MessageProducer
                                    ├── MessageConsumer
                                    └── Message
```

ConnectionFactory

- Интерфейс верхнего уровня для установления соединений
- В типичном приложении — единственный экземпляр (**Singleton**)
- В ActiveMQ — `ActiveMQConnectionFactory`
- Сообщает местонахождение брокера и низкоуровневые детали

Connection

- Долгоживущий объект, похожий на TCP-соединение
- Существует в течение всего жизненного цикла приложения
- **Потокобезопасный** — может работать с несколькими потоками одновременно
- Через него создаются объекты `Session`

Session

- Дескриптор потока при взаимодействии с брокером
- **НЕ потокобезопасный** — не может быть доступен нескольким потокам одновременно
- Основной транзакционный дескриптор: `commit` и `rollback`
- Через него создаются `Message`, `MessageConsumer`, `MessageProducer`, получаются ссылки на `Topic` и `Queue`

MessageProducer и MessageConsumer

- **MessageProducer** — отправляет сообщения адресату
- **MessageConsumer** — получает сообщения, два механизма:
  - **MessageListener** — реализуемый интерфейс обработчика, последовательно обрабатывает сообщения по мере поступления (один поток)
  - **Polling** — опрос через метод `receive()`

Message

Message — самая важная структура, переносит данные. Состоит из двух частей:

1. **Метаданные** — заголовки и свойства
2. **Тело сообщения**

Заголовки — хорошо известные элементы, определённые спецификацией JMS: `JMSDestination`, `JMSTimestamp` и т.д.

Свойства — произвольные фрагменты информации, упрощающие обработку или маршрутизацию без чтения тела.

Типы тела сообщения:

- **TextMessage** — для строк
- **BytesMessage** — для двоичных данных

Модель обмена сообщениями в ActiveMQ

Полезная (хоть и неточная) модель — **«две половинки мозга»**:

- Одна часть отвечает за **приём сообщений от продюсера**
- Другая отправляет сообщения потребителям

В реальности отношения сложнее из-за оптимизаций производительности, но модель достаточна для базового понимания.

Процесс отправки сообщения

1. **Маршалинг.** Отправляющий поток вызывает клиентскую библиотеку, маршализирует сообщение в нужный формат. Сообщение отправляется брокеру.
2. **Запись в хранилище.** Брокер записывает сообщение в своё внутреннее хранилище.
3. **Подтверждение записи.** Персистенс-адаптер получает подтверждение записи. Это **самая медленная часть** взаимодействия.
4. **Ответ клиенту.** Брокер отправляет клиенту подтверждение. Поток клиента может продолжить работу.

Publisher Confirmations

Если у брокера закончилась память или диск:

- Брокер **приостанавливает операцию отправки**, заставляя продюсер ждать (**Producer Flow Control**)
- ИЛИ отправляет **негативное подтверждение** продюсеру (через исключение)

В простом сценарии используется паттерн **Fire-and-forget** — записали и забыли.

Buffering

Когда брокер записывает данные на диск, он взаимодействует с **файловой системой**. Файловая система **буферизует** данные — минимизирует количество операций записи.

> Именно поэтому компьютер ругается на небезопасное извлечение USB — файлы могли не быть записаны.

Уровни кэширования:

1. **Буферный кэш файловой системы**
2. **Кэш контроллера диска** (аппаратный уровень)

ActiveMQ включает свой буфер записи — потоки записывают свои сообщения, ожидая завершения предыдущей записи. Буфер пишется одним действием, максимизируя пропускную способность.

Процесс получения сообщений

1. Потребитель выражает готовность принимать сообщения (через `MessageListener` или `receive()`).
2. ActiveMQ постранично читает (pages) сообщения из хранилища в память.
3. Сообщения перенаправляются (dispatched) консумеру, часто частями для снижения сетевого взаимодействия.
4. Сообщения помещаются в **prefetch buffer** (буфер предварительной выборки) у консумера — выравнивает поток сообщений.
5. Логика приложения вычитывает сообщения из буфера и отправляет подтверждение брокеру.
6. После получения подтверждения брокером сообщение **удаляется** из памяти и хранилища.

   Термин «удаление» вводит в заблуждение — в журнал записывается запись о подтверждении и увеличивается указатель в индексе. **Фактическое удаление** выполняется **сборщиком мусора** в фоновом потоке.

Курсорный механизм

В действительности, ActiveMQ не просто постранично читает данные, а использует **механизм курсора** между принимающей и перенаправляющей частями брокера для минимизации взаимодействия с хранилищем.

> Используемый **протокол согласования (coherency)** — значительная часть того, что делает механизм диспетчеризации ActiveMQ **отличным от Kafka**.

Лекция 14. Apache Kafka

Что такое Kafka

Apache Kafka — distributed event streaming platform (распределённая платформа потоковой обработки событий).

Назначение

- **High-performance data pipelines** — высокопроизводительные пайплайны данных
- **Streaming analytics** — потоковая аналитика
- **Data integration** — интеграция данных
- **Mission-critical applications** — критически важные приложения

Ключевые свойства Kafka

- **Распределённость**
- **Отказоустойчивость**
- **Высокая доступность**
- **Надёжность и согласованность данных**
- **Высокая производительность** (пропускная способность)
- **Горизонтальное масштабирование**
- **Интегрируемость**

Немного истории

- Разработана в **LinkedIn**
- Программный код опубликован в **2011 году**
- С **2012** под крылом Apache
- Реализована на **Scala и Java**
- Название придумал Jay Kreps в честь **Franz Kafka** («известный писатель»)
- Девиз: *«Kafka — a system optimized for writing»*

Задача, которую решает Kafka

Без Kafka (прямые соединения между сервисами):

Сложности при количестве сервисов:

- Надёжность и гарантия доставки

- Подключение новых получателей
- Отправители знают получателей
- Техническая поддержка
- Интеграция разных стеков

С Kafka (брокер посередине):

Удобства:

- Надёжность и гарантия доставки
- Подключение новых получателей
- Отправители не знают получателей
- Техническая поддержка
- Интеграция разных стеков

Основные сущности Kafka

1. **Broker** — сервер Kafka
2. **Zookeeper** — координатор кластера
3. **Message (Record)** — сообщение
4. **Topic / Partition** — топик и его партиции
5. **Producer** — отправитель
6. **Consumer** — получатель

Kafka Broker (Kafka Server / Kafka Node)

Функции брокера

- □ Приём сообщений
- □ Хранение сообщений
- □ Выдача сообщений

Kafka-кластер

- **Масштабирование** — несколько брокеров работают вместе
- **Репликация** — данные дублируются между брокерами

Zookeeper

Zookeeper — координатор кластера Kafka. Его функции:

- □ Состояние кластера

- □ Конфигурация
- □ Адресная книга (data)
- □ Выбор Controller
- □ Обеспечение **консистентности**

Controller

Один из брокеров кластера становится **Controller** — отвечает за управление и принятие решений в кластере. Выбирается через Zookeeper.

Kafka Message (Record / Event)

Сообщение представляет собой **key-value пару** с дополнительными полями:

Поле сообщения	Описание
Key	Ключ (опциональный). Используется для распределения сообщений по кластеру
Value	Содержимое сообщения — массив байт
Timestamp	Время сообщения (от эпохи). Устанавливается при отправке или обработке внутри кластера
Headers	Набор key-value пар с пользовательскими атрибутами сообщения

Kafka Topic — поток данных (stream of data)

Topic — это поток данных, через который сообщения проходят от продюсеров к консумерам.

Свойства Topic

- □ **FIFO** — гарантирует порядок (ordering support)
- Сообщения сохраняются последовательно

Partitions — разделение топика

Топик разделяется на **партиции** для:

- ⚡ **Ускорения чтения/записи данных** (параллелизация)

FIFO per partition — порядок гарантируется **внутри партиции**, но не во всём топике (если партиций больше одной).

Topic placement by Brokers (в кластере)

Партиции топика **размещаются на разных брокерах**:

```
Topic A: partition 0, 1, 2
  Broker 1: P0
  Broker 2: P1
  Broker 3: P2
```

> ⚠ Возможна **несбалансированность** размещения партиций топика — учитывается количество всех партиций всех топиков на брокере.

При наличии нескольких топиков (B, C, D и т.д.) кафка пытается балансировать нагрузку между брокерами.

Storage for Kafka Topic — где хранятся данные

Данные топика хранятся в **файловой системе брокера**, в каталоге `/logs`:

- Для каждой партиции — отдельная директория
- Внутри директории несколько файлов:
  - **`.log`** — сами сообщения
  - **`.index`** — индекс по offset
  - **`.timeindex`** — индекс по timestamp

Segments — сегменты файлов

Файлы партиции разбиваются на **сегменты**:

- Имя файла соответствует **start offset** сегмента
- В **timeindex** хранится **max message timestamp** сегмента

Это нужно для:

- Эффективного управления хранением

- Быстрого поиска по offset и timestamp
- Удаления старых данных

Data removing from Kafka Topic — удаление данных

> ❌ **Операция удаления данных не поддерживается!**

✅ Поддерживается **автоматическое удаление данных по TTL (time-to-live):**

- Удаляются целиком **сегменты** партиций, не отдельные сообщения
- Условие: `segment timestamp expired → to delete`

Data Replication — надёжность данных и отказоустойчивость

Если **один брокер падает** — данные его партиций **потеряны!**

Решение: replication-factor

`Set replication-factor (>1)`

- Каждая партиция реплицируется на **несколько брокеров**
- При падении одного брокера — данные **сохраняются на других**

  Но просто с репликацией не всё так просто. Есть нюансы.

Master-Slave — гарантия согласованности данных

При `replication-factor: 3` — у каждой партиции есть 3 копии.

Leader и Followers

- **Kafka Controller назначает Leader-реплики**
- ✅ **Операции чтения и записи производятся ТОЛЬКО с Leader-репликой**

`producer → Leader → consume`

Остальные реплики (Followers) только синхронизируются с Leader.

Несбалансированность Leaders

> ⚠ Возможна проблема несбалансированности Leaders и Followers — все запросы к одному брокеру, остальные «отдыхают».

Kafka должна перебалансировать Leader’ов между брокерами.

Data sync between Leader and Followers

Followers must fetch data from Leader

Followers **периодически** запрашивают данные у Leader (`fetch periodically`).

Проблема: что если Leader упал?

> Кто теперь Leader? И все ли данные у него есть?

Если новым Leader станет Follower с **отстающими** данными — данные могут быть **потеряны**. Это **ненадёжно**!

Решение — In-Sync Replicas (ISR)

ISR — реплики, которые синхронизированы с Leader.

Настройка: `min.insync.replicas = 3`

- Запись считается успешной только когда она прошла на **минимум N ISR**
- **ISR Follower** — надёжный кандидат на Leader

```
Если запись прошла в Leader + ISR(1) + ISR(2) — данные надёжны.
Если новый Leader выбирается из ISR — данные не теряются.
```

Kafka Producer — отправка сообщений

Kafka Producer — высокопроизводительный отправитель сообщений.

acks — гарантии доставки

Параметр **`acks`** определяет, чего ждёт продюсер:

acks	Что происходит	Гарантия	Производительность
`0`	Producer не ждёт подтверждения отправки	(Самый ненадёжный — могут теряться сообщения)	Самый быстрый

acks	Что происходит	Гарантия	Производительность
1	Producer ждёт подтверждения только от Leader-реплики	Компромиссный — могут теряться сообщения, если лидер упал до репликации	Средний
-1 (all)	Producer ждёт подтверждения от **всех ISR-реплик**, включая Leader	Надёжный — сообщения не теряются	Самый медленный

Delivery semantic support

Kafka поддерживает три семантики доставки:

- **At most once** — не более одного раза (сообщение может потеряться)
- **At least once** — хотя бы один раз (сообщение может дублироваться)
- **Exactly once** — ровно один раз (через **idempotence**)

Конвейер обработки сообщения Producer’ом

Что происходит при `send message`:

1. **fetch metadata** — Producer получает метаданные от Zookeeper:
   - `cluster state`
   - `topic placement`
   - ⚠ **Expensive operation** — блокирует `send()` до получения метаданных (или таймаут 60 сек)
2. **serialize message** — сериализация ключа и значения через `key.serializer` и `value.serializer` (например, `StringSerializer`)
3. **define partition** — определение партиции:
   - `explicit partition` — явное указание
   - `round-robin` — по очереди
   - `key-defined` — `key_hash % n`
4. **accumulate batch** — накопление в батч (контролируется `batch.size` и `linger.ms`)

5. **compress message** — сжатие
6. **send to Broker** — отправка на брокер

   Producer пытается **группировать сообщения** в батчи для эффективности, чтобы лучше использовать сеть и хранилище.

Kafka Consumer — получение сообщений

Kafka Consumer — высокопроизводительный получатель.

Принцип работы

Consumer **сам** запрашивает сообщения у брокера (`poll messages`). В отличие от ActiveMQ — Kafka не пушит сообщения консумерам.

Подключение Consumer

Consumer подключается к **Leader-репликам всех партиций** топика.

 В один поток это может быть медленно, если партиций много.

Consumer Group

Чтобы распараллелить чтение — Consumer’ы объединяются в **Consumer Group** через `<group.id>`:

- Каждая партиция читается **одним Consumer’ом** в группе
- Сообщения распределяются между Consumer’ами группы
- При добавлении/удалении Consumer’а — происходит **rebalance**

Пример: - Topic с 4 партициями - Consumer Group из 4 Consumer’ов - Каждый Consumer читает свою партицию

Kafka Consumer Offset

Offset — last consumed message by Group (последнее прочитанное сообщение группой).

Что хранится

Для каждой Consumer Group хранится:

Field	Value
Partition	A/0 (топик/партиция)
Group	имя группы
Offset	номер последнего обработанного сообщения

Эта информация хранится в **специальном системном топике**: `__consumer_offsets`.

Зачем offsets

- Если Consumer перезапускается — он продолжает с последнего обработанного offset
- Если приходит **новая** группа — она может начать с **начала топика** или с **последнего offset**

Kafka Consumer Offset commit

После обработки сообщения нужно **закоммитить offset**.

Типы коммитов

Тип	Семантика	Особенность
Auto commit	**At most once**	Сообщения могут пропускаться. Коммит происходит автоматически, до обработки
Manual commit	**At least once**	Сообщения могут дублироваться. Коммит после обработки — если упали, прочитаем снова

А как же exactly once?

Custom offset management — управление offset вручную, в той же транзакции, что и обработка. Тогда:

- Обработка и commit в одной транзакции
- При неуспехе — откат всего
- **Exactly once** — не пропустили, не дублировали

Kafka Consumer Offset missing

Что если Consumer Group **неактивна** долгое время?

Параметры

- **`max inactive period`** — `offsets.retention.minutes` (по умолчанию **7 дней**)
- После истечения — offsets **удаляются** из `__consumer_offsets`

После активации группы

После повторной активации **используется параметр `auto.offset.reset`**:

Значение	Что делает
`earliest`	Читать с самого начала топика
`latest`	Читать только новые сообщения

Kafka performance — почему так быстро?

Свойство	Описание
Scalable architecture	Масштабируемая архитектура — партиции, брокеры, consumer groups
Sequential write and read	Последовательная запись и чтение (быстрее случайных операций)
No random read	Нет случайных чтений
Zero-copy	Передача данных от диска в сеть без копирования в user-space
Huge amount of settings	Множество настроек для разных кейсов

Итоги: Apache Kafka

- □ Kafka — **очень популярна**
- □ Kafka — **надёжное проверенное решение**
- □ Kafka — **высокопроизводительный инструмент**
- □ Kafka — **широковещательная с упорядочиванием** (FIFO per partition)
- □ Kafka — **интегрируема со всеми тулами**
- □ Kafka — **гибкая в конфигурации**

- ⚠ Kafka — **нужно понимать нюансы** (acks, ISR, offset management, rebalance)