







# Основы Spring

Урок 2





## Что будет на уроке сегодня

-  Основные принципы систем сборки и их роли в разработке программного обеспечения
-  Краткий обзор и сравнение Maven и Gradle
-  Теория и практика Maven
-  Теория и практика Gradle
-  Сравнение Maven и Gradle
-  Заключение



# Основы Spring



**Spring** — это мощный фреймворк, созданный для упрощения разработки Java-приложений.



Он базируется на нескольких ключевых принципах, и сегодня мы поговорим о двух из них: DI (Dependency Injection) и IoC (Inversion of Control).



**DI**, или внедрение зависимостей, — это концепция, которая позволяет нам делать наши приложения более гибкими и легкими для тестирования.



**IoC**, или инверсия управления, — это еще одна важная концепция в Spring. Она означает, что не наше приложение контролирует жизненный цикл его компонентов (как это обычно происходит), а наоборот — фреймворк контролирует наше приложение.



# Основы Spring

В Spring Framework существует множество аннотаций, которые используются для определения и конфигурации бинов. Вот несколько основных из них:

💡 @Component

💡 @Service

💡 @Repository

💡 @Controller

💡 @Configuration

💡 @Bean



# Основы Spring

В Spring Framework существует множество аннотаций, которые используются для определения и конфигурации бинов. Вот несколько основных из них:

```
1 @Configuration
2 public class AppConfig {
3
4     @Bean
5     public MyService myService() {
6         return new MyServiceImpl();
7     }
8 }
9
10 @Service
11 public class MyServiceImpl implements MyService {
12     // ...
13 }
14
15 @Repository
16 public class MyRepository {
17     // ...
18 }
19
20 @Controller
21 public class MyController {
22     // ...
23 }
```



## Основы Spring

Давайте погрузимся глубже в преимущества Spring, посмотрев на пример кода. Допустим, у нас есть класс Car, который зависит от класса Engine.

```
1 public class Car {  
2     private Engine engine;  
3  
4     public Car() {  
5         this.engine = new Engine();  
6     }  
7 }
```

В этом случае, мы напрямую создаём объект класса Engine внутри класса Car. Вместо этого, мы могли бы воспользоваться преимуществами Spring и внедрить зависимость через конструктор:

```
1 public class Car {  
2     private Engine engine;  
3  
4     public Car(Engine engine) {  
5         this.engine = engine;  
6     }  
7 }
```



## Типы бинов

1

### **Singleton:**

Это область видимости по умолчанию. Когда бин определен как Singleton, Spring IoC контейнер создает единственный экземпляр бина, и все запросы на получение этого бина возвращают один и тот же объект.

2

### **Prototype:**

Когда бин определен как Prototype, Spring IoC контейнер создает новый экземпляр бина каждый раз, когда он запрашивается.

3

### **Request, Session, и Application:**

Эти области видимости применяются только в веб-приложениях. Бин области видимости Request создается для каждого HTTP-запроса.

4

### **WebSocket:**

Эта область видимости доступна для бинов, которые должны быть связаны с жизненным циклом WebSocket.



## Типы бинов

Для определения области видимости бина в Spring, вы можете использовать аннотацию `@Scope`. Например:

```
1 @Component
2 @Scope("prototype")
3 public class PrototypeBean {
4     // ...
5 }
```

В этом примере `PrototypeBean` будет создаваться каждый раз, когда он запрашивается из Spring контейнера.





## Создание простого Spring Boot приложения

Давайте создадим простое веб-приложение. Выберем Java как язык программирования и последнюю версию Spring Boot. В разделе “Dependencies” выберем “Spring Web”. Нажмите “Generate”, и Spring Initializr создаст для вас проект и скачает его в виде .zip-файла.

**Распакуйте этот .zip-файл, и вы увидите структуру проекта Spring Boot.**

**Давайте рассмотрим основные компоненты этой структуры:**

1. `src/main/java`: Здесь находится весь ваш исходный код.
2. `src/main/resources`: Здесь находятся ресурсы вашего приложения, такие как файлы конфигурации, статические веб-ресурсы и т.д.
3. `src/test/java`: Здесь находятся ваши тесты.
4. `pom.xml` или `build.gradle`: Этот файл содержит информацию о вашем проекте и его зависимостях.



## Создание простого Spring Boot приложения

Давайте напишем простой контроллер для нашего веб-приложения. В нашем контроллере будет один метод, который будет отвечать на HTTP-запросы GET на корневой URL ("/"). Этот метод будет возвращать простое текстовое сообщение. Вот как это выглядит:

```
1 package com.example.demo;
2
3 import org.springframework.web.bind.annotation.GetMapping;
4 import org.springframework.web.bind.annotation.RestController;
5
6 @RestController
7 public class HelloController {
8
9     @GetMapping("/")
10    public String hello() {
11        return "Hello, Spring Boot!";
12    }
13 }
```



## Создание простого Spring Boot приложения

Вернемся к нашему классу Car и Engine.  
Если мы хотим, чтобы SpringIoC контейнер управлял нашим объектом Car, мы можем аннотировать его аннотацией @Component:

Мы можем использовать аннотацию @Autowired для того, чтобы Spring IoC контейнер внедрил (или “подключил”) экземпляр Engine в наш Car:

```
1 @Component
2 public class Car {
3     ...
4 }
```

```
1 @Component
2 public class Car {
3     private final Engine engine;
4
5     @Autowired
6     public Car(Engine engine) {
7         this.engine = engine;
8     }
9     ...
10 }
```



# Создание простого Spring Boot приложения

Spring Framework имеет две основные реализации контейнера:



**BeanFactory:** Это базовый контейнер, который предоставляет функциональность IoC. Он определяет основной контракт, который должны выполнять все контейнеры.

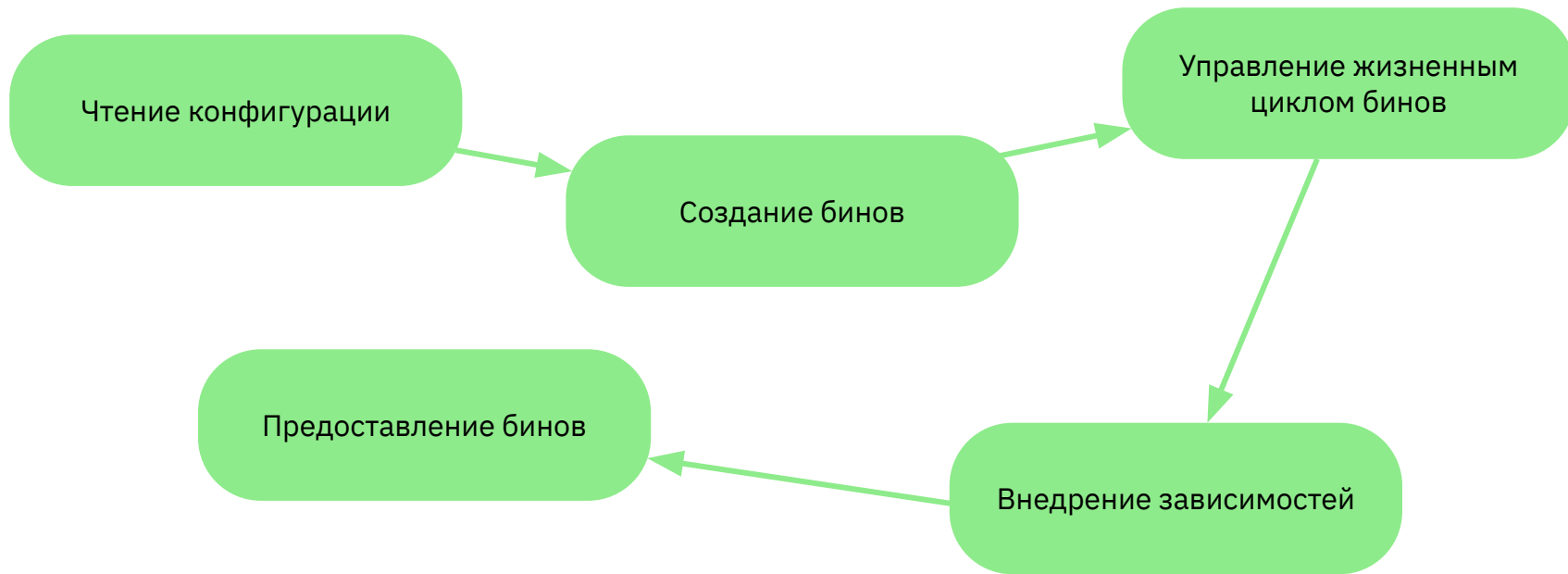


**ApplicationContext:** Это расширение BeanFactory, которое предоставляет дополнительные возможности, такие как интеграция с другими компонентами Spring, поддержка аннотаций и др.



# Создание простого Spring Boot приложения

Контейнер работает следующим образом:





# Конфигурация Spring Boot приложения

1

через файлы `application.properties`  
или `application.yml`

2

через Java-код



## Файлы конфигурации

1

.properties — это старый добрый формат файлов конфигурации Java. Каждый параметр задается в виде пары “ключ-значение”, разделенных знаком равенства.

2

.yaml (YAML Ain't Markup Language, или YAML — это не язык разметки) — это более современный и гибкий формат. Он позволяет задавать структурированные данные с использованием отступов, что может быть более удобно для сложных конфигураций.



## Файлы конфигурации. Параметры

- ✓ `server.port`: порт, на котором будет работать ваше приложение. По умолчанию это 8080, но вы можете задать любой другой порт.
- ✓ `spring.application.name`: имя вашего приложения. Оно может быть полезно для логирования и других вещей.
- ✓ `spring.profiles.active`: активные профили Spring. Мы поговорим об этом чуть позже.
- ✓ `logging.level.root`: уровень логирования для вашего приложения. Вы можете задать его, например, как INFO, WARN, ERROR или DEBUG.
- ✓ `spring.main.banner-mode`: режим баннера при запуске вашего приложения. Вы можете отключить баннер, установив этот параметр в OFF.





## Конфигурация Spring Boot приложения

```
1 @ConfigurationProperties(prefix = "app")
2 public class AppProperties {
3     private String name;
4     private String description;
5
6     // getters
7     public String getName() {
8         return this.name;
9     }
10
11     public String getDescription() {
12         return this.description;
13     }
14
15     // setters
16     public void setName(String name) {
17         this.name = name;
18     }
19
20     public void setDescription(String description) {
21         this.description = description;
22     }
23 }
```



## Профили Spring

Вы можете иметь разные реализации сервиса для разработки и продакшена:

```
1 @Service
2 @Profile("development")
3 public class DevelopmentMyService implements MyService {
4     // реализация для разработки
5 }
```

```
1 @Service
2 @Profile("production")
3 public class ProductionMyService implements MyService {
4     // реализация для производства
5 }
```



## Профили Spring

Создадим интерфейс `MyService` и две его реализации — одну для разработки и одну для производства. Затем мы внедрим `MyService` в контроллер и увидим, какая реализация будет использоваться.

```
1 public interface MyService {
2     String getMessage();
3 }
4
5 @Service
6 @Profile("development")
7 public class DevelopmentMyService implements MyService {
8     @Override
9     public String getMessage() {
10         return "Development Service";
11     }
12 }
```



## Профили Spring

```
1 @Service
2 @Profile("production")
3 public class ProductionMyService implements MyService {
4     @Override
5     public String getMessage() {
6         return "Production Service";
7     }
8 }
9
10 @RestController
11 public class MyController {
12     private final MyService myService;
13
14     public MyController(MyService myService) {
15         this.myService = myService;
16     }
17
18     @GetMapping("/message")
19     public String getMessage() {
20         return myService.getMessage();
21     }
22 }
```



## Конфигурация через Java-код

В Spring Boot вы можете создать классы конфигурации, аннотированные `@Configuration`, и использовать методы с аннотацией `@Bean` для определения компонентов, которые будут управляться Spring IoC контейнером.

```
1 @Configuration
2 public class AppConfig {
3     @Bean
4     public MyService myService() {
5         return new MyServiceImpl();
6     }
7 }
```



# Автоконфигурация

Автоконфигурация — это механизм, который позволяет Spring Boot автоматически настраивать ваше приложение на основе тех библиотек, которые присутствуют в вашем classpath.

В этом заключается одно из основных преимуществ Spring Boot: он упрощает настройку и позволяет вам быстрее приступить к реальной работе. Вместо того чтобы тратить время на настройку различных библиотек, вы можете просто начать использовать их!





## Домашнее задание

Создайте базовое веб-приложение с использованием Spring Boot, которое будет включать в себя основные компоненты: контроллеры, сервисы и репозитории. Приложение может быть простым, например, приложение для управления книжной библиотекой с операциями CRUD (создание, чтение, обновление и удаление) книг.





**Спасибо за внимание**

