Оглавление

[Лонгрид: Введение в Spring Boot 1](#_Toc202903998)

[1. Зачем нужен Spring Boot? 1](#_Toc202903999)

[Проблемы классического Spring 1](#_Toc202904000)

[Вот пример web.xml для классического Spring MVC с минимальной конфигурацией: 1](#_Toc202904001)

[Как решает эти проблемы Spring Boot? 2](#_Toc202904002)

[Концепция и этапы работы Spring Boot 2](#_Toc202904003)

[Как выглядит автоконфигурация внутри spring-boot-autoconfigure? 4](#_Toc202904004)

[3. Профили 5](#_Toc202904005)

[4. Пример работы с профилями в Spring Boot 6](#_Toc202904006)

[6. Заключение 7](#_Toc202904007)

# Лонгрид: Введение в Spring Boot

## 1. Зачем нужен Spring Boot?

### Проблемы классического Spring

Spring Framework — это очень мощный инструмент для создания корпоративных Java-приложений. Однако у классического Spring, как у всякого фреймворка, есть ряд особенностей, которые затрудняют быстрый старт и усложняют поддержку проектов:

* **Обилие конфигурации**: Для запуска даже простого приложения раньше требовалось создавать множество XML-файлов или писать подробные Java-конфигурации. Это занимало много времени и увеличивало вероятность ошибок.
* **Ручная настройка инфраструктуры**: Необходимо было самостоятельно настраивать сервер приложений, подключать зависимости, регистрировать компоненты.
* **Повторяющийся шаблонный код**: Многие настройки и фрагменты кода приходилось копировать из проекта в проект.

**Пример:** Чтобы запустить простое Spring MVC-приложение, нужно было сделать примерно следующее:

- Создать web.xml

- Настроить DispatcherServlet

- Описать бины в XML или Java-конфигурации

- Подключить все зависимости вручную

## Вот пример web.xml для классического Spring MVC с минимальной конфигурацией:

<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee  
 http://xmlns.jcp.org/xml/ns/javaee/web-app\_3\_1.xsd"  
 version="3.1">  
  
 <display-name>Spring MVC Application</display-name>  
  
 <servlet>  
 <servlet-name>dispatcher</servlet-name>  
 <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  
 <init-param>  
 <param-name>contextConfigLocation</param-name>  
 <param-value>/WEB-INF/spring/dispatcher-servlet.xml</param-value>  
 </init-param>  
 <load-on-startup>1</load-on-startup>  
 </servlet>  
  
 <servlet-mapping>  
 <servlet-name>dispatcher</servlet-name>  
 <url-pattern>/</url-pattern>  
 </servlet-mapping>  
  
</web-app>

Примерно такая же картина была не только в мире Spring, но и в других Java фрейворках и технологиях. Пример из практики: фреймворк для работы с базой данных MyBatis, платформа для корпоративных приложений на Java, где также требовалась ручная настройка конфигураций.

### Как решает эти проблемы Spring Boot?

Spring Boot — это расширение Spring, которое автоматизирует и упрощает процесс создания приложений:

- **Автоматическая конфигурация**: Boot сам определяет, какие компоненты и настройки нужны, исходя из ваших зависимостей

- **Могут поставляться встроенные приложения (на примере web сервера)**: Не нужно отдельно устанавливать Tomcat или Jetty - сервер уже включён в приложение

- **Быстрый запуск**: Приложение можно запустить одной командой, без дополнительных шагов по развертыванию

- **Стартеры**: Готовые наборы зависимостей для типовых задач (web, JPA, security и др.), которые избавляют от необходимости вручную подбирать библиотеки

Spring Boot позволяет сосредоточиться на бизнес-логике, а не на инфраструктуре и рутинной настройке.

### Концепция и этапы работы Spring Boot

**Что такое Spring Boot концептуально?**

**Spring Boot можно рассматривать как набор инструментов для того, чтобы запустить приложение на Spring.** Он не заменяет Spring Framework, а является его расширением, которое предлагает готовые решения для типовых задач.

**Физически, Spring Boot — это просто набор библиотек (JAR-файлов), которые вы добавляете в свой проект.** Ключевыми из них являются:

- spring-boot-starter-\*: Это “стартеры”, которые не содержат кода и просто описывают набор зависимостей для конкретной задачи (например, spring-boot-starter-web для веб-приложений)

- spring-boot-autoconfigure: Это “сердце” Spring Boot. Библиотека содержит классы, которые автоматически настраивают ваше приложение на основе подключенных зависимостей

Ключевая идея — **“соглашение вместо конфигурации” (convention over configuration)**. Вместо того чтобы вручную прописывать все настройки, Spring Boot предоставляет конфигурации по умолчанию. Разработчик может вмешаться, только когда ему нужно изменить стандартное поведение.

**Как это работает?** Давайте разберем работу spring boot по этапам.

Когда вы запускаете Spring Boot-приложение, происходит следующее:

1. **Запуск (main метод с SpringApplication.run)**: Это точка входа. Например, при нажатии кнопки “старт” в Idea
2. **Анализ зависимостей (Classpath Scanning)**: Spring Boot “сканирует” ваш проект и видит, какие стартеры вы подключили. Например, он находит spring-boot-starter-web
3. **Активация автоконфигурации**: На основе найденных стартеров, Boot активирует нужные классы автоконфигурации. Для starter-web это будут конфигурации для веб-сервера (Tomcat), обработки HTTP и т.д.
4. **Создание “умных” бинов**: Автоконфигурация создаёт бины (объекты), но делает это “умно”, с помощью аннотаций @ConditionalOn.... Например:

* @ConditionalOnClass(Tomcat.class): “Создать бин для Tomcat, только если в проекте есть класс Tomcat”.
* @ConditionalOnMissingBean: “Создать этот бин, только если пользователь не создал свой собственный с таким же названием”

1. **Сканирование компонентов проекта**: Boot находит ваши классы с аннотациями @RestController, @Service, @Component и добавляет их в контекст приложения
2. **Запуск приложения**: Наконец, Spring Boot запускает встроенный сервер (например, Tomcat) и разворачивает приложение. Теперь оно готово принимать запросы.

Этот процесс позволяет запустить полнофункциональное веб-приложение, написав всего несколько строк кода. И если нужно изменить настройки и поведение, то нужно изменить соответствующие настройки в application.yaml (application.properties). Например, для настроек сервера, которую часто можно видеть в application.yaml - это секция “server”.

Примеры стартеров:

- spring-boot-starter-web - для создания REST API и веб-приложений

- spring-boot-starter-data-jpa - для работы с базой данных через JPA

- spring-boot-starter-security - для добавления аутентификации и авторизации

**Давайте также рассмотрим пример запуска сервера в Spring Boot на примере Tomcat:**

1. **Запуск приложения**

Запускаем приложение командой:

* ./gradlew bootRun
* или
* java -jar myapp.jar

1. **Выполнение метода main**

* В коде вызывается:
* public static void main(String[] args) {  
   SpringApplication.run(DemoApplication.class, args);  
  }
* Это точка входа для Spring Boot.

1. **Создание SpringApplication**

Класс SpringApplication анализирует класс с аннотацией @SpringBootApplication и все зависимости на classpath.

1. **Анализ зависимостей и автоконфигурация**

* Spring Boot видит, что у нас есть зависимость spring-boot-starter-web.
* Это означает, что на classpath присутствует Tomcat (или Jetty/Undertow), Spring MVC и другие необходимые компоненты.

1. **Автоматическое создание и настройка сервера**

* Благодаря автоконфигурации (spring-boot-autoconfigure), Spring Boot находит класс TomcatServletWebServerFactory (или аналогичный для Jetty/Undertow).
* Создаётся бин этого класса, который отвечает за запуск встроенного сервера.

### Как выглядит автоконфигурация внутри spring-boot-autoconfigure?

В библиотеке spring-boot-autoconfigure содержатся специальные классы, которые автоматически настраивают компоненты приложения. Примерно так выглядит класс автоконфигурации для Tomcat:

@Configuration  
public class TomcatWebServerFactoryAutoConfiguration {  
 @Bean  
 public TomcatServletWebServerFactory tomcatServletWebServerFactory() {  
 return new TomcatServletWebServerFactory();  
 }  
}

где @Configuration и @Bean — стандартные аннотации Spring для конфигурации и создания бинов.

**Такие классы лежат внутри spring-boot-autoconfigure и автоматически подключаются, когда вы добавляете стартеры в свой проект.**

1. **Создание ApplicationContext**

* Spring Boot создаёт контекст приложения, сканирует компоненты (@RestController, @Service и т.д.), регистрирует их как бины.

1. **Инициализация сервера**

* Внутри автоконфигурации вызывается метод, который создаёт и настраивает экземпляр Tomcat (или другого сервера).
* Сервер настраивается на порт по умолчанию (8080) или на тот, что указан в application.properties.

1. **Регистрация сервлетов и фильтров**

* Все контроллеры и фильтры, описанные в коде, автоматически регистрируются как сервлеты и фильтры в Tomcat.

1. **Запуск сервера**

* Встроенный Tomcat запускается внутри того же процесса JVM, что и ваше приложение.
* Сервер начинает слушать указанный порт и готов принимать HTTP-запросы.

1. **Приложение запущено**

* Вы видите в консоли сообщение вроде:
* Tomcat started on port(s): 8080 (http)  
  Started DemoApplication in 2.345 seconds (JVM running for 2.789)
* Теперь приложение доступно по адресу http://localhost:8080 (или другому порту, которое указано в application.yaml).

Таким образом всё, что остается сделать программисту - это подключить стартер, сделать настройку (как правило, в application.yaml/application.properties файле) и запустить приложение.

## 3. Профили

Поговорим о профилях. Представим ситуацию, что разработчик решил задачу и написал код. Стандартные пути для выкатки этой задачи на прод:

1. Проверить работу локально (например, запустить приложение на своем компьютере и проверить работу endpoints)
2. Запустить тесты
3. Выкатить задачу на какой-то стенд, например стенд с названием dev, куда разработчики заливают все задачи для первоначального тестирования и отладки
4. После проверки на dev - сделать деплой на prod

Во всех случаях (локальное окружение, прогон тестов, dev и prod стенды) нужны различные настройки: разные хосты и креды для подключения к БД, разные наименования хостов для каких-то внутренних интеграций (например, url для запросов в какой-то микросервис могут выглядеть по разному на dev и prod стенде). Чтобы не указывать каждый раз это вручную, такие настройки описываются где-то один раз для разных стендов (это и называется профили) и далее Spring можно запустить с указанием конкретного профиля (стенда), причем все настройки применятся автоматически.

То есть на помощь нам приходят профили Spring Boot. В Spring это достигается либо указанием аннотации @Profile, либо разделением на профили с помощью наименования файла application (application-local.properties, application-dev.properties, application-prod.properties и т.д.).

## 4. Пример работы с профилями в Spring Boot

Давайте разберем, как работать с профилями на примере. За основу возьмем стандартный файл с настройками в Spring application.properties

1. **Создаем разные файлы настроек со следующим содержимым. Данные файлы обычно размещают там же, где находится стандартный application.properties:**

**application.properties:** (файл настроек без профиля, по умолчанию)

spring.application.name=MyApp  
server.port=8080

**application-local.properties:** (файл настроек без профиля для локального запуска)

spring.application.name=MyAppLocal  
server.port=8080

**application-test.properties:** (файл настроек без профиля для запуска тестов)

spring.application.name=MyAppTest  
server.port=8080

**application-dev.properties:** (dev стенд)

spring.datasource.url=jdbc:h2:mem:testdb  
logging.level.root=DEBUG

**application-prod.properties:** (prod сервер)

spring.datasource.url=jdbc:postgresql://prod-server/mydb  
logging.level.root=INFO

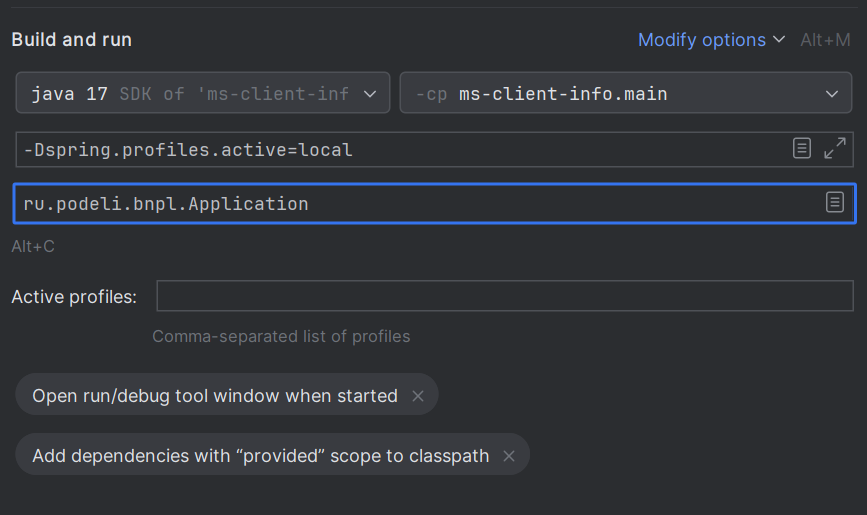
1. **Используем аннотацию @Profile в коде (опционально):**

@Service  
@Profile("dev")  
public class DevOnlyService {  
 // Этот бин будет создан только если активен профиль dev  
}

1. **Запускаем приложение с нужным профилем. Например, на dev стенде примерно следующим образом происходит запуск приложения через kubernetes:**

* Через командную строку:
* java -jar myapp.jar --spring.profiles.active=dev
* Или через переменную окружения:
* export SPRING\_PROFILES\_ACTIVE=dev  
  ./gradlew bootRun

Точно также можно запускать приложение локально. Я, как разработчик, часто тестирую свои приложения с локальным профилем через создание файла application.yml (равносильно application.properties) и указание профиля прямо из Idea:



1. **Spring Boot сам подхватит нужные настройки:**

* Если активен профиль dev, будут применены настройки из application.properties и application-dev.properties.
* Если активен профиль prod, будут применены настройки из application.properties и application-prod.properties.

Итого, профили позволяют легко переключаться между разными конфигурациями без изменения основного кода приложения. Это очень сильно упрощает разработку - от локальной проверки, до запуска приложения на проде.

## 6. Заключение

В данном уроке мы рассмотрели работу Spring Boot с точки зрения его внутренного устройства: изучили, каким образом работают стартеры, из каких компонентов они состоят и как работают изнутри. А также поговорили про профили Spring, для чего это нужно и как запускать профили.