Обратная сторона Spring

* [Java](https://habr.com/ru/hub/java/)

Неделя Spring на Хабре, судя по всему, открыта. Хочется сказать спасибо переводчику и комментаторам статьи ["Почему я ненавижу Spring"](https://habrahabr.ru/post/334118/), которая не смотря на сильный негативный посыл в названии вызвала ряд интересных дискуссий, а так же тем, кто отреагировал на мою прошлую статью [Как писать на Spring в 2017](https://habrahabr.ru/post/333756/). Во многом благодаря комментариям к прошлой статье и появилась эта.

В этот раз мы погрузимся в пучины Spring фреймворка, разоблачим его магию, посмотрим как базовое веб приложение выглядит изнутри, и разберемся, какую-же задачу и как решает Spring Boot.

В комментариях к предыдущей статье несколько человек очень справедливо указали, что пример Hello World-а на Spring все же не очень показателен. Spring, особенно с использованием Spring Boot, дает ощущение простоты и всемогущества, но непонимание основ и внутренностей фреймворка ведет к большой опасности получить стектрейсом по логу. Что ж, чтобы немного развеять ощущение полной магии происходящего, сегодня мы возьмем приложение из предыдущей статьи и разберем, как и что происходит внутри фреймворка и от каких проблем нас отгораживает Boot. Целевая аудитория все же начинающие разработчики, но с некоторым опытом и базовыми знаниями Java и Spring. Хотя, возможно, и опытным пользователям Spring будет интересно освежить знания того, что происходит под капотом.

Ключевые понятия

Бины

Начнем срывать покровы с самых базовых понятий Spring. **Бин (bean)** — это не что иное, как самый обычный объект. Разница лишь в том, что бинами принято называть те объекты, которые управляются Spring-ом и живут внутри его DI-контейнера. Бином является почти все в Spring — сервисы, контроллеры, репозитории, по сути все приложение состоит из набора бинов. Их можно регистрировать, получать в качестве зависимостей, проксировать, мокать и т.п.

DI контейнер

Ключевой и фундаментальный механизм Spring. Внешне очень простой, но внутри он предоставляет очень много механизмов для тонкой настройки зависимостей.

По сути, любое приложение Спринг — это набор бинов, связанных вместе через DI контейнер.

Очень часто при обсуждении Spring звучит аргумент, что его можно легко заменить на любой леговесный DI контейнер (Guice, например) и получить то же самое, но легче и проще. И здесь очень важно понять — ценность Spring DI не в самом факте его наличия, а в его фундаментальности. Все библиотеки в экосистеме Spring, по сути, просто регистрируют свои бины в этом контейнере (включая и сам Spring) — и через иньекцию зависимостей разработчики приложения смогут получить нужные компоненты. Простой пример: при использовании **Spring Security OAuth** если сконфигурить параметры OAuth в application.properties, то Spring Security предоставит бин OAuth2RestTemplate который мы можем просто заинжектить в своем коде. И этот бин при обращении к внешнему API будет знать, куда и как пойти, чтобы получить OAuth токен, как его обновлять, в какое место нашего запроса его добавлять и т.п. Так вот ценность DI тут в том, что это просто механизм общения между нашим кодом и Spring Security. И простой заменой реализации DI на Guice не добиться, чтобы Spring Security тоже начал его использовать. А если в этом новом DI не будет интеграции со всеми библиотеками Spring-а, то и ценность его сильно падает.

Еще один очень важный момент, который многие упускают при обсуждении DI контейнера, это то, что использование инъекции зависимостей не подразумевает создания интерфейсов для **каждого** компонента. Это очень простая мысль, но я много раз видел, что из-за своей простоты она не всегда очевидна. Более того, создание интерфейса, если у него лишь одна реализация — считается плохой практикой. Т.е. классы вполне могут сами по себе быть объектами DI. Более того, отсутствие интерфейса даже не мешает их мокать в тестах, т.к. Mockito, например, вполне умеет мокать классы.

Контекст

Представлен интерфейсом ApplicationContext. По сути, представляет собой само приложение Spring.

Контекст предоставляет возможности реагировать на различные события, которые происходят внутри приложения, управлять жизненным циклом бинов (создавать как синглтон или на каждый запрос, например).

Конфигурация

Итак, если приложение — это набор бинов, чтобы оно заработало нам нужно этот набор описать.

**Конфигурация** — это просто описание доступных бинов. Spring дает несколько вариантов, как можно описать набор бинов, которые сформируют приложение. Исторический вариант — это через набор **xml файлов**.

В наши дни на смену xml файлу пришли **Java аннотации**. Spring Boot полностью построен на аннтациях и большинство современных библиотек в принципе тоже можно сконфигурить через аннотации.

В третьем своем поколении, конфигурация бинов пришла к подходу функциональной регистрации (**functional bean registration**), которая станет одной из важных новых фич готовящегося к выходу Spring 5.

Типичный класс конфигурации может, выглядеть, например так:

**@Configuration**

class PaymentsServiceConfiguration {

**@Bean**

public PaymentProvider paymentProvider() {

return new PayPalPaymentProvider();

}

**@Bean**

public PaymentService paymentService(PaymentProvider paymentProvider) {

return new PaymentService(paymentProvider);

}

}

Эта конфигурация определяет два бина, причем второй зависит от первого. И здесь в игру вступит Spring – когда мы просим предоставить инстанс PaymentProvider — Spring найдет его в контексте и предоставит нам.

Конфигурацию не обязательно описывать в одном огромном файле, можно разбить на несколько и объединять их с помощью **@Import** аннотаций.

Сканирование компонентов

Достаточно важный компонент Spring Framework, еще один подход к упрощению конфигурации приложения. Идея очень простая — если мы знаем, что наш класс MyCoolComponent должен регистрировать бин с именем myCoolComponent, зачем каждый раз писать:

**@Bean**

MyCoolComponent myCoolComponent(dependencies...) {

return new MyCoolComponent(dependencies...);

}?

Почему просто не дать Spring–у автоматом зарегистрировать и создать бин на основании нужного класса? Эту задачу и решает сканирование компонентов. Т.е. если мы объявим наш класс как

**@Component**

class MyCoolComponent {

MyCoolComponent(dependencies...) {

}

}

и разрешим сканирование компонентов — то Spring сам создаст и зарегистрирует бин с именем myCoolComponent, использовав конструктор класса и заинжектив туда все зависимости.

Со сканированием компонентов надо быть осторожным, т.к. по сути оно неявно меняет контекст приложения. Например, если у нас есть интерфейс и две реализации — и на каждом указан @Component, то при попытке заинжектить зависимость на интерфейс Spring бросит исключение, что есть два бина, которые удовлетворяют запросу.

Резюме

Итак, вещи которые нужно запомнить: приложение Spring, описанное интерфейсом ApplicationContext, представляет собой набор объектов (*бинов*), управляемых DI контейнером.

Конфигурация набора бинов осуществляется с помощью:

- классов конфигурации (аннотация @Configuration), которые могут быть комбинированы с помощью импортов (аннотация @Import),

- аннотации @Component при объявлении класса, при этом разрешив сканирование.

Spring Boot

Теперь переходим к следующей части. Допустим, нам надо сконфигурить подключение к MySQL базе данных. Если мы хотим использовать **Spring Data JPA** с **Hibernate** в качестве провайдера, нам потребуется сконфигурировать несколько бинов — EntityManagerFactory (основной класс JPA), DataSource для подключения непосредственно к базе через JDBC драйвер и т.п. Но с другой стороны, если мы это делаем каждый раз и, по сути, делаем одно и то же — почему бы это не автоматизировать? Скажем, если мы указали строку подключения к базе и добавили зависимость на MySQL драйвер — почему бы *чему-то* автоматически не создать все нужные бины для работы с MySQL? Именно это и делает **Spring Boot**. По сути, **Spring Boot** это просто набор классов конфигурации, которые создают нужные бины в контексте. Точно так же их можно создать руками, просто Boot это автоматизирует.

Автоконфигурация

Важное понятие **Spring Boot** это автоконфигурация. По сути, это просто набор конфигурационных классов, которые создают и регистрируют определенные бины в приложении. По большому счету, даже сам **Embedded Servlet Container** — это просто еще один бин, который можно сконфигурировать! Пара важных моментов, которые важно знать об автоконфигурации:

* Включается аннотацией @EnableAutoConfiguration
* Работает в *последнюю очередь*, после регистрации пользовательских бинов
* Принимает решения о конфигурации на основании доступных в classpath классов, свойств в application.properties и т.п.
* Можно включать и выключать разные аспекты автоконфигурации, и применять ее частично (например, только MySQL + JPA, но не веб)
* Всегда отдает приоритет пользовательским бинам. Если ваш код уже зарегистрировал бин DataSource — автоконфигурация не будет его перекрывать

Условия и порядок регистрации бинов

Логика при регистрации бинов управляется набором @ConditionalOn\* аннотаций. Можно указать, чтобы бин создавался при наличии класса в classpath (@ConditionalOnClass), наличии существующего бина (@ConditionalOnBean), отсуствии бина (@ConditionalOnMissingBean) и т.п.

**Spring Boot** активно использует эти аннотации чтобы оставаться как можно более незаметным и не перекрывать пользовательские конфигурации.

Погружение в Hello World

Теперь, имея в запасе базовые теоретические знания, разберем что же происходит при запуске приложения.

Итак, наше приложение включает такой код:

@SpringBootApplication

public class DemoApplication {

public static void main(String[] args) {

SpringApplication.run(DemoApplication.class, args);

}

}

Давайте разберем что здесь происходит по шагам.

Класс DemoApplication

Этот класс помечен аннотацией @SpringBootApplication, что является **мета-аннотацией**, т.е. по сути, является алиасом для нескольких аннотаций:

* @SpringBootConfiguration
* @EnableAutoConfiguration
* @ComponentScan.

Т.е. наличие @SpringBootApplication включает сканирование компонентов, автоконфигурацию и показывает разным компонентам Spring (например, интеграционным тестам), что это **Spring Boot** приложение.

SpringApplication.run()

Это просто хелпер, который делает пару вещей — используя список предоставленных конфигураций (а класс DemoApplication сам по себе конфигурация, см. выше) создает ApplicationContext, конфигурирует его, выводит баннер в консоли и засекает время старта приложения и т.п.

Его можно заменить на ручное создание контекста: new AnnotationConfigApplicationContext(DemoApplication.class). Как можно понять из названия, это контекст приложения, который конфигурируется с помощью аннотаций. Однако, этот контекст не знает ничего об embedded servlet container-ах, и совершенно точно не умеет себя запускать. Его наследник, уже из **Spring Boot** – AnnotationConfig*EmbeddedWeb*ApplicationContext делать это умеет, и если мы в методе main  напишем

@SpringBootApplication

public class DemoApplication {

public static void main(String[] args) throws InterruptedException {

ApplicationContext applicationContext =

new AnnotationConfigEmbeddedWebApplicationContext(DemoApplication.class);

}

}

,то получим точно такое же работающее приложение, т.к. класс AnnotationConfigEmbeddedWebApplicationContext найдет в контексте бин типа EmbeddedServletContainerFactory и через него создаст и запустит встроенный контейнер. Обратите внимание, что все это работает в рамках общего DI контейнера, то есть этот класс можно реализовать самим.

@EnableAutoConfiguration

Эта аннотация включает автоконфигурацию. И здесь, пожалуй, ключевой момент в развенчании магии Spring. Вот как объявлена эта аннотация:

...

@Import(EnableAutoConfigurationImportSelector.class)

public @interface EnableAutoConfiguration {

...

}

Т.е. это самый обычный *импорт* конфигурации, про который мы говорили выше. Класс же EnableAutoConfigurationImportSelector (и его преемник в Spring Boot 1.5+ — AutoConfigurationImportSelector) это просто конфигурация, которая добавит несколько бинов в контекст. Однако, у этого класса есть одна тонкость — он не объявляет бины сам, а использует так называемые *фабрики*.

Класс EnableAutoConfigurationImportSelector смотрит в файл spring.factories и загружает оттуда список значений, которые являются *именами классов (авто)конфигураций, которые Spring Boot импортирует*.

Кусочек файла spring.factories (он находится в папке META-INF внутри spring-boot-autoconfigure.<version>.jar), который нам сейчас нужен это:

org.springframework.boot.autoconfigure.EnableAutoConfiguration=\

org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfiguration,\

... (100 lines)

org.springframework.boot.autoconfigure.webservices.WebServicesAutoConfiguration

Т.е. аннотация @EnableAutoConfiguration просто импортирует все перечисленные конфигурации, чтобы предоставить нужные бины в контекст приложения.

По сути, ее можно заменить на ручной импорт нужных конфигураций:

@Import({

org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfiguration.class,

org.springframework.boot.autoconfigure.aop.AopAutoConfiguration.class,

...})

public class DemoApplication {

...

}

Однако, особенность в том, что Spring Boot пытается применить **все** конфигурации (а их около сотни). Я думаю, у внимательного читателя уже появилась пара вопросов, которые стоит прояснить.

* "Но это же медленно!". И да, и нет — под рукой нет точных цифр, но сам по себе процесс автоконфигурации очень быстрый (порядка сотни миллисекунд на абстрактной машине в вакууме)
* "Но это же излишне, зачем мне конфигурить Rabbit (RabbitAutoConfiguration) если я его не использую?". Наличие автоконфигурции не значит, что бин будет создан. Автоконфигурационные классы активно используют @ConditionalOnClass аннотации, и в большинстве случаев конфигурация ничего делать и создавать не будет (см. выше "Условия и порядок регистрации бинов").

Краткое резюме

В основе "магии" Spring Boot нет ничего магического, он использует совершенно базовые понятия из Spring Framework. В кратком виде процесс можно описать так:

1. Аннотация @SpringBootApplication включает сканирование компонентов и авто-конфигурацию через аннотацию @EnableAutoConfiguration
2. @EnableAutoConfiguration импортирует класс EnableAutoConfigurationImportSelector
3. EnableAutoConfigurationImportSelector загружает список конфигураций из файла META-INF/spring.factories
4. Каждая конфигурация пытается сконфигурить различные аспекты приложения (web, JPA, AMQP etc), регистрируя нужные бины и используя различные условия (наличие / отсутствие бина, настройки, класса и т.п.)
5. Созданный в итоге AnnotationConfigEmbeddedWebApplicationContext ищет в том же DI контейнере фабрику для запуска embedded servlet container
6. Servlet container запускается, приложение готово к работе!

Это может выглядеть сложно, но по большей части разработчикам приложений влезать во внутренности автоконфигурации не нужно, если речь не идет о поддержке авто-конфигурации для своей библиотеке.

Диагностика

Auto-configuration report

В случае, когда что-то идет не так, Spring Boot позволяет запустить диагностику автоконфигурации и посмотреть, какие именно бины были созданы. Чтобы увидеть эту информацию, нужно запустить приложение с ключом --debug.

java -jar my-app.jar --debug

В ответ Spring выдаст детальный *Auto-configuration report*:

=========================

AUTO-CONFIGURATION REPORT

=========================

Positive matches:

-----------------

DataSourceAutoConfiguration matched:

- @ConditionalOnClass found required classes 'javax.sql.DataSource', 'org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType'; @ConditionalOnMissingClass did not find unwanted class (OnClassCondition)

DataSourceAutoConfiguration#dataSourceInitializer matched:

- @ConditionalOnMissingBean (types: org.springframework.boot.autoconfigure.jdbc.DataSourceInitializer; SearchStrategy: all) did not find any beans (OnBeanCondition)

...

Negative matches:

-----------------

ActiveMQAutoConfiguration:

Did not match:

- @ConditionalOnClass did not find required classes 'javax.jms.ConnectionFactory', 'org.apache.activemq.ActiveMQConnectionFactory' (OnClassCondition)

...

Строчка в Positive / Negative matches будет *для каждой примененной автоконфигурации*, более того, Boot сообщит, почему тот или иной бин был создан (т.е. укажет, какие из условий регистрации были выполнены).

Actuator

Spring Boot Actuator это мощный инструмент диагностики работающего приложения, который умеет давать много полезной аналитики (более того, набор этих метрик можно легко расширять из приложения).

После добавления Actuator к проекту, **Spring Boot** опубликует список доступных бинов через URL http://localhost:8080/beans. Этот список так же доступен через JMX (Java Management Extensions), и последняя версия Intellij IDEA умеет показывать все бины приложения прямо из окна запуска.

Резюме

Spring все же остается большим и не самым простым фреймворком, но это цена высокоуровневых абстракций, которые он предоставляет. И хотя знать все тонкости работы фреймворка в ежедневной разработке не нужно, знать, как он работает изнутри, все же, полезно. Надеюсь, что эта статья помогла понять важность и ценность Spring именно как экосистемы и убрала немного "магичности" в происходящем, особенно при использовании Spring Boot. Мой совет — не бойтесь углубляться в недра фреймворка, читайте исходники и документацию, благо они у Spring-a почти эталонные, на мой взгляд.

Так же стоит отметить, что в готовящемся к выходу в сентябре **Spring 5** появится несколько новых концепций, направленных на создание простых приложений, и понижение уровня "магии" (хотя, как мы выяснили, магии там особо и нет). Одна из концепций это **Functional Bean Registration**, которая позволяет регистрировать бины в контексте с помощью функций, или даже с помощью неплохого DSL на Kotlin (а Spring 5 добавит много хорошего для поддержки Kotlin). Следующая, но еще более важная вещь, это комбинация **Functional Web Framework** и **WebFlux (reactive web framework)**, которая позволит создавать веб-приложения вообще без зависимости на Spring MVC и запускать их без сервлет контейнеров. Приложение вполне сможет работать без контекста приложений и DI, и описываться просто как набор функций request -> response. Об этом можно чуть больше почитать [здесь](http://blog.alexnesterov.com/post/spring-your-next-microframework/) (на английском).