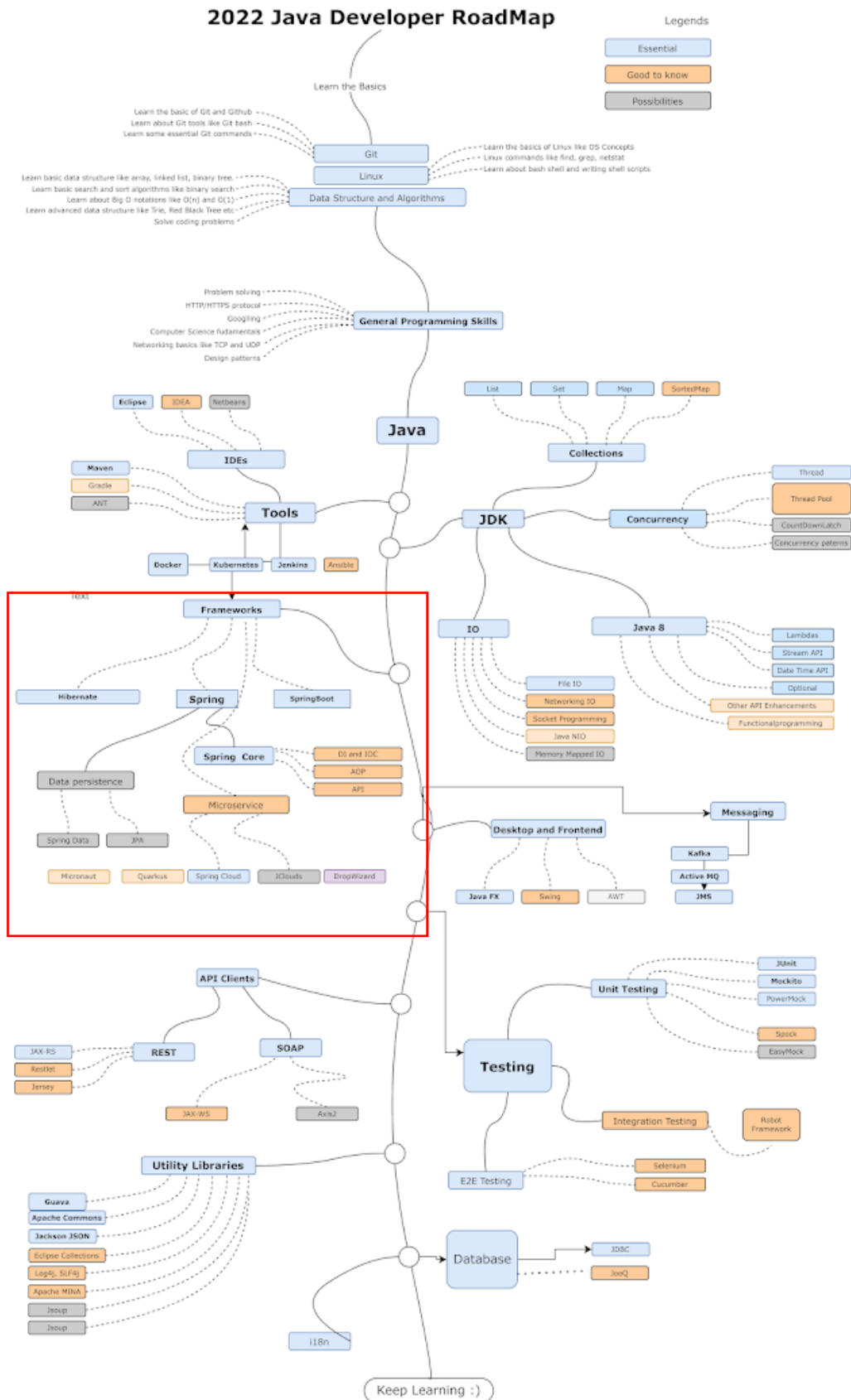


# №1 Base Spring technologies.

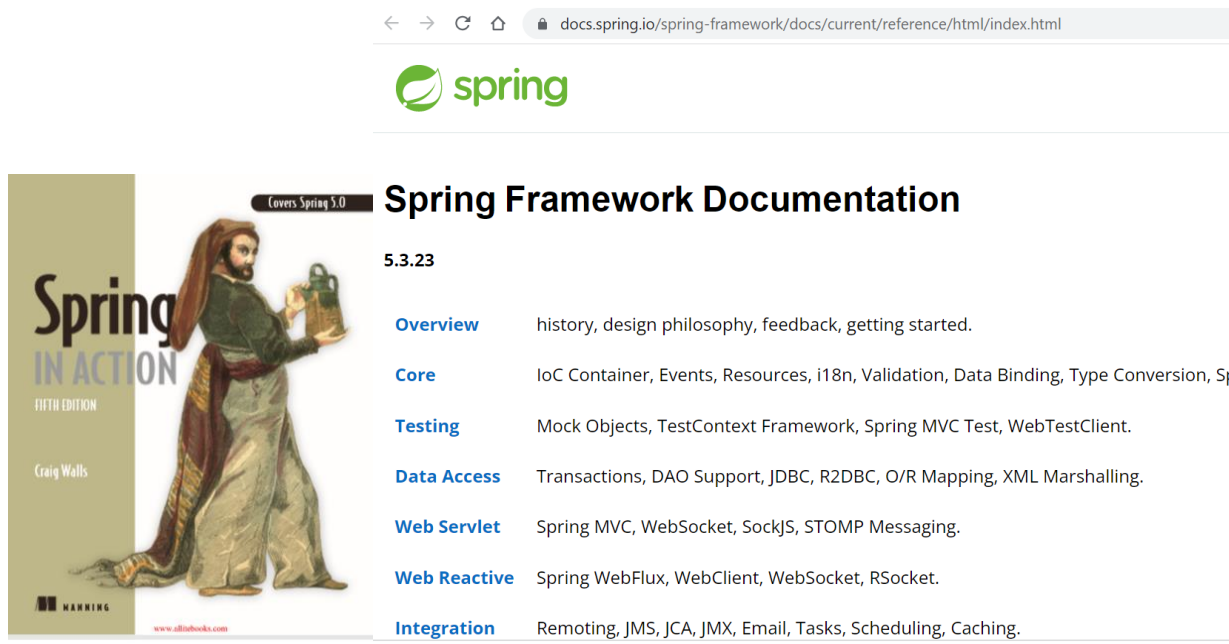
Базовые технологии Spring. Концепция Spring Boot. IoC контейнер. DI, CDI.




<https://spring.io/>

<https://www.manning.com/books/spring-in-action-fourth-edition>

<https://docs.spring.io/spring/docs/current/spring-framework-reference/index.html>



← → ↺ 🏠 docs.spring.io/spring-framework/docs/current/reference/html/index.html



**Covers Spring 5.0** **Spring Framework Documentation**

5.3.23

|                     |   |
|---------------------|---|
| <b>Overview</b>     | history, design philosophy, feedback, getting started.                                |
| <b>Core</b>         | IoC Container, Events, Resources, i18n, Validation, Data Binding, Type Conversion, Sp |
| <b>Testing</b>      | Mock Objects, TestContext Framework, Spring MVC Test, WebTestClient.                  |
| <b>Data Access</b>  | Transactions, DAO Support, JDBC, R2DBC, O/R Mapping, XML Marshalling.                 |
| <b>Web Servlet</b>  | Spring MVC, WebSocket, SockJS, STOMP Messaging.                                       |
| <b>Web Reactive</b> | Spring WebFlux, WebClient, WebSocket, RSocket.  |
| <b>Integration</b>  | Remoting, JMS, JCA, JMX, Email, Tasks, Scheduling, Caching.                           |

**Spring Framework** - облегченная платформа для построения Java-приложений.

### *История Spring*

**Spring** был разработан в июне 2003 года Родом Джонсоном как ответ на сложность ранних спецификаций Java EE под лицензией Apache 2.0 license.

Текущая версия — 5.3.x.

<https://docs.spring.io/spring/docs/current/spring-framework-reference/>

Spring дополняет Java EE. Модель программирования Spring он интегрируется с EE:

Servlet API ([JSR 340](#))

WebSocket API ([JSR 356](#))

Concurrency Utilities ([JSR 236](#))

JSON Binding API ([JSR 367](#))

Bean Validation ([JSR 303](#))

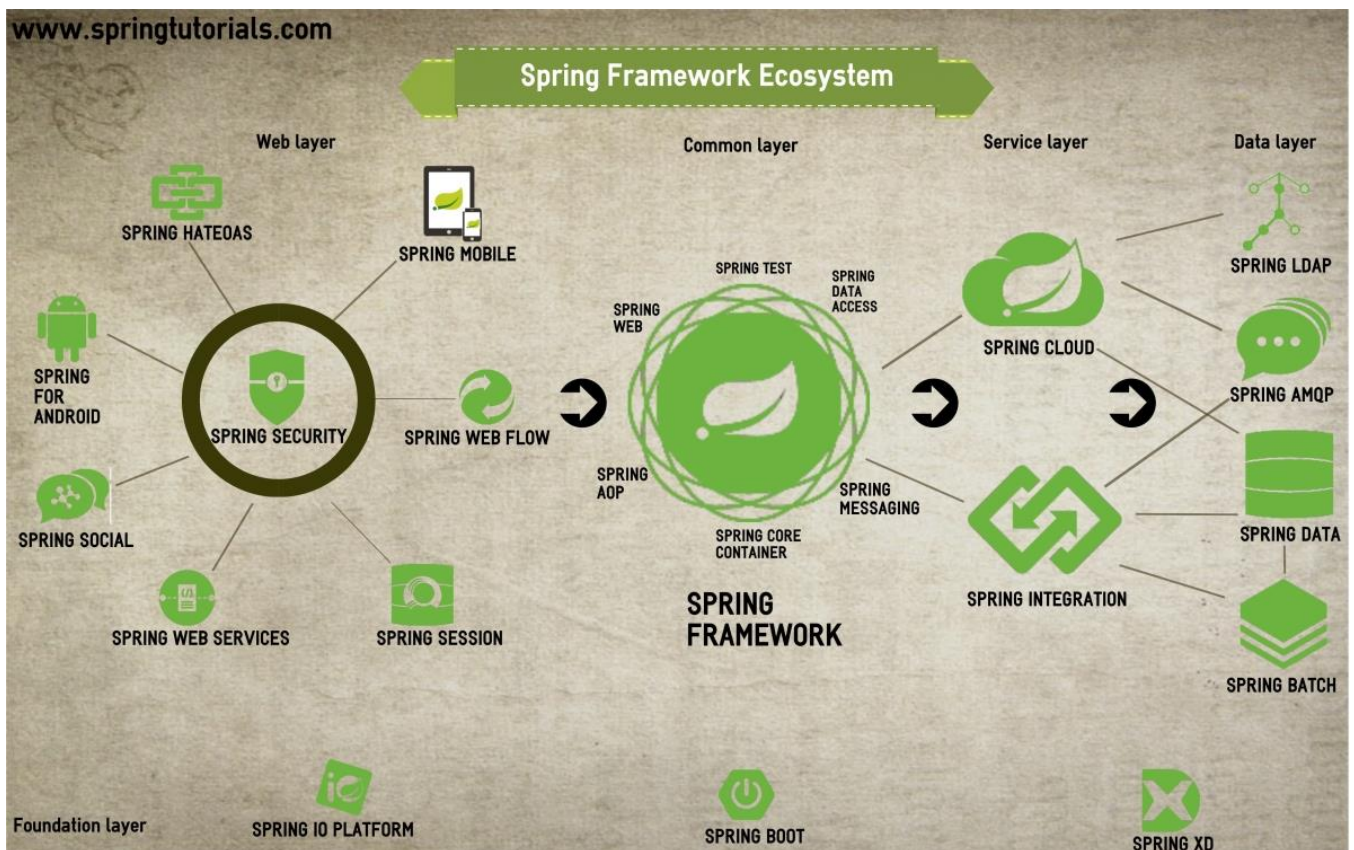
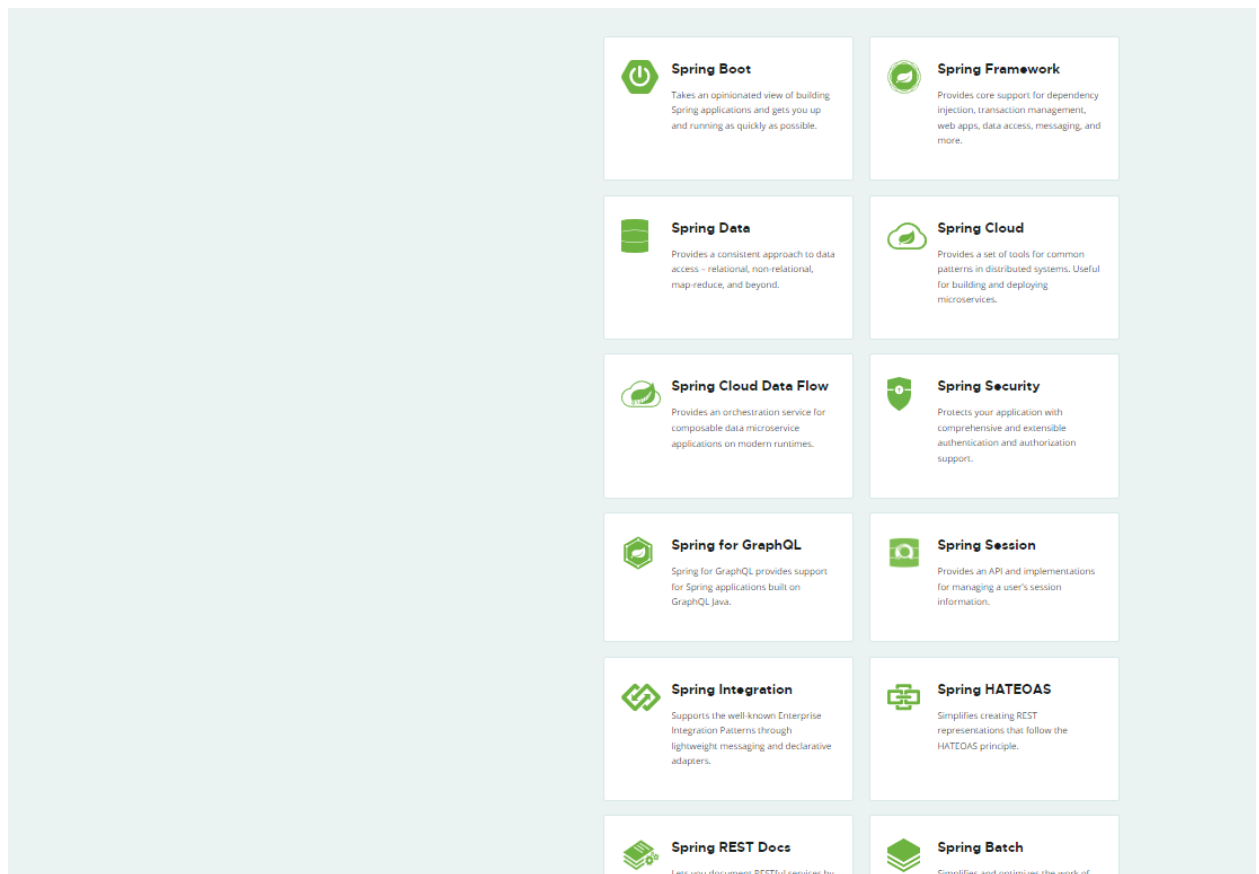
JPA ([JSR 338](#))

JMS ([JSR 914](#))

Dependency Injection ([JSR 330](#))

Common Annotations ([JSR 250](#))

## Spring Экосистема



## Spring Projects на веб-уровне

Следующие проекты Spring так или иначе помогут вам в разработке интерфейса.

**Spring Session** позволяет обрабатывать веб-сессии в вашем приложении.

**Spring Web Services** помогает вам создать сначала контрактный веб-сервис на основе SOAP.

**Spring Social** позволяет Facebook, LinkedIn, Twitter и другим пользователям заходить на сайт.

**Spring для Android** помогает создавать нативные приложения для Android.

**Spring HATEOAS** помогает создавать представления REST, которые следуют принципу HATEOAS.

**Spring Mobile** помогает создавать страницы, удобные для мобильных устройств., если ваши запросы поступают с мобильного устройства.

**Spring Web Flow** разбивает данные на множество экранов для удобной навигации.

**Spring Security** в центре всех этих проектов по причине все проекты веб-слоя могут использовать Spring Security.

### Common layer

**Spring Framework** (или Spring Core) Ядро платформы, предоставляет базовые средства для создания приложений — управление компонентами (бинами, **beans**), внедрение зависимостей, MVC фреймворк, транзакции, базовый доступ к БД. В основном это низкоуровневые компоненты и абстракции. По сути, неявно используется всеми другими компонентами.

### Service layer Projects

**Spring Integration** - подходит для обработки событий. Вы можете подключить его поверх любых проектов Spring Data layer.

Это как обработка данных из разных источников. Если надо раз в час брать файл с FTP, разбивать его на строки, которые потом фильтровать, а дальше отправлять в какую-то очередь — это к Spring Integration.

**Spring Cloud.** - это инструмент для создания распределенных приложений

Много полезного для микросервисной архитектуры — service discovery, трасировка и диагностика, балансировщики запросов, circuit breaker-ы, роутеры и т.п

### Data layer Projects

Проекты в этом слое имеют дело с обработкой данных.

**Spring AMQP** обобщает работу с отправкой и получением сообщений. Он предоставляет низкоуровневый шаблон для взаимодействия с данными. Это обеспечивает фабрику соединений. Он обеспечивает готовую реализацию в RabbitMQ.

**Spring Data** - общий API для выполнения операций CRUD для разных баз данных. Он поддерживает все популярные базы данных SQL, а также базы данных NoSQL.

**Spring Batch** - это фреймворк с множеством функций. Используется для выполнения любой массовой обработки.

**Spring IO Platform** - позволяет легко вводить то, что вам нужно для вашего приложения

**Spring Boot** - дает вам инструмент командной строки для создания приложения. Позволяет избежать XML конфигурации. Boot позволяет быстро создать и сконфигурировать (т.е. настроить зависимости между компонентами) приложение, упаковать его в исполняемый самодостаточный артефакт. Это то связующее звено, которое объединяет вместе набор компонентов в готовое приложение.

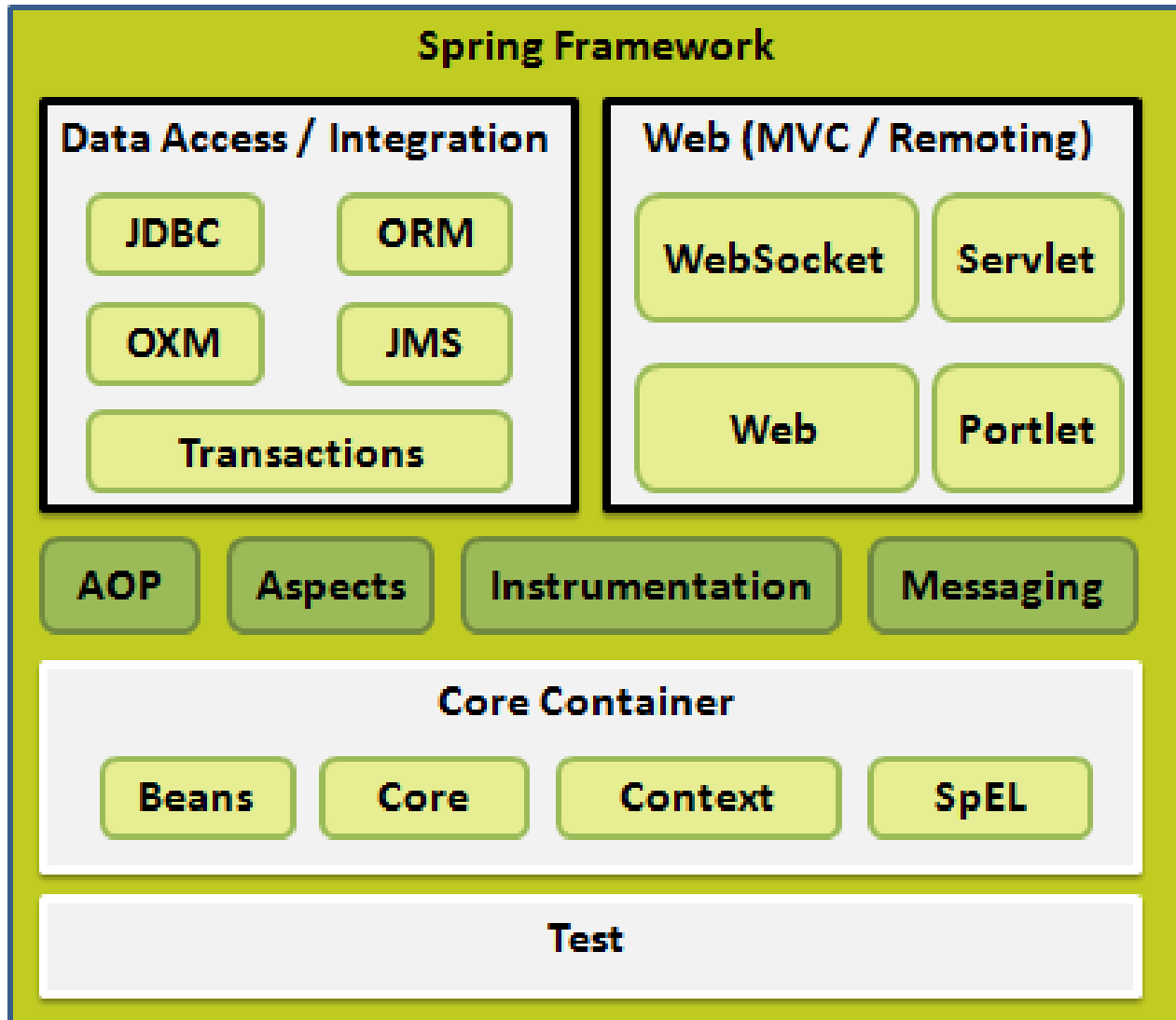
Spring Boot:

- Он не использует кодогенерацию. Из кода, который генерится, присутствует только метод `main`.
- Не использует XML для конфигурации. Все конфигурируется через аннотации
- Используются автоконфигурации по максимуму. Используется *convention over configuration*. Для большинства конфигураций не нужно ничего настраивать.
- Его легко отодвинуть в сторону и "перекрыть" конфигурацию по умолчанию.

Стоит отдельно отметить компонент Spring Boot который называется *DevTools*. Он решает проблему цикла локальной разработки.

**Spring XD** позволяет создавать приложения для больших данных и управлять ими. Помогает вам выполнять анализ больших данных в этих приложениях. Например, использовать Nadoor или другие технологии с минимальными усилиями.

# Spring Framework



На сегодняшний день Spring разделён на некоторое количество отдельных модулей, что позволяет самим решать, какие из них использовать в приложении.

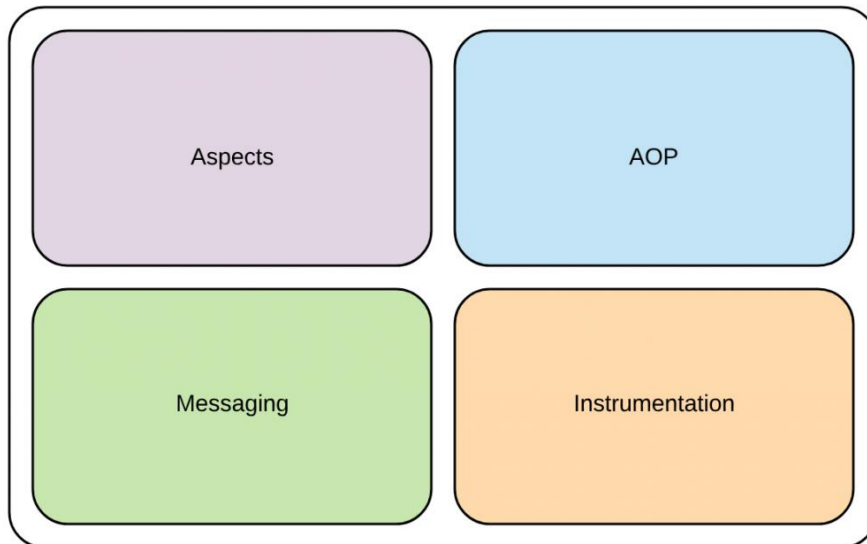
**Основной контейнер (Core Container)** включает в себя Beans, Core, Context и SpEL (expression language).

**AOP** реализует аспекто-ориентированное программирование. Предоставляет элементы для декларирования аспектов, и для автоматического проксирования.

**Модуль Aspects** обеспечивает интеграцию с AspectJ, которая также является фреймворком АОП.

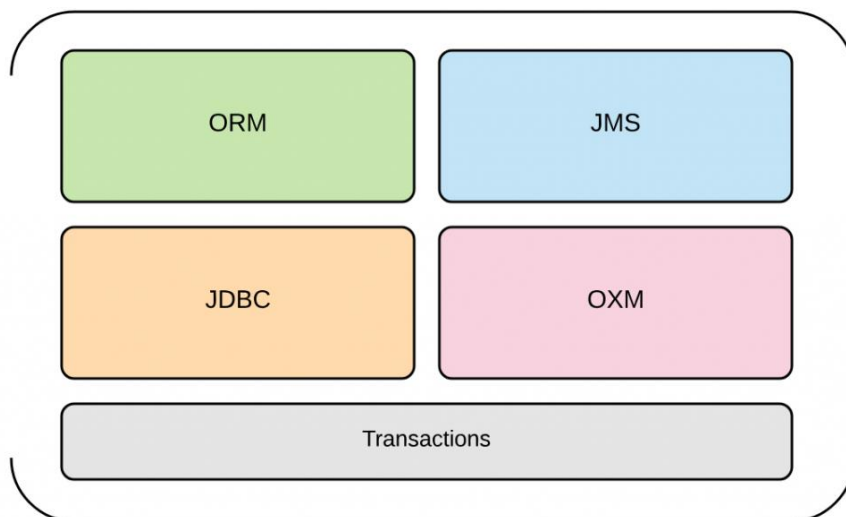
**Instrumentation** отвечает за поддержку class instrumentation и class loader, которые используются в серверных приложениях.

Модуль **Messaging** обеспечивает поддержку STOMP.



**Test** обеспечивает тестирование с использованием TestNG или JUnit Framework.

Контейнер **Data Access/Integration** состоит из JDBC, ORM, OXM, JMS и модуля Transactions.



**JDBC** обеспечивает абстрактный слой JDBC и избавляет разработчика от необходимости вручную прописывать код, связанный с соединением с БД.

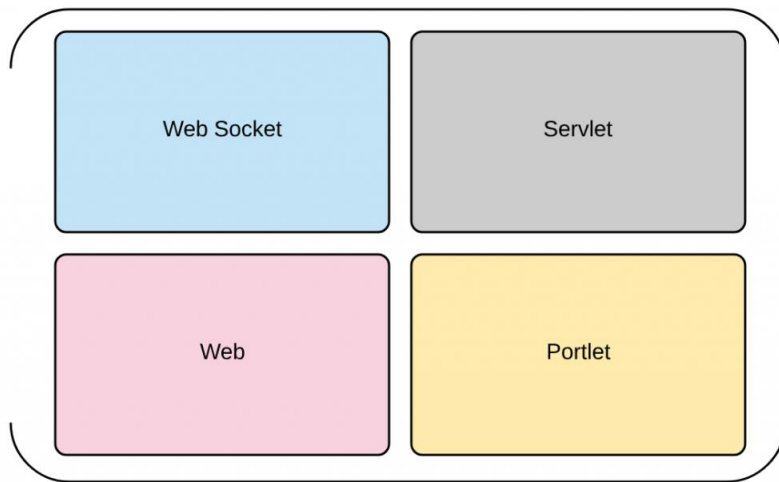
**ORM** обеспечивает интеграцию с такими популярными ORM, как Hibernate, JDO, JPA и т.д.

**Модуль OXM** отвечает за связь Объект/XML – XMLBeans, JAXB и т.д.

**Модуль JMS** (Java Messaging Service) отвечает за создание, передачу и получение сообщений.

**Transactions** поддерживает управление транзакциями для классов, которые реализуют определённые методы.

**Web** слой состоит из Web, Web-MVC, Web-Socket, Web-Portlet



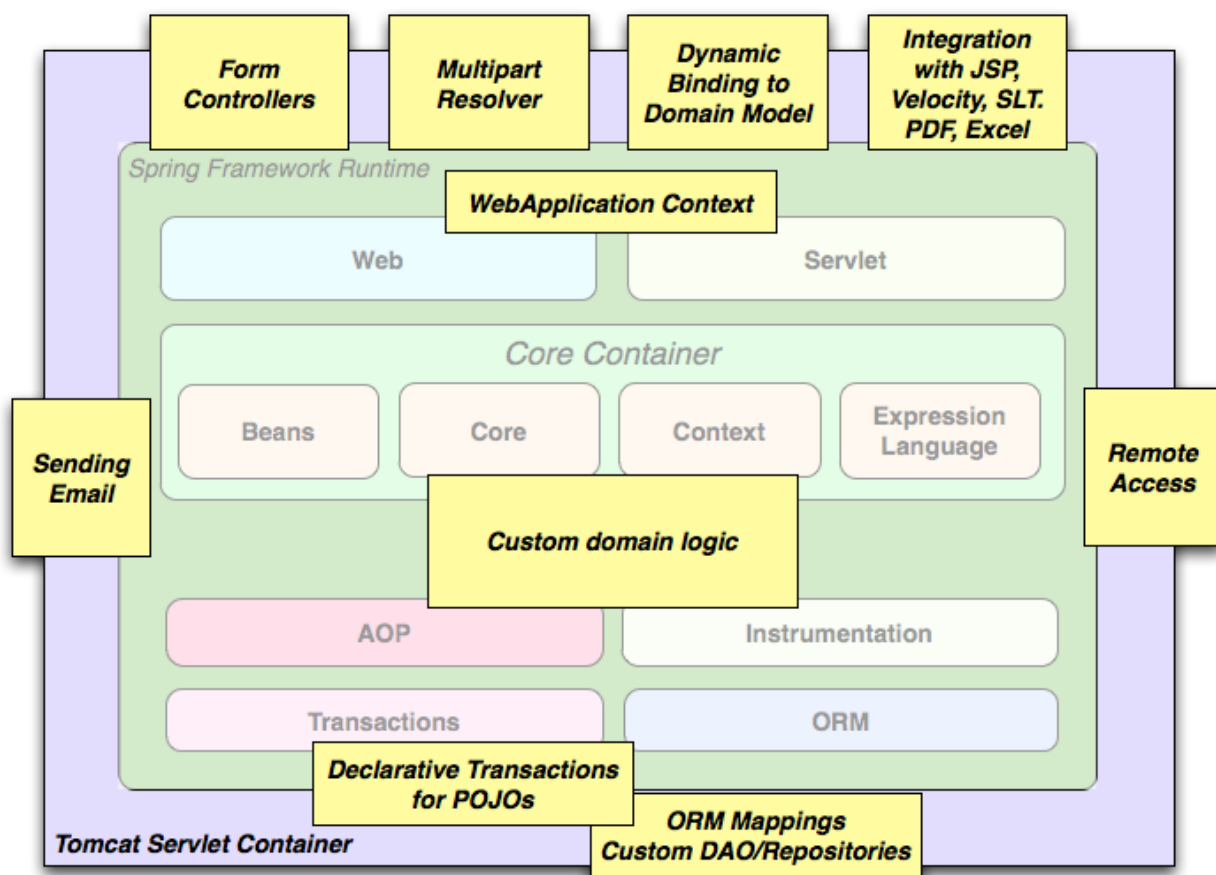
**Web-MVC** содержит реализацию Spring MVC для веб-приложений.

**Spring MVC** стоит упомянуть отдельно, т.к. мы будем вести речь в основном о веб-приложениях.

**Web-Socket** обеспечивает поддержку связи между клиентом и сервером, используя Web-Socket-ы в веб-приложениях.

**Web-Portlet** обеспечивает реализацию MVC в среде портлетов.

*Типичное веб приложение скорее всего будет включать набор вроде Spring MVC, Data, Security.*





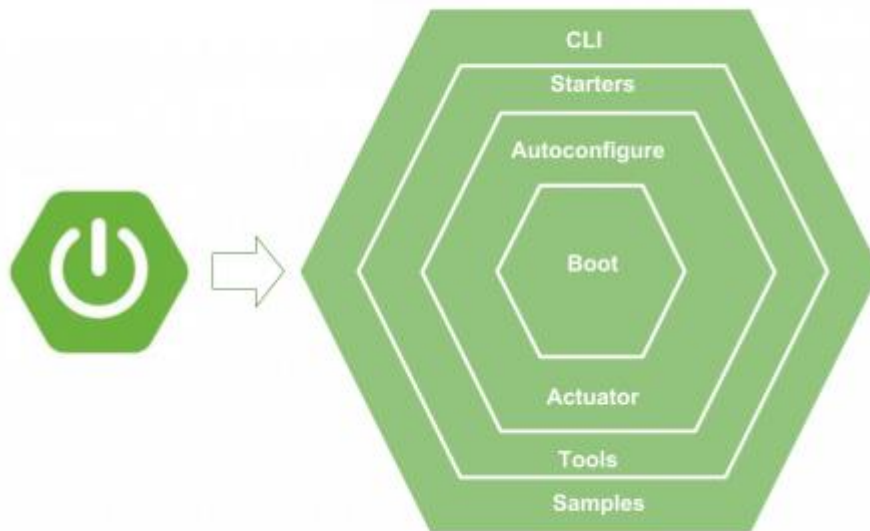
# Spring Boot

Текущая версия 2.7.4-SNAPSHOT

<https://spring.io/projects/spring-boot>

Boot позволяет быстро создать и сконфигурировать (т.е. настроить зависимости между компонентами) приложение, упаковать его в исполняемый самодостаточный артефакт. Это то связующее звено, которое объединяет вместе набор компонентов в готовое приложение. Не использует XML для конфигурации. Все конфигурируется через аннотации.

Чтобы ускорить процесс управления зависимостями, Spring Boot неявно упаковывает необходимые сторонние зависимости для каждого типа приложения на основе Spring и предоставляет их разработчику посредством так называемых **starter**-пакетов (spring-boot-starter-web, spring-boot-starter-data-jpa и т.д.)



**Starter-пакеты** представляют собой набор удобных дескрипторов зависимостей, которые можно включить в свое приложение.

```
<modelVersion>4.0.0</modelVersion>
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.2.0.BUILD-SNAPSHOT</version>
  <relativePath/> <!-- Lookup parent from repository -->
</parent>
<groupId>by.patsei</groupId>
<artifactId>SpringProject1</artifactId>
<version>0.0.1-SNAPSHOT</version>
<packaging>war</packaging>
<name>SpringProject1</name>
<description>Demo project for Spring Boot</description>

<properties>
  <java.version>11</java.version>
</properties>

<dependencies>
  <dependency>
```

```

    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-mustache</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-tomcat</artifactId>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
<!-- https://mvnrepository.com/artifact/org.projectlombok/lombok -->
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.8</version>
    <scope>provided</scope>
</dependency>

</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>

</project>

```

Второй возможностью Spring Boot является автоматическая конфигурация приложения. После выбора подходящего starter-пакета, Spring Boot попытается автоматически настроить Spring-приложение на основе добавленных вами **jar**-зависимостей

Автоматическая конфигурация может быть полностью переопределена в любой момент с помощью пользовательских настроек.

| Name                              | Description   |
|-----------------------------------|---|
| spring-boot-starter               | Core starter, including auto-configuration support, logging and YAML  |
| spring-boot-starter-activemq      | Starter for JMS messaging using Apache ActiveMQ   |
| spring-boot-starter-amqp          | Starter for using Spring AMQP and Rabbit MQ   |
| spring-boot-starter-aop           | Starter for aspect-oriented programming with Spring AOP and AspectJ   |
| spring-boot-starter-cache         | Starter for using Spring Framework's caching support  |
| spring-boot-starter-data-jdbc     | Starter for using Spring Data JDBC  |
| spring-boot-starter-data-jpa      | Starter for using Spring Data JPA with Hibernate  |
| spring-boot-starter-data-mongodb  | Starter for using MongoDB document-oriented database and Spring Data MongoDB  |
| spring-boot-starter-data-rest     | Starter for exposing Spring Data repositories over REST using Spring Data REST  |
| spring-boot-starter-integration   | Starter for using Spring Integration  |
| spring-boot-starter-json          | Starter for reading and writing json  |
| spring-boot-starter-oauth2-client | Starter for using Spring Security's OAuth2/OpenID Connect client features   |
| spring-boot-starter-security      | Starter for using Spring Security   |
| spring-boot-starter-test          | Starter for testing Spring Boot applications with libraries including JUnit, Hamcrest and Mockito                         |
| spring-boot-starter-validation    | Starter for using Java Bean Validation with Hibernate Validator   |
| spring-boot-starter-web           | Starter for building web, including RESTful, applications using Spring MVC. Uses Tomcat as the default embedded container |

Поддержка сборки предоставляется для следующих инструментов сборки:

| Build Tool | Version   |
|------------|-----------|
| Maven      | 3.5+      |
| Gradle     | 7.x (7.5) |

Каждое Spring Boot web-приложение включает встроенный web-сервер. Ниже приведен список контейнеров сервлетов, которые поддерживаются "из коробки"

| Name                                | Servlet Version |
|-------------------------------------|-----------------|
| Tomcat 10.0                         | 5.0             |
| Jetty 11.0                          | 5.1             |
| Undertow 2.2 (Jakarta EE 9 variant) | 5.0             |

## Свойства Spring Boot

1) **SpringApplication**. Аннотация `@SpringBootApplication` неявно определяет базовый «пакет поиска» для определенных элементов

```
@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

```
com
+- example
  +- myapplication
    +- Application.java
    |
    +- customer
    |   +- Customer.java
    |   +- CustomerController.java
    |   +- CustomerService.java
    |   +- CustomerRepository.java
    |
    +- order
    |   +- Order.java
    |   +- OrderController.java
    |   +- OrderService.java
    |   +- OrderRepository.java
```

SpringApplication можно конфигурировать с XML, но обычно рекомендуется, чтобы основным источником конфигурации был один класс помеченный `@Configuration`. Обычно класс, который определяет метод `main`.

**@Import** - может использоваться для импорта дополнительных классов конфигурации.

**@ComponentScan** - автоматического подбора всех компонентов Spring, включая классы **@Configuration**.

**@ImportResource** используется для загрузки файлов конфигурации XML.

**@EnableAutoConfiguration** или **@SpringBootApplication** добавленная к одному из классов **@Configuration** - включает автоматическую настройку.

# Spring Beans and Dependency Injection

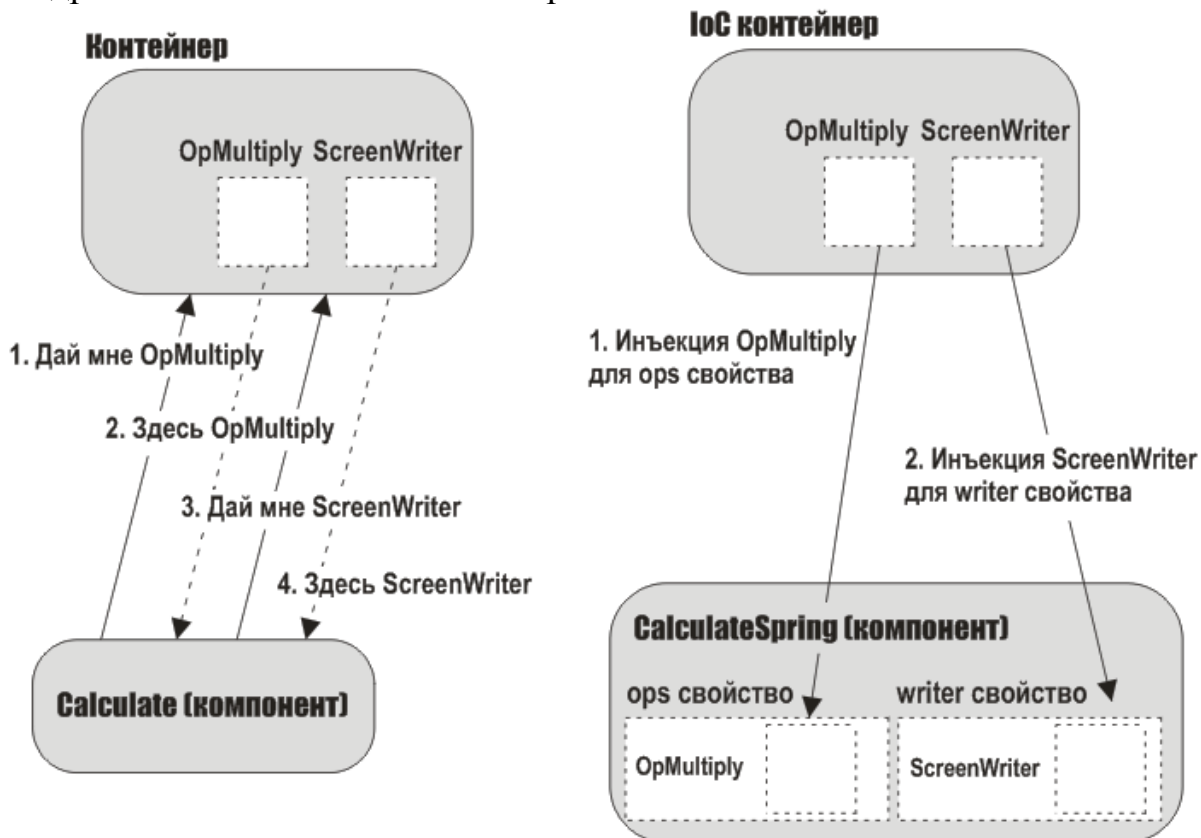
<https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#spring-core>

Ядро Spring Framework основано на принципе *инверсии управления* (Inversion of Control - IoC), когда создание и управление зависимостями между компонентами становятся внешними. По своей сути IoC и, следовательно, DI (Dependency Injection) направлены на то, чтобы предложить простой механизм для предоставления зависимостей компонента и управления этими зависимостями на протяжении всего их жизненного цикла.

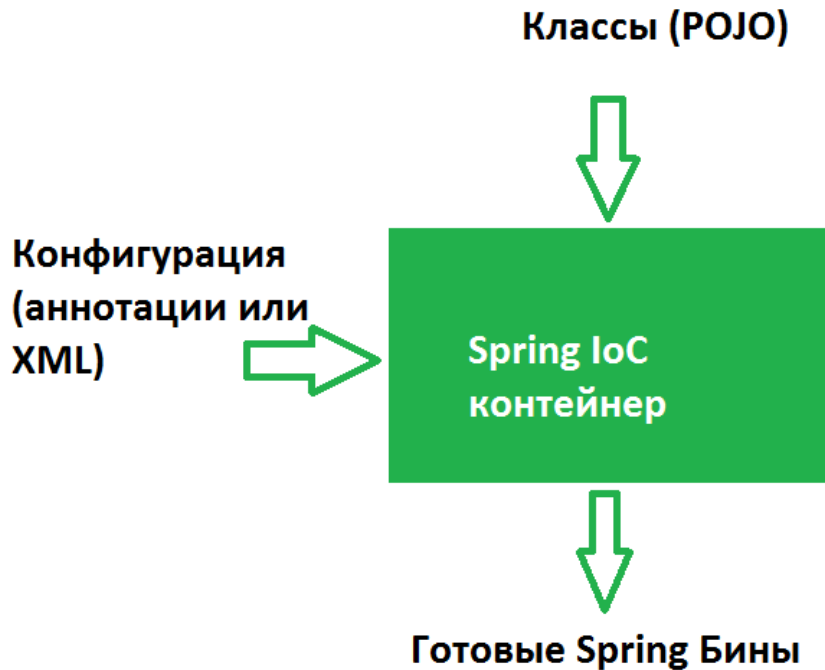
инверсия управления может быть разделена на два подтипа:

- 1) внедрение зависимостей (Dependency Injection) и
- 2) поиск зависимостей (Dependency Lookup).

Тип **Dependency Lookup** является намного более традиционным подходом и на первый взгляд выглядит более знакомым Java-программистам. Тип **Dependency Injection**, обеспечивает более высокую гибкость и удобство в использовании. В случае применения IoC в стиле Dependency Lookup компонент должен получить ссылку на зависимость, тогда как в стиле Dependency Injection зависимости внедряются в компонент контейнером IoC



IoC проиллюстрировать это можно так:



На вход контейнер Spring принимает: обычные классы (которые впоследствии будут бинами); конфигурацию (неважно как именно ее задавать – либо в специальном файле XML, либо с помощью специальных аннотаций).

А на выходе он производит **объекты – бины**. То есть экземпляры классов, созданные в соответствии с конфигурацией и внедренные куда нужно (в другие бины). После этого никакие операторы *new* нам не понадобятся, мы будем работать в классе-бине с его полями-бинами так, будто они уже инициированы. Конечно, не со всеми полями, а только с теми, которые сконфигурированы как бины. Остальные инициализируются как обычно, в том числе с помощью оператора *new*.

**JavaBeans** — классы в языке Java, написанные по определённым правилам. Чтобы класс мог работать как *bean*, он должен соответствовать определённым соглашениям об именах методов, конструкторе и поведении..

Правила описания гласят:

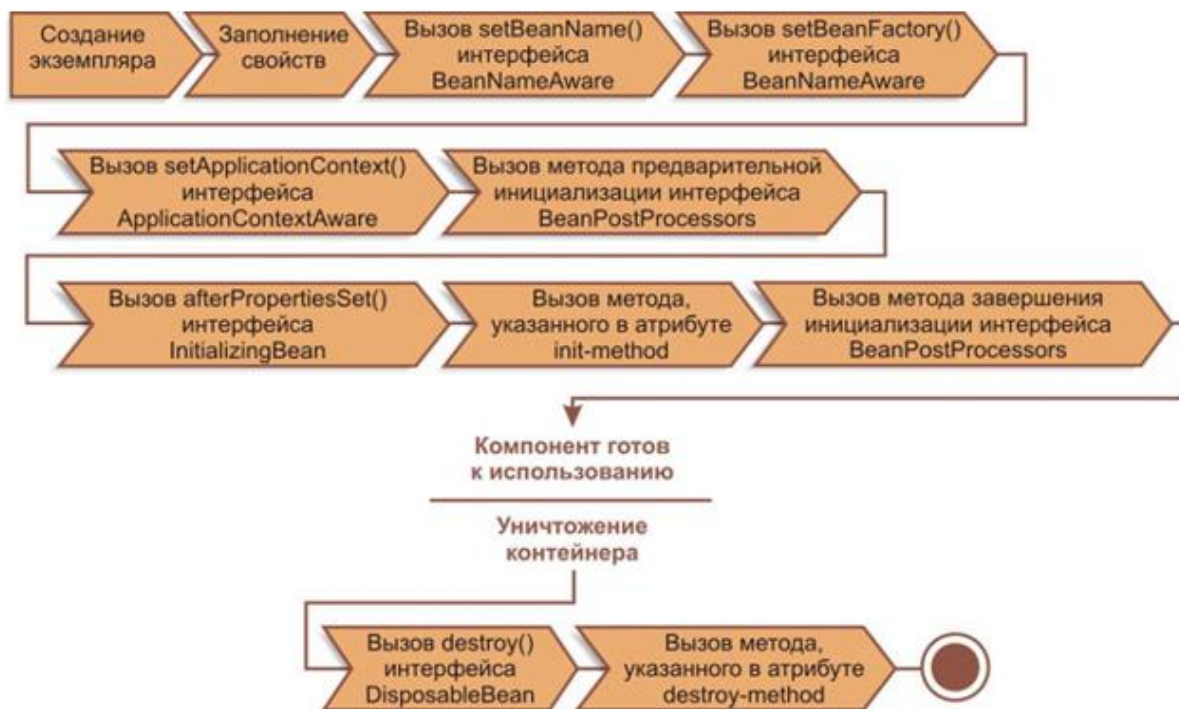
- Класс должен иметь конструктор без параметров, с модификатором доступа *public*. Такой конструктор позволяет инструментам создать объект без дополнительных сложностей с параметрами.
- Свойства класса должны быть доступны через *get*, *set* и другие методы (так называемые методы доступа), которые должны подчиняться стандартному соглашению об именах. Это легко позволяет инструментам автоматически определять и обновлять содержание *bean*’ов. Многие инструменты даже имеют специализированные редакторы для различных типов свойств.
- Класс должен быть сериализуем. Это даёт возможность надёжно сохранять, хранить и восстанавливать состояние *bean* независимым от платформы и виртуальной машины способом.
- Класс должен иметь переопределённые методы *equals()*, *hashCode()* и *toString()*.

Все компоненты вашего приложения: **@Component**, **@Service**, **@Repository**, **@Controller** и т.д.) автоматически регистрируются как Spring Beans.

### Жизненный цикл бина

Основными признаками конфигурации IoC контейнера являются классы с аннотацией **@Configuration** и методы с аннотацией **@Bean**. Аннотация **@Bean** используется для указания того, что метод создает, настраивает и инициализирует новый объект, управляемый Spring IoC контейнером.

```
@Configuration
public class LessonsConfiguration {
    @Bean
    GreetingService greetingService() {
        return new GreetingServiceImpl();
    }
}
```



Для управления контейнером жизненным циклом бина, вы можете реализовать метод `afterPropertiesSet()` интерфейса `InitializingBean` и метод `destroy()` интерфейса `DisposableBean`. Метод `afterPropertiesSet()` позволяет выполнять какие-либо действия после инициализации всех свойств бина контейнером, метод `destroy()` выполняется при уничтожении бина контейнером. Однако их не рекомендуется использовать, поскольку они дублируют код Spring. Как вариант, предпочтительно использовать методы с JSR-250 аннотациями **@PostConstruct** и **@PreDestroy**. Также существует вариант определить аналогичные методы как параметры аннотации **@Bean**, например так: **@Bean(initMethod = "initMethod", destroyMethod = "destroyMethod")..**



При совместном использовании методов, интерфейсов и аннотаций, описанных выше, учитывайте их порядок вызовов. Для методов инициализации порядок будет следующий:

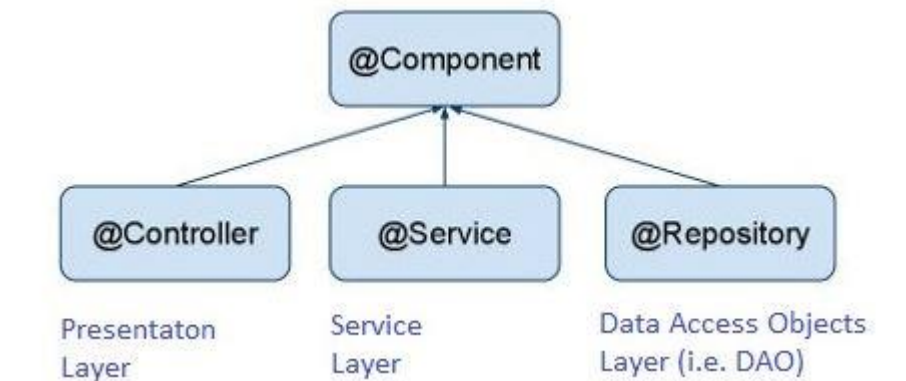
- Методы с аннотациями `@PostConstruct` в порядке их определения в классе
- Метод `afterPropertiesSet()`
- Метод, указанный в параметре `initMethod` аннотации `@Bean`

Для методов разрушения порядок будет следующий:

- Методы с аннотациями `@PreDestroy` в порядке их определения в классе
- Метод `destroy()`
- Метод, указанный в параметре `destroyMethod` аннотации `@Bean`

Spring IoC контейнеру требуются метаданные для конфигурации. Одну из таких аннотаций мы уже рассмотрели, это `@Bean`, рассмотрим теперь и другие.

Другой основной аннотацией является `@Component`, а также её наследники `@Repository`, `@Service` и `@Controller`.



Для того, чтобы ваша конфигурация могла знать о таких компонентах и вы могли бы их использовать, существует специальная аннотация для класса вашей конфигурации `@ComponentScan`.

```
@Configuration
@ComponentScan
public class LesConfiguration {
    @Bean
    GreetingService greetingService() {
        return new GreetingServiceImpl();
    }
}
```

Стоит упомянуть ещё одну мета-аннотацию `@Required`. Данная аннотация применяется к setter-методу бина и указывает на то, чтобы соответствующее свойство метода было установлено на момент конфигурирования значением из определения бина или автоматического связывания. Если же значение не будет установлено, будет выброшено исключение. Использование аннотации позволит избежать `NullPointerException` в процессе использования свойства бина.

```
private ApplicationContext context;
```

```
@Required  
public void setContext(ApplicationContext context) {  
    this.context = context;  
}
```

## Области видимости(scopes) бинов

Изначально, Spring Framework поддерживает несколько вариантов, некоторые доступны, только если вы используете web-aware ApplicationContext. Также вы можете создать свою собственную область видимости. Ниже приведен список областей видимостей:

**singleton** - По умолчанию. Spring IoC контейнер создает единственный экземпляр бина. Как правило, используется для бинов без сохранения состояния(stateless)

```
@Scope(value = ConfigurableBeanFactory.SCOPE_SINGLETON)  
@Component  
public class DDao {}
```

```
<bean id="dDao" class="com.spring.dao.DDao" scope="singleton" />
```

**prototype** - Spring IoC контейнер создает любое количество экземпляров бина. Новый экземпляр бина создается каждый раз, когда бин необходим в качестве зависимости, либо через вызов getBean(). Как правило, используется для бинов с сохранением состояния(stateful)

**request** - Жизненный цикл экземпляра ограничен единственным HTTP запросом; для каждого нового HTTP запроса создается новый экземпляр бина. Действует, только если вы используете web-aware ApplicationContext

**session** - Жизненный цикл экземпляра ограничен в пределах одной и той же HTTP Session. Действует, только если вы используете web-aware ApplicationContext

**global session** - Жизненный цикл экземпляра ограничен в пределах глобальной HTTP Session(обычно при использовании portlet контекста). Действует, только если вы используете web-aware ApplicationContext

**application** - Жизненный цикл экземпляра ограничен в пределах ServletContext. Действует, только если вы используете web-aware ApplicationContext.

Для того, чтобы указать область видимости бина, отличный от singleton, необходимо добавить аннотацию @Scope("область\_видимости") методу объявления бина или классу с аннотацией @Component:

```
@Component  
@Scope("prototype")  
public class GreetingServiceImpl implements GreetingService {  
    //...  
}
```

Классы с аннотацией `@Configuration` не стремятся на 100% заменить конфигурации на XML, при этом, если вам удобно или имеется какая-то необходимость в использовании XML конфигурации, то к вашей Java-конфигурации необходимо добавить аннотацию `@ImportResource`, в параметрах которой необходимо указать нужное вам количество XML-конфигураций. Выглядит это следующим способом:

```
@Configuration
@ImportResource("classpath:/web-java/xml-config.xml")
public class LesConfiguration {
    @Value("${jdbc.url}")
    String url;
    //...
}\
    xml-config.xml
<beans>
    <context:property-placeholder
location="classpath:/jdbc.properties"/>
</beans>
```

## Dependency Injection

DI была подтверждена официальным принятием процессом сообщества Java (Java Community Process - JCP) документ JSR-330 ("Dependency Injection for Java" - "Внедрение зависимостей для Java") в 2009 году.

**Внедрения зависимостей (DI) - является специализированной формой IoC.** Внедрение зависимости – это и есть инициализация полей бинов другими бинами (зависимостями).

Вариант Dependency Injection также имеет разновидности:

- Constructor Dependency Injection (Внедрение зависимостей через конструктор)
- Setter Dependency Injection (Внедрение зависимостей через метод установки).
- Через поле класса.

В Spring Boot в месте, где будет осуществлена инъекция необходимо ставить аннотацию `@Autowired` (автосвязывание). Однако, если вы используете DI с помощью конструктора использовать эту аннотацию необязательно.

Тип Property Injection имеет следующие недостатки: поле – публичное (можно и не публичное) нельзя указать интерфейс, только конкретную реализацию.

```
public class PropertyInjected {

    @Autowired
    public GreetingServiceImpl greetingService;
```

Тип Constructor Dependency Injection относится к ситуации, когда зависимости предоставляются компоненту в его конструкторе (или конструкторах). Компонент объявляет конструктор или набор конструкторов, получающих в качестве аргументов

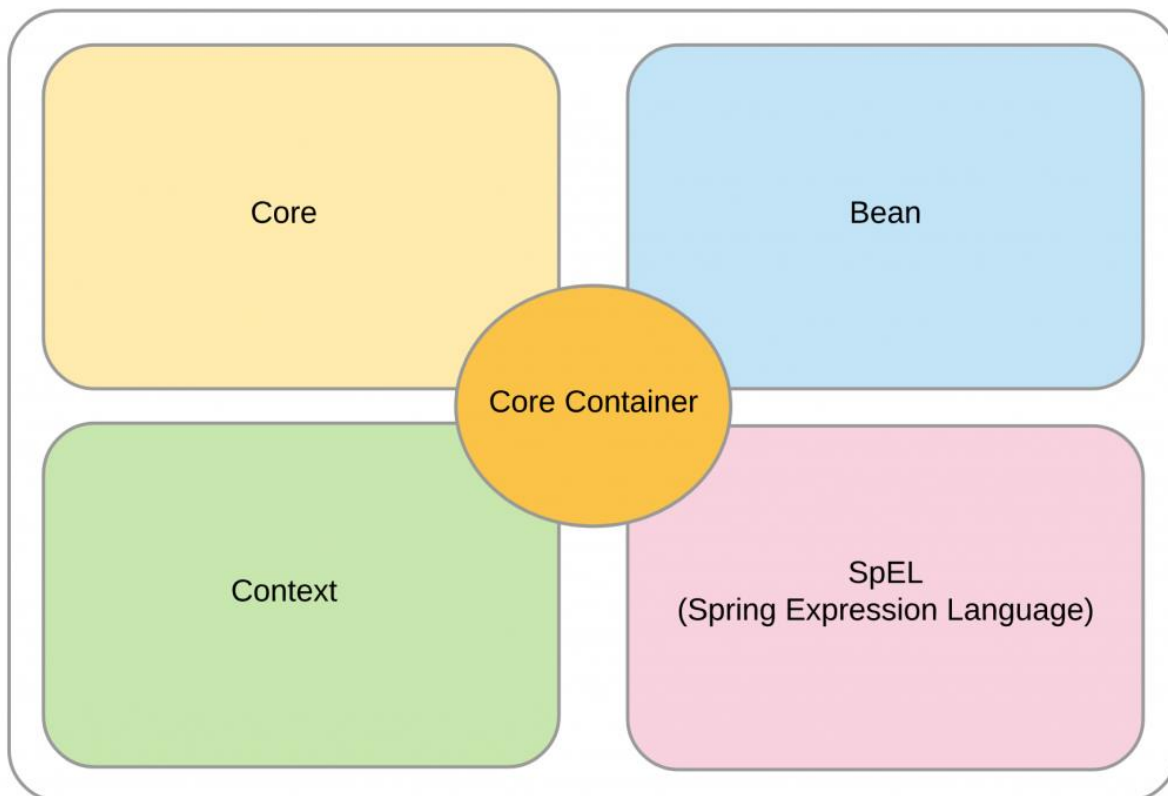
его зависимости, и контейнер IoC передает зависимости компоненту при создании его экземпляра

```
public class ConstructorInjection {  
  
    private Dependency dependency;  
  
    @Autowired  
    public ConstructorInjection(Dependency dependency) {  
        this.dependency = dependency;  
    }  
}
```

Этот способ обладает одним недостатком: до того момента, как Spring поставит все зависимости в бин, экземпляр класса находится в некорректном состоянии. И обращение к зависимостям может привести к `NullPointerException`.

```
public class SetterInjection {  
    private Dependency dependency;  
  
    @Autowired  
    public void setDependency(Dependency dependency) {  
        this.dependency = dependency;  
    }  
}
```

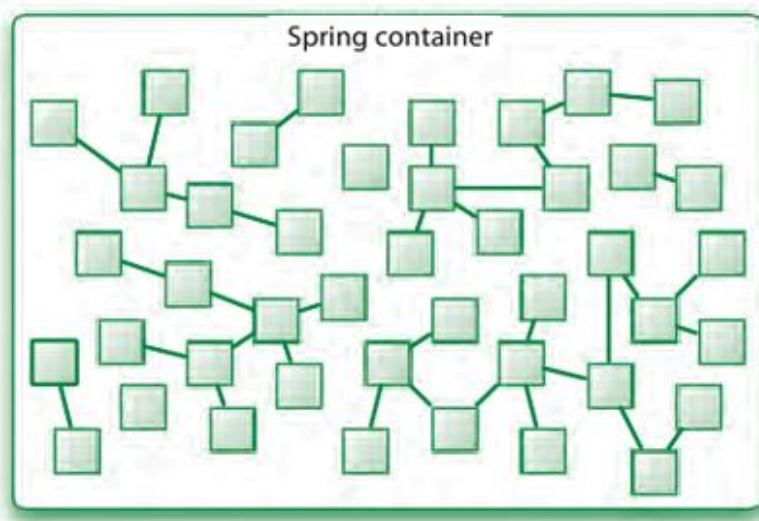
## Контейнеры Spring. Spring Core



Core содержит несколько базовых классов и инструментов. Вся Spring Framework основан на Core контейнере.

Модуль Core содержит базовые классы Spring Framework, включая внедрение зависимостей (DI) и инверсию управления (IOC).. Независимо от того, какой тип Spring Application вы создаете, у вас всегда будет прямая или косвенная зависимость от Spring Core.

Прикладные объекты располагаются внутри контейнера Spring. Контейнер создает объекты, связывает их друг с другом, конфигурирует и управляет их полным жизненным циклом



Контейнер находится в ядре фреймворка Spring Framework. Для управления компонентами, составляющими приложение, он использует прием внедрения зависимостей.

Интерфейс **BeanFactory** предоставляет инфраструктуру конфигурации и базовые функциональные возможности, а **ApplicationContext** является полным надмножеством BeanFactory и добавляет больше специфических функций. Удобнее всего работать с интерфейсом *ApplicationContext*.

Чтобы инициализировать контейнер и создать в нем бины, нужно создать экземпляр *ApplicationContext*.

Существуют два подкласса *ApplicationContext*:

*ClassPathXmlApplicationContext* берет конфигурацию из XML-файла, а

*AnnotationConfigApplicationContext* – из аннотаций:

Пример XML-based конфигурац. Метаданных

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="..." class="...">
    <!-- collaborators and configuration for this bean go here -->
  </bean>

  <bean id="..." class="...">
    <!-- collaborators and configuration for this bean go here -->
  </bean>

  <!-- more bean definitions go here -->

</beans>
```

Атрибут *id* - это строка, которая идентифицирует определение отдельного компонента. Атрибут *class* определяет тип компонента и использует полное имя класса.

```
<bean id="myDataSource" class="org.apache.commons
    .dbcp.BasicDataSource" destroy-method="close">
  <!-- results in a setDriverClassName(String) call -->
  <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
  <property name="url" value="jdbc:mysql://localhost:3306/mydb"/>
  <property name="username" value="root"/>
  <property name="password" value="masterkaoli"/>
</bean>
```

Как уже сказано, контейнеру для создания бинов требуется конфигурация, так что конструктор контейнера принимает аргументы:

```
ApplicationContext context =
    new ClassPathXmlApplicationContext("services.xml", "daos.xml");
```

```
// retrieve configured instance
PetStoreService service = context.getBean("petStore", PetStoreService.class);
```

## Процесс разрешения зависимостей

IoC контейнер выполняет разрешение зависимостей бинов в следующем порядке:

- 1) Создается и инициализируется ApplicationContext с метаданными конфигурации, которые описывают все бины. Эти метаданные могут быть описаны через XML, Java-код или аннотации
- 2) Для каждого бина и его зависимостей вычисляются свойства, аргументы конструктора или аргументы статического фабричного метода, либо обычного(без аргументов) конструктора. Эти зависимости предоставляются бину, когда он(бин) уже создан. Сами зависимости инициализируются рекурсивно, в зависимости от вложенности в себе других бинов. Например, при инициализации бина А, который имеет зависимость В, а В зависит от С, сначала инициализируется бин С, потом В, а уже потом А
- 3) Каждому свойству или аргументу конструктора устанавливается значение или ссылка на другой бин в контейнере
- 4) Для каждого свойства или аргумента конструктора подставляемое значение конвертируется в тот формат, который указан для свойства или аргумента. По умолчанию Spring может конвертировать значения из строкового формата во все встроенные типы, такие как int, long, String, boolean и др.

Spring каждый раз при создании контейнера проверяет конфигурацию каждого бина. И только бины с областью видимости(scope) singleton создаются сразу вместе со своими зависимостями, в отличие от остальных, которые создаются по запросу и в соответствии со своей областью видимости. В случае цикличной зависимости(когда класс А требует экземпляр В, а классу В требуется экземпляр А) Spring IoC контейнер обнаруживает её и выбрасывает исключение BeanCurrentlyInCreationException.

Spring контейнер может разрешать зависимости между бинами через autowiring(далее, автоматическое связывание).

Т.к. кандидатов для автоматического связывания может быть несколько, то для установки конкретного экземпляра необходимо использовать аннотацию **@Qualifier**. Данная аннотация может быть применена как к отдельному полю класса, так и к отдельному аргументу метода или конструктора:

```
public class AutowiredClass {
    //...

    @Autowired //к полям класса
    @Qualifier("main")
    private GreetingService greetingService;

    @Autowired //к отдельному аргументу конструктора или метода
    public void prepare(@Qualifier("main") GreetingService greetingService){
```



```

        /* что-то делаем... */
    };

```

Соответственно, у одной из реализации GreetingService должна быть установлена соответствующая аннотация **@Qualifier**:

```

@Component
@Qualifier("main")
public class GreetingServiceImpl implements GreetingService {
    //...
}

```

Spring также поддерживает использование JSR-250 **@Resource** аннотации автоматического связывания для полей класса или параметров setter-методов:

```

public class AutowiredClass {
    //...

    @Resource //По умолчанию поиск бина с именем "context"
    private ApplicationContext context;

    @Resource(name="greetingService") //Поиск бина с именем "greetingService"
    public void setGreetingService(GreetingService service) {
        this.greetingService = service;
    }

    //...
}

```

## Использование стандартных JSR-330 аннотаций

Spring Framework поддерживает JSR-330 аннотации. Эти аннотации работают таким же способом, как и Spring аннотации. Для того, чтобы работать с ними, необходимо добавить в pom.xml следующую зависимость:

```

<dependency>
<groupId>javax.inject</groupId>
<artifactId>javax.inject</artifactId>
<version>1</version>
</dependency>

```

Ниже приведена таблица сравнения JSR-330 и Spring аннотаций для DI:



| Spring              | javax.inject.* | Различия   |
|---------------------|----------------|--|
| @Autowired          | @Inject        | @Inject не имеет параметра <i>required</i>   |
| @Component          | @Named         | -  |
| @Scope("singleton") | @Singleton     | По умолчанию, область видимости у JSR-330 похожа на Spring <code>prototype</code> . Поэтому, если вы хотите использовать область видимости в соответствии с Spring, стоит использовать именно Spring аннотацию <code>@Scope</code> . |
| @Qualifier          | @Named         | -  |
| @Value              | -              | Нет аналога  |
| @Required           | -              | Нет аналога  |
| @Lazy               | -              | Нет аналога  |