# 1. SPRING CORE

## 1.1 Spring architecture

### Почему Spring?

**Легковесный фреймворк**. Состоит из набора jar-файлов небольшого размера. Например, если не надо работать с базой, то можно просто не подключать spring-jdbc. Во втором смысле легковесный потому, что у наших классов слабая связность со Spring.

**Spring – это контейнер**. Он управляет жизненным циклом объектов, которые он создает, и которые в нем живут. Не надо писать new, dispose, создавать зависимые объекты. То есть, они все в контейнере будут жить, крутиться, и там же и вариться.

**Spring – это фреймворк**. Он предоставляет кучу готовых велосипедов, которые очень хорошо ездят. В нем есть много утилитарных классов, упрощающих работу с базой данных, с почтой, веб-сервисами, и так далее.

**Dependency Injection (Inversion of Control)**. В Spring реализован принцип инверсии контроля, когда объекты не создают свои зависимости, а получают их.

**AOP (аспектно-ориентированное проиграммированое)**. Спринг сам реализует поддержку АОП, а многие его утилитарные классы и функции работают с помощью этой технологии.

### Основные модули Spring.

**Core container**, состоит из ключевых модулей.

Основные 2 модуля – Beans и Core, которые управляют бинами или реализуют внедрение зависимостей. Context – управляет контекстом, где хранятся бины, обеспечивает доступ к ним. Expression – специальный язык выражений, который может использоваться для поиска и модификаций графа бинов во время выполнения программы.

**AOP** – модуль поддержки AOP. Сюда входит внутренний встроенный модуль AOP, который позволяет вам использовать собственно АОП. Плюс есть модуль Aspects, обеспечивает поддержку библиотеки AspectJ.

**Instrumentation** – необходим, когда вы используете Спринг в каком-то application server (Tomcat, JBoss). Обеспечивает загрузку классов в контейнер и позволяет ему управлять Спринг контекстом и бинами. Содержит единственный модуль Instrument.

**Data access and integration** – модуль доступа к данным и БД в частности. Это и ORM, и JDBC, и JMS, и Transaction (поддержка транзакций), и других интеграционных вещей.

**Web & Remoting** – позволяют создавать компоненты для веб-приложений, и непосредственно веб-приложения. Есть и поддержка базовых вещей – например, реализация загрузки файлов на сервер. (Web, Servlet, Struts). Есть реализация паттерна MVC для создания веб-приложения. И компоненты для создания веб-сервисов, обеспечения безопасности и интеграции с другими веб-технологиями.

**Test** – упрощает тестирование Spring-приложений. С помощью него можно поднять Spring Context, создать бины, вызвать их методы и протестировать.

### Проблемы приложения, сделанного в рамках ветки SpringArchitecture

1. Внесение изменений проблематично. Данные внутри кода (захотим другого клиента, например, и придется ради этого перекомпилировать код). Решить проблему несложно – вынести в .property файлы, например.

2. Невозможность масштабирования. Жестко прописанный логгер, поменять на лету логику его работы без перекомпиляции приложения нельзя.

3. Сложно тестировать. Юнит-тест для метода App.logEvent() непрямо будет также тестировать ConsoleEventLogger.logEvent().

Можно, конечно, решить все три проблемы в отдельности, но их общая проблема в наличии **сильной связности** в приложении. Когда классы друг друга создают, когда в код зашиты данные, логи, которые обрабатывают эти данные, тоже создают новые классы.

## 1.2 Dependency Injection

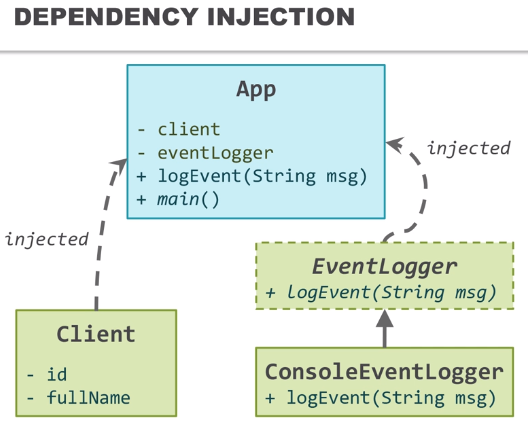
### Как решить описанные выше проблемы?

1. Вынести все статические данные во внешние файлы. Все, что когда-нибудь может измениться, нужно выносить прочь от кода.

2. Разъединение на интерфейсы. Это поможет избежать сильной связности. Если есть сервисные классы, их можно вызвать через интерфейс. Таким образом, конкретный класс не будет зависеть от конкретной реализации, а только от интерфейса сервиса. Перебарщивать с ними тоже не надо – нет смысла создавать интерфейс для класса, только хранящего данные (DAO).

3. Внедрить зависимости (dependency injection). Позволить объектам получать, а не создавать объекты. Сделать так, чтобы не мы создавали класс, а брали созданный кем-то (в нашем случае – Spring Framework). Вообще, существуют и другие фреймворки.

### Dependency Injection



Выглядеть будет примерно так. У нас есть класс App. У него есть все те же самые функции, те же методы, но класс не создает клиента или логгер, а получает их, а к логгеру обращается через интерфейс. Таким образом, App не знает, какой логгер он получит, что позволит нам изменить функциональность в случае необходимости. Такой подход позволяет решить все проблемы, которые были в нашем приложении.

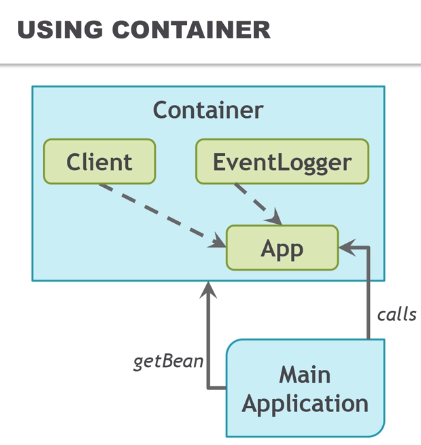
Но кто же нам создаст все эти объекты?

### Spring container

Контейнер будет наши классы создавать, соединять, передавать один в другой, а вы будете обращаться к нему за необходимой информацией. То есть, он создаст объекты классов, которые вы опишете. Данные объекты в Spring называются бинами.

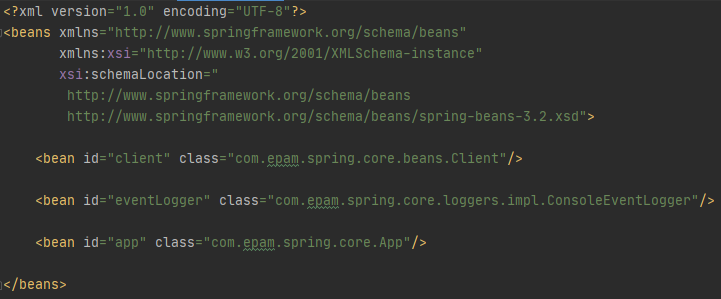
Спринг проинджектит нужные зависимости. То есть, он поймет, что нужно клиента и логгера засунуть в App, возможно, другие объекты между собой соединит, проверит, чтобы не было циклических зависимостей, и все. Объекты живут, зависимость у них есть. Осталось только к нему обратиться.

Наше приложение обратится к контейнеру, попросит у него определенный бин, а дальше будет вызывать его методы. Теперь, если нам нужен какой-то объект, то мы не создаем его, а обращаемся к контейнеру, чтобы его получить, либо делаем навигацию по графу объектов (получаем первый, через него получаем следующий, и так далее). Обращаться к контейнеру нам понадобится только в исходной точке нашей программы. Остальные объекты получат бины через внедрение зависимостей.



Как же Спринг поймет, какие объекты ему создавать? Для этого в Спринг используется xml файл. В нем содержится описание контейнера – тех бинов, которые вы хотите создать. Кроме xml, можно использовать аннотации. Ими вы отмечаете классы, которые станут бинами, а также места, где их надо проинджектить. В принципе, конфигурацию Спринг можно описать и полностью без xml, используя для этого только Java код.

### Spring.xml



Давайте разберем его по частям.

Главным образом вы определяете тег beans, а внутри него уже остальные бины. Также в этом теге указываются, какие xml схемы мы будем использовать. Основные схемы, которые нужно подключить – это перечисленные здесь. Без них ничего работать не будет.

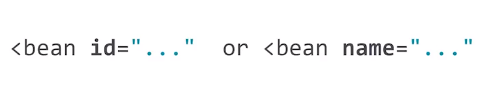
Дальше вы определяете ваши бины. Назначаете им какие-то id, и указываете class, объект которого хотите создать.

Такой файл лучше создавать в каталоге ресурсов (src/main/resources).

## 1.3 Bean Naming and Context Start-Up

### Bean naming

В Спринг есть два способа, как можно именовать бины. Можно указать имя бина либо в атрибуте **id**, либо в атрибуте **name**.



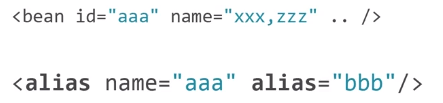
*В чем разница между ними?*

**id** – это формальный XML атрибут, то есть, XML тэги в документах, созданные согласно XML схеме, могут его содержать. Однако, он может быть только один и уникальный. В старой версии Спринга, до 3.2, было ограничение на то, какие символы могут присутствовать в id. Разрешались только буквы английского алфавита, цифры и некоторые другие.

То, что айди должен быть уникальным, с одной стороны, хорошо – нам нужно, чтобы бины имели уникальные имена, чтобы мы могли к ним правильно обращаться. Также XML парсер сможет выполнять дополнительные проверки, связанные с айди.

**name** – можно написать несколько имен, разделяя их пробелами, запятыми или точкой с запятой. В таком случае, к одному бину можно будет обращаться по разным именам. Основное имя будет первое, а остальные будут alias бина. Это может быть полезным для описания контекста программисту. Например, если есть класс, который реализует два интерфейса, то есть, выполняет роль двух сущностей (интерфейсы ClientService, LogonService, например, а реализующий их класс один). В таком случае, вы можете назначить 2 имени для бина, и в разных местах контекста обращаться к соответствующему имени. В будущем вы, возможно, разделите класс на 2 отдельных, и достаточно будет определить 2 отдельных бина с разными именами, при этом, не надо будет везде в описании контекста делать изменения.

*В определении бина можно указать и id, и name*. Оба имени будут работать и станут алиасами. Но лучше все-таки придерживаться какого-нибудь одного выбранного способа.

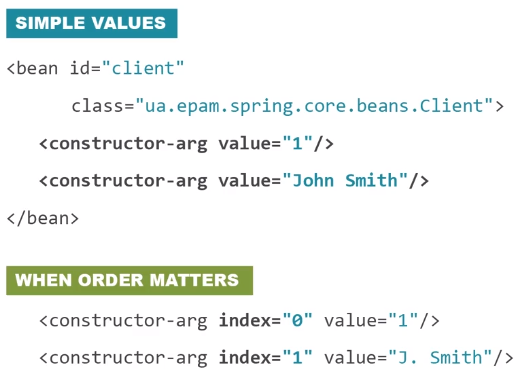


**alias** бина можно задать, используя атрибут **name**, но также можно использовать отдельный тег **alias**. В принципе, разницы нет, но, отдельный тег может пригодиться, если вы подключаете сторонний контекст к своему, и, при этом, изменить его вы не можете, то есть, вы не можете указать другие имена в атрибуте name для бина, а обращаться к нему по другому имени хочется.

Имя бина важно. Именно по нему мы будем обращаться к нему и доставать из контекста, и именно это имя мы будем использовать для внедрения зависимостей в другой бин.

Конечно, Спринг может совершить внедрение зависимостей, используя только типы объектов. Однако, в больших системах, где несколько классов могут реализовывать один и тот же интерфейс, с этим может быть сложно. Поэтому имена все равно понадобятся.

### Constructor injecting

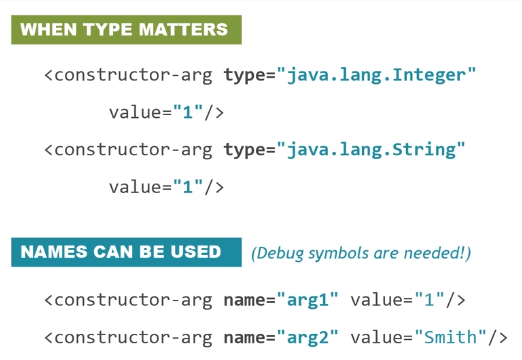


Спринг поддерживает два способа инджекта данных в бин – через конструктор, или через геттеры-сеттеры.

Начнем с конструкторов. Если мы хотим заинджектать какую-то статическую информацию, то можно просто использовать тег **constructor-arg**, как на изображении выше.

Обратите внимание на вещь, что конструкторы могут иметь объекты одного типа, например, String. Тогда в каком порядке Спринг будет инджектить эти значения в бин? *По умолчанию он инджектит в том порядке, в котором они описаны в XML файле*. Если вы хотите изменить порядок, то можно использовать атрибут **index**. В таком случае вы жестко зададите, какой переменной какое значение хотите задать.

Индексы – это хорошо, пока в конструкторе немного аргументов. Но что, если их там штук 20? Для каждого зададите индекс, а потом вам понадобится всунуть еще один аргумент где-то посередине. Придется менять все индексы, которые идут за ним. А еще лучше – просто не писать конструкторы с таким большим количеством аргументов. Если же типы аргументов разные, можно указывать их в разном порядке, ибо Спринг поймет, какой параметр куда должен попасть.

Правда, может быть ситуация, когда для параметра Integer вы указали значение 1, и для параметра String вы указали значение 1. 

В таком случае, можно указать тип аргумента с помощью атрибута **type**. Это позволит не привязываться к индексам.

Можно устанавливать значение и по имени аргумента с помощью **name**. Но работать это будет только в случае, если включены debug-символы при компиляции приложения. Тогда Спринг сможет через рефлексию узнать имена аргументов и вставить значения. *На продакшн-коде такого делать никто не будет (а жаль :с)*. Очень странно, кстати, но у меня работает и без этого. Надо разобраться, почему. Никаких дебаг-символов и препроцессоров не включено.

В документации Спринг точно так же указано, что нужен дебаг-флаг ([https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-some-examples](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html" \l "beans-some-examples)), но по факту все успешно компилится, и нигде этот флаг не нужен. Не ИДЕ в этом виновата, ибо запуск в консоли mvn clean install и java -jar тоже работают без проблем.

Таким образом, мы вынесли статическую информацию из кода. Это еще не проперти файл, конечно, но, все-таки, текстовый, и уже гораздо легче поддается изменениям.

*Как же проинджектить бины в друг друга?*



Для этого используется атрибут **ref** в **constructor-arg**. Значением этого атрибута будет имя бина, который вы хотите заинджектить.

*Если Спринг по какой-то причине не сможет найти конструктор, принимающий указанное количество аргументов, вы получите ошибку*.

Если Спрингу не передать никаких аргументов, он будет вызывать конструктор по умолчанию. Притом, он должен быть явно задан, если существуют конструкторы с параметрами. Интересно и то, что модификатор доступа конструктора может быть любой – Спринг все равно сможет им воспользоваться.

### Spring container

С Spring Framework есть 2 типа контейнеров.

Первый – **Bean Factory**. Это простейший контейнер. Он делает только одно – dependency injection. То есть, он поднимается, создает бины и инджектит их, куда надо. Больше он ничего не умеет делать.

Второй – это **ApplicationContext**. Он делает все то же самое, что и Bean Factory, но также предоставляет и сервисы фреймворка по управлению этими бинами:

- ClassPathXmlApplicationContext;

- FileSystemXmlApplicationContext;

- AnnotationConfigApplicationContext;

- XmlWebApplicationContext;

- StaticApplicationContext.

Как же создать контекст в программе?



Мы определяем переменную ApplicationContext (это интерфейс, кстати). И мы используем new, создаем соответствующий контекст, и указываем ему, где искать соответствующий spring.xml файл. Дальше мы вызываем метод getBean, которому передаем имя бина. Вообще, аппликейшн контекст содержит разные методы для получения бина. Можно их получать по имени, по классу, по классу и имени, можно даже запросить бины, у которых есть определенная аннотация. Если вы запрашиваете бин по имени, то вам придется явно приводить возвращаемый объект к нужному типу. Если вы запрашиваете бин по классу, либо по имени и классу, тогда кастить не придется благодаря Generics.

При использовании аппликейшн контекста можно указать сразу несколько xml файлов. В каждом из них могут быть описаны разные бины. Они даже могут инджектить друг друга. В таком случае Спринг объединит описания и создаст один контекст со всеми бинами вместе.

## 1.4 Scopes and inner beans