# **Spring – общее описание**

## **Spring – история, определение**

истоки Spring уходят корнями в книгу Expert One-to-One J2ЕЕ Design and Development, в которой Род Джонсон представил lnterface 21 Framework. После выпуска в свет открытого кода этот фреймворк послужил основанием для известного нам теперь Spring Framework. Первая версия Spring была выпущена в октябре 2002 года и состояла из небольшого ядра с контейнером управления инверсией (loC), который нетрудно было настроить и применять. Первая стабильная версия (Spring Framework 1.0) появилась 24 марта 2004 года. Со временем каркас Spring пришел на смену серверам JEE (Java Enterprise Edition - корпоративная версия Java) и дорос до уровня полноценной технологии.

Обычно Spring описывается как легковесный каркас ( **lightweight framework**) для построения приложений на Java. Ядро каркаса Spring Framework основано на принципе инверсии управления (**lnversion of Control - IoC**)

Главный принцип всей философии Spring - минимальное воздействие. Spring является легковесным каркасом в том смысле, что для использования всех преимуществ ядра Spring вам придется внести лишь минимальные изменения в свой прикладной код.

По сути, Spring предлагает контейнер, часто называемый контекстом приложения Spring, который создает компоненты приложения и  управляет ими. Эти компоненты, или bean-компоненты, объединяются внутри контекста Spring, образуя полноценное приложение, подобно тому, как кирпичи, известковый раствор, древесина, гвозди, водопроводные трубы и проводка соединяются вместе, образуя дом. Акт объединения bean-компонентов основан на шаблоне, известном как внедрение зависимостей (Dependency Injection, DI). В технологии внедрения зависимостей компоненты не создают и не поддерживают жизненный цикл других компонентов, от которых они зависят, а полагаются в этом на отдельный объект (контейнер), который создаст все нужные компоненты и внедрит их в другие компоненты, которые в них нуждаются. Обычно это делается с помощью аргументов конструктора или методов доступа к свойствам

## **Инверсия управления и внедрение зависимостей**

Спринг реализует паттерн композиция.

**lnversion of Control (инверсия управления) – IoC -**  когда создание зависимостей между компонентами и управление ими осуществляется внешним образом. Например, при использовании библиотеки вы сами прописываете в своем коде какой метод какого объекта вызвать, а в случае с фреймворками — чаще всего уже фреймворк будет вызывать в нужный ему момент тот код, который вы написали. То есть, тут уже не вы управляете процессом выполнения кода/программы, а фреймворк это делает за вас. Вы передали ему управление (инверсия управления).

Понятие инверсии управления было переименовано Мартином Фаулером в более описательное понятие внедрения зависимостей (**Dependency lnjection** - DI).

Компонент, которому **требуются** определенные **зависимости**, зачастую называется **зависимым объектом**, а в случае инверсии управления - **целевым объектом**.

Реализация внедрения зависимостей в Spring основана на двух ключевых понятиях Java: компонентах JavaBeans и интерфейсах. Написание кода по интерфейсам способствует повышению гибкости приложений, но сложность связывания при этом весьма велика (и требует много кода), но применяя внедрение зависимостей, можно сильно сократить объем кода. А с помощью интерфейсов можно получить максимальную отдачу от внедрения зависимостей, потому что в компонентах Spring Beans можно использовать любую реализацию интерфейса для удовлетворения их зависимости. Кроме того, применение интерфейсов позволяет задействовать в Spring динамические объекты-заместители из комплекта JDK по проектному шаблону "Заместитель" (Proxy), чтобы предоставить эффективные средства вроде сквозной функциональности.

### Преимущества внедрения зависимостей:

* Сокращение объема связующего кода (кода, который должен быть написан для связывания вместе компонентов приложения.).
* Упрощенное конфигурирование приложений.
* Возможность управлять общими зависимостями в одном хранилище.
* Улучшенная тестируемость. (например, применяя внедрение зависимостей, можно сымитировать реализацию объекта DАО, возвращающую наборы тестовых данных, а затем передать их объекту предметной области для проверки.)
* Стимулирование качественных проектных решений для приложений.

Недостаток:

* Внедрение зависимостей способно затруднить тем, кто не особенно хорошо ориентируется в прикладном коде, возможность выяснить, какая именно реализация отдельной зависимости привязана к конкретным объектам.

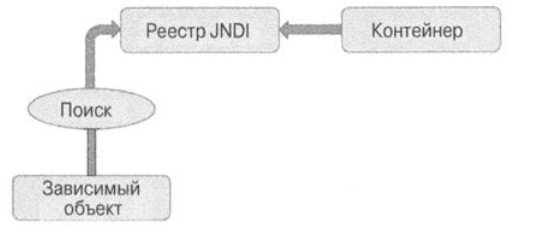
### Dependency injection и Dependency lookup

Инверсия управления может быть разделена на два подтипа: **внедрение зависимостей (**Dependency injection**)** и **поиск зависимостей** (**Dependency lookup**). Когда речь идет о внедрении зависимостей, всегда имеется в виду инверсия управления. Но когда речь идет об инверсии управления, то не всегда имеется в виду внедрение зависимостей. Например, поиск зависимостей - это также форма инверсии управления. Второй тип - внедрения зависимостей - в действительности обеспечивает более высокую гибкость и удобство применения по сравнению с поиском зависимостей.

Если инверсия управления реализуется как поиск зависимостей, то компонент должен получить ссылку на зависимость, тогда как при внедрении зависимостей последние внедряются в компонент контейнером инверсии управления. У поиска зависимостей имеются две разновидности: извлечение зависимостей и контекстный поиск зависимостей (CDL). И у внедрения зависимостей имеются две разновидности : через конструктор и через метод установки (setter dependency injection).

У поиска зависимостей имеются две разновидности:

**1. Извлечение зависимостей** (Dependency Pull), один из видов внедрения зависимостей. В этом случае зависимости извлекаются из реестра по мере необходимости. Извлечение зависимостей предоставляется и в Spring как механизм для извлечения компонентов, которыми он управляет.

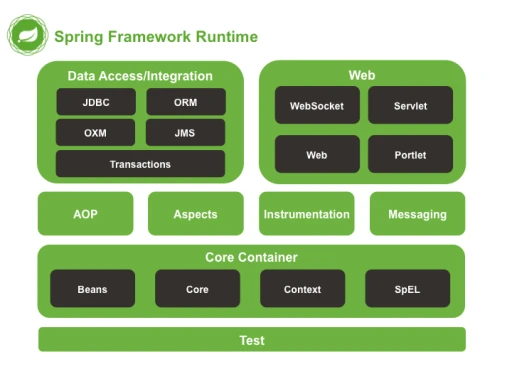


**2. Контекстный поиск зависимостей** (Contextualized Dependency Lookup, CDL). В этом случае поиск осуществляется в контейнере, управляющем ресурсом, а не только в каком-то центральном реестре.



Механизм контекстного поиска приводится в действие через реализацию в компоненте интерфейса. Реализуя этот интерфейс, комnонент извещает контейнер, что ему требуется получить зависимость.

## **Структура Spring**



**Модули**:

Аор - Содержит все классы, требующиеся для применения в приложении средств АОП из Spriпg. Этот архивный JАR-файл должен быть также включен в приложение, если планируется пользоваться другими средствами Spriпg, в которых применяется АОП

Beans - Содержит все классы, поддерживающие манипулирование компонентами Spriпg Веапs.

Context - Содержит классы, которые предоставляют многие расширения для ядра Spring. Все классы должны использовать интерфейс ApplicationContext из Spriпg. В этом модуле содержатся также классы Spriпg для удаленного взаимодействия, классы для интеграции с языками динамических сценариев (например, JRuby, Groovy, BeaпShell), классы из прикладного интерфейса API по спецификации JSR-303 (Веапs Validatioп), классы для планирования и выполнения заданий и т.д.

Core - Основной модуль, требующийся для каждого приложения Spring. В его архивном JАR-файле находятся классы, общие для всех остальных модулей Spring

jdЬc - В этот модуль входят все классы, предназначенные для поддержки JDBC. Он необходим для всех приложений, которым требуется доступ к базам данных.

Test

Tx – Предоставляет все классы, предназначенные для поддержки инфраструктуры транзакций в Spriпg.

wеb - Содержит основные классы для применения Spriпg в веб-приложениях, в том числе классы для автоматической загрузки контекста типа ApplicationContext, классы для поддержки выгрузки файлов и ряд полезных классов для выполнения таких повторяющихся заданий, как извлечение целочисленных значений из строки запроса

web-reactive – Содержит базовые интерфейcы и классы для модели реактивного вебпрограммирования в Spriпg

web - Содержит все классы для собственного каркаса по проектному шаблону MVC в Spriпg. А если применяется отдельный каркас по шаблону MVC, то классы из архивного JАR-файла этого модуля не требуются.

**Проекты**:

Ядро Spring Framework

Spring Security - обеспечивает управление доступом к ресурсам приложения на основе ролей, а в приложениях с более сложными требованиями к безопасности поддерживается ведение списка управления доступом (ACL), Spring Security применяется главным образом в защищенных веб-приложениях

Spring Boot – содержит готовые конфигурации для различных типов приложений Spring, организованные в стартовые пакеты, внедряет все зависимости, требующиеся в приложении Spring, принимая во внимание совместимость версий.

Spring Batch предоставляет общую структуру и разнообразные правила для реализации пакетных заданий, сокращая объем стереотипного кода.

Spring lntegration помогает упростить интеграцию приложений Spring с внешними системами.

Spring MVC – веб-фреймворк Spring

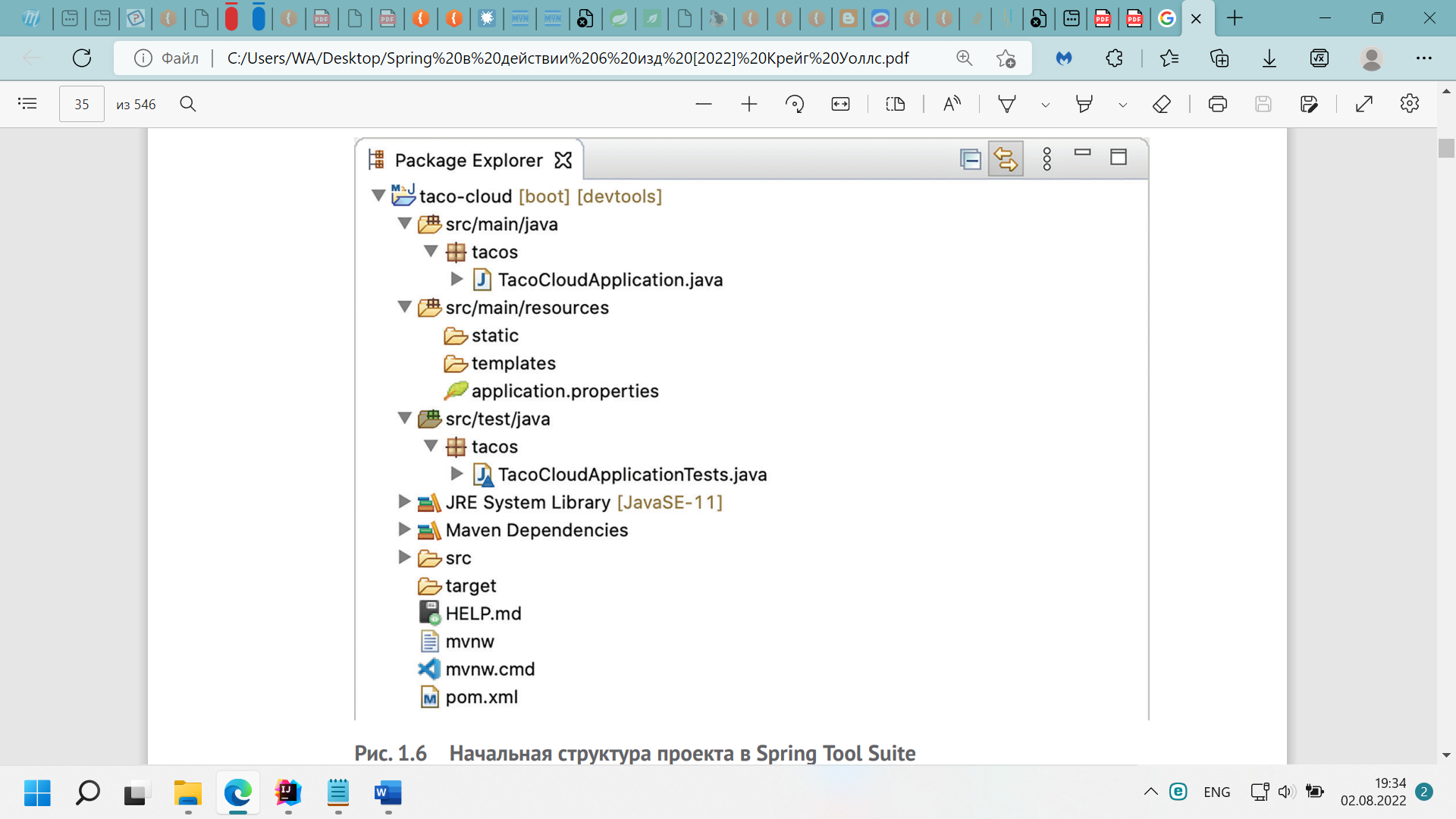
Spring Data

Spring Cloud - набор проектов для разработки облачных приложений с помощью Spring.

Spring Native – позволяет компилировать проекты Spring Boot в  двоичные выполняемые файлы с  помощью компилятора GraalVM

# **Конфигурирование Spring**

Можно делать вручную, а можно использоватьSpring Initializr (из Spring Boot) – это веб-приложение, помогающее создать скелетную структуру проекта Spring



исходный код помещается в каталог src/main/java, код тестов – в src/test/java, а ресурсы, не являющиеся Java-ресурсами, – в  src/main/resources.

mvnw и  mvnw.cmd  – это сценарии-обертки Maven, эти сценарии можно использовать для создания нового проекта, даже если на вашем компьютере не установлен Maven;

pom.xml – это параметры сборки Maven

application.properties  – это файл используется для определения конфигурационных свойств. М

static – в эту папку можно поместить любой статический контент (изображения, таблицы стилей, JavaScript и т. д.), который должен отображаться в браузере.

templates  – тут мы разместим файлы шаблонов, используемые для отображения контента в браузере.

## **Конфигурации**

### Виды конфигураций

1) **Автоконфигурация** - наиболее приоритетный способ, которому стоит отдавать предпочтение. Spring Boot может делать обоснованные предположения о том, какие компоненты следует настроить и связать вместе, опираясь на элементы в пути поиска классов, переменные окружения и другие факторы. Автоконфигурация, предлагаемая Spring Boot, значительно сократила объем явного описания конфигурации (с помощью XML или на Java), необходимого для создания приложения.

2) **Конфигурационный класс Java**. Используется, только если Spring не может автоматически настроить компоненты. Аннотация @Configuration подсказывает фреймворку Spring, что это класс конфигурации, который создает bean-компоненты для контекста Spring. Методы класса конфигурации снабжены аннотацией @Bean, указывающей, что возвращаемые ими объекты должны быть добавлены в контекст приложения как bean-компоненты (где эти компоненты по умолчанию будут доступны по идентификаторам, совпадающим с именами определяющих их методов). Определение конфигурации в  Java-коде имеет определенные преимущества перед описанием в XML-файлах, в том числе более высокий уровень безопасности типов и  простоту рефакторинга.

3) **ХМL-файл** - раньше способ управления контекстом приложения Spring для связывания bean-компонентов был основан на одном или нескольких XML-файлах, описывающих компоненты и их взаимосвязи с другими компонентами. Сейчас этот способ устарел.

Spring позволяет комбинировать эти способы.

### **Бины и контекст**

**Бин**. По сути, это просто объект какого-то класса.

**Контекст** — это набор бинов (объектов). Обращаясь к контексту — мы можем получить нужный нам бин (объект) по его имени например, или по его типу, или еще как-то.

### Виды внедрения зависимостей:

**Constructor Dependency Injection**

Внедрение зависимостей через конструктор происходит в том случае, когда зависимости предоставляются компоненту в его конструкторе (или нескольких конструкторах). С этой целью в компоненте объявляется один или ряд конструкторов, получающих в качестве аргументов его зависимости, а контейнер инверсии управления передает зависимости компоненту при получении его экземпляра.

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

@Override  
 public String toString() {  
 return dependency.toString();  
 }  
}

**Setter method dependency injection**

При внедрении зависимостей через метод установки контейнер инверсии управления внедряет зависимости компонента через методы установки в стиле компонентов JavaBeans. Методы установки компонента отражают зависимости, которыми может управлять контейнер инверсии управления. На практике внедрение зависимостей через метод установки оказывается наиболее широко применяемым механизмом инверсии управления, который проще всего реализовать.

public class SetterInjection {  
  
 private Dependency dependency;  
  
 public void setDependency(Dependency dependency) {  
 this.dependency = dependency;  
 }  
  
 @Override  
 public String toString() {  
 return dependency.toString();  
 }  
}

**Field dependency injection**

Внедрение зависимостей через поле

**Выбор между внедрением и поиском зависимостей**  Как правило, выбор типа инверсии управления зависит от применяемого контейнера. В Spring компоненты и их зависимости всегда связываются вместе с помощью инверсии управления в стиле внедрения, за исключением первоначальных поисков компонентов Spring Beans.

Если есть выбор, нужно использовать **внедрение зависимостей**. Оно не оказывает никакого влияния на исходный код компонентов, тогда как, код, реализующий извлечение зависимостей, должен активно получать ссылку на реестр и взаимодействовать с ним при получении зависимостей, а для применения контекстного поиска зависимостей требуется, чтобы в классах был реализован конкретный интерфейс и поиск зависимостей вручную. Если же выбрать внедрение зависимостей, то в классах достаточно разрешить их внедрение через конструкторы или же через методы установки. Выбрав внедрение зависимостей, можно пользоваться своими классами полностью отдельно от контейнера инверсии управления, который поставляет зависимые объекты и взаимодействующие с ними объекты вручную, тогда как при поиске зависимостей классы всегда будут зависеть от классов и интерфейсов, определяемых в контейнере инверсии управления. Еще один недостаток поиска зависимостей состоит в том, что он сильно затрудняет тестирование классов отдельно от контейнера. А при внедрении зависимостей тестирование компонентов не составит особого труда, поскольку для этого достаточно предоставить зависимости с помощью подходящего конструктора или метода установки. Главной причиной для выбора внедрения, а не поиска зависимостей является значительное упрощение задачи разработки. Код внедрения зависимостей является пассивным в том смысле, что он не пытается активно выполнить какую-то задачу, таким образом, код оказывается намного более простым и менее подверженным ошибкам.

**Выбор между внедрением зависимостей через конструктор и метод установки**

внедрение зависимостей через конструктор окажется удачным выбором в том случае, когда требуется гарантировать передачу зависимостей компоненту и когда проектирование выполняется для неизменяемых объектов.

Внедрение зависимостей через конструктор особенно удобно в том случае, если экземпляр класса зависимости должен существовать перед применением компонента. Кроме того, внедрение зависимостей через конструктор помогает в применении неизменяемых объектов.

Внедрение зависимостей через метод установки позволяет менять местами зависимости, не создавая новые объекты, а также разрешает классу выбирать подходящие стандартные настройки, не прибегая к явному внедрению объекта.

Внедрение зависимостей через метод установки полезно если компонент предоставляет свои зависимости контейнеру, но готов обеспечить для них свои стандартные настройки. Внедрение зависимостей через метод установки состоит в том, что оно позволяет объявлять зависимости в интерфейсе. Но если нет абсолютной уверенности, что все реализации отдельного интерфейса предметной области требуют конкретной зависимости, то в каждом классе реализации лучше определить его собственные зависимости, а в интерфейсе предметной области оставить одни только методы предметной области. Внедрение зависимостей через метод установки позволяет также оперативно сменять зависимости для различных реализаций, не создавая нового экземпляра родительского компонента. Пожалуй, самое значительное преимущество внедрения зависимостей через метод установки заключается в том, что оно является наименее навязчивым из всех существующих механизмов внедрения зависимостей.

**Инверсия управления в Spring**

Ядро реализации каркаса Spriпg основано на внедрения зависимостей, хотя в нем обеспечиваются также возможности для поиска зависимостей. Каркас Spriпg автоматически предоставляет взаимодействующие объекты зависимому объекту, используя внедрение зависимостей .



Контейнер инверсии управления в Spring примечателен тем, что он может выполнять функцию адаптера между его собственным контейнером внедрения зависимостей и внешними контейнерами поиска зависимостей.

**Внедрение зависимостей в Spring**

Пользоваться внедрением зависимостей вместе с платформой Spriпg следует при всякой возможности. А в крайнем случае можно обратиться к возможностям поиска зависимостей.

Ядром контейнера внедрения зависимостей в Spring служит интерфейс Bean Factory, который отвечает за управление компонентами Spring Beans, в том числе их зависимостями и жизненными циклами. Термин компонент Spring Веап употребляется в Spring для обозначения любого компонента, управляемого контейнером.

Внутренне конфигурация компонентов Spring Beans представлена экземплярами классов, реализующих интерфейс BeanDefini tion. В конфигурации компонента Spring Bean хранятся сведения не только о самом компоненте, но и о тех компонентах Spring Beans, от которых он зависит. Для любых классов реализации интерфейса BeanFactory, в которых также реализуется интерфейс Bean DefinitionReader, данные типа BeanDefini tion можно прочитать из файла конфигурации, используя классы PropertiesBeanDefinitionReader или XmlBean DefinitionReader.

Итак, компоненты Spring Beans можно идентифицировать в интерфейсе Bean Factory, и каждому компоненту Spring Bean может быть назначен идентификатор, имя или то и другое. Экземпляр компонента Spring Bean можно получить и без идентификатора или имени (это так называемый анонимный компонент Spring Bean) или как один компонент внутри другого компонента Spring Bean. У каждого компонента Spring Bean имеется по крайней мере одно имя, но их может быть сколько угодно, причем дополнительные имена разделяются запятыми. Все имена после первого считаются псевдонимами того же самого компонента Spring Bean. Чтобы изв.лечь компоненты Spring Beans из интерфейса BeanFactory и установить отношения зависимости, когда, например, компонент Х зависит от компонента У, можно воспользоваться их идентификаторами или именами.

**Интерфейс ApplicationContext**

ApplicationContext, который предоставляет намного больше возможнос;тей для конфигурирования, чем традиционный интерфейс BeanFactory.

Интерфейс ApplicationContext служит расширением интерфейса BeanFactory в Spring. Помимо услуг no внедрению зависимостей, интерфейс Application Context предоставляет такие услуги, как транзакции и АОП, источник сообщений для интернационализации (i 18n), обработка событий в приложениях и пр. При разработке приложений, основанных на Spring, рекомендуется взаимодействовать с Spring через интерфейс ApplicationContext. Начальная загрузка интерфейса ApplicationContext поддерживается в Spring посредством ручного программирования (получения экземпляра вручную и загрузки подходящей конфигурации) или в среде веб-контейнера через класс ContextLoaderListener.

Чтобы определить компоненты Spring Beans с помощью аннотаций, классы этих компонентов необходимо снабдить соответствующими стереотипными аннотациями2, а методы и конструкторы - аннотацией @Autowired. В последнем случае контейнер инверсии управления уведомляется, где именно следует искать компонент Spring Bean конкретного типа, чтобы задать его в качестве аргументов при вызове данного метода. В качестве параметра стереотипных аннотаций можно задать имя результирующего компонента Spring Bean. 2Такие аннотации называются стереотипными потому, что они входят в состав nакета org. springframework. stereotype. В этом nакете собраны все аннотации, nрименяемые дпя оnределения комnонентов Spring Beans.

**Конфигурирование на языке Java**

конфигурационный класс снаб-жается аннотацией @Configuration и содержит методы, объявляемые с аннотацией @Bean и вызываемые непосредственно из контейнера инверсии управления для получения экземпляров компонентов Spring Beans. Имя компонента Spring Bean будет совпадать с именем метода, применяемого для его создания.

Вместо экземпляра класса Defaul tListaЫeBeanFactory в данном случае получается экземпляр класса AnnotationConfigApplicationContext. Класс AnnotationConfigApplicationContext реализует интерфейс ApplicationContext. Конфигурационный класс может служить и для чтения определений компонентов Spring Beans, снабженных аннотациями. В данном случае в конфигурационном классе не потребуются методы с аннотациями @Bean, поскольку конфигурирование определения компонента Spring Bean является составной частью его класса. Но для того, чтобы найти определения компонентов Spring Beans в классах Java, придется активизировать просмотр этих компонентов. Это делается в конфигурационном классе с помощью аннотации @ComponentScanning, равнозначной элементу разметки ,как показано ниже.

, конфигурирование в формате XML можно по-разному сочетать с конфигурированием на языке Java. Например, в конфигурационном классе можно импортировать определения компонентов Spring Beans из одного или нескольких ХМL-файлов конфигурации, используя аннотацию @ImportResource

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

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

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

**Соображения по поводу внедрения зависимостей через метод поиска**

Внедрение зависимостей через метод поиска предназначено для тех случаев, когда требуется работать с двумя компонентами Spriпg Beans, имеющими разные жизненные циклы. Не поддавайтесь искушению применять внедрение зависимостей через метод поиска, если компоненты Spring Beans разделяют один и тот же жизненный цикл, особенно когда они являются одиночными.

**Замена метода**

Применяя замену метода, можно произвольно заменить реализацию любого метода в каком угодно компоненте Spring Веап, не внося коррективы в исходный код этого компонента. Допустим, что в приложении применяется сторонняя библиотека и требуется изменить логику действия определенного метода. Но изменить исходный код нельзя, поскольку он предоставлен сторонней организацией, и поэтому единственным решением будет замена метода, позволяющая внедрить собственную реализацию вместо логики действия данного метода.

Внутренне это достигается путем динамического создания подкласса, производного от класса компонента Spring Bean. Применяя библиотеку CGLIВ, обращения к заменяемому методу следует переадресовать другому компоненту Spring Bean, в котором реализуется интерфейс MethodReplacer.

Чтобы заменить метод, необходимо реализовать сначала интерфейс Method Replacer

**Именование компонентов Spring Beans с помощью аннотаций в конфигурациях**

Если определения компонентов Spring Beans объявляются с помощью аннотаций, то именование компонентов несколько отличается от принятого при конфигурировании в формате XML

Начнем, однако, с простого объявления определений компонентов Spring Beans с помощью стереотипной аннотации @Component и всех ее специальных разновидностей вроде @Service, @Repository и @Controller.

компонент Spring Bean именуется так же, как и сам класс, но первая буква в его имени становится строчной.

Если же требуется присвоить имя компоненту Bean по-другому, это имя следует указать в качестве аргумента аннотации

@Component("johnMayer")

А как насчет псевдонимов? Если задать в качестве аргумента аннотации @Component однозначный идентификатор компонента Spring Bean, то назначение псевдонимов окажется невозможным при таком способе объявления компонента.

Чтобы объявить nсевдонимы, можно восnользоваться атрибутом name аннотации @Bean, который исnользуется в данной аннотации no умолчанию.

@Bean(name = {"johnMayer", "john", "jonathan", "johnny"})

Для поддержки nсевдонимов в версии Spring 4.2 была внедрена аннотация @AliasFor, nозволяющая объявлять псевдонимы для атрибутов большинства других аннотаций, доступных в Spring. Например, у аннотации @Bean имеются два атрибута, name и value, неявно объявляемых как nсевдонимы друг для друга, а с помощью аннотации @AliasFor они явно становятся псевдонимами.

public @interface Award {  
  
 @AliasFor("prize")  
 String[] value() default {};  
  
 @AliasFor("value")  
 String[] prize() default {};  
}

**Режим получения экземпляров компонентов Spring Beans**

По умолчанию все компоненты Spring Beans являются одиночными. Это означает, что Spring обслуживает одиночный экземпляр компонента, во всех зависимых объектах используется один и тот же экземпляр, а в результате всех вызовов метода ApplicationContext. getBean () возвращается один тот же экземпляр.

Все компоненты Spring Beans по умолчанию создаются как одиночные экземпляры, и поэтому для всех запросов к конкретному компоненту Spring Bean применяется один и тот же экземпляр. Каркас Spring, конечно, не ограничивается применением только одиночного экземпляра; для удовлетворения каждой зависимости и каждого обращения к методу getBean ( ) можно по-прежнему получить новый экземпляр компонента Spring Bean.

Сменить режим получения экземпляров с одиночного на неодиночный в режиме XML:

scope="prototype"

в режиме аннотаций:

@Component  
@Scope("prototype")  
public class Singer {

По умолчанию каркас Spring устанавливает в атрибуте scope значение singleton (одиночный). Область видимое-ти на уровне nрототиnа вынуждает Spring nолучать новый экземnляр комnонента Spring Bean всякий раз, когда он заnрашивается в nриложении.

**Выбор режима получения экземпляров**

одиночные экземпляры должны nрименяться в следующих случаях.

Общий объект без состояния. Имеется объект, не поддерживающий состояние, и множество зависимых от него объектов. В связи с тем что состояние не поддерживается, синхронизация не нужна, а следовательно, не придется получать новый экземпляр компонента Spring Bean всякий раз, когда он требуется зависимому объекту для какой-нибудь обработки.

Общий объект с состоянием только для чтения. Этот случай похож на предыдущий, но отличается поддержанием состояния только для чтения. Но и в этом случае синхронизация не нужна, и получение экземпляра по каждому запросу компонента Spring Bean только влечет дополнительные издержки.

• Общий объект с разделяемым состоянием. Если требуется компонент Spring Bean, состояние которого должно разделяться с другими объектами, то одиночный экземпляр будет идеальным выбором. Но в этом случае придется обеспечить как можно более точную синхронизацию при записи состояния.

• Высокопроизводительные объекты с записываемым состоянием. Если имеется компонент Spring Bean, который интенсивно применяется в приложении, то может оказаться, что сохранение его одиночным и синхронизация всего доступа к нему для записи позволит добиться более высокой производительности, чем постоянное получение сотен экземпляров этого объекта.

Неодиночные экземпляры должны применяться в перечисленных ниже случаях.

• Объекты с записываемым состоянием. Если имеется компонент Spring Bean с большой долей записываемого состояния, то может оказаться, что затраты на синхронизацию превышают затраты на получение нового экземпляра для обработки каждого запроса от зависимого объекта.

• Объекты с закрытым состоянием. В ряде случаев зависимым объектам требуется компонент Spring Bean с закрытым состоянием, чтобы зависимые объекты могли выполнять свою обработку раздельно. В таком случае одиночный экземпляр совсем не подходит, а следовательно, требуется неодиночный экземпляр.

Области видимости компонентов Spring Beans

Одиночный экземпляр. Стандартная область видимости. В этом случае объекты будут создаваться лишь по одному на каждый контейнер инверсии управления в Spring.

Прототип. Каркас Spring получит новый экземпляр по запросу из приложения.

Запрос. Служит для применения в веб-приложениях. Если для построения вебприложений применяется модуль Spring MVC, то компоненты Spring Beans с областью видимости на уровне запроса будут создаваться по каждому НТТРзапросу и уничтожаться по окончании его обработки.

Сеанс связи. Служит для применения в веб-приложениях. Если для построения веб-приложений применяется модуль Spring MVC, то компоненты Spring Beans с областью видимости на уровне сеанса связи будут создаваться для каждого НТТР-сеанса и уничтожаться по его завершении.

Глобальный сеанс связи. Служит для применения в веб-приложениях, основанных на портлетах. Компоненты Spring Beans с областью видимости на уровне глобального сеанса связи могут совместно применяться всеми портлетами в портальном приложении, приводимом в действие модулем Spring MVC.

Поток исполнения. Каркас Spring получит новый экземпляр компонента Spring Bean по запросу из нового потока исполнения. А по запросу из одного и того же потока исполнения возвратится тот же самый экземпляр компонента Spring Bean. Однако эта область видимости не регистрируется по умолчанию.

Специальная. Это специальная область видимости компонента Spring Bean, которую можно создать, реализовав интерфейс org. springframework. beans. factory. config. Scope и зарегистрировав ее в конфигурации Spring (для конфигурирования в формате XML следует применять класс org. spring framework.beans.factory.config.CustomScopeConfigurer~

Разрешение зависимостей

При нормальном функционировании каркас Spring способен разрешать зависимости, просто просматривая файл конфигурации или аннотации в классах.

Подобным образом каркас Spring может гарантировать, что все его компоненты сконфигурированы в правильном порядке и каждый компонент имеет корректно настроенные зависимости. Ефигурирует их в произвольном порядке, то вполне возможно, что какой-нибудь его компонент будет создан и настроен прежде своих зависимостей.

@Component("johnMayer")  
@DependsOn("gopher")  
public class Singer

**Автосвязывание компонентов Spring Beans**

В каркасе поддерживается пять режимов автосвязывания. Ниже приведено их краткое описание.

• Режим byName. Если применяется этот режим, каркас Spring пытается связать каждое свойство с одноименным компонентом Spring Bean. Так, если у целевого компонента Spring Bean имеется свойство f оо и в контексте типа ApplicationContext определен компонент foo, то этот компонент присваивается свойству foo целевого компонента Spring Bean.

• Режим bуТуре. Если применяется этот режим, каркас Spring пытается связать каждое свойство целевого компонента Spring Bean с компонентом того же самого типа, автоматически выбираемым из контекста типа АррliсаtiоnСоntехt.

• Режим constructor. Этот режим действует подобно режиму ЬуТуре, за исключением того, что зависимости внедряются в нем через конструкторы, а не методы установки. Каркас Spring пытается обнаружить совпадение с как можно большим числом аргументов в конструкторе.

Режим default. В этом режиме каркас Spring автоматически делает выбор между режимами constructor и ЬуТуре. Если у компонента Spring Bean имеется конструктор по умолчанию (т.е. без аргументов), то в Spring выбирается режим автосвязывания ЬуТуре, а иначе - режим constructor.

• Режим no. Выбирается по умолчанию.

конфигурированию с помощью аннотаций. Для атрибута lazy-ini t имеется равнозначная аннотация @Lazy, применяемая на уровне класса для объявления компонентов Spring Beans, экземпляры которых получаются при первом обращении к ним. С помощью стереотипных аннотаций можно составить лишь одну конфигурацию для компонента Spring Bean, и поэтому кажется вполне логичным, что имена компонентов Spring Beans не имеют особого значения, поскольку для каждого типа предполагается лишь один компонент. Так, если составить конфигурацию с помощью упомянутой выше аннотации, то по умолчанию будет выбран режим автосвязывания ЬуТуре. А если имеются компоненты Spring Beans родственных типов, то целесообразно выбрать режим автосвязывания byName. Это можно сделать с помощью аннотации @Qualifier вместе с аннотацией @Autowired, указав имя внедряемого компонента Spring Bean в качестве аргумента.

Аннотация @Primary,в класс, определяющий компонент Spring Bean. Эта аннотация предписывает каркасу Spring отдать предпочтение данному компоненту Spring Bean при автосвязывании по типу Аннотация @Primary действует как маркерный интерфейс и не имеет атрибутов. Ее наличие в конфигурации компонента Spring Bean помечает его как приоритетный, когда требуется автоматически привязать компонент Spring Bean данного типа в режиме ЬуТуре.

**Аннотация @Qualifier**. Эта аннотация размещается сразу же после аннотации @Autowired в объявлениях неоднозначно функционирующих методов установки.

**Настройка наследования компонентов Spring Beans**

# Lifecycle

**Управление жизненным циклом компонентов Spring Beans**

Важной особенностью любого контейнера инверсии управления, в том числе и в Spring, является возможность построения компонентов таким образом, чтобы получать уведомления в определенные моменты своего жизненного цикла. Это дает компонентам возможность выполнять надлежащую обработку в такие моменты их жизненного цикла. В общем, непосредственное отношение к компонентам имеют два события, наступающие в течение жизненного цикла: после инициализации и перед уничтожением компонентов.

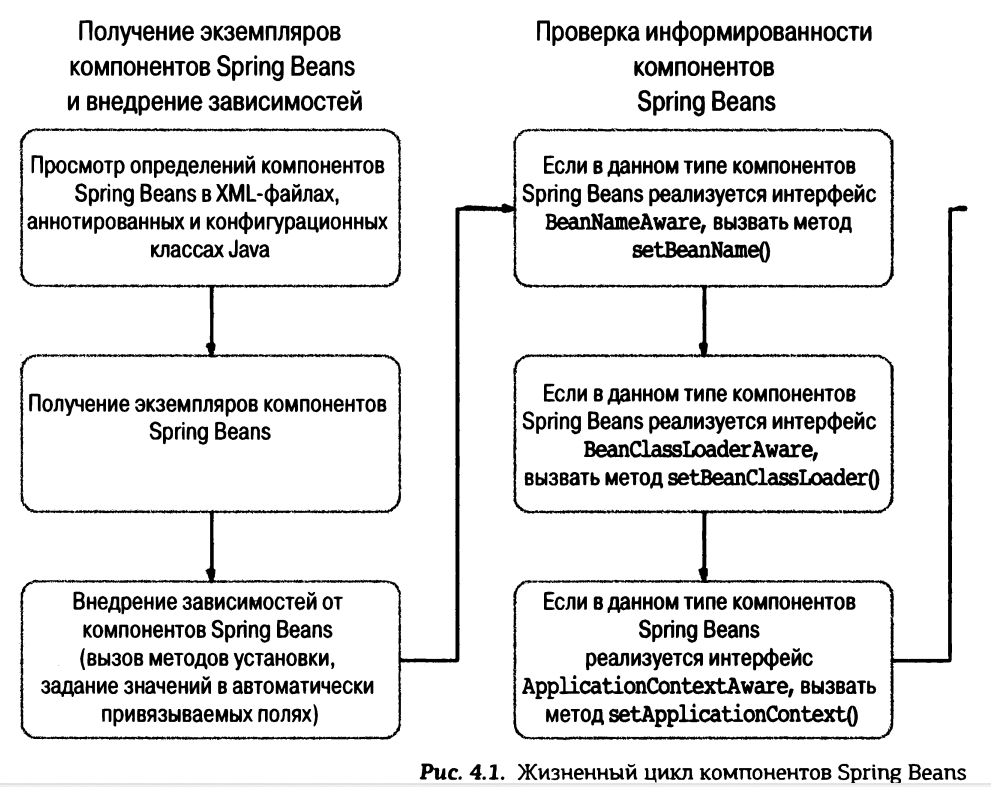
В контексте Spring событие после инициализации наступает в тот момент, когда завершается установка всех значений свойств компонента Spring Bean и все проверки зависимостей, сконфигурированные для выполнения. А событие перед уничтожением наступает непосредственно перед тем, как Spring приступит к уничтожению экземпляра компонента Spring Bean. Но для компонентов Spring Beans с областью видимости на уровне прототипа событие перед уничтожением не будет инициировано в каркасе Spring. Этот каркас спроектирован таким образом, чтобы методы обратного вызова жизненного цикла, выполняющие инициализацию компонента Spring Bean, вызывались для объектов независимо от области видимости данного компонента. Хотя для компонентов Spring Beans с областью видимости на уровне прототипа методы обратного вызова, выполняющие уничтожение компонента Spring Bean, вызываться не будут. В каркасе Spring предоставляются три механизма, которые можно применять в компоненте Spring Bean для привязки к каждому из этих событий и выполнения дополнительной обработки. Эти механизмы основаны на интерфейсах, методах и аннотациях соответственно.

Если в компоненте Spriпg Bean применяется механизм, основанный на интерфейсах, то в нем реализуется интерфейс, характерный для того типа уведомлений, которые необходимо получать, а каркас Spring посылает уведомления данному комnонен1)' через метод обратного вызова, определенный в этом интерфейсе. Если же применяется механизм, основанный на методах, то в Spring допускается указывать в конфигурации интерфейса ApplicationContext имя метода, который должен быть вызван при инициализации компонента Spring Bean, а также имя метода, который должен быть вызван при уничтожении данного компонента. А механизм, основанный на аннотациях, предполагает применение аннотаций по спецификации JSR-250 для указания методов, которые должны вызываться после построения и перед уничтожением компонентов Spring Beans.

Механизм, основанный на интерфейсах, широко применяется непосредственно в Spring, поэтому не обязательно помнить, что следует указывать инициализацию и уничтожение всякий раз, когда применяются компоненты Spring Beans. Но в собственных компонентах, возможно, лучше применять механизм, основанный на методах или аннотациях, чтобы не реализовывать в них специальные интерфейсы из Spring. И хотя выше утверждалось, что переносимость зачасrую оказывается не настолько важной, как обычно заявляется в многочисленной литературе на данную тему, это совсем не означает, что переносимостью можно пожертвовать, когда имеется вполне пригодная альтернатива. Иными словами, если вы привязываете свое приложение к каркасу Spring другими способами, то, применяя интерфейсный метод, можете определить обратный вызов лишь один раз и просто забыть о нем. А если вы определяете целый ряд однотипных компонентов Spring Beans, которым требуются уведомления в течение их жизненного цикла, то, применяя механизм, основанный на интерфейсах, вы можете избавить себя от необходимости указывать в ХМL-файле конфигурации методы обратного вызова жизненного цикла для каждого компонента Spring Bean. Применение аннотаций по спецификации JSR-250 - еще один приемлемый способ, поскольку они соответствуют стандарrу, определенному в организации JCP, а вы не привязываетесь непосредственно к аннотациям, характерным только для Spring. Для этого достаточно убедиться, что в том контейнере инверсии управления, где выполняется приложение, поддерживается стандарт JSR-250

**Создание компонентов Spring Beans**

Если компонент Spring Bean информирован о том, когда он был инициализирован, такой компонент может проверить, все ли требуемые зависимости удовлетворены.



Используя обратный вызов при инициализации, компонент может проверить требующиеся ему зависимости и сгенерировать, если потребуется, исключение или же предоставить значение, устанавливаемое по умолчанию. Обратный вызов при инициализации инициируется после того, как каркас Spring завершит предоставление зависимостей и выполнит любые запрашиваемые проверки зависимостей.

**Выполнение метода при создании компонента Spriпg Веап**

один из способов получения обратного вызова при инициализации предусматривает назначение конкретного метода из компонента Spring Bean в качестве метода инициализации и уведомление об этом каркаса Spring.

@Configuration  
static class SingerConfig{  
  
 @Lazy  
 @Bean(initMethod = "init")  
 Singer singerOne() {  
 Singer singerOne = new Singer();  
 singerOne.setName("John Mayer");  
 singerOne.setAge(39);  
 return singerOne;  
 }

**Реализация интерфейса InitializingBean**

Интерфейс InitializingBean, определяемый в Spring, позволяет задать в компоненте Spring Bean код, который будет выполнен, когда компонент получит уведомление, что каркас Spring завершил его конфигурирование. Как и в методе инициализации, это дает возможность проверить конфигурацию компонента Spring Bean и попутно предоставить любые значения по умолчанию.

**Применение аннотации @PostConstruct по спецификации ]SR-250**

Достичь той же самой цели можно и с помощью аннотации жизненного цикла @PostConstruct, определенной в спецификации JSR-250. Начиная с версии Spring 2.5 поддерживаются аннотации по спецификации JSR-250, предназначенные для указания метода, который каркас Spring должен вызвать, если соответствующая аннотация, связанная с жизненным циклом компонента Spring Bean, существует в классе.

Аннотация PostConstruct используется для методов, которые необходимо выполнить после завершения внедрения зависимости для выполнения любой инициализации.

@PostConstruct  
public void afterPropertiesSet() throws Exception {

**Объявление метода инициализации с помощью аннотации @Bean**

Еще один способ объявить метод инициализации для компонента Spring Bean состоит в том, чтобы указать в аннотации @Bean атрибут ini tMethod вместе с именем данного метода. С помощью этой аннотации объявляются компоненты Spring Beans в конфигурационных классах Java

public class SingerConfigDemo {  
  
 @Configuration  
 static class SingerConfig{  
  
 @Lazy  
 @Bean(initMethod = "init")  
 Singer singerOne() {

**Описание порядка разрешения зависимостей**

Все упомянутые выше механизмы инициализации могут применяться к одному и тому же экземпляру компонента Spring Bean. В данном случае Spring сначала вызывает метод, снабженный аннотацией @PostConstruct, затем метод afterProperties Set (),а далее - метод инициализации, указанный в файле конфигурации.

можно выявить следующие стадии в процессе создания компонентов Spring Beans:

1. Сначала вызывается конструктор для создания компонента Spring Bean.

2. Затем вызываются методы установки для внедрения зависимостей.

3. Как только появятся компоненты Spring Beans с внедренными зависимостями, делается запрос компонентов типа BeanPostProcessor из инфраструктуры предварительной инициализации, чтобы выяснить, требуется ли им вызвать что-нибудь из данного компонента Spring Bean. Это компоненты специальной инфраструктуры Spring, выполняющие модификацию компонентов Spring Beans после их создания. Аннотация @PostConstruct регистрируется средствами класса CommonAnnotationBeanPostProcessor, и поэтому из данного компонента Spring Bean вызывается метод, снабженный аннотацией @Post Construct. Этот метод выполняется сразу же после построения компонента Spring Bean, но прежде ввода данного класса в действие и до фактической инициализации компонента Spring Bean (т.е. до вызова метода afterProperties Set () и атрибута init-method).

4. Метод afterPropertiesSet (),определяемый в интерфейсе Initializing Bean, выполняется сразу же после внедрения зависимостей. Этот метод вызывается из интерфейса BeanFactory после установки всех предоставленных свойств компонента Spring Bean и удовлетворения требований интерфейсов BeanFactoryAware и ApplicationContextAware.

5. Атрибут init-method вызывается в последнюю очередь, поскольку в нем фактически указан конкретный метод инициализации компонента Spring Bean.

**Уничтожение компонентов Spring Beans**

Когда применяется реализация интерфейса ApplicationContext, заключающая в оболочку интерфейс DefaultListaЬleBeanFactory (например, через метод get Defaul tListaЬleBeanFactory () из класса GenericXmlApplicationContext), то, вызвав метод ConfigurableBeanFactory. destroySingletons (), можно сообщить интерфейсу BeanFactory, что требуется уничтожить все одиночные экземпляры. Обычно это делается при завершении приложения, чтобы очистить любые ресурсы, которые компоненты Spring Beans могут удерживать открытыми, и тем самым корректно закрыть приложение. Разрешить компонентам Spring Beans получать уведомления о вызове метода destroySingletons () можно тремя способами, которые похожи на механизмы для получения обратного вызова при инициализации.

Выполнение метода при уничтожении компонента Spriпg Веап

------

Реализация интерфейса DisposableBean

Как и для обратных вызовов при инициализации, в Spring предоставляется интерфейс DisposaЬleBean, который может быть реализован в компонентах Spring Beans как механизм для получения обратных вызовов при уничтожении.

public class DestructiveBeanWithInterface implements InitializingBean, DisposableBean {

@Override  
public void destroy() {

Применение аннотации @PreDestroy по спецификации ]SR-250

Третий способ уничтожения компонентов Spring Beans предусматривает применение аннотации жизненного цикла @PreDestroy, которая определена в спецификации JSR-250 и полностью противоположна по своему действию @PostConstruct.

@PreDestroy  
public void destroy() {

Объявление метода уничтожения с помощью аннотации @Bean

Еще один способ объявить метод уничтожения для компонента Spring Bean состоит в том, чтобы указать в аннотации @Bean атрибут destroyMethod вместе с именем данного метода. С помощью этой аннотации объявляются компоненты Spring Beans в конфигурационных классах Java

public class DestructiveBeanConfigDemo {  
  
 @Configuration  
 static class DestructiveBeanConfig {  
  
 @Lazy  
 @Bean(initMethod = "afterPropertiesSet", destroyMethod = "destroy")  
 DestructiveBeanWithJSR250 destructiveBean() {  
 DestructiveBeanWithJSR250 destructiveBean = new DestructiveBeanWithJSR250();  
 destructiveBean.setFilePath(System.*getProperty*("java.io.tmpdir") +  
 System.*getProperty*("file.separator") + "test.txt");  
 return destructiveBean;  
 }  
  
 }

**Описание порядка разрешения зависимостей**

Как при создании, так и при уничтожении компонентов Spring Beans все упомянутые выше механизмы можно применять к одному и тому же экземпляру компонента. С этой целью в Spring сначала вызывается метод, снабженный аннотацией @PreDestroy, затем метод DisposaЫeBean. destroy () и метод уничтожения, сконфигурированный в определении данного компонента в формате XML.

**Применение перехватчика завершения**

Единственный недостаток обратных вызовов при уничтожении компонентов в Spring заключается в том, что они не запускаются автоматически, т.е. перед закрытием приложения следует вызвать мeтoдAbstractApplicationContext. destroy (). Если приложение выполняется как сервлет, указанный метод destroy () можно вызвать в методе destroy () данного сервлета. Но в автономном приложении дело обстоит не так просто, особенно при наличии многих точек выхода из приложения. Правда, из этого положения есть выход. В языке Java допускается создание перехватчика завершения - потока, который исполняется непосредственно перед завершением приложения. Задействовать этот механизм проще всего с помощью метода registerShutdown Hook () из класса, реализующего интерфейс AbstractApplicationContext. Определение и конфигурация компонента Spring Bean остаются прежними, за исключением следующих изменений в методе main ():добавления вызова ctx. registerShutdown Hook () и удаления вызовов ctx. destroy () или ctx. close (),как показано ниже.

public static void main(String... args) throws Exception {  
 GenericXmlApplicationContext ctx = new GenericXmlApplicationContext();  
 ctx.load("classpath:spring/app-context-annotation.xml");  
 ctx.registerShutdownHook();  
 ctx.refresh();  
  
 ctx.getBean(DestructiveBeanWithHook.class);

**Информирование компонентов Spring Beans об их контексте**

при определенных обстоятельствах может потребоваться такой компонент Spring Bean, в котором внедрение зависимостей применяется для получения его зависимостей.

Применение интерфейса BeanNameAware

В интерфейсе BeanNameAware, который компонент Spring Bean может реализовать, чтобыполучитьсвоеимя,определен единственный мeтoдsetBeanName ( String). Каркас Spring вызывает метод setBeanName () по окончании конфигурирования компонента Spring Bean, но перед любыми обратными вызовами жизненного цикла (при инициализации или уничтожении), как показано на рис. 4.1. Как правило, реализация метода setBeanName () сводится к одной строке, в которой значение, переданное контейнером, сохраняется в каком-нибудь поле для дальнейшего применения. Например когда компонент Spring Bean, получает свое имя с помощью интерфейса BeanNameAware и затем выводит его на консоль.

Применение интерфейса ApplicationContextAware

Через интерфейс ApplicationContextAware компоненты Spring Beans могут получить ссылку на экземпляр контекста типа ApplicationContext, в котором они сконфигурированы. Основной причиной для создания этого интерфейса была необходимость предоставить компоненrу Spring Bean досrуп к контексrу типа Application Context в приложении (например, для получения других компонентов Spring Beans программным способом}, используя метод getBean (). Но подобной нормы практики следует все же избегать, а вместо нее внедренять зависимости, чтобы предоставить свои компоненты Spring Beans, а также взаимодействующие с ними объекты.

Интерфейс ApplicationContext служит, конечно, не только для поиска компонентов Spring Beans, но и для решения многих других задач. Как упоминалось ранее, к числу таких задач относится уничтожение всех одиночных экземпляров с предварительным их уведомлением по очереди. В предыдущем разделе было показано, как создать перехватчик завершения, чтобы предписать интерфейсу Application Context уничтожить все одиночные экземпляры перед закрытием приложения.

**Применение фабрик компонентов Spring Beans**

Пользуясь каркасом Spring, вы непременно столкнетесь с трудностями реализации и внедрения зависимостей, которые нельзя просто создать с помощью операции new. Для преодоления подобных трудностей в Spring предоставляется интерфейс FactoryBean, действующий в качестве адаптера для тех объектов, которые нельзя создавать и манипулировать ими с помощью стандартной семантики Spring. Как правило, фабрики применяются для создания тех компонентов Spring Beans, которые нельзя создать с помощью операции new.

Попросту говоря, интерфейс FactoryBean и реализующий его компонент Spring Bean выполняет функции фабрики для других компонентов Spring Beans. Такие фабрики конфигурируются в контексте типа ApplicationContext подобно обычным компонентам Spring Beans, но когда интерфейс FactoryBean применяется в Spring для удовлетворения запроса на внедрение или поиск зависимости, вместо возврата экземпляра компонента, реализующего интерфейс FactoryBean, вызывается метод FactoryBean. getObject () и возвращается результат этого вызова.

Класс МessageDigestFactoryBean как пример фабрики компонентов Spriпg Веапs

В языке Java имеется класс MessageDigest, предоставляющий функциональные средства для формирования свертки сообщения из произвольных данных.

public class MessageDigesterConfigDemo {  
  
 @Configuration  
 static class MessageDigesterConfig {  
  
 @Bean  
 public MessageDigestFactoryBean shaDigest() {  
 MessageDigestFactoryBean factoryOne = new MessageDigestFactoryBean();  
 factoryOne.setAlgorithmName("SHA1");  
 return factoryOne;  
 }  
  
 @Bean  
 public MessageDigestFactoryBean defaultDigest() {  
 return new MessageDigestFactoryBean();  
 }  
  
 @Bean MessageDigester digester() throws Exception {  
 MessageDigester messageDigester = new MessageDigester();  
 messageDigester.setDigest1(shaDigest().getObject());  
 messageDigester.setDigest2(defaultDigest().getObject());  
 return messageDigester;  
 }  
 }  
  
 public static void main(String... args) {  
 GenericApplicationContext ctx = new AnnotationConfigApplicationContext(MessageDigesterConfig.class);  
  
 MessageDigester digester = (MessageDigester) ctx.getBean("digester");  
 digester.digest("Hello World!");  
 ctx.close();  
 }  
}

**Непосредственный доступ к фабрике компонентов Spring Beans**

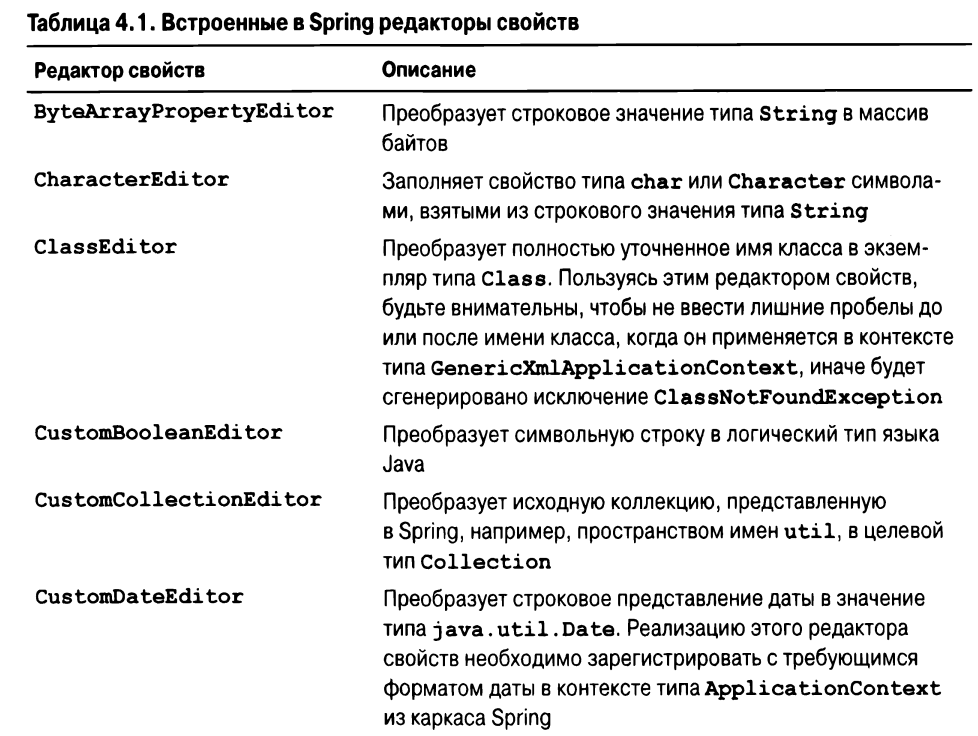
Непосредственный Доступ к фабрике компонентов Spring Beans осуществляется просто. Для этого имя компонента Spring Bean достаточно предварить в вызове метода getBean () знаком амперсанда,

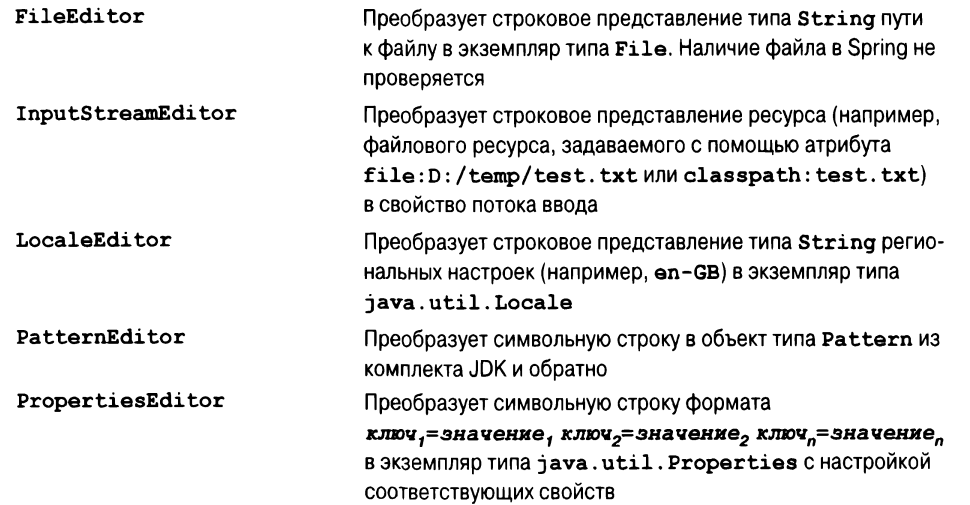
MessageDigestFactoryBean factoryBean =  
 (MessageDigestFactoryBean) ctx.getBean("&shaDigest");

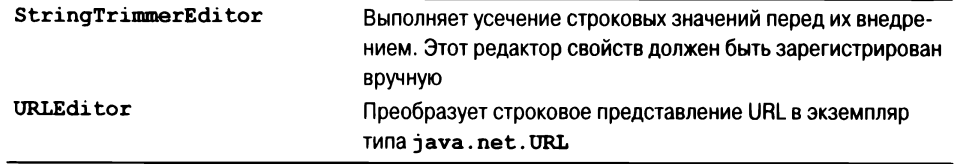
**Применение атрибутов factory-bean и factory-method**

Иногда требуется получить экземпляры компонентов JavaBeans, предоставляемых сторонним приложением, не поддерживающим Spring. Вам, например, неизвестно, как получить их экземпляры, но в то же время известно, что стороннее приложение предоставляет класс, который можно использовать для получения экземпляра компонента JavaBean, требующегося в вашем приложении Spring.

PropertyEditor (редактор свойств) - это интерфейс, nредназначенный для взаимного преобразования типов внутреннего представления значений свойств и типа String.







**Доступ к ресурсам**

В каркасе Spring предоставляется единообразный механизм для доступа к ресурсам независимо от конкретного протокола.

В основу всей поддержки ресурсов в Spring положен интерфейс org. springframework. core. io. Resource. В интерфейсе Resource определены десять самоочевидных методов: contentLength (), exists (), getDescription (), getFile (), getFileName (), getURI (), getURL (), isOpen (), isReadaЫe () и lastModified ().Но имеется еще один, не столь самоочевидный метод create Relati ve (),в котором получается новый экземпляр типа Resource по относительному пути к тому экземпляру, для которого он вызывается.

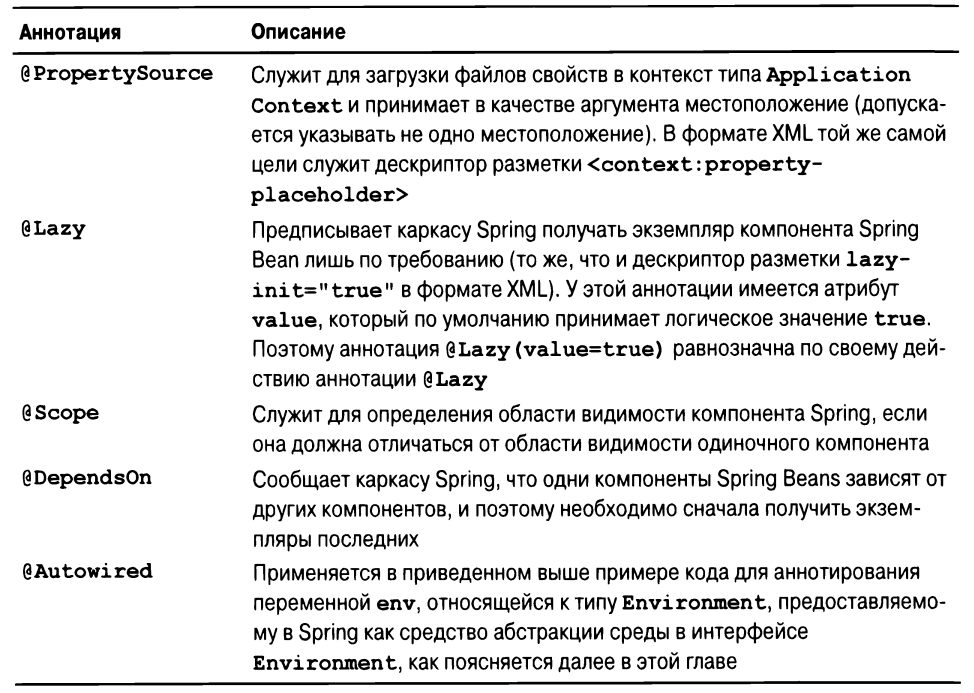
**Конфигурирование с помощью классов Java**

Чтобы избавиться от конфигурации в формате XML, придется заменить файл app-context-xml. xml специальным классом, который называется конфигурационным и снабжен аннотацией @Configuration, которая извещает Spring, что данный файл конфигурации построен на основе классов Java. Этот конфигурационный класс будет содержать методы с аннотациями @Bean, представляющими определения компонентов Spring Beans. С помощью аннотации @Bean объявляется компонент Spring Bean и требования к внедрению зависимостей. Аннотация @Bean равнозначна дескриптору разметки , а имя метода - атрибуту id в дескрипторе <bean> и когда получается экземпляр компонента Spring Bean, внедрение зависимостей через метод установки достигается благодаря вызову соответствующего метода для получения поставщика сообщений. А это равнозначно тому, чтобы воспользоваться дескриптором разметки при конфигурировании в формате XML.

для инициализации экземпляра контекста типaApplicationContext вызывается конструктор клaccaAnnotation ConfigApplicationContext, которому в качестве аргумента передается упомянутый выше конфигурационный класс (используя аргументы переменной длины, этому конструктору можно передать несколько конфигурационных классов). После этого возвращаемым экземпляром контекста типа ApplicationContext можно воспользоваться, как обычно. Иногда для целей тестирования конфигурационный класс может быть объявлен как внутренний статический следующим образом:

public class JavaConfigSimpleDemo {  
  
 @Configuration  
 static class AppConfig {  
 @Bean  
 public MessageProvider messageProvider() {  
 return new ConfigurableMessageProvider();  
 }  
  
 @Bean  
 public MessageRenderer messageRenderer() {  
 MessageRenderer renderer = new StandardOutMessageRenderer();  
 renderer.setMessageProvider(messageProvider());  
  
 return renderer;  
 }  
 }  
  
  
 public static void main(String... args) {  
 ApplicationContext ctx = new   
 AnnotationConfigApplicationContext(AppConfig.class);  
  
 MessageRenderer renderer =  
 ctx.getBean("messageRenderer", MessageRenderer.class);  
  
 renderer.render();  
 }  
}

**Аннотации для конфигурирования на языке Java**



В аннотации @ComponentScan определяются пакеты, в которых каркас Spring должен просмотреть аннотации к определениям компонентов Spring Beans. По своему действию она равнозначна дескриптору разметки при конфигурировании в формате XML.

В приложении можно также применять несколько конфигурационных классов для развязки конфигурации и организации компонентов Spring Beans по назначению (например, один такой класс может быть выделен для объявления компонентов ОАО, а другой - для объявления компонентов служб и т.д.). Определим для примера компонент provider с помощью конфигурационного класса AppConfigFour. Этот компонент может быть доступен из другого конфигурационного класса благодаря импорту компонентов Spring Beans, определенных в данном классе. И для этого целевой конфигурационный класс AppConfigThree снабжается аннотацией @Import, как показано ниже.

@Configuration  
@Import(AppConfigFour.class)  
public class AppConfigThree {

@Configuration  
@ComponentScan(basePackages={"com.apress.prospring5.ch4.annotated"})  
public class AppConfigFour {  
  
}

**Смешанное конфигурирование в Spring**

Чтобы импортировать объявления компонентов Spring Beans из ХМL-файла конфигурации, достаточно воспользоваться аннотацией @ImportResource.

@Configuration  
@ImportResource(value="classpath:spring/app-context-xml-01.xml")  
public class AppConfigFive {

**Профили**

профиль вынуждает Spring конфигурировать только тот экземпляр контекста типа ApplicationContext, который был определен, когда указанный профиль стал активным.

@Configuration  
@Profile("kindergarten")  
public class KindergartenConfig {  
  
 @Bean  
 public FoodProviderService foodProviderService(){  
 return new FoodProviderServiceImpl();  
 }  
}

public class ProfileJavaConfigExample {  
 public static void main(String... args) {  
 GenericApplicationContext ctx = new AnnotationConfigApplicationContext(  
 KindergartenConfig.class,  
 HighschoolConfig.class);

лишь один из профилей будет фактически использован для получения экземпляра контекста типа ApplicationContext в зависимости от значения пapaмeтpa-Dspring .profiles. active настройки виртуальной машины JVM.

Вместо параметра -Dspring.profiles.active виртуальной машины JVM можно также воспользоваться специальной аннотацией @ActiveProfiles для конфигурирования профилей, хотя это можно сделать только в тестовых классах.

@RunWith(SpringJUnit4ClassRunner.class)  
@ContextConfiguration(classes={KindergartenConfig.class, HighschoolConfig.class})  
@ActiveProfiles("kindergarten")  
public class ProfilesJavaConfigTest {  
  
 @Autowired FoodProviderService foodProviderService;  
  
 @Test  
 public void testProvider(){

Профили в Spring предоставляют разработчикам приложений еще один способ конфигурирования запуска приложений на выполнение, который обычно реализуется в инструментальных средствах сборки (например, в Maven, где имеется специальная поддержка профилей). Инструментальные средства сборки упаковывают нужные файлы конфигурации/свойств в архив Java (формата JAR или WAR в зависимости от типа приложения), исходя из переданных им аргументов, а затем развертывают его в целевой среде. Разработчики приложений могут самостоятельно определять профили и активизировать их программно или с помощью специального параметра виртуальной машины JVM.

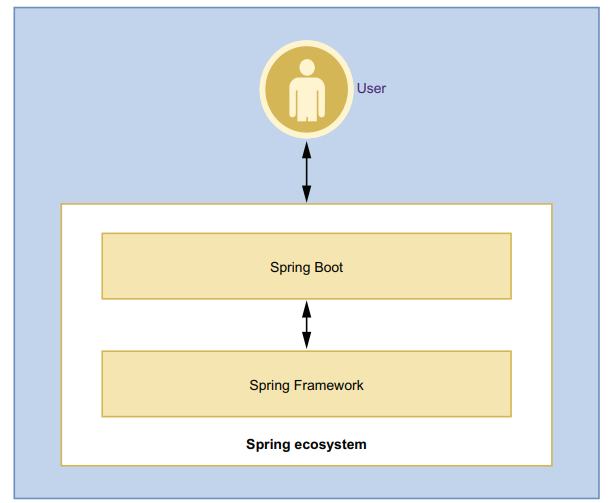
**Абстракция через интерфейсы Environment и PropertySource**

Для установки активного профиля необходимо обратиться к интерфейсу Environment, обеспечивающему уровень абстракции для инкапсуляции среды выполняемого приложения Spring.

# Spring Boot

## What is Spring Boot

Spring Boot was released in April 2014 to reduce some of the burdens of developing a Java web application. It allowed developers to focus more on the business logic rather than the boilerplate technical code and associated configurations. Spring Boot intends to create Spring-based, production-ready, standalone applications with little configuration changes on behalf of the application developer. It takes an opinionated view of the Spring Framework, so the application developers can quickly get started with what they need. It provides an additional layer between the Spring Framework for the user to simplify certain configuration aspects.



Spring Boot has several notable features that make it stand out from the crowd of other frameworks:

Fast bootstrapping—One of the primary goals of Spring Boot is to provide a fast startup experience in Spring application development. With Spring Boot, you can generate an application by specifying the dependencies you need in your application, and Spring Boot takes care of the rest.

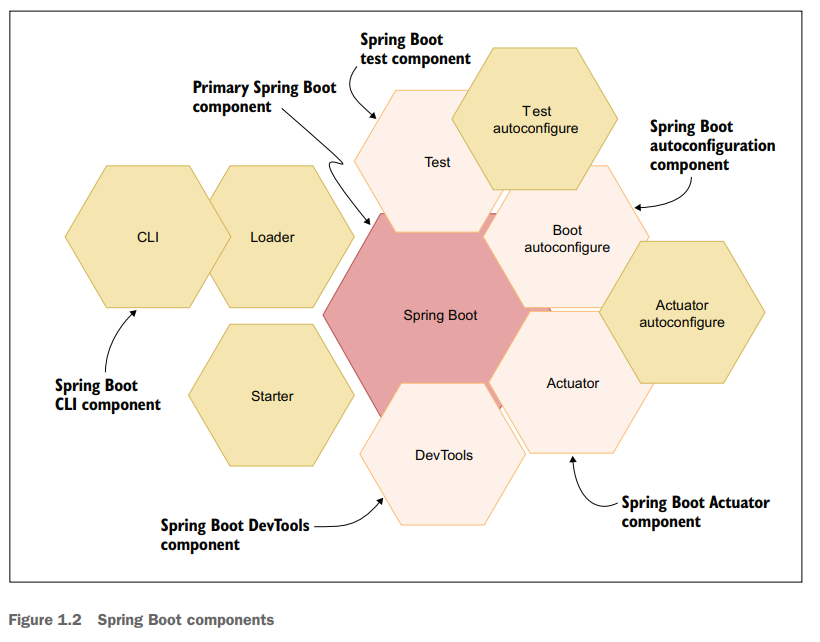
Autoconfiguration—Spring Boot automatically configures the bare minimum components of a Spring application. It does this based on the presence of the JAR files in the classpath or properties configured in the various property files. For instance, if Spring Boot detects the presence of a database driver JAR file (e.g., H2 in-memory database JAR) in the classpath, it automatically configures the corresponding data source to connect to the database.

Opinionated—Spring Boot is opinionated. It automatically configures several components to start with a Spring application. Spring Boot does this with a set of starter dependencies. A starter dependency targets a specific area of application development and provides the related dependencies. For example, if you need to develop a web application, you can configure the spring-boot-starterweb dependency, which ensures that all related dependencies for developing a web application, such as spring-web and spring-webmvc, are available in the application classpath.

Standalone—Spring Boot applications embed a web server, so they can run standalone and do not necessarily require an external web or application server. This enables Spring Boot applications to be packaged as an executable JAR file and run with the java -jar command. This also allows Spring Boot applications to be easily containerized and candidates for cloud-native application development.

Production-ready—Spring Boot provides several useful production-ready features out of the box to monitor and manage the application once it is pushed to production, such as health checks, thread dumps, and other useful metric

## Spring Boot components



spring-boot—This is the primary Spring Boot component that provides support to other components. For example, it contains the SpringApplication class, which contains several static methods to create a standalone Spring Boot application. It also provides support for embedded web servers (e.g., Tomcat) and supports externalized application configurations (e.g., database details of your application), etc.

spring-boot-autoconfigure—This component provides the necessary support for the automatic configuration of a Spring Boot application. Spring Boot autoconfiguration guesses and configures the spring beans based on the dependencies present in classpath and the properties configured.

spring-boot-starters—Starters are a set of prepackaged dependency descriptors provided for developer convenience. A Spring Boot starter assists in providing a set of Spring and related technologies to the developer, which otherwise, the developer needs to manage themselves.

spring-boot-CLI—This is a developer-friendly command-line utility that compiles and runs groovy codes. It can also watch files for changes, so you do not need to restart your application on modifications. This CLI tool exempts you from the need for dependency management tools, such as Maven or Gradle.

spring-boot-actuator—This component provides the actuator endpoints to interact with, monitor, and audit a Spring Boot application. An actuator in Spring Boot can be managed through JMX or HTTP endpoints. Spring Boot provides a predefined list of actuator endpoints that cover a range of application aspects. If that does not satisfy your need, you can also create your custom actuator endpoints specific to your application. Spring Boot actuator also provides configurations to let you decide which actuator endpoints you want to enable and provides several means to secure them from unauthorized access.

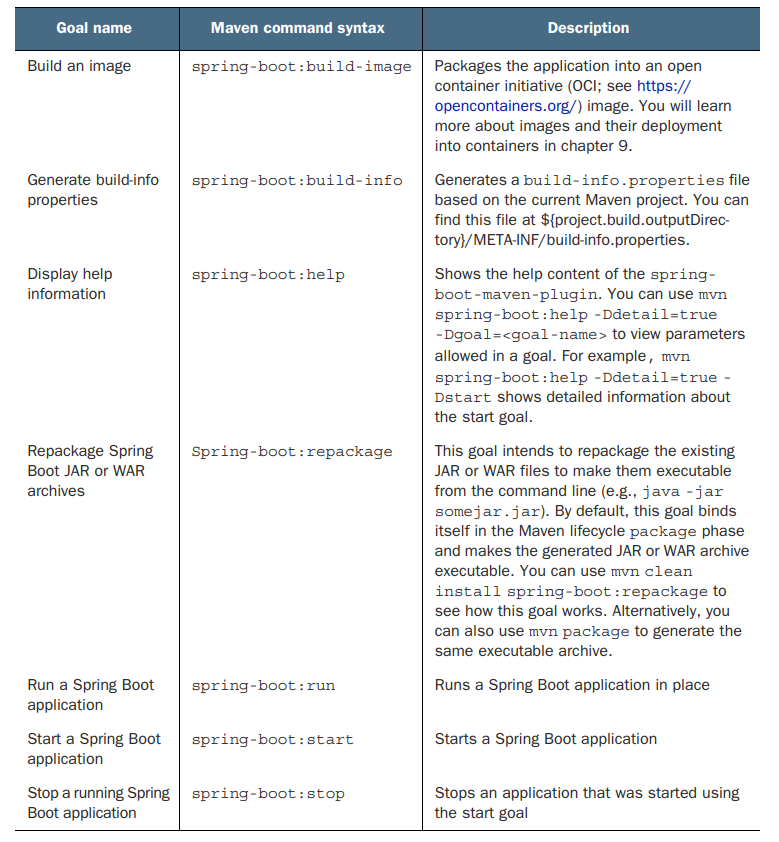
spring-boot-actuator-autoconfigure—This component provides support to autoconfigure the actuator endpoints based on the classpath. For instance, if the Micrometer (https://micrometer.io) dependency is present in the classpath, Spring Boot automatically configures the MetricsEndpoint.

spring-boot-test—This module contains annotations and methods to write test cases for the Spring Boot application. ϒ spring-boot-test-autoconfigure—This component supports the autoconfiguration of the test cases of your application.

spring-boot-loader—This component allows a Spring Boot application to be packaged as a single fat JAR file, including all dependencies and the embedded web servers that can be run standalone. You don’t use this module independently; instead, it is used along with Maven or Gradle plugins.

spring-boot-devtools—This component contains an additional developer toolkit for a smooth development experience of Spring Boot applications. The toolkit includes features such as automatic detection of application code changes and LiveReload server to automatically refresh any HTML changes to the browser.

List of Spring Boot Maven plugin goals



### THE SPRING BOOT MAIN CLASS

In general, to run a web application, you build and package the application components in a WAR or EAR archive file and deploy it into a web (e.g., Apache Tomcat) or application server (e.g., Red Hat JBoss). Spring Boot simplifies this process to a certain degree. It does not enforce you to build a WAR or EAR file of your application. Instead, it lets you run the Spring Boot application like a regular Java application using a conventional main() method.

the class in the generated Java file is annotated with the @SpringBootApplication annotation. This is a convenient annotation that consists of three annotations: @EnableAutoConfiguration, @ComponentScan, and @SpringBootConfiguration, each of which is performing a specific task in the application.

Let’s understand these annotations based on their actions:

* @EnableAutoConfiguration—Spring Boot provides several @Enable\* annotations to enable specific features in your Spring Boot application. The @EnableAutoConfiguration annotation provides the necessary support for Spring Boot to autoconfigure your application based on the JAR dependencies present in the application classpath.
* @ComponentScan—Provides support to scan the packages for Spring components in the application. A component in Spring is a Java bean that is managed by Spring and annotated with the @Component, @Bean, or specialized component annotations. With the presence of @ComponentScan annotation, the Spring Boot application scans for all components present in the root package and subpackages under it to manage their lifecycle. The key point to remember with ComponentScan is that the scan starts from a root package and continues to all child packages. Thus, if you have packages that are not in the root or its subpackage, none of those components will be scanned by the component scan.
* @SpringBootConfiguration—This annotation indicates that the annotated class provides the Spring Boot application configuration. It is meta-annotated with Spring @Configuration annotation so that the configurations in the annotated class can be found automatically by Spring Boot. Thus, the beans defined in this main class can be autodetected and loaded by Spring.

Also, note that the Spring Boot application main class needs to be in your application root package, as the @SpringBootApplication annotation is configured in this class. @SpringBootApplication annotation uses the root package as the base package. This base package and all other subpackages are automatically scanned by Spring Boot to load Spring components (e.g., classes configured with @Component, @Configuration, and other Spring annotations) and other types.

The next, and final, component is the use of the SpringApplication class in the generated Java file. This class is provided by Spring Boot to conveniently bootstrap a Spring Boot application. Most of the time, you’ll use the static run() method of SpringApplication to bootstrap and launch your application. Spring Boot performs several activities while it executes the run() method:

1. Creates an instance of an ApplicationContext based on the libraries present in the classpath
2. Registers a CommandLinePropertySource to expose command line arguments as Spring properties
3. Refreshes the ApplicationContext created at step 1 to load all singleton beans
4. Triggers the ApplicationRunners and CommandRunners configured in the application

The SpringApplication class attempts to create an instance of ApplicationContext based on the JAR dependencies present in the classpath. A Spring Boot web application can be either Servlet-based or reactive type. Leveraging Spring’s class loading techniques, and based on the availability of the classes in the classpath, Spring deduces the current application’s type. Once the application type is known, Spring Boot applies the below strategy to load the application context:

1. If the application is identified as a Servlet-based web application, Spring Boot attempts to create an instance of AnnotationConfigServletWebServerApplicationContext class.
2. Alternatively, if the application is reactive type, Spring Boot creates an instance of the AnnotationConfigReactiveWebServerApplicationContext class.
3. If the application is neither a Servlet-based nor a reactive application, Spring Boot attempts to create an instance of AnnotationConfigApplicationContext class.

### Creating an executable JAR file

The easiest way of creating an executable JAR file from your Spring Boot project is by using the mvn package command. Recall that you’ve selected the packaging type while generating the project. Based on the selection, a JAR file is created in the project’s target directory. The generated JAR file can be executed with the java -jar command from your command line to start the application.

By default, the Maven package goal does not generate an executable JAR or WAR file on its own. It’s the spring-boot-maven-plugin’s repackage goal that binds itself in the package phase and prepares the executable file.

### Spring Boot startup events

Spring framework’s event management mechanism promotes decoupling event publishers and subscribers in an application. It allows you to subscribe to the framework’s built-in events as well as define your custom events.

The Spring Boot framework also provides several built-in events that you can subscribe to perform certain actions. For instance, there might be a requirement that you need to invoke an external REST API if your Spring Boot application initializes completely. In this section, we’ll introduce several Spring Boot events, which are published at various stages of an application startup and initialization:

* ApplicationStartingEvent—Published at the beginning of the application startup once the Listeners are registered. Spring Boot’s LoggingSystem uses this event to perform any action that needs to be taken up before application initialization.
* ApplicationEnvironmentPreparedEvent—Published when the application is starting up and the Environment is ready for inspection and modification. Spring Boot internally uses this event to preinitialize several services, such as MessageConverter, ConversionService, Initialize Jackson, and others.
* ApplicationContextInitializedEvent—Published when the ApplicationContext is prepared, ApplicationContextInitializers are executed, but none of the bean definitions are loaded. This event can be used to perform a task before beans are initialized in the Spring container.
* ApplicationPreparedEvent—Published when the ApplicationContext is prepared, bean definitions are loaded but not refreshed. The Environment is ready for use at this stage.
* ContextRefreshedEvent—Published when the ApplicationContext is refreshed. This event comes from Spring—not Spring Boot. This event does not extend SpringApplicationEvent. The Spring Boot ConditionEvaluationReportLoggingListener listens to this event and prints the autoconfiguration report once this event is published.
* WebServerInitializedEvent—Published when the webserver is ready. This event has two variants based on the type of the application: ServletWebServerInitializedEvent for Servlet-based applications and ReactiveWebServerInitializedEvent for reactive applications. This event does not extend SpringApplicationEvent.
* ApplicationStartedEvent—Published when the ApplicationContext is refreshed but before the ApplicationRunner and CommandLineRunners are called.
* ApplicationReadyEvent—Published by SpringApplication to indicate the application is ready to service requests. It is not advised to change the internal state of the application, as all application initialization steps are finished.
* ApplicationFailedEvent—Published when there are some exceptions, and the application has failed to start. This event is useful to perform tasks like script execution or notifying startup failures.

These events are useful if you need programmatic control on the application startup behavior. The easiest approach is to subscribe to these events and take necessary actions. For instance, if you need to modify any parameter in the Environment, you can subscribe to ApplicationEnvironmentPreparedEvent and do so. Spring Boot uses these events internally to initialize several components of the application.

Let’s discuss different approaches to subscribe to these events. The easiest way to use Spring Framework’s @EventListener annotation. For instance, to listen to the ApplicationReadyEvent, you can use the code snippet shown in the following listing.

**@EventListener(ApplicationReadyEvent.class)** public void applicationReadyEvent(ApplicationReadyEvent applicationReadyEvent) { System.out.println("Application Ready Event generated at "+new Date(applicationReadyEvent.getTimestamp())); }

Although @EventListener works well in most circumstances, it does not work for events that are published very early in the application start-up, such as ApplicationStartingEvent and ApplicationEnvironmentPreparedEvent. In this section, we’ll discuss two additional approaches to listening to events in a Spring Boot application.

Using SpringApplication - Typically, in the generated Spring Boot project, the application class invokes the static run() method of SpringApplication to start the application. However, the SpringApplication class also provides several setter methods to customize the application startup behavior. For instance, it lets you add ApplicationContextInitializer, set ApplicationListener, and many others with the various setter methods. To use SpringApplication to listen to events, you can create an appropriate ApplicationListener class and implement the onApplicationEvent() method.

Using the spring.factories file - The spring.factories file provides you with an extension point in the Spring Boot framework to configure and customize certain application features. Spring Boot provides an approach to configure certain custom components, such as the ApplicationListener, through this file. The spring.factories is located inside the META-INF folder, which is located inside the src\main\resources folder. The following listing shows a sample spring.factories file.

### Spring Boot actuator

Spring Boot actuator lets you monitor and interact with your Spring Boot application. For instance, you can perform a health check in an application to determine whether the application is up. Besides, you can also capture the thread dump or heap dump of your application to perform a variety of analyses. To enable the Spring Boot actuator, you need to include the spring-boot-starter-actuator dependency in the pom.xml.

By default, Spring Boot exposes the /actuator as the base endpoint to access the other endpoints. Only the /health and /info endpoints are enabled for HTTP by default.

### Managing configurations

Spring Boot provides several approaches to let you externalize application configurations without altering the application source code. The various approaches include property files, YAML files, environment variables, and command-line arguments.

# Databases – general

## Ссылки

Transactional:

<https://medium.com/@rameez.s.shaikh/spring-boot-transaction-tutorial-understanding-transaction-propagation-ad553f5d85d4>

<https://hellokoding.com/spring-transactional/>

<https://hellokoding.com/transaction-isolation-levels/>

<http://akorsa.ru/2016/08/rukovodstvo-dlya-nachinayushhih-acid-i-tranzaktsii-bd/>

<http://akorsa.ru/2017/01/sovety-i-oshibki-v-spring-transactions/>

<http://akorsa.ru/2016/08/kak-na-samom-dele-rabotaet-transactional-spring/>

## DAO

Проектный шаблон DАО (Data Access Object - объект доступа к данным) предназначен для того, чтобы отделить низкоуровневые прикладные интерфейсы АРI или операции доступа к данным от высокоуровневых услуг бизнес-логики. Для реализации проектного шаблона DАО требуются следующие компоненты.

• Интерфейс DAO. Определяет стандартные операции, выполняемые над отдельным объектом или несколькими объектами модели.

• Реализация DAO. Это класс, в котором предоставляется конкретная реализация интерфейса ОАО. Как правило, в нем применяется соединение через JOBC или источник данных для манипулирования отдельным объектом или несколькими объектами модели.

• Объекты модели, иначе называемые объектами данных или сущностями. Это результат простого преобразования объекта POJO в запись таблицы базы данных.

## @Transactional

### ACID

**Транзакции** — это группа операций на чтение/запись, выполняющихся только если все операции из группы успешно выполнены.

ACID stands for Atomic, Consistency, Isolation, and Durability.

По сути транзакции характеризуются следующими четырьмя свойствами (также известными как ACID):

* Атомарность - позволяет объединить единые и независимые операции в «единицу работы», которая работает по принципу «все-или-ничего», успешно выполняющаяся только если все содержащиеся операции успешно выполнятся.
* Консистентность (согласованность) - означает что все требования уникальности были соблюдены для каждой совершенной транзакции. Это подразумевает, что требования по всем ключам (primary и foreign key), типам данных, триггерам успешно пройдены и не было найдено нарушений требования уникальности.
* Изоляция - Во время выполнения транзакции параллельные транзакции не должны оказывать влияние на её результат.
* Долговечность - Успешная транзакция должна всегда изменять состояние системы и прежде чем закончить ее изменения состояния сохраняются в лог транзакций. Если Ваша система внезапно выключится или возникнет перебой с электричеством, тогда все незавершенные транзакции можно воспроизвести.

В реляционной БД каждое SQL выражение должно исполняться в пределах транзакции. Без явного определения границ транзакции БД будет использовать неявную транзакцию, которая покрывает каждое SQL выражение. Неявная транзакция начинается до того как выражение запустится и заканчивается (коммит или откат) после того как выражение выполнится. Режим неявной транзакции известен так же как режим autocommit.

### Уровни изолированности

Стандарты SQL определяют 4 уровня изолированности:

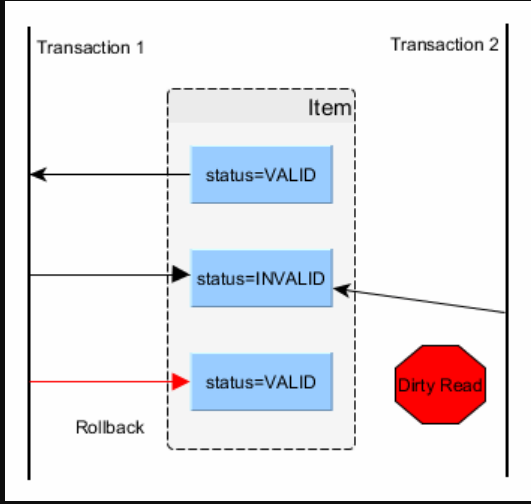
* READ\_UNCOMMITTED
* READ\_COMMITTED
* REPEATABLE\_READ
* SERIALIZABLE

Все уровни, кроме SERIALIZABLE уровня подвержены аномалии данных (феномен), которые могут произойти согласно следующей схеме:

|  |  |  |  |
| --- | --- | --- | --- |
| **Isolation Level** | **Dirty read** | **Non-repeatable read** | **Phantom read** |
| READ\_UNCOMMITTED — чтение незавершенных транзакций | да | да | да |
| READ\_COMMITTED — чтение завершенных транзакций | нет | да | да |
| REPEATABLE\_READ — повторяемое чтение | нет | нет | да |
| SERIALIZABLE — последовательное чтение | нет | нет | нет |

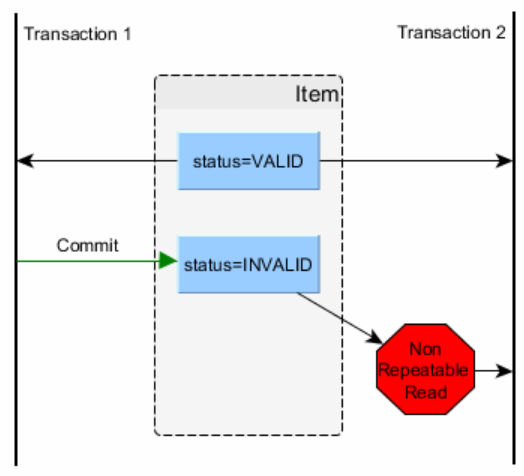
**Dirty Read («Грязное» чтение) -** происходит когда транзакция может прочитать незаконченные изменения некоторой другой выполняемой транзакции. Явление случается из-за того что нет блокировки предотвращающей это.

A dirty read occurs when a transaction is allowed to read data from a row that has been modified but not yet committed by another running transaction. Say T2 modifies a row. T1 then reads that row before T2 performs a COMMIT. If T2 then performs a ROLLBACK, T1 will have read a row that was never committed and that may thus be considered to have never existed.



**Non-repeatable read (Неповторяющееся чтение)** - Неповторяющееся чтение проявляется, когда повторные чтения в рамках одной транзакции дают разные результаты из-за параллельных транзакций, которые обновили запись, которую мы читаем. Это нежелательно, так как мы в конечном итоге используем устаревшие данные. Это предотвращается размещением разделяемой блокировки (блокировка на чтение) на чтение записи на все протяжении выполнения текущей транзакции.

A non-repeatable read occurs, when during the course of a transaction, a row is retrieved twice and the values within the row differ between reads. Say T1 reads a row. T2 then modifies or deletes that row and performs a COMMIT. If T1 then attempts to reread the row, it may receive the modified value or discover that the row has been deleted.



**Phantom read (Фантомное чтение)** - Фантомное чтение происходит когда вторая транзакция вставляет запись, которая подходит по критериям предыдущей выборки первой транзакции. В результате получится, что одни и те же выборки в первой транзакции дают разные множества строк. Это предотвращается с помощью использования блокировки по диапазону или[блокировки предикатами](http://research.microsoft.com/en-us/um/people/gray/papers/On%20the%20Notions%20of%20Consistency%20and%20Predicate%20Locks%20in%20a%20Database%20System%20CACM.pdf).

A phantom read occurs when, in the course of a transaction, two identical queries are executed, and the collection of rows returned by the second query is different from the first. Say T1 reads the set of rows that satisfy some search condition S. T2 then adds or removes rows that satisfy S used by T1. If T1 repeats the initial read with the same S, it obtains a different collection of rows.

### **Уровни изолированности по умолчанию**

Даже если SQL стандарт обязывает использовать уровень изоляции SERIALIZABLE, большинство СУБД используют разный уровень по умолчанию.

|  |  |
| --- | --- |
| **СУБД** | **Уровень по умолчанию** |
| Oracle | READ\_COMMITTED |
| MySQL | REPEATABLE\_READ |
| Microsoft SQL Server | READ\_COMMITTED |
| PostgreSQL | READ\_COMMITTED |

Обычно, READ\_COMMITED является правильным выбором, поскольку даже SERIALIZABLE не может защитить Вас от потери обновлений во время выполнения чтения/записей в разных транзакций и веб-реквестов.

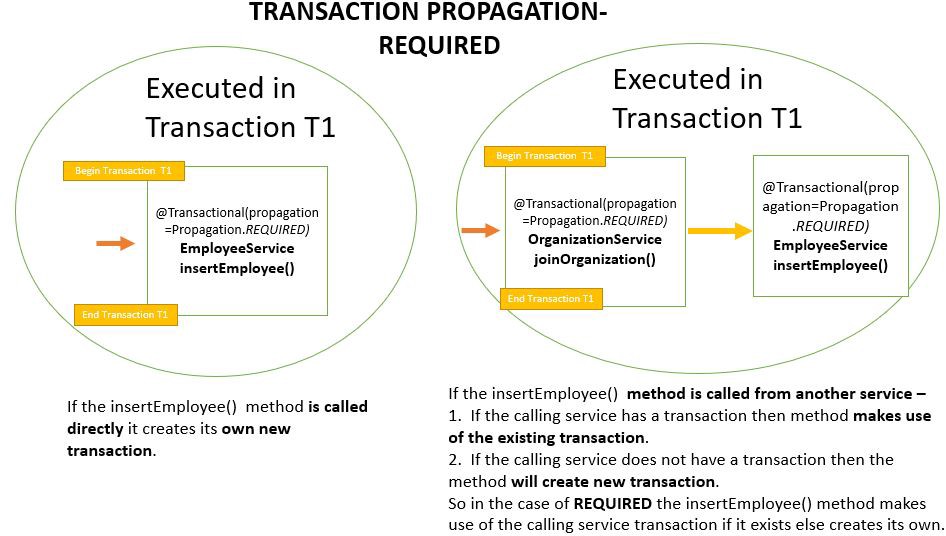
### Transaction Propagation

Any application involves a number of services or components making a call to other services or components. Transaction Propagation indicates if any component or service will or will not participate in transaction and how will it behave if the calling component/service already has or does not have a transaction created already.

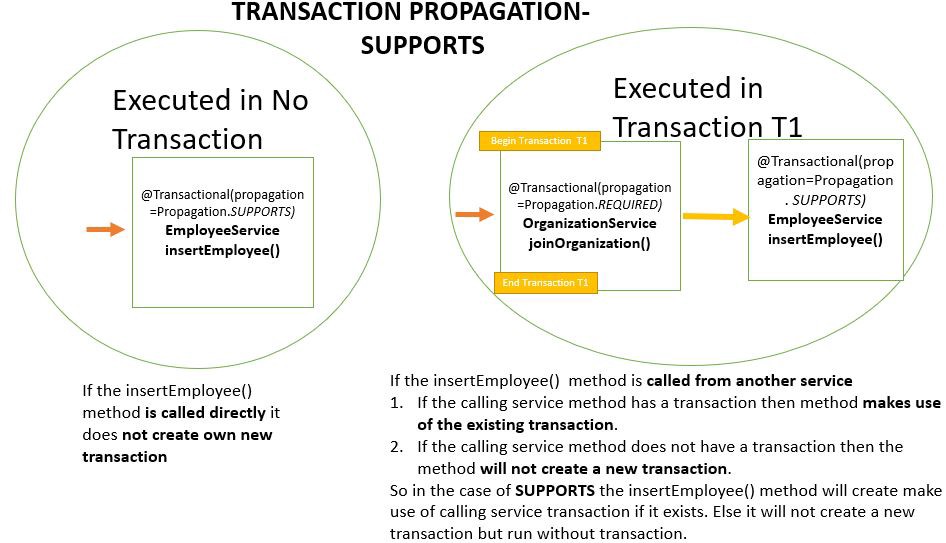
There are six types of Transaction Propagations:

* **REQUIRED**
* **SUPPORTS**
* **NOT\_SUPPORTED**
* **REQUIRES\_NEW**
* **NEVER**
* **MANDATORY**

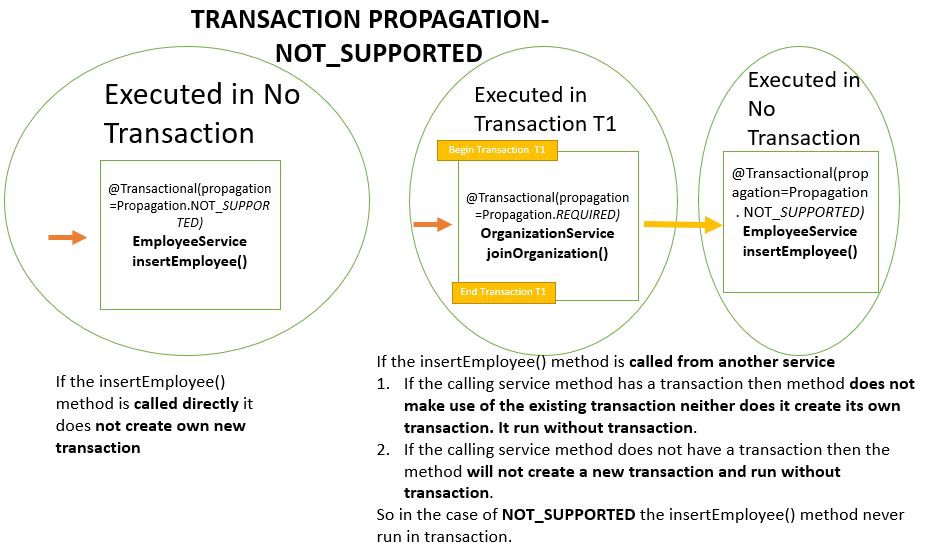
Transaction Propagation — REQUIRED (Default Transaction Propagation)



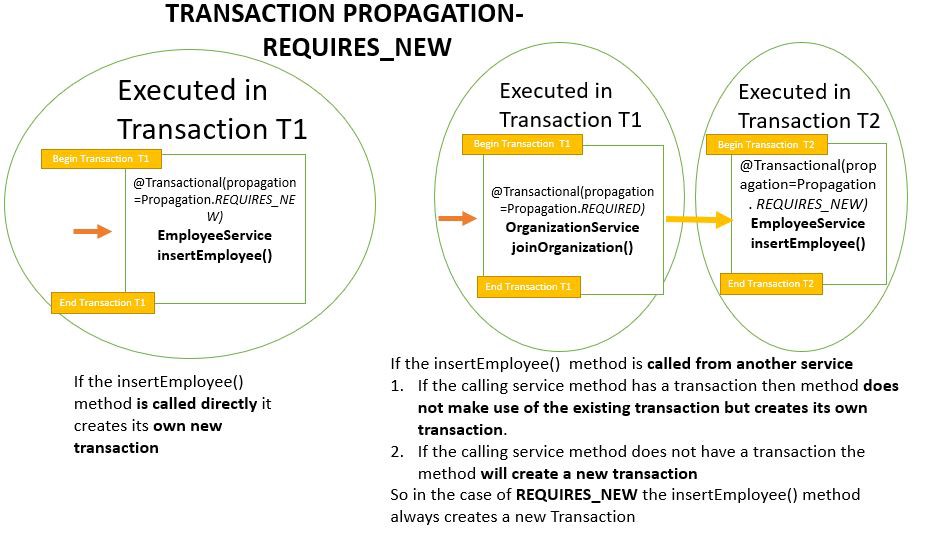
Transaction Propagation — SUPPORTS



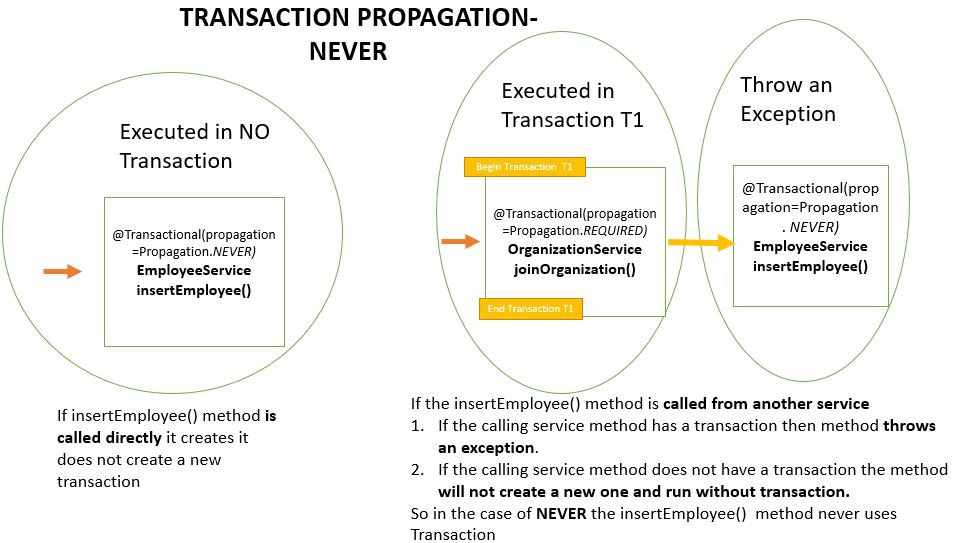
Transaction Propagation — NOT\_SUPPORTED



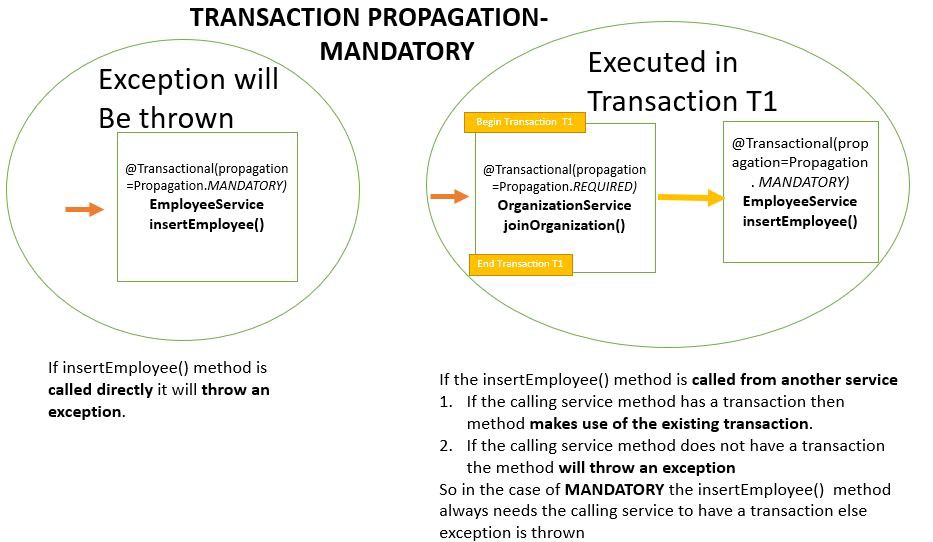
Transaction Propagation — REQUIRES\_NEW



Transaction Propagation — NEVER



Transaction Propagation — MANDATORY



# Аспектно-ориентированное программирование

## Основные понятия АОП

**Точка соединения (joinpoint).** Это вполне определенная точка во время выполнения приложения. Характерными примерами точек соединения служат вызов метода, обращение к методу, инициализация класса и получение экземпляра объекта. Точки соединения относятся к базовым понятиям АОП и определяют места в приложении, где можно вставлять дополнительную логику средствами АОП.

**Совет (advice).** Фрагмент кода, который должен выполняться в отдельной точке соединения, представляет собой совет, определенный в методе класса.

**Срез (pointcut)** - это совокупность точек соединения, предназначенная для определения момента, когда следует выполнить совет. Создавая срезы, можно приобрести очень точный контроль над тем, как применять совет к компонентам приложения.

**Аспект (aspect)** - это сочетание совета и срезов, инкапсулированных в классе. Такое сочетание приводит в итоге к определению логики, которую следует внедрить в приложение, а также тех мест, где она должна выполняться.

**Связывание (weawing)** - это процесс вставки аспектов в определенном месте прикладного кода. Для решений АОП на стадии компиляции связывание обычно делается статически во время сборки. А для решений АОП на стадии выполнения связывание происходит динамически во время выполнения.

**Цель (target)** - это объект, поток исполнения которого изменяется каким-нибудь процессом АОП. Зачастую целевой объект обозначается как снабженный советом объект.

**Введение (introduction)** - это процесс, посредством которого можно изменить структуру объекта, введя в него дополнительные методы или поля. Введение можно применять в АОП для того, чтобы реализовать в любом объекте определенный интерфейс, не прибегая к явной его реализации в классе данного объекта.

## Типы АОП

Имеются два разных типа АОП - статическое и динамическое.

**Реализация статического АОП**

При статическом АОП связывание образует еще одну стадию в процессе сборки приложения. Недостаток такого механизма заключается в том, что любые изменения, вносимые в аспекты, даже если они касаются только ввода еще одной точки соединения, требуют перекомпиляции всего приложения в целом.

**Реализация динамического АОП**

процесс связывания происходит динамически во время выполнения. Добиться этого можно по-разному в зависимости от конкретной реализации, но, как станет ясно в дальнейшем, применяемый в Spring подход состоит в создании заместителя для всех целевых объектов, что дает возможность вызывать совет по мере надобности. Недостаток динамического АОП заключается в том, что оно обычно не выполняется так же хорошо, как и статическое АОП, хотя производительность при этом неуклонно растет. А главным преимуществом динамических реализаций АОП является простота, с которой можно изменить целый ряд аспектов в приложении, не прибегая к повторной компиляции основного прикладного кода.

**АОП в Spring**

Реализацию АОП в Spring можно представить как состоящую из двух логических частей. Первая часть содержит ядро АОП, обеспечивающее полностью развязанные, чисто программные функциональные возможности АОП, иначе называемые Spring АОР АРI, т.е. прикладным интерфейсом АОП в Spring. А вторая часть реализации АОП содержит ряд служб каркаса, упрощающих применение АОП в приложениях.

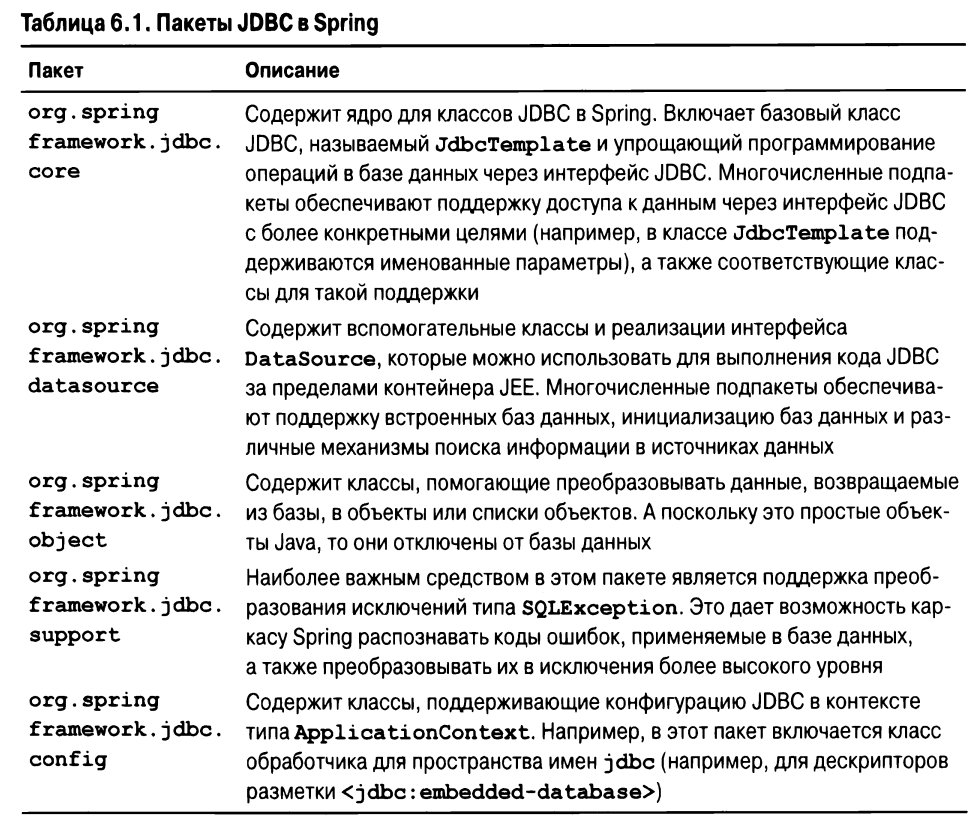
# Spring - JDBC

## ССЫЛКИ

<https://www.logicbig.com/tutorials/spring-framework/spring-data-access-with-jdbc.html>

Spring в действии 6 изд [2022] Крейг Уоллс, глава 3

## Инфраструктура JDBC в Spring



## Подключение через DataSource

Для управления автоматическим подключением к базе данных можно воспользоваться каркасом Spring, предоставив компонент Spring Bean, реализующий интерфейс javax. sql. DataSource. Отличие между интерфейсами DataSource и Connection заключается в том, что интерфейс DataSource предоставляет реализации Connection и управляет ими. Можно использовать:

DriverManagerDataSource – используется в основном для тестовых целей;

BasicDataSource (<http://commons.apache.org/dbcp/>) – используется в реальных приложениях.

Нужно создать соединение (Connection), запрос (PreparedStatement) и результирующий набор (ResultSet), а затем закрыть соединение, запрос и результирующий набор. Это делается с использованием try - catch (SQLException). SQLException  – это контролируемое исключение, которое требует обработки в блоке catch.

В Spring поддерживается также встроенная база данных **(**EmbeddedDatabase**)**, которая автоматически запускается и становится доступной приложению в виде источника данных типа DataSource. Встроенные базы данных можно сконфигурировать и с помощью конфигурационных классов Java. Для этой цели, в частности, служит класс EmbeddedDatabase Builder, в котором сценарии создания и загрузки базы данных указываются в качестве аргументов при получении экземпляра класса EmbeddedDatabase, реализующего интерфейс DataSource.

## Подключение через JdbcTemplate

Такой вариант проще, чем DataSource, т.к. здесь не создаются ни запросы, ни соединения, после завершения не выполняется никаких заключительных операций, полностью отсутствует обработка исключений. Остается код, ориентированный исключительно на выполнение запроса.

Запросы JdbcTemplate содержатся в Dao (Repository), DataSource – в файле конфигурации.

Класс JdbcTemplate представляет ядро поддержки интерфейса JDBC в Spring. Класс JdbcTemplate позволяет выдать базе данных оператор SQL любого типа, а также возвратить результат любого типа.

**Инициализация объекта типа JdbcTemplate в классе DAO**

Сначала нужно подготовить объект класса JdbcTemplate к применению в классе DAO. Как правило, достаточно сконструировать объект класса JdbcTemplate, передав его конструктору объект источника данных, который должен быть внедрен каркасом Spring в класс DАО. Как правило, инициализация объекта типа JdbcTemplate происходит в методе setDataSource (). И как только источник данных будет внедрен средствами Spring, объект типа JdbcTemplate будет также инициализирован и готов к применению.

**Извлечение одиночного значения средствами класса JdbcTemplate**. Пример:

@Override public String findNameById(Long id) {  
 return jdbcTemplate.queryForObject(  
 "select first\_name || ' ' || last\_name from singer where id = ?",  
 new Object[]{id}, String.class);  
}

В приведенном выше примере кода для извлечения имени певца используется метод queryForObject () из класса JdbcTemplate. В качестве первого аргумента указывается строка с оператором SQL, а в качестве второго аргумента - параметры, передаваемые оператору SQL для привязки параметров в формате массива объектов. Последний аргумент обозначает возвращаемый тип (в данном случае - String).

**Применение именованных параметров запроса с помощью класса NamedParameterJdbcTemplate**

Можно пользоваться именованными параметрами запроса, чтобы гарантировать точную привязку каждого параметра, это обеспечивает расширение класса JdbcTemplate nод названием NamedParameterJdbcTemplate (из nакета org.springframework.jdbc.core.namedparam).

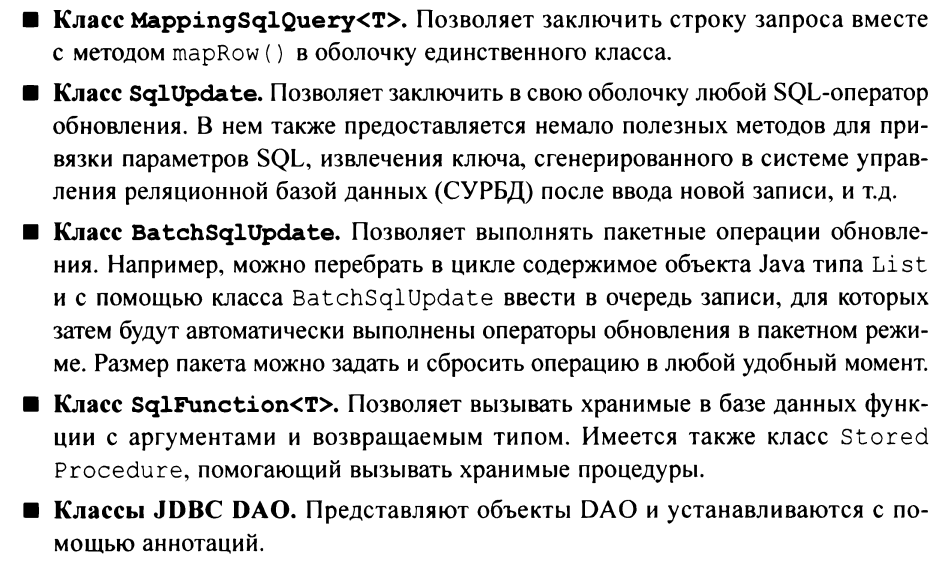
public class JdbcSingerDao implements SingerDao, InitializingBean {  
  
 private NamedParameterJdbcTemplate namedParameterJdbcTemplate;  
  
 @Override  
 public String findNameById(Long id) {  
 String sql = "SELECT first\_name ||' '|| last\_name FROM singer WHERE id = :singerId";  
 Map<String, Object> namedParameters = new HashMap<>();  
 namedParameters.put("singerId", id);  
 return namedParameterJdbcTemplate.queryForObject(sql,  
 namedParameters, String.class);  
 }  
  
 public void setNamedParameterJdbcTemplate(NamedParameterJdbcTemplate namedParameterJdbcTemplate) {  
 this.namedParameterJdbcTemplate = namedParameterJdbcTemplate;  
 }

**Извлечение объектов предметной области с помощью интерфейса RowМapper**

Вместо извлечения одиночного значения обычно требуется запрашивать одну или несколько строк из таблицы базы данных, а затем преобразовать каждую строку в соответствующий объект предметной области. В каркасе Spring предоставляется интерфейс RowMapper (из пакета org.springframework.jdbc.core), предоставляющий простой способ преобразования результирующего набора, получаемого через интерфейс JDBC, в объекты POJO.

public class JdbcSingerDao implements SingerDao, InitializingBean {  
  
 private NamedParameterJdbcTemplate namedParameterJdbcTemplate;  
  
 public void setNamedParameterJdbcTemplate(NamedParameterJdbcTemplate namedParameterJdbcTemplate) {  
 this.namedParameterJdbcTemplate = namedParameterJdbcTemplate;  
 }  
  
 @Override  
 public List<Singer> findAll() {  
 String sql = "select id, first\_name, last\_name, birth\_date from singer";  
 return namedParameterJdbcTemplate.query(sql, (rs, rowNum) -> {  
 Singer singer = new Singer();  
 singer.setId(rs.getLong("id"));  
 singer.setFirstName(rs.getString("first\_name"));  
 singer.setLastName(rs.getString("last\_name"));  
 singer.setBirthDate(rs.getDate("birth\_date"));  
 return singer;  
 });  
 }  
  
 @Override  
 public String findNameById(Long id) {  
 String sql = "SELECT first\_name || ' ' || last\_name FROM singer WHERE id = :singerId";  
 Map<String, Object> namedParameters = new HashMap<>();  
 namedParameters.put("singerId", id);  
 return namedParameterJdbcTemplate.queryForObject(sql,  
 namedParameters, String.class);  
 }

**Классы Spring, моделирующие операции в JDBC**



**Выборка данных с помощью класса МappinqSqlQuery**

Для моделирования операций обработки запросов в Spring предоставляется класс MappingSqlQuery. По существу, экземпляр класса MappingSqlQuery конструируется с помощью источника данных и строки запроса, а затем реализуется метод mapRow ( ) для преобразования каждой записи из результирующего набора в соответствующий объект предметной области.

Public class SelectAllSingers extends MappingSqlQuery<Singer> {  
 private static final String *SQL\_SELECT\_ALL\_SINGER* =  
 “select id, first\_name, last\_name, birth\_date from singer”;  
  
 public SelectAllSingers(DataSource dataSource) {  
 super(dataSource, *SQL\_SELECT\_ALL\_SINGER*);  
 }  
  
 protected Singer mapRow(ResultSet rs, int rowNum) throws SQLException {  
 Singer singer = new Singer();  
  
 singer.setId(rs.getLong(“id”));  
 singer.setFirstName(rs.getString(“first\_name”));  
 singer.setLastName(rs.getString(“last\_name”));  
 singer.setBirthDate(rs.getDate(“birth\_date”));  
  
 return singer;  
 }  
}

**Обновление данных с помощью класса SqlUpdate**

public class UpdateSinger extends SqlUpdate {  
 private static final String *SQL\_UPDATE\_SINGER* =  
 “update singer set first\_name=:first\_name, last\_name=:last\_name, birth\_date=:birth\_date where id=:id”;

**Группирование операций с помощью класса BatchSqlUpdate - д**ля выполнения групповых операций служит класс BatchSqlUpdate.

# ORM

## Ссылки

https://www.javatpoint.com/spring-with-orm-frameworks

## Description

ORM is an acronym for Object/Relational mapping. It is a programming strategy to map object with the data stored in the database. It simplifies data creation, data manipulation, and data access.

ORM (англ. Object-Relational Mapping, рус. объектно-реляционное отображение, или преобразование) — технология программирования, которая связывает базы данных с концепциями объектно-ориентированных языков программирования, создавая «виртуальную объектную базу данных». The object-relational mapping is a mechanism which is used to develop and maintain a relationship between an object and the relational database by mapping an object state into the database column. It converts attributes of programming code into columns of the table. It is capable of handling various database operations easily such as insertion, updation, deletion, etc.

ORM избавляет программиста от написания большого количества кода, часто однообразного и подверженного ошибкам, тем самым значительно повышая скорость разработки. Кроме того, большинство современных реализаций ORM позволяют программисту при необходимости самому жёстко задать код SQL-запросов, который будет использоваться при тех или иных действиях (сохранение в базу данных, загрузка, поиск и т. д.) с постоянным объектом.





The ORM tool internally use advantages of using ORM over JDBC.

ORM has the following advantages over JDBC:

* Application development is fast.
* Management of transaction.
* Generates key automatically.
* Details of SQL queries are hidden the JDBC API to interact with the database.

## Mapping

### Mapping Directions

Mapping Directions are divided into two parts: -

* **Unidirectional relationship -** In this relationship, only one entity can refer the properties to another. It contains only one owing side that specifies how an update can be made in the database.
* **Bidirectional relationship -** This relationship contains an owning side as well as an inverse side. So here every entity has a relationship field or refer the property to other entity.

### Types of Mapping

Following are the various ORM mappings: -

* **One-to-one -** This association is represented by @OneToOne annotation. Here, instance of each entity is related to a single instance of another entity.
* **One-to-many -** This association is represented by @OneToMany annotation. In this relationship, an instance of one entity can be related to more than one instance of another entity.
* **Many-to-one -** This mapping is defined by @ManyToOne annotation. In this relationship, multiple instances of an entity can be related to single instance of another entity.
* **Many-to-many -** This association is represented by @ManyToMany annotation. Here, multiple instances of an entity can be related to multiple instances of another entity. In this mapping, any side can be the owing side.

## Spring and ORM

Spring provides API to easily integrate Spring with ORM frameworks such as Hibernate, JPA (Java Persistence API), JDO(Java Data Objects), Oracle Toplink and iBATIS.

There are a lot of advantage of Spring framework in respect to ORM frameworks. There are as follows:

* **Less coding is required**: By the help of Spring framework, you don't need to write extra codes before and after the actual database logic such as getting the connection, starting transaction, commiting transaction, closing connection etc.
* **Easy to test**: Spring's IoC approach makes it easy to test the application.
* **Better exception handling**: Spring framework provides its own API for exception handling with ORM framework.
* **Integrated transaction management**: By the help of Spring framework, we can wrap our mapping code with an explicit template wrapper class or AOP style method interceptor.

# Hibernate

## Ссылки

<https://www.javatpoint.com/hibernate-tutorial>

<https://habr.com/ru/post/320542/>

<https://www.baeldung.com/learn-jpa-hibernate>

## Description

Hibernate is a Java framework that simplifies the development of Java application to interact with the database. It is an open source, lightweight, ORM (Object Relational Mapping) tool. Hibernate implements the specifications of JPA (Java Persistence API) for data persistence.

Some of the databases supported by Hibernate are: DB2, MySQL, Oracle, PostgreSQL.

### Advantages of Hibernate Framework

**1) Open Source and Lightweight -** Hibernate framework is open source under the LGPL license and lightweight.

**2) Fast Performance** - The performance of hibernate framework is fast because cache is internally used in hibernate framework. There are two types of cache in hibernate framework first level cache and second level cache. First level cache is enabled by default.

**3) Database Independent Query** - HQL (Hibernate Query Language) is the object-oriented version of SQL. It generates the database independent queries. So you don't need to write database specific queries. Before Hibernate, if database is changed for the project, we need to change the SQL query as well that leads to the maintenance problem.

**4) Automatic Table Creation** - Hibernate framework provides the facility to create the tables of the database automatically. So there is no need to create tables in the database manually.

**5) Simplifies Complex Join** - Fetching data from multiple tables is easy in hibernate framework.

**6) Provides Query Statistics and Database Status** - Hibernate supports Query cache and provide statistics about query and database status.

## Hibernate Architecture

Hibernate architecture comprises of many interfaces such as Configuration, SessionFactory, Session, Transaction, etc.

The Hibernate architecture is categorized in four layers.

* Java application layer
* Hibernate framework layer
* Backhand api layer
* Database layer





### Core interfaces of Hibernate framework

* Configuration
* SessionFactory
* Session
* Query
* Criteria
* Transaction

### Description of elements of Hibernate Architecture

**SessionFactory** is a factory of session and client of ConnectionProvider. It holds second level cache (optional) of data. The org.hibernate.SessionFactory interface provides factory method to get the object of Session. SessionFactory is a thread-safe object.

**Session**. The session object provides an interface between the application and data stored in the database. It is a short-lived object and wraps the JDBC connection. It is factory of Transaction, Query and Criteria. It holds a first-level cache (mandatory) of data. The org.hibernate.Session interface provides methods to insert, update and delete the object. It also provides factory methods for Transaction, Query and Criteria. Session is not a thread-safe object.

**Transaction**. The transaction object specifies the atomic unit of work. It is optional. The org.hibernate.Transaction interface provides methods for transaction management.

**ConnectionProvider**. It is a factory of JDBC connections. It abstracts the application from DriverManager or DataSource. It is optional.

**TransactionFactory**. It is a factory of Transaction. It is optional.

**Criteria**. The objects of criteria are used for the creation and execution of the object-oriented criteria queries.

### States of the object (instance)

There are 3 states of the object (instance) in hibernate.

1. **Transient**: The object is in a transient state if it is just created but has no primary key (identifier) and not associated with a session.
2. **Persistent**: The object is in a persistent state if a session is open, and you just saved the instance in the database or retrieved the instance from the database.
3. **Detached**: The object is in a detached state if a session is closed. After detached state, the object comes to persistent state if you call lock() or update() method.

## Creation ofHibernate application

### Without IDE

For creating the first hibernate application, we need to follow the following steps:

1. Create the Persistent class
2. Create the mapping file for Persistent class
3. Create the Configuration file
4. Create the class that retrieves or stores the persistent object
5. Load the jar file
6. Run the first hibernate application by using command prompt

#### Persistent class:

A simple Persistent class should follow some rules:

* **A no-arg constructor:** It is recommended that you have a default constructor at least package visibility so that hibernate can create the instance of the Persistent class by newInstance() method.
* **Provide an identifier property:** It is better to assign an attribute as id. This attribute behaves as a primary key in database.
* **Declare getter and setter methods:** The Hibernate recognizes the method by getter and setter method names by default.
* **Prefer non-final class:** Hibernate uses the concept of proxies, that depends on the persistent class. The application programmer will not be able to use proxies for lazy association fetching.

### Hibernate using Annotation

The hibernate application can be created with annotation. There are many annotations that can be used to create hibernate application such as @Entity, @Id, @Table etc.

Hibernate Annotations are based on the JPA 2 specification and supports all the features.

All the JPA annotations are defined in the **javax.persistence** package. Hibernate EntityManager implements the interfaces and life cycle defined by the JPA specification.

The core advantage of using hibernate annotation is that you don't need to create mapping (hbm) file. Here, hibernate annotations are used to provide the meta data.

**Create the Persistence class.**

@Entity annotation marks this class as an entity.

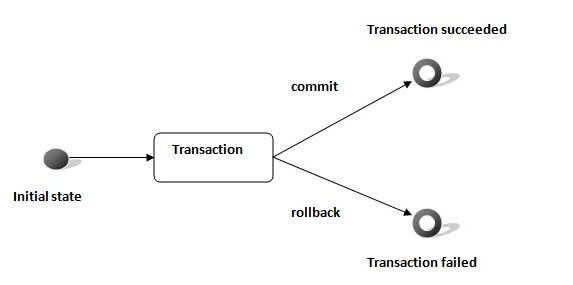
@Table annotation specifies the table name where data of this entity is to be persisted. If you don't use @Table annotation, hibernate will use the class name as the table name by default.

@Id annotation marks the identifier for this entity.

@Column annotation specifies the details of the column for this property or field. If @Column annotation is not specified, property name will be used as the column name by default.

### Hibernate Transaction Management

A **transaction** simply represents a unit of work. In such case, if one step fails, the whole transaction fails (which is termed as atomicity). A transaction can be described by ACID properties (Atomicity, Consistency, Isolation and Durability).



In hibernate framework, we have **Transaction** interface that defines the unit of work. It maintains abstraction from the transaction implementation (JTA,JDBC). A transaction is associated with Session and instantiated by calling **session.beginTransaction()**.

### SQL Dialects in Hibernate

The dialect specifies the type of database used in hibernate so that hibernate generate appropriate type of SQL statements. For connecting any hibernate application with the database, it is required to provide the configuration of SQL dialect.

## Hibernate Mapping

There can be 4 types of association mapping in hibernate.

1. One to One
2. One to Many
3. Many to One
4. Many to Many

There are 3 ways of inheritance mapping in hibernate.

1. Table per hierarchy
2. Table per concrete class
3. Table per subclass

## Caching in Hibernate

Hibernate caching improves the performance of the application by pooling the object in the cache. It is useful when we have to fetch the same data multiple times.

**First Level Cache** Session object holds the first level cache data. It is enabled by default. The first level cache data will not be available to entire application. An application can use many session object.

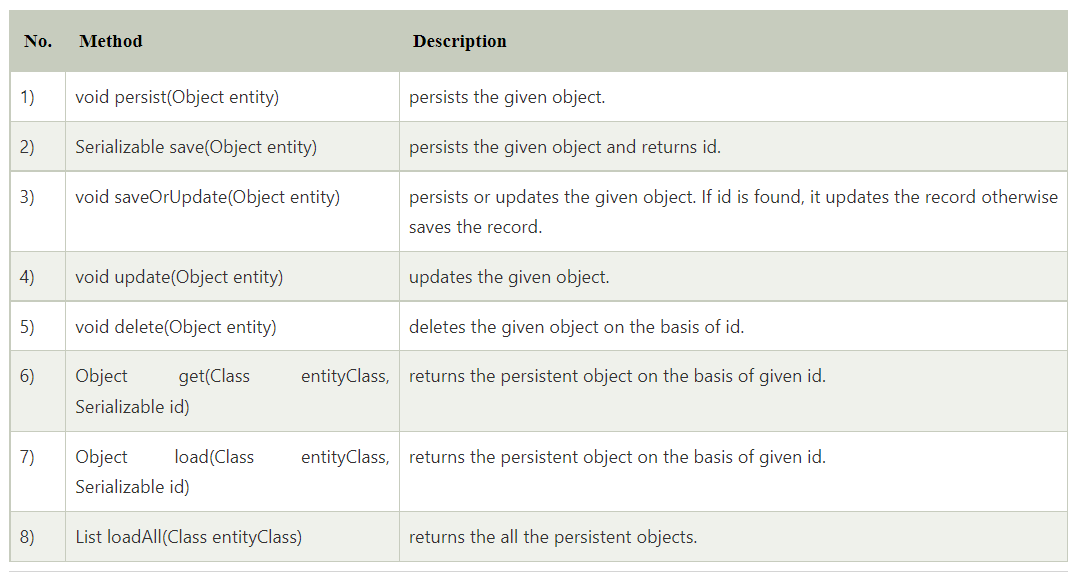
**Second Level Cache** SessionFactory object holds the second level cache data. The data stored in the second level cache will be available to entire application. But we need to enable it explicitely.

|  |  |  |
| --- | --- | --- |
| 1) | First Level Cache is **associated with Session**. | Second Level Cache is associated with **SessionFactory**. |
| 2) | It is **enabled** by default. | It is **not enabled** by default. |

## Hibernate and Spring Integration

The Spring framework provides HibernateTemplate class, so you don't need to follow so many steps like create Configuration, BuildSessionFactory, Session, beginning and committing transaction etc.

### Methods of HibernateTemplate class



### Steps for integration

create table in the database - It is optional.

create applicationContext.xml file - It contains information of DataSource, SessionFactory etc.

create Employee.java file - It is the persistent class

create employee.hbm.xml file - It is the mapping file.

create EmployeeDao.java file - It is the dao class that uses HibernateTemplate.

create InsertTest.java file - It calls methods of EmployeeDao class.

# Spring Data JDBC

Проект Spring Data – это довольно большой зонтичный проект, включающий несколько подпроектов, большинство из которых сосредоточены на хранении данных в различных типах баз данных. Вот некоторые из самых популярных проектов Spring Data:

* Spring Data JDBC – поддержка интерфейса JDBC для реляционных баз данных;
* Spring Data JPA – поддержка интерфейса JPA для реляционных баз данных;
* Spring Data MongoDB  – поддержка документной базы данных Mongo;  Spring Data Neo4j – поддержка графовой базы данных Neo4j;
* Spring Data Redis – поддержка хранилища ключей Redis;
* Spring Data Cassandra – поддержка колоночной базы данных Cassandra.

Одной из самых интересных и полезных функций, общей для всех проектов в Spring Data, является возможность автоматического создания репозиториев на основе интерфейса спецификации.

# JPA

## Ссылки

<https://www.javatpoint.com/jpa-tutorial>

<https://docs.spring.io/spring-data/jpa/docs/current/reference/html/>

<https://www.youtube.com/watch?v=8SGI_XS5OPw>

## Description

Java Persistence (JPA) is a Java specification that provides certain functionality and standard to ORM tools. The **javax.persistence** package contains the JPA classes and interfaces.

Java Persistence API (JPA) — спецификация API Java EE, предоставляет возможность сохранять в удобном виде Java-объекты в базе данных

JPA can be seen as a bridge between object-oriented domain models and relational database systems. Being a specification, JPA doesn't perform any operation by itself. Thus, it requires implementation. So, ORM tools like Hibernate, TopLink, and iBatis implements JPA specifications for data persistence.

The advantages of JPA are given below.

* The burden of interacting with the database reduces significantly by using JPA.
* The user programming becomes easy by concealing the O/R mapping and database access processing.
* The cost of creating the definition file is reduced by using annotations.
* We can merge the applications used with other JPA providers
* Using different implementations can add the features to the standard Implementation which can later be the part of JPA specification.

**JPQL** is the Java Persistence query language defined in JPA specification. It is used to construct the queries. Some of the essential features of JPQL are: -

* It is simple and robust.
* It is a platform-independent query language.
* JPQL queries can be declared statically into metadata or can also be dynamically built in code.
* It can be used with any database such as MySQL, Oracle.

Embeddable classes represent the state of an entity but do not have a persistent identity of their own. The objects of such classes share the identity of the entity classes that owns it. An Entity may have single-valued or multivalued embeddable class attributes.

### JPA advantages

There are following features of JPA:

* It is a powerful repository and custom **object-mapping abstraction.**
* It supports for **cross-store persistence**. It means an entity can be partially stored in MySQL and Neo4j (Graph Database Management System).
* It dynamically generates queries from queries methods name.
* The domain base classes provide basic properties.
* It supports transparent auditing.
* Possibility to integrate custom repository code.
* It is easy to integrate with Spring Framework with the custom namespace.

### JPA vs. Hibernate

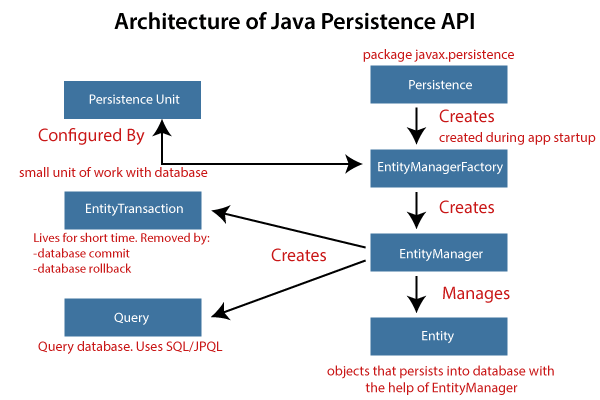
|  |  |
| --- | --- |
| **JPA** | **Hibernate** |
| Java Persistence API (JPA) defines the management of relational data in the Java applications. | Hibernate is an Object-Relational Mapping (ORM) tool which is used to save the state of Java object into the database. |
| It is just a specification. Various ORM tools implement it for data persistence. | It is one of the most frequently used JPA implementation. |
| It is defined in **javax.persistence** package. | It is defined in **org.hibernate** package. |
| The **EntityManagerFactory** interface is used to interact with the entity manager factory for the persistence unit. Thus, it provides an entity manager. | It uses **SessionFactory** interface to create Session instances. |
| It uses **EntityManager** interface to create, read, and delete operations for instances of mapped entity classes. This interface interacts with the persistence context. | It uses **Session** interface to create, read, and delete operations for instances of mapped entity classes. It behaves as a runtime interface between a Java application and Hibernate. |
| It uses **Java Persistence Query Language** (JPQL) as an object-oriented query language to perform database operations. | It uses **Hibernate Query Language** (HQL) as an object-oriented query language to perform database operations. |
| JPA is a **Java specification** for mapping relation data in Java application. | Hibernate is an **ORM framework** that deals with data persistence. |
| JPA does not provide any implementation classes. | It provides implementation classes. |
| It is implemented in various ORM tools like **Hibernate, EclipseLink,** etc. | Hibernate is the **provider** of JPA. |

## JPA Architecture

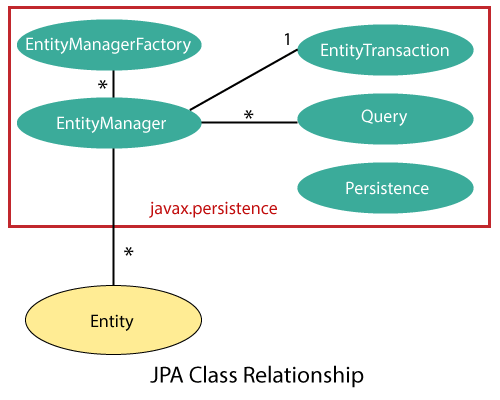
JPA is a source to store business entities as relational entities. It shows how to define a POJO as an entity and how to manage entities with relation.

The following figure describes the class-level architecture of JPA that describes the core classes and interfaces of JPA that is defined in the **javax persistence** package. The JPA architecture contains the following units:

* **Persistence:** It is a class that contains static methods to obtain an EntityManagerFactory instance.
* **EntityManagerFactory:** It is a factory class of EntityManager. It creates and manages multiple instances of EntityManager.
* **EntityManager:** It is an interface. It controls the persistence operations on objects. It works for the Query instance.
* **Entity:** The entities are the persistence objects stores as a record in the database.
* **Persistence Unit:** It defines a set of all entity classes. In an application, EntityManager instances manage it. The set of entity classes represents the data contained within a single data store.
* **EntityTransaction:** It has a **one-to-one** relationship with the EntityManager class. For each EntityManager, operations are maintained by EntityTransaction class.
* **Query:** It is an interface that is implemented by each JPA vendor to obtain relation objects that meet the criteria.

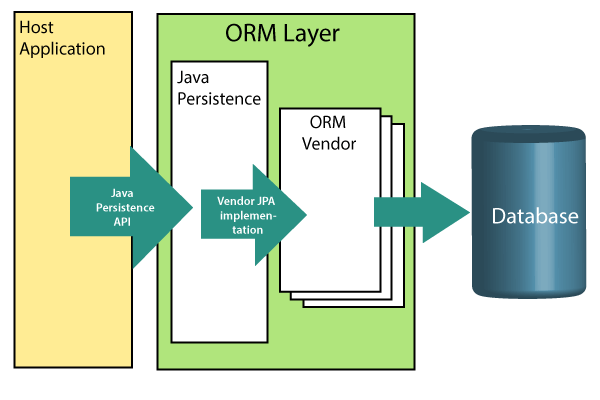


#### JPA Class Relationships



* The relationship between EntityManager and EntityTransaction is **one-to-one**. There is an EntityTransaction instance for each EntityManager operation.
* The relationship between EntityManageFactory and EntityManager is **one-to-many**. It is a factory class to EntityManager instance.
* The relationship between EntityManager and Query is **one-to-many**. We can execute any number of queries by using an instance of EntityManager class.
* The relationship between EntityManager and Entity is **one-to-many**. An EntityManager instance can manage multiple Entities.

#### Object-Relation Mapping (ORM)



### Entity

In general, entity is a group of states associated together in a single unit. On adding behaviour, an entity behaves as an object and becomes a major constituent of object-oriented paradigm. So, an entity is an application-defined object in Java Persistence Library.

These are the properties of an entity that an object must have: -

* **Persistability -** An object is called persistent if it is stored in the database and can be accessed anytime.
* **Persistent Identity -** In Java, each entity is unique and represents as an object identity. Similarly, when the object identity is stored in a database then it is represented as persistence identity. This object identity is equivalent to primary key in database.
* **Transactionality -** Entity can perform various operations such as create, delete, update. Each operation makes some changes in the database. It ensures that whatever changes made in the database either be succeed or failed atomically.
* **Granuality -** Entities should not be primitives, primitive wrappers or built-in objects with single dimensional state.

A persistence entity is a lightweight Java class with its state typically persisted to a table in a relational database. Instances of such an entity correspond to individual rows in the table. Entities typically have relationships with other entities, and these relationships are expressed through object/relational mapping (ORM) metadata. This metadata may be specified directly in the entity class file by using annotations or in a separate XML descriptor file distributed with the application.

Entity (Сущность) — POJO-класс, связанный с БД с помощью аннотации (@Entity) или через XML. К такому классу предъявляются следующие требования:

* Должен иметь пустой конструктор (public или protected)
* Не может быть вложенным, интерфейсом или enum
* Не может быть final и не может содержать final-полей/свойств
* Должен содержать хотя бы одно @Id-поле

При этом entity может:

* Содержать непустые конструкторы
* Наследоваться и быть наследованным
* Содержать другие методы и реализовывать интерфейсы

Entities могут быть связаны друг с другом (один-к-одному, один-ко-многим, многие-к-одному и многие-ко-многим)

The @Entity annotation declares that the class represents an entity. @Id declares the attribute which acts as the primary key of the entity. Additional annotations may be used to declare additional metadata (for example changing the default table name in the @Table annotation), or to create associations between entities.

### JPA Entity Manager

Following are some of the important roles of an entity manager: -

* The entity manager implements the API and encapsulates all of them within a single interface.
* Entity manager is used to read, delete and write an entity.
* An object referenced by an entity is managed by entity manager.

The **EntityManagerFactory** interface present in **java.persistence** package is used to provide an entity manager.

### JPA Cascading Operations

In JPA, if any operation is applied on an entity then it will perform on that particular entity only. These operations will not be applicable to the other entities that are related to it.

To establish a dependency between related entities, JPA provides **javax.persistence.CascadeType** enumerated types that define the cascade operations. These cascading operations can be defined with any type of mapping i.e. One-to-One, One-to-Many, Many-to-One, Many-to-Many.

### JPA Criteria API

The Criteria API is a specification that provides type-safe and portable criteria queries written using Java programming language APIs. It is one of the most common ways of constructing queries for entities and their persistent state. It is just an alternative method for defining JPA queries. Criteria API defines a platform-independent criteria queries, written in Java programming language. It was introduced in JPA 2.0. The main purpose behind this is to provide a type-safe way to express a query.

To create a Criteria query, create an object of **CriteriaBuilder** interface.

Methods of Criteria API Query Clauses

|  |  |  |
| --- | --- | --- |
| **Clause** | **Criteria API Interface** | **Methods** |
| SELECT | CriteriaQuery | select() |
| FROM | AbstractQuery | from() |
| WHERE | AbstractQuery | where() |
| ORDER BY | CriteriaQuery | orderBy() |
| GROUP BY | AbstractQuery | groupBy() |
| HAVING | AbstractQuery | having() |

## Spring and JPA

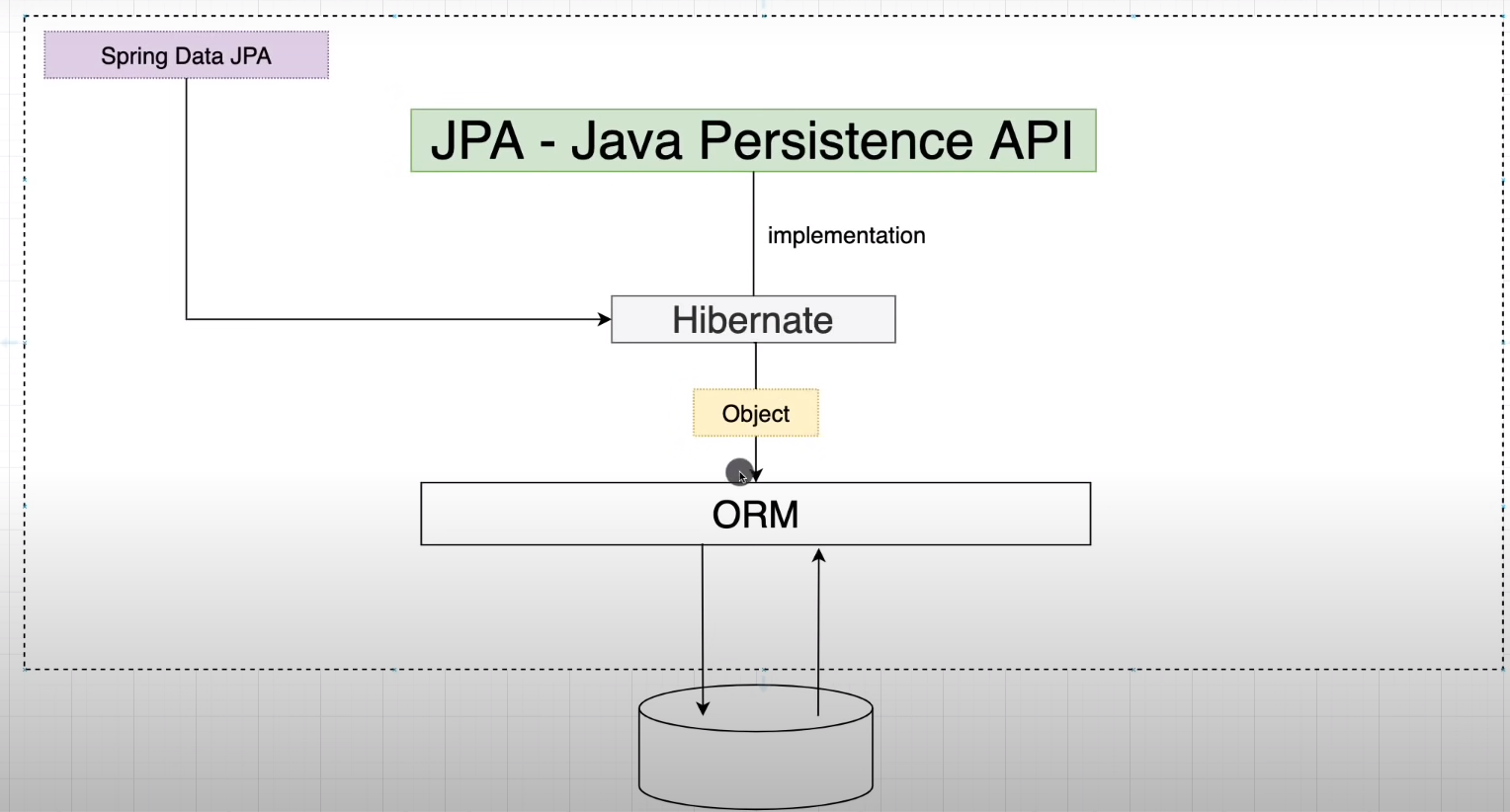
Spring Data JPA API provides JpaTemplate class to integrate spring application with JPA.

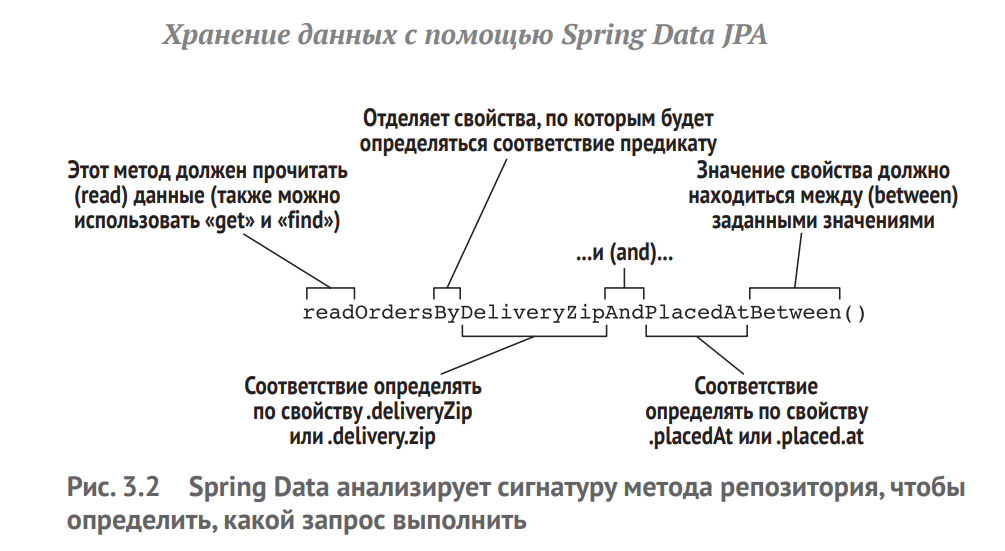
JPA (Java Persistent API) Is the sun specification for persisting objects in the enterprise application. It is currently used as the replacement for complex entity beans.

**Advantage of Spring JpaTemplate**:

You don't need to write the before and after code for persisting, updating, deleting or searching object such as creating Persistence instance, creating EntityManagerFactory instance, creating EntityTransaction instance, creating EntityManager instance, commiting EntityTransaction instance and closing EntityManager.

## Spring Data JPI





На рис. 3.2 показано, как Spring Data анализирует имя метода readOrdersByDeliveryZipAndPlacedAtBetween() при создании реализации репозитория. Как видите, глаголом в  данном случае является read (прочитать). Spring Data воспринимает глаголы find (найти), read (прочитать) и get (получить) как синонимы, обозначающие выборку одного или нескольких объектов. Кроме того, в качестве глагола можно также использовать count (подсчитать), если нужно, чтобы метод вернул только целое число с количеством сущностей, соответствующих предикату. Подлежащее в имени метода может отсутствовать, но в данном случае оно указано – это Orders (заказы). Spring Data игнорирует большинство слов, служащих подлежащим, поэтому с таким же успехом методу можно было дать имя readPuppiesBy..., и он все равно отыскал бы сущности TacoOrder, потому что этим типом параметризован CrudRepository

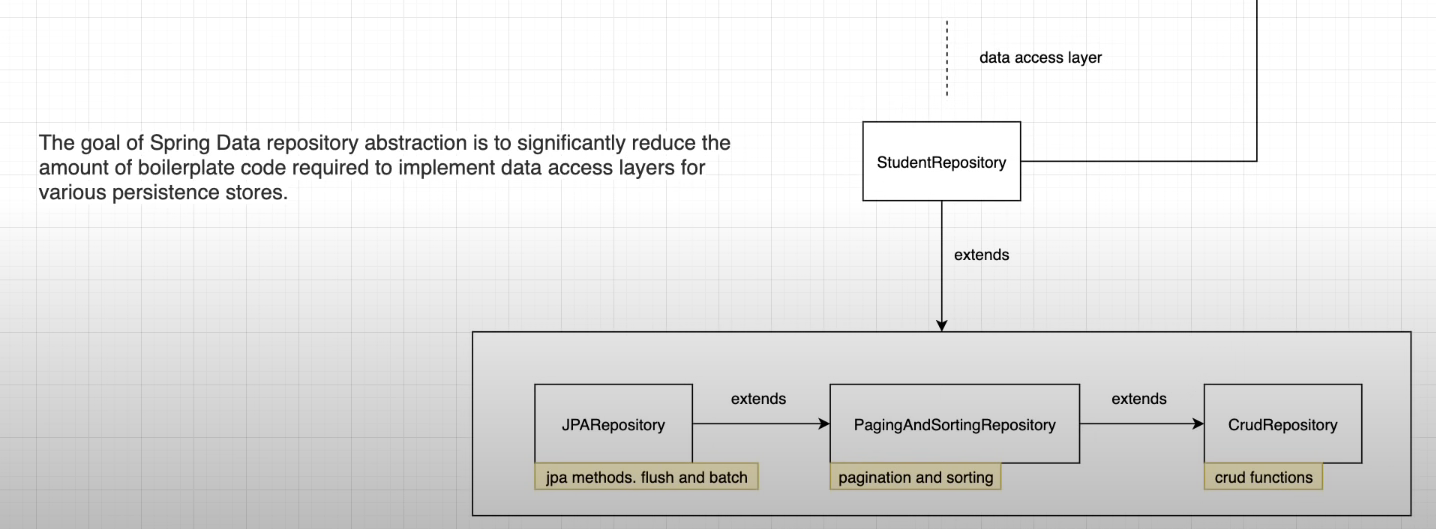
Предикат следует за словом By в имени метода и является самой интересной частью сигнатуры метода. В  данном случае предикат ссылается на два свойства класса TacoOrder: deliveryZip и PlacedAt. Свойство deliveryZip должно быть равно значению, переданному в  первом параметре. Ключевое слово Between указывает, что значение свойства PlacedAt должно находиться между значениями, переданными в двух последних параметрах.

Такое соглашение об именах может быть очень удобным для относительно простых запросов, но не нужно обладать большим воображением, чтобы заметить, что для более сложных запросов имена методов могут получиться весьма неудобоваримыми. В  подобных случаях просто давайте методам имена по своему вкусу и аннотируйте их аннотацией @Query с явно указанным запросом, который должен выполняться при вызове метода, как показано ниже: @Query("Order o where o.deliveryCity=’Seattle’") List readOrdersDeliveredInSeattle(); В этом простом примере запрос в  аннотации @Query выберет все заказы, доставленные в Сиэтл. Но вообще в @Query можно определить практически любой запрос JPA, который только можно придумать, но трудно или невозможно выразить в виде имени метода.

Специализированные методы запросов также можно использовать совместно с Spring Data JDBC, но в этом случае необходимо учитывать следующие важные отличия:  все специализированные методы запросов должны снабжаться аннотацией @Query. Это связано с тем, что, в отличие от JPA, Spring Data JDBC не поддерживает анализ имен методов и не может автоматически конструировать запросы;  все запросы, указанные в @Query, должны быть SQL-запросами, а не запросами JPA

### JPA repositories

JPA repository is data layer.



The central interface in the Spring Data repository abstraction is Repository.

This interface acts primarily as a marker interface to capture the types to work with and to help you to discover interfaces that extend this one. The [CrudRepository](https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html) interface provides sophisticated CRUD functionality for the entity class that is being managed.

On top of the CrudRepository, there is a [PagingAndSortingRepository](https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/PagingAndSortingRepository.html) abstraction that adds additional methods to ease paginated access to entities:

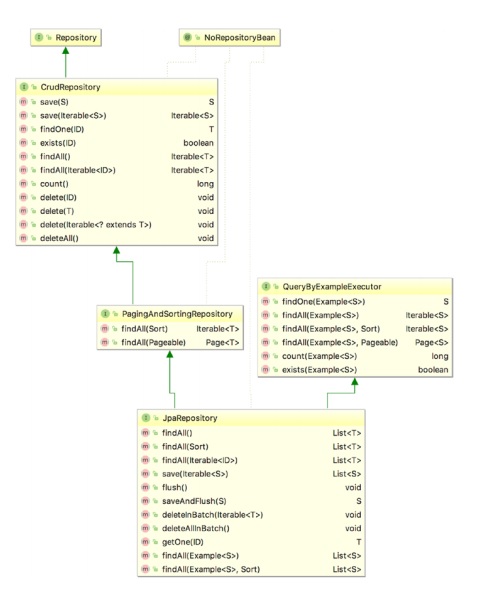
Typically, your repository interface extends Repository, CrudRepository, or PagingAndSortingRepository. Alternatively, if you do not want to extend Spring Data interfaces, you can also annotate your repository interface with @RepositoryDefinition. Extending CrudRepository exposes a complete set of methods to manipulate your entities. If you prefer to be selective about the methods being exposed, copy the methods you want to expose from CrudRepository into your domain repository.

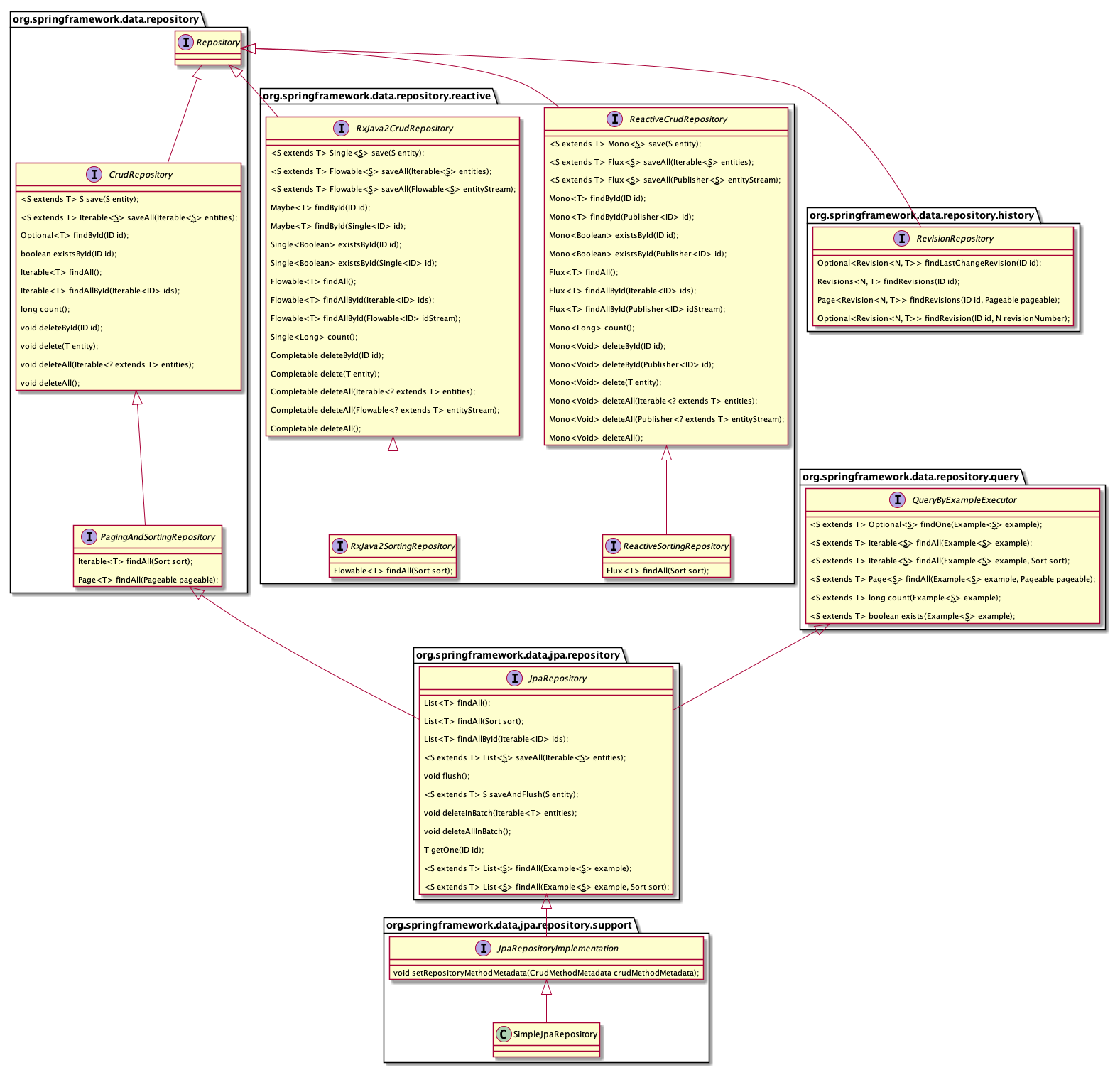
Their main functions are:

**CrudRepository** mainly provides CRUD functions.

**PagingAndSortingRepository** provides methods to do pagination and sorting records.

**JpaRepository** provides some JPA-related methods such as flushing the persistence context and deleting records in a batch.





The intermediate repository interface is annotated with @NoRepositoryBean. Make sure you add that annotation to all repository interfaces for which Spring Data should not create instances at runtime.

@NoRepositoryBean

interface MyBaseRepository<T, ID> extends Repository<T, ID> {

Optional<T> findById(ID id);

<S extends T> S save(S entity);

}

interface UserRepository extends MyBaseRepository<User, Long> {

User findByEmailAddress(EmailAddress emailAddress);

}

### **Spring Data JPA repositories using JavaConfig**

**Example. Spring Data JPA repositories using JavaConfig**

@Configuration

@EnableJpaRepositories

@EnableTransactionManagement

class ApplicationConfig {

@Bean

public DataSource dataSource() {

EmbeddedDatabaseBuilder builder = new EmbeddedDatabaseBuilder();

return builder.setType(EmbeddedDatabaseType.HSQL).build();

}

@Bean

public LocalContainerEntityManagerFactoryBean entityManagerFactory() {

HibernateJpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();

vendorAdapter.setGenerateDdl(true);

LocalContainerEntityManagerFactoryBean factory = new LocalContainerEntityManagerFactoryBean();

factory.setJpaVendorAdapter(vendorAdapter);

factory.setPackagesToScan("com.acme.domain");

factory.setDataSource(dataSource());

return factory;

}

@Bean

public PlatformTransactionManager transactionManager(EntityManagerFactory entityManagerFactory) {

JpaTransactionManager txManager = new JpaTransactionManager();

txManager.setEntityManagerFactory(entityManagerFactory);

return txManager;

}

}

|  |  |
| --- | --- |
|  | You must create LocalContainerEntityManagerFactoryBean and not EntityManagerFactory directly, since the former also participates in exception translation mechanisms in addition to creating EntityManagerFactory. |

### **Query Methods**

With Spring Data, declaring those queries becomes a four-step process:

1. Declare an interface extending Repository or one of its subinterfaces and type it to the domain class and ID type that it should handle, as shown in the following example:

interface PersonRepository extends Repository<Person, Long> { … }

1. Declare query methods on the interface.

interface PersonRepository extends Repository<Person, Long> {

List<Person> findByLastname(String lastname);

}

1. Set up Spring to create proxy instances for those interfaces, either with [JavaConfig](https://docs.spring.io/spring-data/jpa/docs/current/reference/html/" \l "repositories.create-instances.java-config) or with [XML configuration](https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#repositories.create-instances).

To use Java configuration, create a class similar to the following:

import org.springframework.data.jpa.repository.config.EnableJpaRepositories;

@EnableJpaRepositories

class Config { … }

1. Inject the repository instance and use it, as shown in the following example:

class SomeClient {

private final PersonRepository repository;

SomeClient(PersonRepository repository) {

this.repository = repository;

}

void doSomething() {

List<Person> persons = repository.findByLastname("Matthews");

}

}

##### Using JPA Named Queries

**Annotation-based named query configuration**

@Entity

@NamedQuery(name = "User.findByEmailAddress",

query = "select u from User u where u.emailAddress = ?1")

public class User {

}

To allow these named queries, specify the UserRepository as follows:

**Example 60. Query method declaration in UserRepository**

public interface UserRepository extends JpaRepository<User, Long> {

List<User> findByLastname(String lastname);

User findByEmailAddress(String emailAddress);

}

Spring Data tries to resolve a call to these methods to a named query, starting with the simple name of the configured domain class, followed by the method name separated by a dot. So the preceding example would use the named queries defined earlier instead of trying to create a query from the method name.

**Using @Query**

Using named queries to declare queries for entities is a valid approach and works fine for a small number of queries. As the queries themselves are tied to the Java method that runs them, you can actually bind them directly by using the Spring Data JPA @Query annotation rather than annotating them to the domain class. This frees the domain class from persistence specific information and co-locates the query to the repository interface.

Queries annotated to the query method take precedence over queries defined using @NamedQuery or named queries declared in orm.xml.

The following example shows a query created with the @Query annotation:

**Example 61. Declare query at the query method using**@Query

public interface UserRepository extends JpaRepository<User, Long> {

@Query("select u from User u where u.emailAddress = ?1")

User findByEmailAddress(String emailAddress);

}

**Using Advanced LIKE Expressions**

The query running mechanism for manually defined queries created with @Query allows the definition of advanced LIKE expressions inside the query definition, as shown in the following example:

**Example 62. Advanced**like**expressions in @Query**

public interface UserRepository extends JpaRepository<User, Long> {

@Query("select u from User u where u.firstname like %?1")

List<User> findByFirstnameEndsWith(String firstname);

}

In the preceding example, the LIKE delimiter character (%) is recognized, and the query is transformed into a valid JPQL query (removing the %). Upon running the query, the parameter passed to the method call gets augmented with the previously recognized LIKE pattern.

**Native Queries**

The @Query annotation allows for running native queries by setting the nativeQuery flag to true, as shown in the following example:

**Example 63. Declare a native query at the query method using @Query**

public interface UserRepository extends JpaRepository<User, Long> {

@Query(value = "SELECT \* FROM USERS WHERE EMAIL\_ADDRESS = ?1", nativeQuery = true)

User findByEmailAddress(String emailAddress);

}

**Using Sort**

Sorting can be done by either providing a PageRequest or by using Sort directly. The properties actually used within the Order instances of Sort need to match your domain model, which means they need to resolve to either a property or an alias used within the query. The JPQL defines this as a state field path expression.

|  |  |
| --- | --- |
|  | Using any non-referenceable path expression leads to an Exception. |

However, using Sort together with [@Query](https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods.at-query) lets you sneak in non-path-checked Order instances containing functions within the ORDER BY clause. This is possible because the Order is appended to the given query string. By default, Spring Data JPA rejects any Order instance containing function calls, but you can use JpaSort.unsafe to add potentially unsafe ordering.

The following example uses Sort and JpaSort, including an unsafe option on JpaSort:

**Example 65. Using**Sort**and**JpaSort

public interface UserRepository extends JpaRepository<User, Long> {

@Query("select u from User u where u.lastname like ?1%")

List<User> findByAndSort(String lastname, Sort sort);

@Query("select u.id, LENGTH(u.firstname) as fn\_len from User u where u.lastname like ?1%")

List<Object[]> findByAsArrayAndSort(String lastname, Sort sort);

}

repo.findByAndSort("lannister", Sort.by("firstname"));

repo.findByAndSort("stark", Sort.by("LENGTH(firstname)"));

repo.findByAndSort("targaryen", JpaSort.unsafe("LENGTH(firstname)"));

repo.findByAsArrayAndSort("bolton", Sort.by("fn\_len"));

|  |  |
| --- | --- |
|  | Valid Sort expression pointing to property in domain model. |
|  | Invalid Sort containing function call. Throws Exception. |
|  | Valid Sort containing explicitly *unsafe* Order. |
|  | Valid Sort expression pointing to aliased function. |

**Using Named Parameters**

By default, Spring Data JPA uses position-based parameter binding, as described in all the preceding examples. This makes query methods a little error-prone when refactoring regarding the parameter position. To solve this issue, you can use @Param annotation to give a method parameter a concrete name and bind the name in the query, as shown in the following example:

**Example 66. Using named parameters**

public interface UserRepository extends JpaRepository<User, Long> {

@Query("select u from User u where u.firstname = :firstname or u.lastname = :lastname")

User findByLastnameOrFirstname(@Param("lastname") String lastname,

@Param("firstname") String firstname);

}

### Transactional

By default, CRUD methods on repository instances inherited from [SimpleJpaRepository](https://docs.spring.io/spring-data/data-jpa/docs/current/api/org/springframework/data/jpa/repository/support/SimpleJpaRepository.html) are transactional. For read operations, the transaction configuration readOnly flag is set to true. All others are configured with a plain @Transactional so that default transaction configuration applies. Repository methods that are backed by transactional repository fragments inherit the transactional attributes from the actual fragment method.

If you need to tweak transaction configuration for one of the methods declared in a repository, redeclare the method in your repository interface, as follows:

**Example 107. Custom transaction configuration for CRUD**

public interface UserRepository extends CrudRepository<User, Long> {

@Override

@Transactional(timeout = 10)

public List<User> findAll();

// Further query method declarations

}

Doing so causes the findAll() method to run with a timeout of 10 seconds and without the readOnly flag.

Another way to alter transactional behaviour is to use a facade or service implementation that (typically) covers more than one repository. Its purpose is to define transactional boundaries for non-CRUD operations. The following example shows how to use such a facade for more than one repository:

**Example 108. Using a facade to define transactions for multiple repository calls**

@Service

public class UserManagementImpl implements UserManagement {

private final UserRepository userRepository;

private final RoleRepository roleRepository;

public UserManagementImpl(UserRepository userRepository,

RoleRepository roleRepository) {

this.userRepository = userRepository;

this.roleRepository = roleRepository;

}

@Transactional

public void addRoleToAllUsers(String roleName) {

Role role = roleRepository.findByName(roleName);

for (User user : userRepository.findAll()) {

user.addRole(role);

userRepository.save(user);

}

}

}

This example causes call to addRoleToAllUsers(…) to run inside a transaction (participating in an existing one or creating a new one if none are already running). The transaction configuration at the repositories is then neglected, as the outer transaction configuration determines the actual one used. Note that you must activate <tx:annotation-driven /> or use @EnableTransactionManagement explicitly to get annotation-based configuration of facades to work. This example assumes you use component scanning.

Note that the call to save is not strictly necessary from a JPA point of view, but should still be there in order to stay consistent to the repository abstraction offered by Spring Data.

#### Transactional query methods

To let your query methods be transactional, use @Transactional at the repository interface you define, as shown in the following example:

**Example 109. Using @Transactional at query methods**

@Transactional(readOnly = true)

interface UserRepository extends JpaRepository<User, Long> {

List<User> findByLastname(String lastname);

@Modifying

@Transactional

@Query("delete from User u where u.active = false")

void deleteInactiveUsers();

}

Typically, you want the readOnly flag to be set to true, as most of the query methods only read data. In contrast to that, deleteInactiveUsers() makes use of the @Modifying annotation and overrides the transaction configuration. Thus, the method runs with the readOnly flag set to false.

|  |  |
| --- | --- |
|  | You can use transactions for read-only queries and mark them as such by setting the readOnly flag. Doing so does not, however, act as a check that you do not trigger a manipulating query (although some databases reject INSERT and UPDATE statements inside a read-only transaction). The readOnly flag is instead propagated as a hint to the underlying JDBC driver for performance optimizations. Furthermore, Spring performs some optimizations on the underlying JPA provider. For example, when used with Hibernate, the flush mode is set to NEVER when you configure a transaction as readOnly, which causes Hibernate to skip dirty checks (a noticeable improvement on large object trees). |

@Transactional isolation

You can declare which isolation level you'd like to use with this property. The options are DEFAULT, READ\_UNCOMMITTED, READ\_COMMITTED, REPEATABLE\_READ, SERIALIZABLE

The default and most practical option is Isolation.DEFAULT which delegates the setting to the underlying database

**@Transactional propagation**

You can control either to use a current transaction to execute CRUD operations against a database or suspend it with the following options

* Use a current transaction, if none exists then create a new one with Propagation.REQUIRED, execute non-transactionally with Propagation.SUPPORTS, and throw an exception with Propagation.MANDATORY
* Suspend a current transaction, if none exists then create a new transaction with Propagation.REQUIRED\_NEW, execute non-transactionally with Propagation.NOT\_SUPPORTED

The default and most practical option is Propagation.REQUIRED

The below example set the isolation level to READ\_COMMITTED and propagation to REQUIRED

@Transactional(

isolation = Isolation.READ\_COMMITTED,

propagation = Propagation.REQUIRED)

public void crudAgainstDatabase() {

...

}

**@Transactional rollbackFor and noRollbackFor**

* You can use rollbackFor to indicate which exception types must cause a transaction rollback. By default, they are unchecked exceptions including RuntimeException, Error and their subclasses

Say you throw a RuntimeException in your custom Transactional method

@Transactional

public void updateWithThrowingRuntimeException(Long id, String name) {

Product product = findById(id).get();

product.setName(name);

throw new MyRuntimeException();

}

static class MyRuntimeException extends RuntimeException {

}

Then the transaction is rollbacked by the throwing MyRuntimeException as expected

@Test

public void testRollbackRuntimeException() {

try {

productService.updateWithThrowingRuntimeException(1L, "updated");

} catch (RuntimeException e) {

System.out.println(e.getMessage());

}

Optional<Product> updatedProduct = productService.findByName("updated");

assertThat(updatedProduct).isNotPresent();

}

* Checked exceptions, Exception and its subclasses, don't cause transaction rollback by default

**@Transactional readOnly**

A boolean flag served as a hint for the actual transaction subsystems

The default value is false.

**@Transactional pitfalls**

* You don't have to call save/update method against database explicitly inside a @Transactional method

#### Советы - Transactional

Устанавливайте аннотацию @Transactional в слое сервиса (Service layer), а не в слое DAO. Сервисные бины могут использовать несколько DAO, соблюдая ACID под одной и той же транзакцией. Иначе если только в DAO определен механизм транзакций, то сервисные бины увеличат расходы на создание множественных транзакций для концептуально сгруппированных операций, не говоря уже о неконсистентном состоянии данных, которое мы рискуем получить.

Мы можем задать @Transactional(propagation = Propagation.MANDATORY) на уровне класса в наших DAO, таким образом заставляя потребителей наших DAO инициировать управление транзакциями.

Необходимо знать дефолтное поведение аннотации @Transactional и не использовать ее вслепую. А именно, если не указано явно, уровень распространения (propagation) установится в Propagation.REQUIRED, что означает использование  текущей транзакции, иначе создать новую; изоляция установится в значение Isolation.DEFAULT, это значение определяется нижележащей БД (по умолчанию у многих БД это значение равно Isolation.READ\_COMMITED); readOnly флаг выключен по умолчанию; rollbackFor может быть задано с Throwable классом, но будьте осторожны: по умолчанию откат (rollback) происходит только когда выбрасывается RuntimeException (если не установлен этот параметр).

Будьте осторожны с флагом readOnly. Хотя @Transactional(readOnly = true, propagation=Propagation.REQUIRED) выбросит исключение  при попытке выполниться команде вставки/обновления через  JDBC в транзакции, она может иметь неожиданное поведение при попытке вставки/обновления через ORM, где операция скрытно выполнится и успешно выполнится коммит транзакции. В ORM  окружении необходимо использовать этот флаг вместе с Propagation.SUPPORTS. В этом случае мы не будем платить стоимость создания новой транзакции в простой операции выборки. Или даже рассмотрите возможность избавиться от управления транзакциями для операций SELECT.

Spring автоматически откатывает транзакции для unchecked (Runtime) исключений.

Не используйте аннотацию @Transactional на приватных, protected или default методах. Этот пункт происходит из первого пункта о том как Spring управляет транзакциями. Добавление аннотации к методами с private, protected или default модификаторами доступа не выбросит исключение. Однако аннотация будет проигнорирована.

Очень важно в случае возникновения непредвиденных ситуаций смотреть и понимать логи. Для включения логов необходимо добавить в log4j.properties

org.springframework.transaction=TRACE

Это позволит Вам отследить активности происходящие внутри транзакционных процессов.

**Transactional Propagation**

**@Transactional(propagation=Propagation.REQUIRED)**

Если не указано, то стратегия распространения по дефолту это REQUIRED.

Другие опции это REQUIRES\_NEW, MANDATORY, SUPPORTS, NOT\_SUPPORTED, NEVER, и NESTED.

**REQUIRED**

* Означает что целевой метод не может быть запущен без активной транзакции. Если транзакция уже стартовала до вызова  этого метода, тогда методы выполнится в ней или иначе при вызове метода новая транзакция будет создана.

**REQUIRES\_NEW**

* Означает что новая транзакция должна начинаться всякий раз, как целевой метод будет вызываться. Если уже есть начатая транзакция, то она будет приостановлена, до начала новой.

**MANDATORY**

* Означает что целевой метод требует активной транзакции для старта. Если ее нет, то выполнение не будет произведено, и выбросится исключение.

**SUPPORTS**

* Означает что целевой метод может быть исполнен вне транзакции. Если есть начатая транзакция, то метод запустится в ней. Если нет запущенной транзакции, то метод выполнится все равно, только не в транзакционном контексте.
* Методы, которые выполняют выборку данных наилучшие кандидаты для этой опции.

**NOT\_SUPPORTED**

* Означает что целевой метод не требует транзакционного контекста для выполнения. Если есть начатая транзакция, то она будет приостановлена.

**NEVER**

* Означает что целевой метод выбросит исключение, если выполнится в транзакционном процессе.
* Не советую использовать эту опцию.

**@Transactional (rollbackFor=Exception.class)**

* По умолчанию откат происходит при rollbackFor=RunTimeException.class
* В спринге, все API классы выбрасывают RuntimeException, что означает что если любой метод упал, то контейнер всегда произведет откат текущей транзакции.
* Проблема только с checked exceptions. Так что данная опция может использоваться для декларативного отката транзакции если выбросится Checked Exception.

**@Transactional (noRollbackFor=IllegalStateException.class)**

* Означает что откат не должен происходить если целевой метод выбросил это исключение.

### Defining Repository Interfaces

To define a repository interface, you first need to define a domain class-specific repository interface. The interface must extend Repository and be typed to the domain class and an ID type. If you want to expose CRUD methods for that domain type, extend CrudRepository instead of Repository.

Typically, your repository interface extends Repository, CrudRepository, or PagingAndSortingRepository. Alternatively, if you do not want to extend Spring Data interfaces, you can also annotate your repository interface with @RepositoryDefinition. Extending CrudRepository exposes a complete set of methods to manipulate your entities. If you prefer to be selective about the methods being exposed, copy the methods you want to expose from CrudRepository into your domain repository.

### Defining Query Methods

The repository proxy has two ways to derive a store-specific query from the method name:

* By deriving the query from the method name directly.
* By using a manually defined query.

Example. Query creation from method names

interface PersonRepository extends Repository<Person, Long> {

List<Person> findByEmailAddressAndLastname(EmailAddress emailAddress, String lastname);

// Enables the distinct flag for the query

List<Person> findDistinctPeopleByLastnameOrFirstname(String lastname, String firstname);

List<Person> findPeopleDistinctByLastnameOrFirstname(String lastname, String firstname);

// Enabling ignoring case for an individual property

List<Person> findByLastnameIgnoreCase(String lastname);

// Enabling ignoring case for all suitable properties

List<Person> findByLastnameAndFirstnameAllIgnoreCase(String lastname, String firstname);

// Enabling static ORDER BY for a query

List<Person> findByLastnameOrderByFirstnameAsc(String lastname);

List<Person> findByLastnameOrderByFirstnameDesc(String lastname);

}

Parsing query method names is divided into subject and predicate. The first part (find…By, exists…By) defines the subject of the query, the second part forms the predicate. The introducing clause (subject) can contain further expressions. Any text between find (or other introducing keywords) and By is considered to be descriptive unless using one of the result-limiting keywords such as a Distinct to set a distinct flag on the query

The appendix contains the [full list of query method subject keywords](https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#appendix.query.method.subject) and [query method predicate keywords including sorting and letter-casing modifiers](https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#appendix.query.method.predicate). However, the first By acts as a delimiter to indicate the start of the actual criteria predicate.

The actual result of parsing the method depends on the persistence store for which you create the query. However, there are some general things to notice:

* The expressions are usually property traversals combined with operators that can be concatenated. You can combine property expressions with AND and OR. You also get support for operators such as Between, LessThan, GreaterThan, and Like for the property expressions. The supported operators can vary by datastore, so consult the appropriate part of your reference documentation.
* The method parser supports setting an IgnoreCase flag for individual properties (for example, findByLastnameIgnoreCase(…)) or for all properties of a type that supports ignoring case (usually String instances — for example, findByLastnameAndFirstnameAllIgnoreCase(…)). Whether ignoring cases is supported may vary by store, so consult the relevant sections in the reference documentation for the store-specific query method.
* You can apply static ordering by appending an OrderBy clause to the query method that references a property and by providing a sorting direction (Asc or Desc). To create a query method that supports dynamic sorting, see “[Special parameter handling](https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#repositories.special-parameters)”.

#### Special parameter handling

To handle parameters in your query, define method parameters as already seen in the preceding examples. Besides that, the infrastructure recognizes certain specific types like Pageable and Sort, to apply pagination and sorting to your queries dynamically. The following example demonstrates these features:

**Using**Pageable**,**Slice**, and**Sort**in query methods**

Page<User> findByLastname(String lastname, Pageable pageable);

Slice<User> findByLastname(String lastname, Pageable pageable);

List<User> findByLastname(String lastname, Sort sort);

List<User> findByLastname(String lastname, Pageable pageable);

**Defining sort expressions**

Sort sort = Sort.by("firstname").ascending()

.and(Sort.by("lastname").descending());

#### Limiting Query Results

You can limit the results of query methods by using the first or top keywords, which you can use interchangeably. You can append an optional numeric value to top or first to specify the maximum result size to be returned. If the number is left out, a result size of 1 is assumed. The following example shows how to limit the query size:

**Example 18. Limiting the result size of a query with**Top**and**First

User findFirstByOrderByLastnameAsc();

User findTopByOrderByAgeDesc();

Page<User> queryFirst10ByLastname(String lastname, Pageable pageable);

Slice<User> findTop3ByLastname(String lastname, Pageable pageable);

List<User> findFirst10ByLastname(String lastname, Sort sort);

List<User> findTop10ByLastname(String lastname, Pageable pageable);

The limiting expressions also support the Distinct keyword for datastores that support distinct queries. Also, for the queries that limit the result set to one instance, wrapping the result into with the Optional keyword is supported.

#### Repository Methods Returning Collections or Iterables

Query methods that return multiple results can use standard Java Iterable, List, and Set. Beyond that, we support returning Spring Data’s Streamable, a custom extension of Iterable, as well as collection types provided by [Vavr](https://www.vavr.io/). Refer to the appendix explaining all possible [query method return types](https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#appendix.query.return.types).

You can use Streamable as alternative to Iterable or any collection type. It provides convenience methods to access a non-parallel Stream (missing from Iterable) and the ability to directly ….filter(…) and ….map(…) over the elements and concatenate the Streamable to others:

**Example 19. Using Streamable to combine query method results**

interface PersonRepository extends Repository<Person, Long> {

Streamable<Person> findByFirstnameContaining(String firstname);

Streamable<Person> findByLastnameContaining(String lastname);

}

Streamable<Person> result = repository.findByFirstnameContaining("av")

.and(repository.findByLastnameContaining("ea"));

#### Null Handling of Repository Methods

You can express nullability constraints for repository methods by using [Spring Framework’s nullability annotations](https://docs.spring.io/spring-framework/docs/5.3.22/reference/html/core.html#null-safety). They provide a tooling-friendly approach and opt-in null checks during runtime, as follows:

* [@NonNullApi](https://docs.spring.io/spring/docs/5.3.22/javadoc-api/org/springframework/lang/NonNullApi.html): Used on the package level to declare that the default behavior for parameters and return values is, respectively, neither to accept nor to produce null values.
* [@NonNull](https://docs.spring.io/spring/docs/5.3.22/javadoc-api/org/springframework/lang/NonNull.html): Used on a parameter or return value that must not be null (not needed on a parameter and return value where @NonNullApi applies).
* [@Nullable](https://docs.spring.io/spring/docs/5.3.22/javadoc-api/org/springframework/lang/Nullable.html): Used on a parameter or return value that can be null.

### Creating Repository Instances

Each Spring Data module includes a repositories element that lets you define a base package that Spring scans for you

You can also trigger the repository infrastructure by using a store-specific @Enable${store}Repositories annotation on a Java configuration class. For an introduction to Java-based configuration of the Spring container, see [JavaConfig in the Spring reference documentation](https://docs.spring.io/spring-framework/docs/5.3.22/reference/html/core.html" \l "beans-java).

A sample configuration to enable Spring Data repositories resembles the following:

**Example. Sample annotation-based repository configuration**

@Configuration

@EnableJpaRepositories("com.acme.repositories")

class ApplicationConfiguration {

@Bean

EntityManagerFactory entityManagerFactory() {

// …

}

}

### Customizing Individual Repositories

To enrich a repository with custom functionality, you must first define a fragment interface and an implementation for the custom functionality, as follows:

**Example 29. Interface for custom repository functionality**

interface CustomizedUserRepository {

void someCustomMethod(User user);

}

**Example 30. Implementation of custom repository functionality**

class CustomizedUserRepositoryImpl implements CustomizedUserRepository {

public void someCustomMethod(User user) {

// Your custom implementation

}

}

The implementation itself does not depend on Spring Data and can be a regular Spring bean. Consequently, you can use standard dependency injection behavior to inject references to other beans (such as a JdbcTemplate), take part in aspects, and so on.

Then you can let your repository interface extend the fragment interface, as follows:

**Example 31. Changes to your repository interface**

interface UserRepository extends CrudRepository<User, Long>, CustomizedUserRepository {

// Declare query methods here

}

При создании entity и т.д. нужно выбирать аннотации (@Entity, @Id и т.д.) из javax.persistence , что если поменяется имплементация (например с Hibernate на что-другое), не пришлось менять аннотации.

## Spring Boot JPA

**Spring Boot JPA**is a Java specification for managing **relational** data in Java applications. It allows us to access and persist data between Java object/ class and relational database. JPA follows **Object-Relation Mapping**(ORM). It is a set of interfaces. It also provides a runtime **EntityManager** API for processing queries and transactions on the objects against the database. It uses a platform-independent object-oriented query language JPQL (Java Persistent Query Language).

In the context of persistence, it covers three areas:

* The Java Persistence API
* **Object-Relational** metadata
* The API itself, defined in the **persistence** package

JPA is not a framework. It defines a concept that can be implemented by any framework.

# HTTP

У протокола HTTP есть очень важное достоинство: он предоставляет стороннему наблюдателю довольно подробную информацию о том, что произошло с запросом и ответом, даже если этот наблюдатель ничего не знает о семантике операции:

* URL идентифицирует некоего конечного получателя запроса, какую-то единицу адресации;
* по статусу ответа можно понять, выполнена ли операция успешно или произошла ошибка; если имеет место быть ошибка — то можно понять, кто виноват (клиент или сервер), и даже в каких-то случаях понять, что же конкретно произошло;
* по методу запроса можно понять, является ли операция модифицирующей; если операция модифицирующая, то можно выяснить, является ли она идемпотентной;
* по методу запроса и статусу и заголовкам ответа можно понять, кэшируем ли результат операции, и, если да, то какова политика кэширования.

# REST API

## Ссылки

<https://www.restapitutorial.com/lessons/restquicktips.html>

https://habr.com/ru/post/560590/

## What Is REST

API — это спецификация/интерфейс, написанный нами, разработчиками, чтобы наш код мог использовать другой код: библиотеки, прочие приложения или сервисы.

REST представляет собой акроним фразы «передача состояния представления» (representational state transfer) — довольно загадочного способа заявить, что наше приложение взаимодействует с другим. Приложение A не ожидает от приложения Б сохранения своего состояния — текущего и накопленной информации о процессах — между вызовами связи. Приложение A передает представление нужной части своего состояния с каждым запросом к приложению Б. Это повышает живучесть и отказоустойчивость: состояние взаимодействий приложения Б с приложением A сохраняется в случае аварийного сбоя и перезапуска первого — приложение A может повторно выполнить запрос и продолжить с того же места, где произошел сбой. Ресурсы, которые придерживаются этого общего принципа, часто называют приложениями/сервисами без сохранения состояния (stateless), поскольку каждый сервис поддерживает собственное состояние даже в ходе последовательности взаимодействий и не ждет от других сервисов/приложений, что они сделают это вместо него.

## API в стиле HTTP-глаголов

REST API обычно создаются на основе следующих HTTP-команд:

**GET**

Read a specific resource (by an identifier) or a collection of resources.

**PUT**

Update a specific resource (by an identifier) or a collection of resources. Can also be used to create a specific resource if the resource identifier is known before-hand.

**DELETE**

Remove/delete a specific resource by an identifier.

**POST**

Create a new resource. Also a catch-all verb for operations that don't fit into the other categories.

Иногда применяются также следующие две команды:

OPTIONS;

HEAD.

Их можно использовать для извлечения параметров обмена информацией для пар «запрос — ответ» (OPTIONS) и заголовка ответа без его тела (HEAD).

Here are some quick-hit rules for URL path (resource name) design:

* Use identifiers in your URLs instead of in the query-string. Using URL query-string parameters is fantastic for filtering, but not for resource names.
  + **Good:** /users/12345
  + **Poor:** /api?type=user&id=23
* Leverage the hierarchical nature of the URL to imply structure.
* Design for your clients, not for your data.
* Resource names should be nouns. Avoid verbs as resource names, to improve clarity. Use the HTTP methods to specify the verb portion of the request.
* Use plurals in URL segments to keep your API URIs consistent across all HTTP methods, using the collection metaphor.
  + **Recommended:** /customers/33245/orders/8769/lineitems/1
  + **Not:** /customer/33245/order/8769/lineitem/1
* Avoid using collection verbiage in URLs. For example 'customer\_list' as a resource. Use pluralization to indicate the collection metaphor (e.g. customers vs. customer\_list).
* Use lower-case in URL segments, separating words with underscores ('\_') or hyphens ('-'). Some servers ignore case so it's best to be clear.
* Keep URLs as short as possible, with as few segments as makes sense.

Response status codes are part of the HTTP specification. There are quite a number of them to address the most common situations. In the spirit of having our RESTful services embrace the HTTP specification, our Web APIs should return relevant HTTP status codes. For example, when a resource is successfully created (e.g. from a POST request), the API should return HTTP status code 201. A list of valid [HTTP status codes](https://www.restapitutorial.com/httpstatuscodes.html) is available [here](https://www.restapitutorial.com/httpstatuscodes.html) which lists detailed descriptions of each.

Suggested usages for the "Top 10" HTTP Response Status Codes are as follows:

**200 OK**

General success status code. This is the most common code. Used to indicate success.

**201 CREATED**

Successful creation occurred (via either POST or PUT). Set the Location header to contain a link to the newly-created resource (on POST). Response body content may or may not be present.

**204 NO CONTENT**

Indicates success but nothing is in the response body, often used for DELETE and PUT operations.

**400 BAD REQUEST**

General error for when fulfilling the request would cause an invalid state. Domain validation errors, missing data, etc. are some examples.

**401 UNAUTHORIZED**

Error code response for missing or invalid authentication token.

**403 FORBIDDEN**

Error code for when the user is not authorized to perform the operation or the resource is unavailable for some reason (e.g. time constraints, etc.).

**404 NOT FOUND**

Used when the requested resource is not found, whether it doesn't exist or if there was a 401 or 403 that, for security reasons, the service wants to mask.

**405 METHOD NOT ALLOWED**

Used to indicate that the requested URL exists, but the requested HTTP method is not applicable. For example, POST /users/12345 where the API doesn't support creation of resources this way (with a provided ID). The Allow HTTP header must be set when returning a 405 to indicate the HTTP methods that are supported. In the previous case, the header would look like "Allow: GET, PUT, DELETE"

**409 CONFLICT**

Whenever a resource conflict would be caused by fulfilling the request. Duplicate entries, such as trying to create two customers with the same information, and deleting root objects when cascade-delete is not supported are a couple of examples.

**500 INTERNAL SERVER ERROR**

Never return this intentionally. The general catch-all error when the server-side throws an exception. Use this only for errors that the consumer cannot address from their end.

Below is a table summarizing recommended return values of the primary HTTP methods in combination with the resource URIs:

| **HTTP Verb** | **CRUD** | **Entire Collection (e.g. /customers)** | **Specific Item (e.g. /customers/{id})** |
| --- | --- | --- | --- |
| POST | Create | 201 (Created), 'Location' header with link to /customers/{id} containing new ID. | 404 (Not Found), 409 (Conflict) if resource already exists.. |
| GET | Read | 200 (OK), list of customers. Use pagination, sorting and filtering to navigate big lists. | 200 (OK), single customer. 404 (Not Found), if ID not found or invalid. |
| PUT | Update/Replace | 405 (Method Not Allowed), unless you want to update/replace every resource in the entire collection. | 200 (OK) or 204 (No Content). 404 (Not Found), if ID not found or invalid. |
| PATCH | Update/Modify | 405 (Method Not Allowed), unless you want to modify the collection itself. | 200 (OK) or 204 (No Content). 404 (Not Found), if ID not found or invalid. |
| DELETE | Delete | 405 (Method Not Allowed), unless you want to delete the whole collection—not often desirable. | 200 (OK). 404 (Not Found), if ID not found or invalid. |

## **The six constraints**

1. Uniform Interface

The uniform interface constraint defines the interface between clients and servers. It simplifies and decouples the architecture, which enables each part to evolve independently. The four guiding principles of the uniform interface are:

* 1. **Resource-Based**

Individual resources are identified in requests using URIs as resource identifiers. The resources themselves are conceptually separate from the representations that are returned to the client. For example, the server does not send its database, but rather, some HTML, XML or JSON that represents some database records expressed, for instance, in Finnish and encoded in UTF-8, depending on the details of the request and the server implementation.

* 1. **Manipulation of Resources Through Representations**

When a client holds a representation of a resource, including any metadata attached, it has enough information to modify or delete the resource on the server, provided it has permission to do so.

* 1. **Self-descriptive Messages**

Each message includes enough information to describe how to process the message. For example, which parser to invoke may be specified by an Internet media type (previously known as a MIME type). Responses also explicitly indicate their cache-ability.

* 1. **Hypermedia as the Engine of Application State (HATEOAS)**

Clients deliver state via body contents, query-string parameters, request headers and the requested URI (the resource name). Services deliver state to clients via body content, response codes, and response headers. This is technically referred-to as hypermedia (or hyperlinks within hypertext).

Aside from the description above, HATEOS also means that, where necessary, links are contained in the returned body (or headers) to supply the URI for retrieval of the object itself or related objects. We'll talk about this in more detail later.

The uniform interface that any REST services must provide is fundamental to its design.

1. Stateless

As REST is an acronym for REpresentational State Transfer, statelessness is key. Essentially, what this means is that the necessary state to handle the request is contained within the request itself, whether as part of the URI, query-string parameters, body, or headers. The URI uniquely identifies the resource and the body contains the state (or state change) of that resource. Then after the server does it's processing, the appropriate state, or the piece(s) of state that matter, are communicated back to the client via headers, status and response body.

Most of us who have been in the industry for a while are accustomed to programming within a container which provides us with the concept of “session” which maintains state across multiple HTTP requests. In REST, the client must include all information for the server to fulfill the request, resending state as necessary if that state must span multiple requests. Statelessness enables greater scalability since the server does not have to maintain, update or communicate that session state. Additionally, load balancers don't have to worry about session affinity for stateless systems.

So what's the difference between state and a resource? State, or application state, is that which the server cares about to fulfill a request—data necessary for the current session or request. A resource, or resource state, is the data that defines the resource representation—the data stored in the database, for instance. Consider application state to be data that could vary by client, and per request. Resource state, on the other hand, is constant across every client who requests it.

Ever had back-button issues with a web application where it went AWOL at a certain point because it expected you to do things in a certain order? That's because it violated the statelessness principle. There are cases that don't honor the statelessness principle, such as three-legged OAuth, API call rate limiting, etc. However, make every effort to ensure that application state doesn't span multiple requests of your service(s).

1. Cacheable

As on the World Wide Web, clients can cache responses. Responses must therefore, implicitly or explicitly, define themselves as cacheable, or not, to prevent clients reusing stale or inappropriate data in response to further requests. Well-managed caching partially or completely eliminates some client–server interactions, further improving scalability and performance.

1. Client-Server

The uniform interface separates clients from servers. This separation of concerns means that, for example, clients are not concerned with data storage, which remains internal to each server, so that the portability of client code is improved. Servers are not concerned with the user interface or user state, so that servers can be simpler and more scalable. Servers and clients may also be replaced and developed independently, as long as the interface is not altered.

1. Layered System

A client cannot ordinarily tell whether it is connected directly to the end server, or to an intermediary along the way. Intermediary servers may improve system scalability by enabling load-balancing and by providing shared caches. Layers may also enforce security policies.

1. Code on Demand (optional)

Servers are able to temporarily extend or customize the functionality of a client by transferring logic to it that it can execute. Examples of this may include compiled components such as Java applets and client-side scripts such as JavaScript.

Complying with these constraints, and thus conforming to the REST architectural style, will enable any kind of distributed hypermedia system to have desirable emergent properties, such as performance, scalability, simplicity, modifiability, visibility, portability and reliability.

**NOTE:** The only optional constraint of REST architecture is code on demand. If a service violates any other constraint, it cannot strictly be referred to as RESTful.