

JAVA PROFESSIONAL

SPRING FRAMEWORK И SPRING BOOT

Spring, или Spring Framework — фреймворк для языка программирования Java. Он нужен, чтобы разработчикам было легче проектировать и создавать приложения. Spring не связан с конкретной парадигмой или моделью программирования, поэтому его могут использовать как каркас для разных видов приложений.

Если Spring Framework фокусируется на предоставлении гибкости, то Spring Boot стремится сократить длину кода и упростить разработку web-приложения. Используя конфигурацию при помощи аннотаций и стандартного кода, Spring Boot сокращает время, затрачиваемое на разработку приложений. Данная возможность помогает создать автономные приложения с меньшими или почти нулевыми затратами на их конфигурацию.

ПРЕИМУЩЕСТВА SPRING FRAMEWORK

- Spring Framework может быть задействован на всех архитектурных слоях, применяемых при разработке web-приложений
- Использует модель POJO при написании классов, а это очень легкая структура
- Позволяет свободно связывать модули и легко их тестировать
- Поддерживает декларативное программирование
- Избавляет от самостоятельного создания фабричных и синглтон-классов
- Поддерживает различные способы конфигурации
- Предоставляет сервис уровня middleware

Несмотря на наличие стольких преимуществ, которыми обладает Spring, длительная процедура подготовки, связанная с его настройкой, способствовала появлению Spring Boot.

ПРЕИМУЩЕСТВА *SPRING BOOT*

- Spring Boot не требует развертывания war-файлов
- Создает автономные приложения
- Помогает напрямую встроить в приложение Tomcat, Jetty или Undertow
- Не требует XML-конфигурации
- Направлен на уменьшение объема исходного кода
- Имеет дополнительную функциональность «из коробки»
- Простота запуска
- Простая настройка и управление

СИСТЕМЫ СБОРКИ

Сборка (англ. assembly) - двоичный файл, содержащий исполняемый код программы или другой, подготовленный для использования информационный продукт.

В Spring Boot выбор системы сборки является важной задачей. Исторически у разработчиков приложений Java имелось несколько вариантов утилит сборки проектов. Некоторые со временем стали непопулярными, и не без причин, и теперь сообщество разработчиков сконцентрировалось на двух: Maven и Gradle. Spring Boot поддерживает обе с одинаковым успехом.

MAVEN

Maven используется для автоматизации сборки проектов с использованием Java. Он помогает составить схему сборки конкретного программного обеспечения, а также его различных зависимостей. Он использует XML-файл для описания проекта, который вы собираете, зависимостей программного обеспечения от сторонних модулей и частей, порядка сборки, а также необходимых плагинов. Существуют заранее определенные цели для таких задач, как упаковка и компиляция. Для абсолютного большинства проектов определяемая соглашением структура Maven прекрасно подходит, так что менять ее обычно смысла не имеет.

MAVEN

1. Здесь есть четкая структура каталогов, и вы обязательно должны ей следовать. (При использовании плагинов IDE, она создается автоматически).
2. Каждый билд имеет атрибуты groupId, artifactId и version.

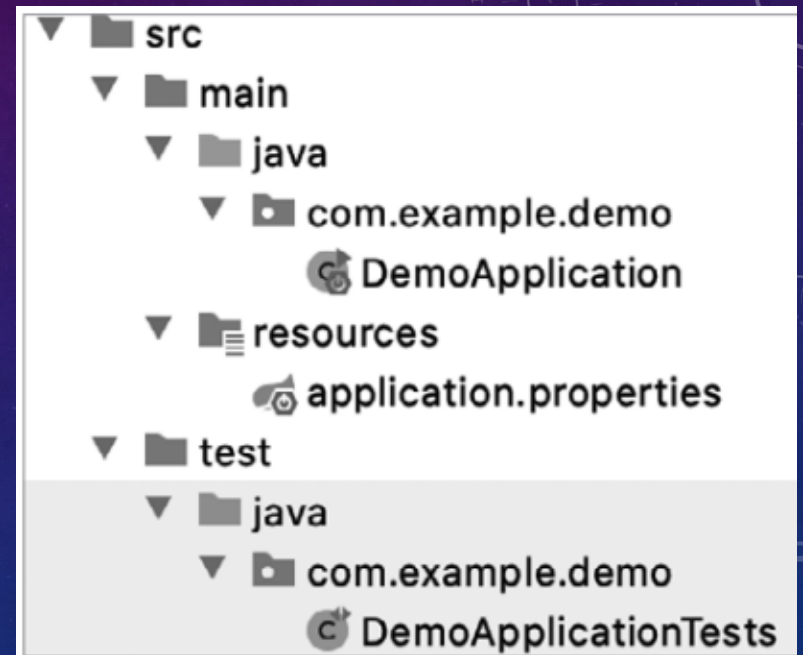
Первые два уникальны и используются для определения в репозитории.

Обычно groupId - доменное имя организации, а artifactId - название текущего проекта. Поэтому эта пара и является определяющей: нет двух компаний с одинаковым доменом, как и не существует двух одинаковых проектов в одной компании.

MAVEN

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
  </dependency>
</dependencies>
</project>
```

Управление зависимости проекта
Maven



Spring Boot с типичной структурой
проекта Maven

GRADLE

Maven использует XML для конфигурирования сборки. Gradle - предметно-ориентированный язык на основе Groovy. И это его основное отличие от Maven. В остальном Gradle руководствуется теми же принципами. Здесь за выполнение всей работы также ответственны плагины и нет широкой свободы действий.

```
apply plugin: 'java'

repositories {
    mavenCentral()
}

jar {
    baseName = 'gradleExample'
    version = '0.0.1-SNAPSHOT'
}

dependencies {
    compile 'junit:junit:4.12'
}
```

Пример файла build.gradle проекта с HelloWorld

ВЫБОР МЕЖДУ MAVEN И GRADLE

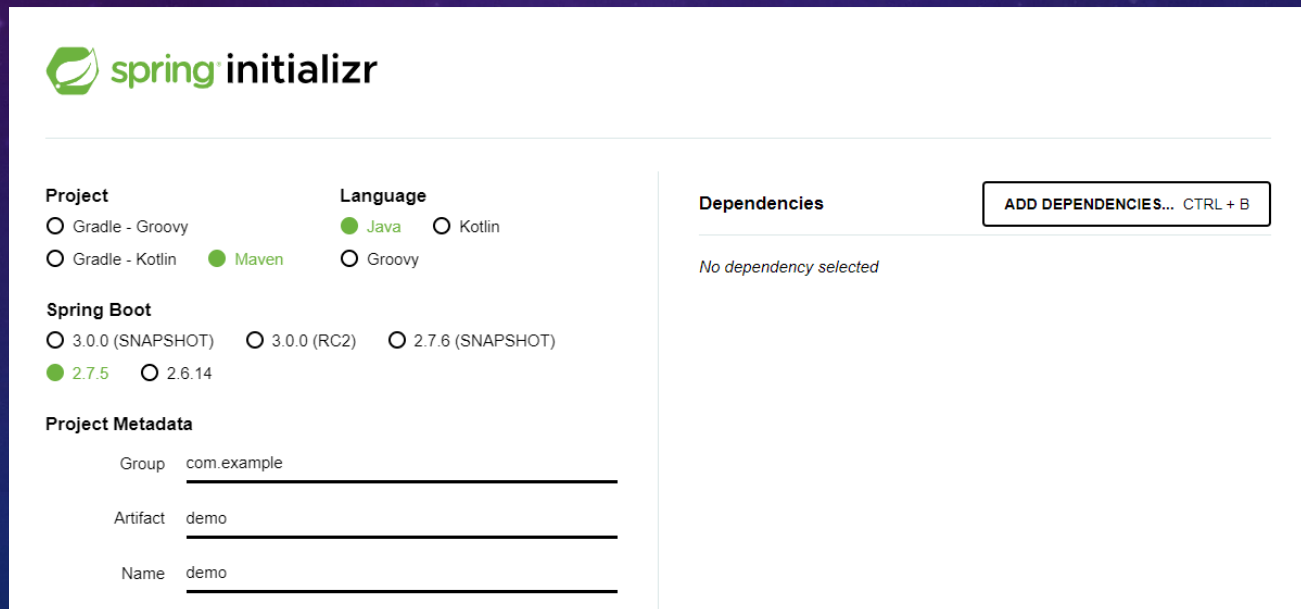
Более жесткий декларативный, можно даже сказать, догматичный подход Maven значительно повышает единообразие процесса сборки для различных проектов и сред. Если следовать пути Maven, проблемы практически исключены и можно сосредоточиться на коде, не возясь со сборкой.

Для простых проектов, а также проектов с очень сложными требованиями к сборке гибкость Gradle может оказаться лучшим вариантом. Но дополнительная гибкость Gradle, особенно в сложных проектах, означает и большее количество времени, потраченного на тонкую настройку, диагностику и устранение неполадок, когда все работает не так, как ожидалось.

ПЕРВЫЙ SPRING BOOT REST API

Существует множество способов создания приложений Spring Boot, но отправная точка у большинства одна — Spring Initializr. Для того, чтобы воспользоваться им достаточно перейти по URL:

<https://start.spring.io/>



The image shows the Spring Initializr web form. It has a white background with a green Spring logo and the text 'spring initializr'. The form is divided into several sections: 'Project' with radio buttons for 'Gradle - Groovy', 'Gradle - Kotlin', and 'Maven' (selected); 'Language' with radio buttons for 'Java' (selected), 'Kotlin', and 'Groovy'; 'Spring Boot' with radio buttons for '3.0.0 (SNAPSHOT)', '3.0.0 (RC2)', '2.7.6 (SNAPSHOT)', '2.7.5' (selected), and '2.6.14'; 'Project Metadata' with input fields for 'Group' (com.example), 'Artifact' (demo), and 'Name' (demo); and 'Dependencies' with a button 'ADD DEPENDENCIES... CTRL + B' and the text 'No dependency selected'.

spring initializr

Project

☐ Gradle - Groovy ☒ Gradle - Kotlin ☒ Maven

Language

☒ Java ☐ Kotlin ☐ Groovy

Spring Boot

☐ 3.0.0 (SNAPSHOT) ☐ 3.0.0 (RC2) ☐ 2.7.6 (SNAPSHOT) ☒ 2.7.5 ☐ 2.6.14

Project Metadata

Group

Artifact

Name

Dependencies ADD DEPENDENCIES... CTRL + B

No dependency selected

ЧТО ТАКОЕ REST И API?

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) — это набор правил того, как программисту организовать написание кода серверного приложения, чтобы все системы легко обменивались данными и приложение можно было масштабировать.

REST API — репрезентативная передача состояния (REST) — это архитектурный стиль, который осуществляет реализацию клиента и сервера независимо друг от друга. Сервисы в REST API взаимодействуют по протоколу HTTP.

HTTP-КОМАНДЫ

REST API обычно создаются на основе следующих HTTP-команд:

- POST
- GET
- PUT
- PATCH
- DELETE

Эти команды соответствуют типовым операциям над ресурсами: созданию (POST), чтению (GET), обновлению (PUT и PATCH) и удалению (DELETE).

JAVA PROFESSIONAL

ПРИНЦИПЫ ПРОГРАММИРОВАНИЯ: DRY, YAGNI, KISS, SOLID.

ПРИНЦИПЫ ПРОГРАММИРОВАНИЯ.SOLID

Принцип **SOLID** в упрощенном варианте означает, что когда при написании кода используется несколько принципов вместе, то это значительно облегчает дальнейшую поддержку и развитие программы. Полностью акроним расшифровывается так:

Single responsibility principle – принцип единственной обязанности на каждый класс должна быть возложена одна-единственная обязанность. Класс должен быть ответственен лишь за что-то одно.

Open/closed principle – принцип открытости/закрытости (программные сущности должны быть закрыты для изменения, но открыты для расширения)

SOLID

Liskov substitution principle – принцип подстановки Барбары Лисков (функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа, не зная об этом. Подклассы не могут замещать поведения базовых классов. Подтипы должны дополнять базовые типы)

Interface segregation principle – принцип разделения интерфейса (много специализированных интерфейсов лучше, чем один универсальный)

Dependency inversion principle – принцип инверсии зависимостей (зависимости внутри системы строятся на основе абстракций. Модули верхнего уровня не зависят от модулей нижнего уровня. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций)

YAGNI. KISS. DRY. WET.

Термин **YAGNI** значит You Ain't Gonna Need It – **вам это не понадобится!** Его суть в том, чтобы реализовать только поставленные задачи и отказаться от избыточного функционала.

KISS – Keep It Simple, Stupid – не усложняй! Смысл этого принципа программирования заключается в том, что стоит делать максимально простую и понятную архитектуру, применять шаблоны проектирования и не изобретать велосипед.

DRY - don't repeat yourself
Or (DIE) duplication is evil

Нарушения принципа **DRY** называют **WET** — «Write Everything Twice» (Пиши всё по два раза) или «We enjoy typing» (Нам нравится печатать). Это игра английских слов «dry» (сухой) и «wet» (влажный).

OVERENGINEERING В ПРОГРАММИРОВАНИИ

Overengineering (Избыточный инжиниринг) - процесс разработки продукта, который должен быть более надежным или иметь больше функций, чем часто необходимо для его предполагаемого использования, или чтобы процесс был излишне сложным или неэффективным. Другими словами: **«Код или дизайн, решающий проблемы, которых у вас нет».**

Честно говоря, это не очень полезно — особенно для разработчиков программного обеспечения.

КАК ИЗБЕЖАТЬ ОШИБКИ?

Ваши методы должны быть небольшими, не превышающими 40-50 строк.

Каждый метод должен решать только одну проблему.

У вас в проекте много условий? Убедитесь, что вы разбили их на более мелкие блоки кода.

Дважды проверьте все требования проекта, чтобы убедиться, что вы ничего не упускаете и не добавляете лишнего в свой код.

При написании кода не повторяйтесь. То есть избегайте копирования кода в разные места. В противном случае дальнейшее обслуживание будет трудным. Причина в том, что вам придется изменять код в разных местах.

JAVA PROFESSIONAL

REFLECTIONS. ANNOTATIONS.

JAVA REFLECTION API

Java Reflection — это особенный функционал, который позволяет программе получить доступ к приватным частям объектов или поменять поведение некоторых методов классов. Созданный таким образом код будет адаптироваться к входным данным и, например, не будет зависеть от типов, с которыми работает.

Это дает возможность писать код, который со временем будет эволюционировать, то есть не зависеть от текущих имплементаций методов или переменных. Главные преимущества рефлексии — свобода и адаптивность. При необходимости вызвать приватный метод класса можно не переписывать его, а вызвать через Java Reflection. Фактически рефлексия позволяет не следовать написанному коду, вводя новые правила. Можно пойти чуть дальше и начать перехватывать вызовы методов, подменяя их другой логикой, или создать программу, которая будет работать с еще не написанным классом.

Чтобы использовать Java Reflection API, не нужно подключать сторонние библиотеки. Все расположено в пакете [java.lang.reflect](#).

METHODS IN JAVA.LANG.REFLECT

- `getName()` — возвращает имя и пакет класса;
- `getSimpleName()` — возвращает имя класса без пакета;
- `getModifiers()` — возвращает модификаторы класса;
- `getSuperclass()` — возвращает родительский класс;
- `getInterfaces()` — возвращает список интерфейсов, которые наследует класс;
- `getConstructors()` — возвращает список конструкторов класса;
- `getFields()` — возвращает список публичных полей класса;
- `getDeclaredFields()` — возвращает список всех полей класса, в том числе приватных;
- `getMethods()` — возвращает массив публичных методов класса;
- `getDeclaredMethods()` — возвращает массив всех методов класса, в том числе приватных;
- `getPackage()` — возвращает имя пакета класса.

ПРИМЕНЕНИЕ РЕФЛЕКСИИ В JAVA

С помощью интерфейса Java Reflection API можно делать следующее:

1. Определить класс объекта.
2. Получить информацию о модификаторах класса, полях, методах, конструкторах и суперклассах.
3. Выяснить, какие константы и методы принадлежат интерфейсу.
4. Создать экземпляр класса, имя которого неизвестно до момента выполнения программы.
5. Получить и установить значение свойства объекта.
6. Вызвать метод объекта.
7. Создать новый массив, размер и тип компонентов которого неизвестны до момента выполнения программ.

АННОТАЦИИ В JAVA

Аннотации - это форма метаданных. Они предоставляют информацию о программе, при том сами частью программы не являются.

Метаданные — это способ добавить дополнительную информацию к исходному коду

Применение:

Информация для компилятора. Могут использоваться компилятором для обнаружения ошибок и подавления предупреждений.

Обработка во время компиляции и развертывания. Программа может создавать код, XML-файлы и т.п. на основе аннотаций.

Обработка во время выполнения. Некоторые аннотации могут использоваться во время выполнения программы.

КЛАССИФИКАЦИЯ АННОТАЦИЙ

Аннотации можно классифицировать по следующим признакам:

- аннотации для аннотаций
- аннотации типов
- аннотации для кода
- нативные аннотации
- аннотации, написанные программистом

КЛАССИФИКАЦИЯ АННОТАЦИЙ

Аннотации для аннотаций еще называют мета-аннотациями:

- @Target: указывает контекст, для которого применима аннотация

Аннотации типов:

@Retention: указывает, до какого шага во время компиляции аннотация будет доступна

- @Documented: указывает, что аннотация должна быть задокументирована с помощью javadoc
- @Inherited: позволяет реализовать наследование аннотаций родительского класса классом-наследником
- @Repeatable: указывает, что аннотация может быть использована повторно в том же месте

КЛАССИФИКАЦИЯ АННОТАЦИЙ

Аннотации для кода:

- **@Override**: указывает, что метод переопределяет, объявленный в суперклассе или интерфейсе метод
- **@Deprecated**: помечает код, как устаревший
- **@SuppressWarnings**: отключает для аннотированного элемента предупреждения компилятора. Обратите внимание, что если необходимо отключить несколько категорий предупреждений, их следует добавить в фигурные скобки, например `@SuppressWarnings ({"unchecked", "cast"})`.
- **@SafeVarargs**: отключает предупреждения для всех методов или конструкторов, принимающих в качестве параметра `varargs`
- **@FunctionalInterface**: помечает интерфейсы, имеющие только один абстрактный метод (при этом они могут содержать любое количество методов по умолчанию или статических)

КЛАССИФИКАЦИЯ АННОТАЦИЙ

Аннотация @Native

Начиная с Java 8, в пакете `java.lang.annotation` появилась новая аннотация под названием `@Native`, применимая только к полям. Она указывает, что аннотированное поле является константой, на которую можно ссылаться с нативного кода.

КАК НАПИСАТЬ СВОЮ АННОТАЦИЮ?

Чтобы создать аннотацию, мы используем ключевое слово `interface` и добавляем перед ним символ `@`. Символ `@` сообщит компилятору, что это аннотация.

Все аннотации расширяют `java.lang.annotation.Annotation` интерфейс, что означает, что `java.lang.annotation.Annotation` - это суперинтерфейс всех аннотаций.

Аннотация должна иметь `RetentionPolicy` область действия аннотации, в которой в этот момент аннотация будет игнорироваться или отбрасываться. Если политика хранения не определена, будет использоваться политика хранения по умолчанию, которая представляет собой файл `RetentionPolicy.CLASS`.

JAVA PROFESSIONAL

SPRING FRAMEWORK INTRODUCTION, CONTEXT AND BEANS.

SPRING FRAMEWORK

Технология, с которой больше всего ассоциируется Spring, — это внедрение зависимостей (DI) в Inversion of Control.

По сути Spring Framework представляет собой просто **контейнер внедрения зависимостей**, с несколькими удобными слоями.

Например: доступ к базе данных, прокси, **аспектно-ориентированное программирование**, RPC, веб-инфраструктура MVC). Это все позволяет нам быстрее и удобнее создавать Java-приложения.

IoC

Инверсия управления — это принцип разработки программного обеспечения, который отделяет управление объектами или то, как вызываются части программы, от реализации. Подобно голливудскому принципу «Не звоните нам, мы позвоним вам», инфраструктура IoC освобождает разработчика от управления вызовом кода. Это «инверсионный» аспект принципа инверсии управления.

Преимущества этой архитектуры:

- отделение выполнения задачи от ее реализации
- упрощение переключения между различными реализациями
- большая модульность программы
- проще тестировать программу, изолируя компонент или имитируя его зависимости и позволяя компонентам взаимодействовать через контракты.

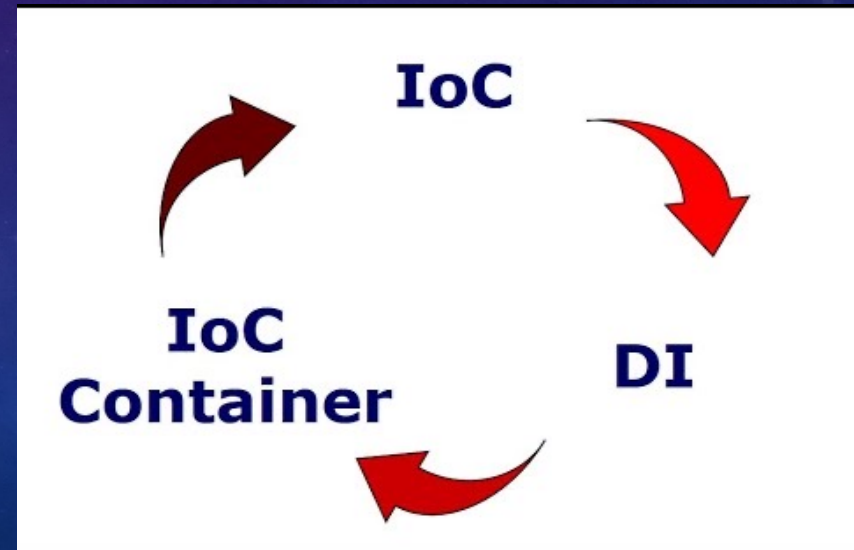
DI AND IOC-CONTAINER

Внедрение зависимостей в основном предоставляет объекты, которые нужны объекту (его зависимости), вместо того, чтобы создавать их.

Контейнер Spring IoC — это реализация Spring, использующая шаблон IoC в паре с DI. По сути, это программный контейнер, предоставляющий настраиваемый контекст приложения, в котором создаются, инициализируются, кэшируются и управляются подключаемые объекты, известные как bean -компоненты.

В Spring имеется 2 различных вида контейнеров:

1. Spring BeanFactory Container
2. Spring ApplicationContext Container



BEANS AND BEANFACTORY

Spring-бины – это классы, созданием экземпляров которых и установкой в них зависимостей управляет контейнер фреймворка Spring. Бины предназначены для реализации бизнес-логики приложения. Spring Bean представляет собой singleton, то есть в некотором блоке приложения существует только один экземпляр данного класса.

BeanFactory содержит определения **bean-компонентов** и создает их экземпляры всякий раз, когда этого требует клиентское приложение, что означает:

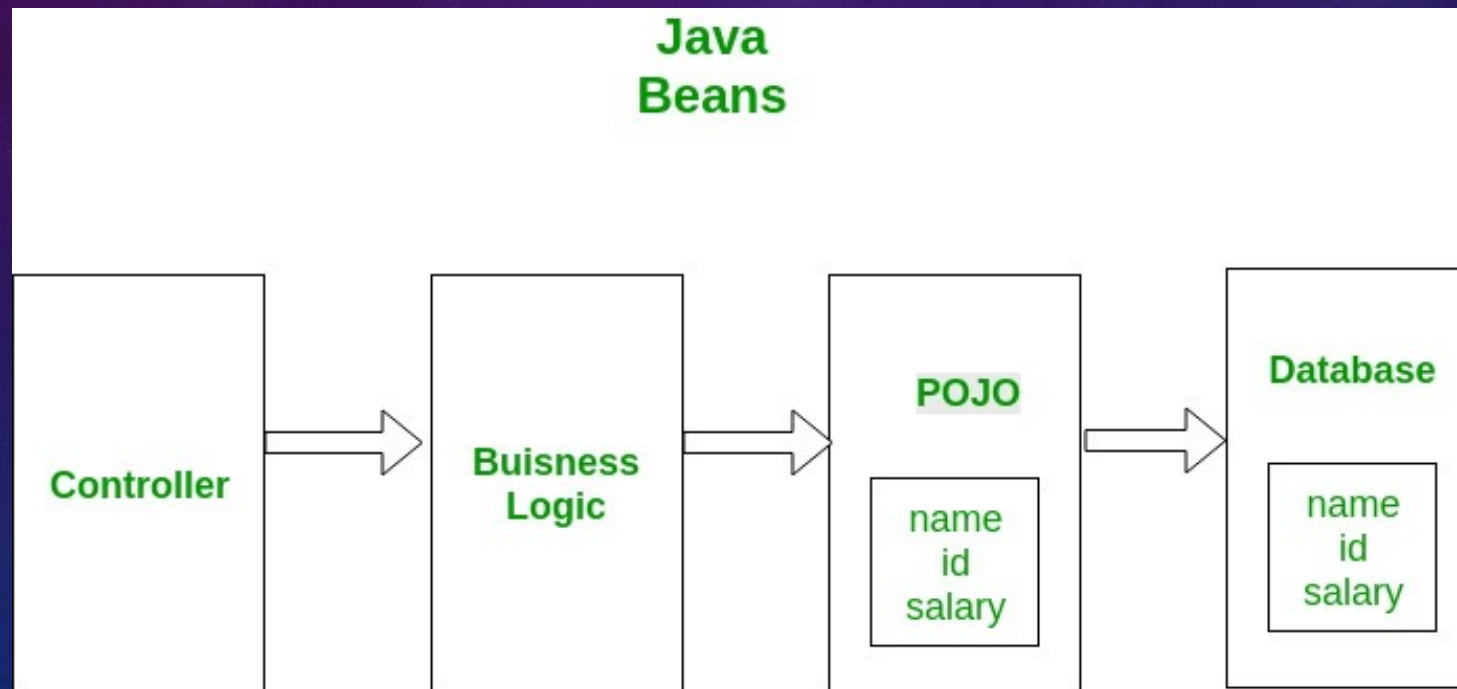
Он заботится о жизненном цикле компонента, создавая его экземпляры и вызывая соответствующие методы уничтожения.

Он способен создавать ассоциации между зависимыми объектами при их создании.

Важно отметить, что BeanFactory не поддерживает внедрение зависимостей на основе аннотаций, в то время как ApplicationContext, надмножество BeanFactory, поддерживает.

POJO

POJO(Plain Old Java Object) — «старый добрый Java-объект», простой Java-объект, не унаследованный от какого-то специфического объекта и не реализующий никаких служебных интерфейсов сверх тех, которые нужны для бизнес-модели.



JAVA PROFESSIONAL

SPRING FRAMEWORK: BEANS CONFIGURATION

BEAN LIFECYCLE

Жизненный цикл любого объекта означает следующее: как и когда он появляется, как он себя ведет во время жизни и как и когда он исчезает. Жизненный цикл бина ровно про это же: «как и когда».

Жизненным циклом управляет спринг-контейнер.

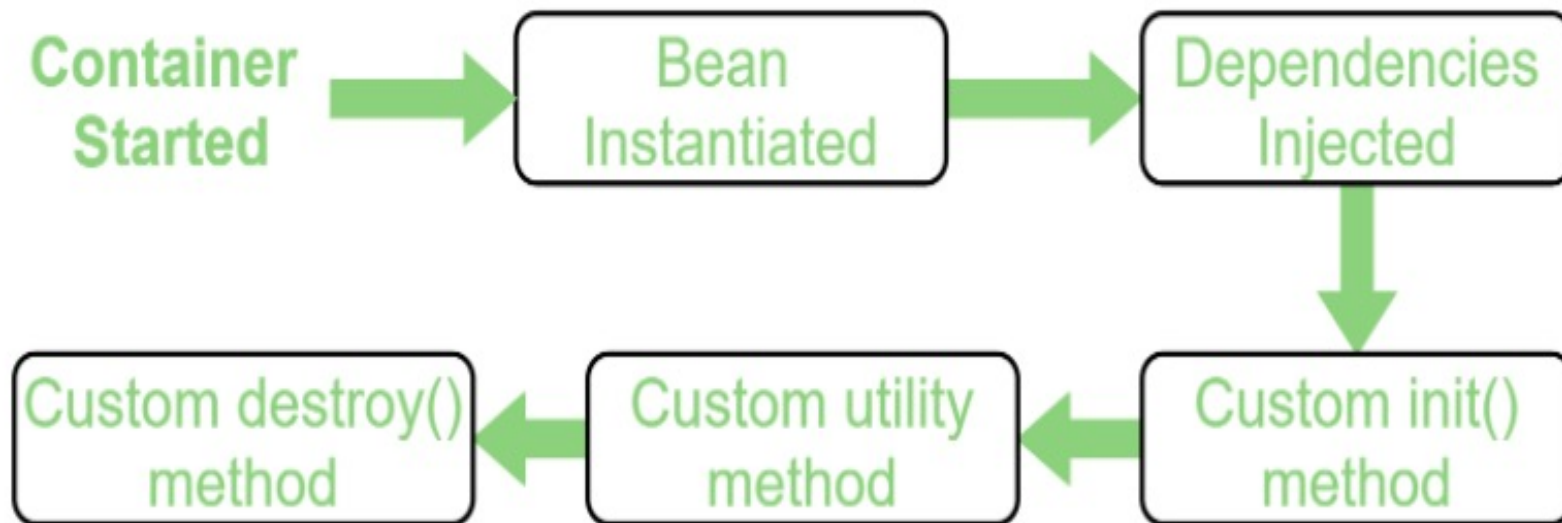
В первую очередь после запуска приложения запускается именно он.

После этого контейнер по необходимости и в соответствии с запросами создает экземпляры бинов и внедряет необходимые зависимости.

И, наконец, бины, связанные с контейнером, уничтожаются когда контейнер завершает свою работу.

Поэтому, если мы хотим выполнить какой-то код во время инстанцирования бина или сразу после завершения работы контейнера, один из самых доступных нам вариантов это вынести его в специальные `init()` и `destroy()` методы.

BEAN LIFECYCLE



CONFIGURATION WITH JAVA

Стоит упомянуть, что в Spring Framework поддерживается конфигурация с помощью Java, что временами бывает удобно. Это позволяет нам настроить большую часть Spring-приложения без использования конфигурационного файла XML, используя специальные аннотации.

В конфигурации с помощью аннотаций Java, ключевыми являются **@Configuration** и **@Bean**

CONFIGURATION WITH JAVA

@Configuration

Эта аннотация, прописанная перед классом, означает, что класс может быть использован контейнером Spring IoC как конфигурационный класс для бинов.

@Bean

Аннотация @Bean, прописанная перед методом, информирует Spring о том, что возвращаемый данным методом объект должен быть зарегистрирован, как бин.

CONFIGURATION WITH ANNOTATIONS

Благодаря конфигурации с помощью аннотаций мы можем связывать между собой бины вставив аннотации непосредственно в Java-класс (доступны аннотации к классам, методом и полям).

Существует несколько видов распространённых аннотаций:

- @Required
- @Autowired
- @Qualifier
- JSR-250 Annotations (@PostConstruct и @PreDestroy)

@REQUIRED, @AUTOWIRED, @QUALIFIER

Аннотации **@Required** применяется к методам-сеттерам и означает, что значение метода должно быть установлено в XML-файле. Если этого не будет сделано, то мы получим `BeanInitializationException`.

Аннотация **@Autowired** обеспечивает контроль над тем, где и как автосвязывание должны быть осуществлено. Мы можем использовать `@Autowired` как для методов, так и для конструкторов.

В реальной жизни может сложиться ситуация, когда вы создаёте несколько бинов одного и того же типа, но в конкретном случае вам необходим конкретный бин. Для того, чтобы указать Spring, какой именно бин вам необходим, применяется аннотация **@Qualifier**.

@POSTCONSTRUCT, @PREDESTROY

Для того, чтобы настроить создание и уничтожения бина мы просто указываем методы `init-method` и `destroy-method` при объявлении бина.

Аттрибут `init-method` определяет метод, который будет вызван сразу, после создания экземпляра бина, а `destroy-method` – определяет метод, который будет вызван непосредственно перед уничтожением.

В Spring Framework также предусмотрена возможность использовать аннотации **@PostConstruct** и **@PreDestroy**, которые равнозначный `init-method` и `destroy-method` соответственно.

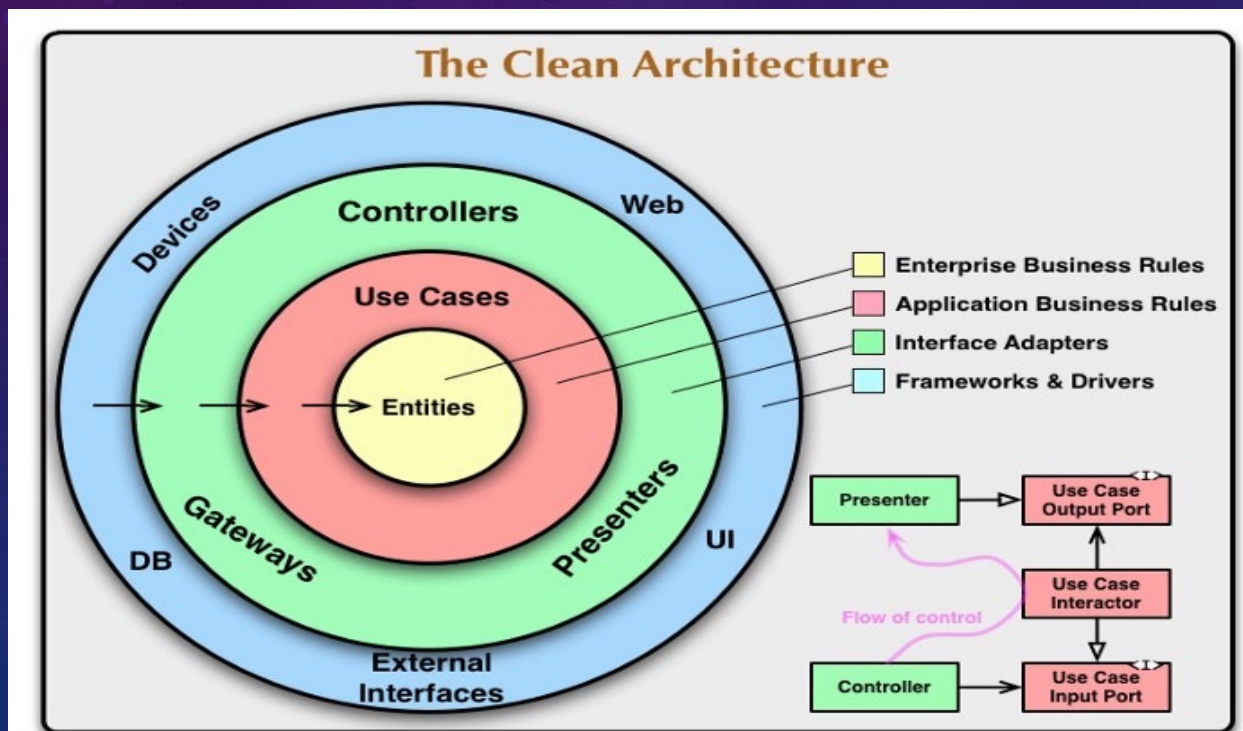
JAVA PROFESSIONAL

SPRING BOOT: ONION (HEXAGONAL) ARCHITECTURE

CLEAN ARCHITECTURE

Чистая архитектура — это философия дизайна программного обеспечения, которая разделяет компоненты на определенные уровни.

Если компонент зависит от всех других компонентов, мы не знаем, какие побочные эффекты будет иметь изменение одного компонента, что затрудняет поддержку кодовой базы и еще более затрудняет ее расширение и декомпозицию.



DDD

Domain Drive Design(предметно-ориентированное проектирование) - подход к разработке приложений, основанный на выделении доменов (domain).

Использование подхода «Domain driven design» при проектировании архитектуры сложных корпоративных информационных систем позволяет сделать проще поддержание изменений в проекте и эффективнее тестировать новые релизы.

Домен — область знаний/деятельности, для которой разрабатывается приложение.

Модель — система абстракций, которая описывает отдельные характеристики домена. Как и физическая модель, упрощающая понимание и изучение объекта, помогает решить проблемы, связанные с данным доменом.

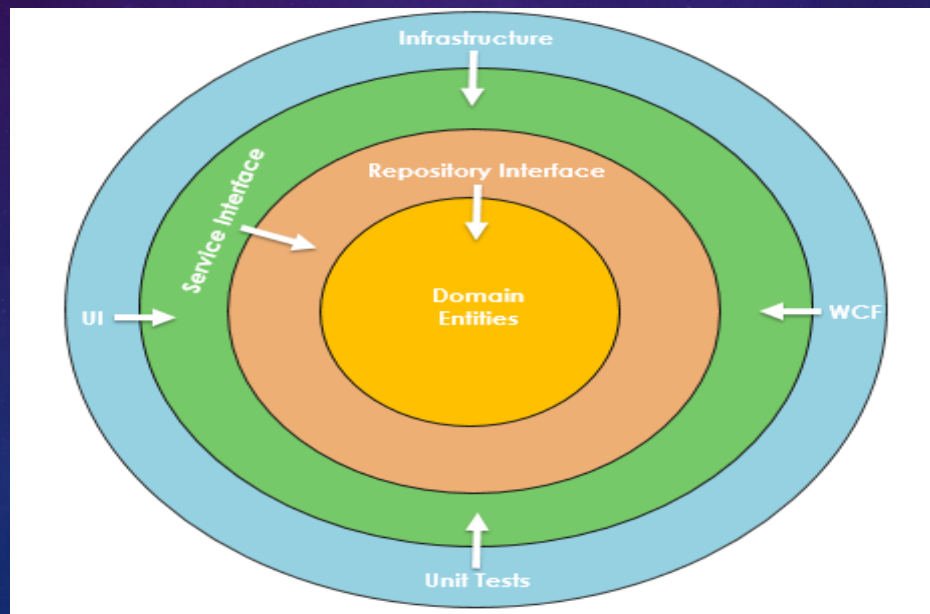
Сервисы — это функции, которые не привязаны к сущностям или ценностным объектам. Грубо говоря, действие в себе.

Репозитории — это такие сервисы, которые используют глобальный интерфейс, чтобы обеспечить доступ ко всем сущностям и ценностным объектам, находящимся в конкретной группе агрегатов.

HEXAGONAL ARCHITECTURE

Шестиугольная архитектура — это модель разработки программных приложений на основе доменной логики, чтобы изолировать ее от внешних факторов.

Логика предметной области указана в бизнес-ядре, которое мы назовем внутренней частью, а остальные — внешними частями. Доступ к логике домена извне возможен через порты и адаптеры.



LAYERS OF THE ONION ARCHITECTURE

Слой домена - в центральной части луковой архитектуры существует доменный уровень; этот слой представляет объекты бизнеса и поведения. Идея состоит в том, чтобы иметь все объекты домена в этом ядре. Он содержит все объекты домена приложения. Помимо объектов домена, у вас также могут быть интерфейсы домена. Эти объекты домена не имеют никаких зависимостей.

Слой репозитория - этот уровень создает абстракцию между объектами предметной области и бизнес-логикой приложения. На этом уровне мы обычно добавляем интерфейсы, которые обеспечивают поведение сохранения и извлечения объектов, как правило, с использованием базы данных. Этот уровень состоит из шаблона доступа к данным, который представляет собой более слабо связанный подход к доступу к данным. Мы также создаем общий репозиторий и добавляем запросы для извлечения данных из источника, сопоставления данных из источника данных с бизнес-объектом и сохранения изменений в бизнес-объекте в источнике данных.

LAYERS OF THE ONION ARCHITECTURE

Уровень сервисов - Уровень сервиса содержит интерфейсы с общими операциями, такими как «Добавить», «Сохранить», «Редактировать» и «Удалить». Кроме того, этот уровень используется для связи между уровнем пользовательского интерфейса и уровнем репозитория. Уровень сервисов также может содержать бизнес-логику для сущности. На этом уровне интерфейсы служб отделены от их реализации, учитывая слабую связь и разделение задач.

Слой пользовательского интерфейса - Это самый внешний уровень, в котором хранятся периферийные функции, такие как пользовательский интерфейс и тесты. Для веб-приложения он представляет проект веб-API или модульного тестирования. На этом уровне реализован принцип внедрения зависимостей, так что приложение создает слабосвязанную структуру и может взаимодействовать с внутренним уровнем через интерфейсы.

JAVA PROFESSIONAL

SPRING BOOT: IMPLEMENTING REST CONTROLLERS

DISPATCHERServlet

DispatcherServlet (Controller) - это главный контроллер в приложении Spring MVC, который обрабатывает все входящие запросы и передает их для обработки в различные методы в контроллеры.

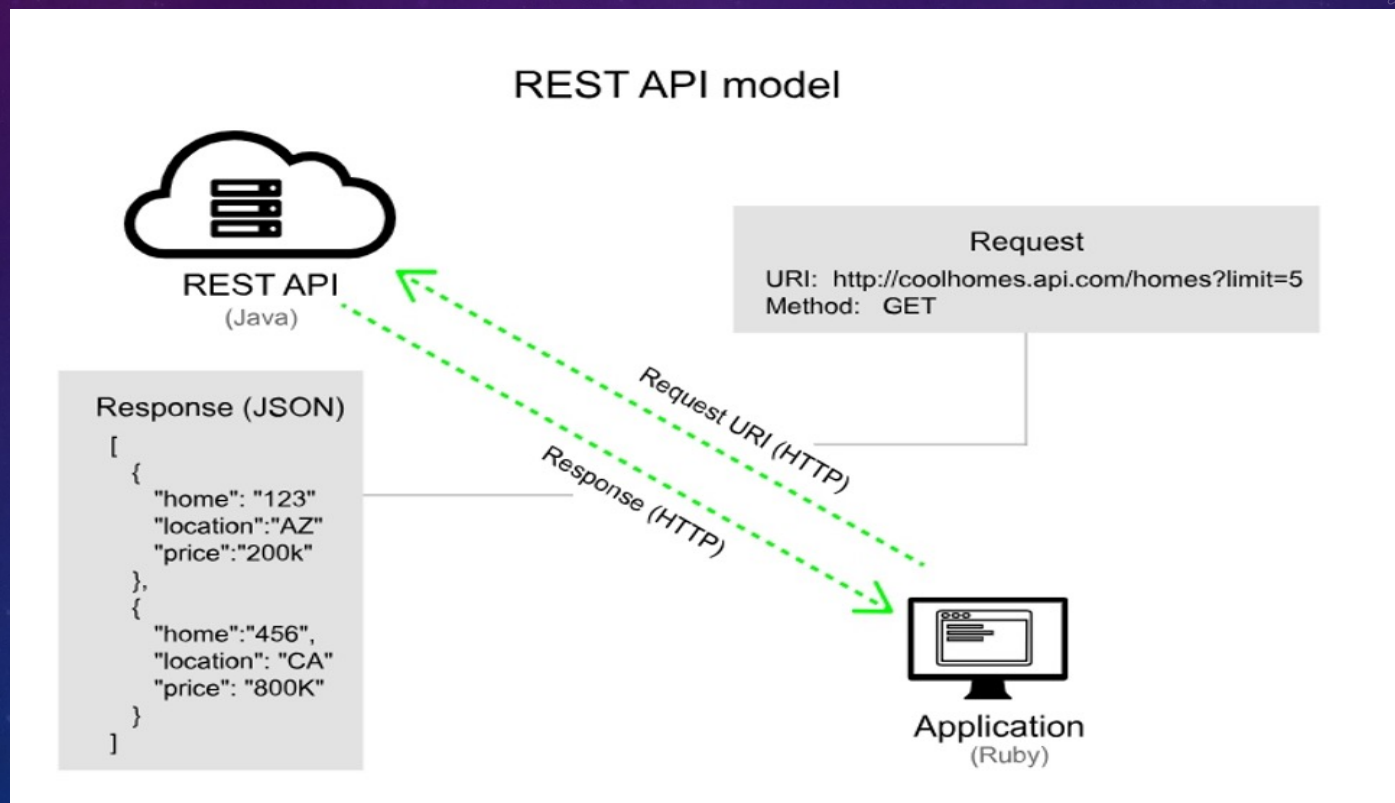
HTTPSERVLETS

При написании веб-приложений на Java с использованием Spring или без него (MVC/Boot) вы в основном имеете в виду написание приложений, которые возвращают два разных формата данных:

- **HTML** → Ваше веб-приложение создает HTML-страницы, которые можно просматривать в браузере.
- **JSON/XML** → Ваше веб-приложение предоставляет сервисы RESTful, которые генерируют JSON или XML. Сайты с большим количеством Javascript или даже другие веб-сервисы могут затем использовать данные, которые предоставляют эти сервисы.

WHAT IS REST?

REST (от англ. Representational State Transfer — «передача состояния представления») – это общие принципы организации взаимодействия приложения/сайта с сервером посредством протокола HTTP.



WHAT IS REST?

Всё взаимодействие с сервером сводится к 4 операциям (4 — это необходимый и достаточный минимум, в конкретной реализации типов операций может быть больше):

- Получение данных с сервера (обычно в формате JSON, или XML);
- Добавление новых данных на сервер;
- Модификация существующих данных на сервере;
- Удаление данных на сервере

REST CONTROLLERS

Аннотация Spring **RestController** — это удобная аннотация, которая сама по себе снабжена аннотациями `@Controller` и `@ResponseBody`.

Эта аннотация применяется к классу, чтобы пометить его как обработчик запросов.

Аннотация Spring `RestController` используется для создания веб-сервисов RESTful с использованием Spring MVC.

Spring `RestController` заботится о сопоставлении данных запроса с определенным методом обработчика запросов. Как только тело ответа генерируется из метода обработчика, оно преобразует его в ответ JSON или XML.

JAVA PROFESSIONAL

JAVA JDBC API, РАБОТА С БАЗАМИ ДАННЫХ

JDBC DRIVER MANAGER

JDBC (Java Database Connectivity) — это платформенно независимый промышленный стандарт взаимодействия Java-приложений с реляционными базами данных.

JDBC управляет:

- подключением к базе данных;
- выдачей запросов и команд;
- обработкой данных, полученных из базы.

DriverManager - это синглтон, который содержит информацию о всех зарегистрированных драйверах. Метод `getConnection` на основании параметра URL находит `java.sql.Driver` соответствующей базы данных и вызывает у него метод `connect`.

HOW IT WORKS?

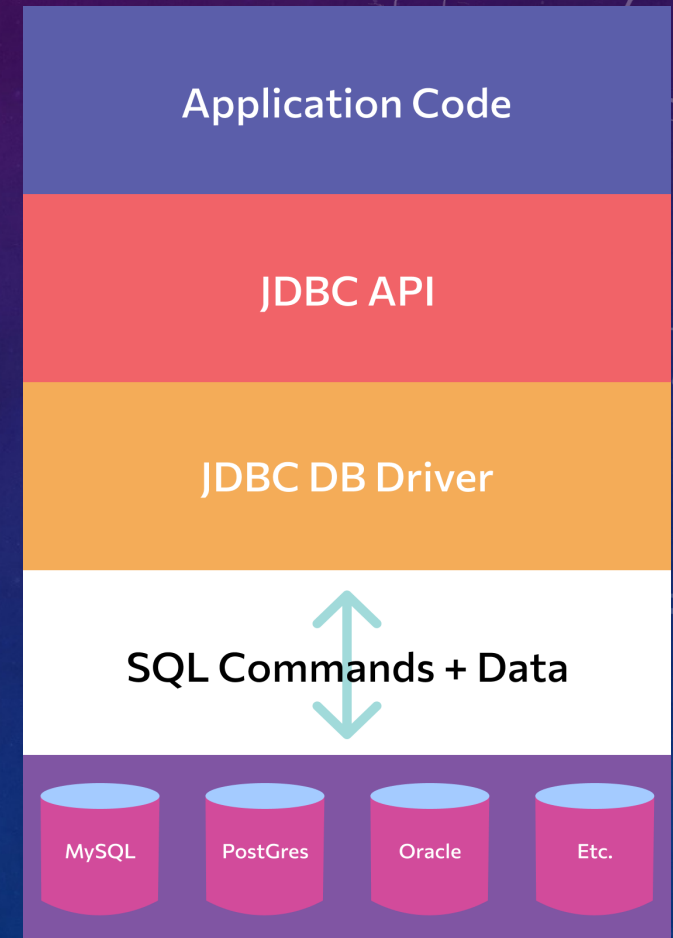
Пакет JDBC состоит из двух главных компонентов:

1. **API** (программного интерфейса), который поддерживает связь между Java-приложением и менеджером JDBC;
2. **Драйвера JDBC**, который поддерживает связь между менеджером JDBC и драйвером базы данных.

Соединение с базой устанавливается по особому URL.

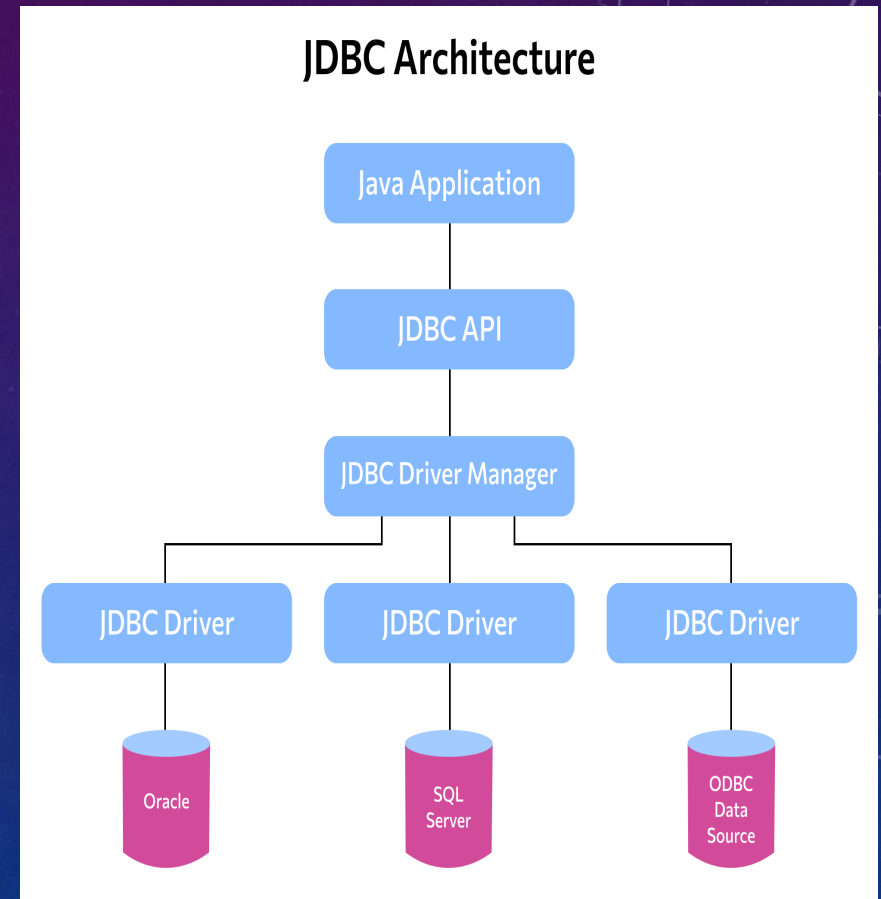
При этом разработчику не нужно знать специфику конкретной базы — API выступает в качестве посредника между базой и приложением.

Это упрощает как процесс создания приложения, так и переход на базу данных другого типа.



JDBC ARCHITECTURE

1. Установка базы данных на сервер или выбор облачного сервиса, к которому нужно получить доступ.
2. Подключение библиотеки JDBC.
3. Проверка факта нахождения необходимого драйвера JDBC в classpath.
4. Установление соединения с базой данных с помощью библиотеки JDBC.
5. Использование установленного соединения для выполнения команд SQL.
6. Закрытие соединения после окончания сеанса.



JDBC INTERFACES

В JDBC есть 3 основных интерфейса:

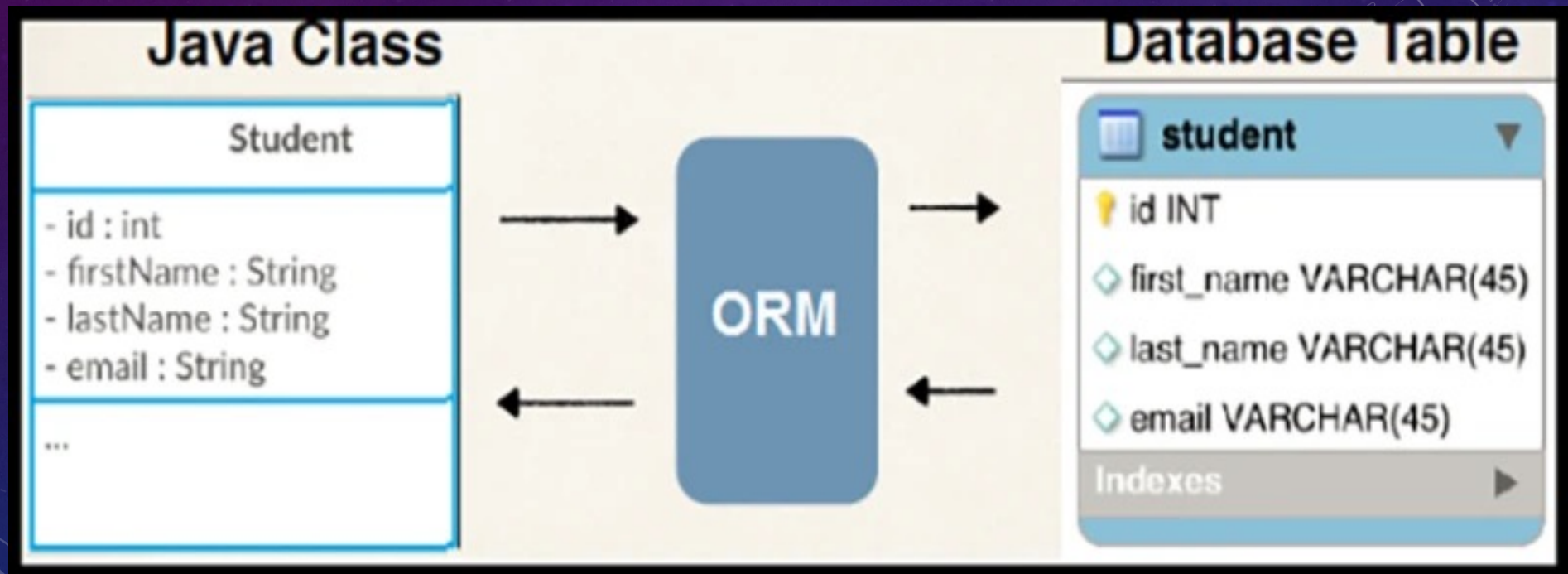
1. Connection – отвечает за соединение с базой данных
2. Statement – отвечает за запрос к базе данных
3. ResultSet – отвечает за результат запроса к базе данных

JAVA PROFESSIONAL

SPRING BOOT: JPA AND DATABASES

ORM

ORM — Object-Relational Mapping или в переводе на русский объектно-реляционное отображение. Это технология программирования, которая связывает базы данных с концепциями объектно-ориентированных языков программирования. Если упростить, то ORM это связь Java объектов и записей в БД



SPRING DATA JPA

Репозитории **Spring Data JPA** - это интерфейсы, которые вы можете определить для доступа к данным.

Запросы JPA создаются автоматически из имен ваших методов.

Например, интерфейс `CityRepository` может объявить метод `findAllByState(String state)`, чтобы найти все города (`city`) в данном штате (`state`).

ENTITIES

Сущности в JPA — это не что иное, как РОЮ, представляющие данные, которые могут быть сохранены в базе данных.

Сущность представляет собой таблицу, хранящуюся в базе данных. Каждый экземпляр сущности представляет собой строку в таблице.

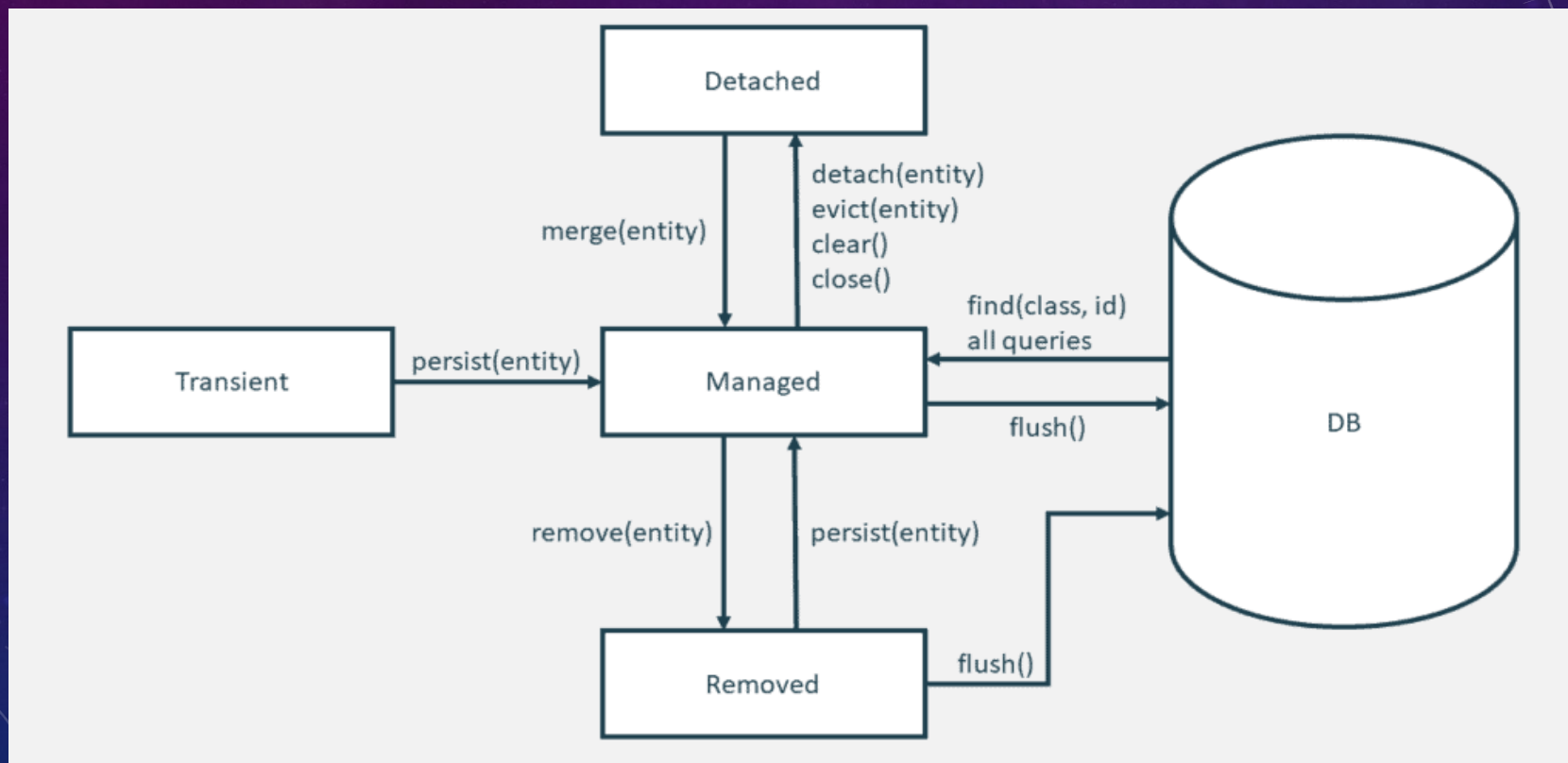
JPA'S PERSISTENCE CONTEXT

Контекст персистентности — одна из основных концепций JPA. Вы можете думать об этом как о наборе всех объектов сущностей, которые вы использовали в своем Use Case. Каждый из них представляет собой запись в базе данных.

Но напрямую мы не работаем с Persistence Context. Для этого мы используем Entity Manager или "менеджер сущностей". Именно он знает про контекст и про то, какие там живут сущности. Мы же взаимодействуем с Entity Manager'ом.

JPA'S 4 LIFECYCLE STATES

Модель жизненного цикла состоит из 4 состояний: transient(переходный), managed(управляемый), removed(удаленный), and detached(отсоединенный)



JPA'S 4 LIFECYCLE STATES

1. Transient

Состояние жизненного цикла вновь созданного объекта сущности называется переходным . Сущность еще не была сохранена, поэтому она не представляет никакой записи базы данных.

```
Student student = new Student();  
student.setFirstName("Andrew");  
student.setLastName("Brown");
```

Это меняется, когда вы предоставляете его методу EntityManager.find . Затем объект сущности меняет свое состояние жизненного цикла на управляемое и присоединяется к текущему контексту сохраняемости.

JPA'S 4 LIFECYCLE STATES

2. Managed

Все сущностные объекты, присоединенные к текущему контексту постоянства, находятся в управляемом состоянии жизненного цикла. Это означает, что ваш поставщик постоянства, например Hibernate, обнаружит любые изменения в объектах и сгенерирует необходимые операторы SQL INSERT или UPDATE, когда он сбрасывает persistence context.

```
Student student = new Student();  
student.setFirstName("Andrew");  
student.setLastName("Brown");  
em.persist(student);
```


JPA'S 4 LIFECYCLE STATES

3. Detached

Сущность, которая ранее управлялась, но больше не привязана к текущему persistence context, находится в состоянии жизненного цикла detached .

Сущность отсоединяется, когда вы закрываете persistence context. Обычно это происходит после обработки запроса. Затем транзакция базы данных фиксируется, persistence context закрывается, и объект сущности возвращается вызывающей стороне. Затем вызывающая сторона извлекает объект сущности в состоянии жизненного цикла detached .

Вы также можете программно отсоединить сущность, вызвав метод detach в EntityManager .

```
em.detach(student);
```

JPA'S 4 LIFECYCLE STATES

4. Removed

Когда вы вызываете метод удаления в вашем EntityManager, сопоставленная запись базы данных не удаляется немедленно. Сущностный объект только меняет свое состояние жизненного цикла на remove.

Во время следующей операции сброса Hibernate сгенерирует оператор SQL DELETE для удаления записи из таблицы базы данных.

```
em.remove(student);
```


JAVA PROFESSIONAL

SPRING BOOT: VALIDATION AND ERROR HANDLING

JSR-303

Проверка bean -компонентов **JSR-303** — это спецификация, целью которой является стандартизация проверки Java-бинов с помощью аннотаций. Целью стандарта JSR-303 является использование аннотаций непосредственно в классе компонентов Java.

Спецификация JSR 303 позволяет указывать правила проверки непосредственно в полях внутри любого класса Java, для проверки которых они предназначены, вместо создания правил проверки в отдельных классах.

COMMON VALIDATION ANNOTATIONS

Некоторые из наиболее распространенных аннотаций валидации:

@NotNull: поле не должно быть нулевым.

@NotEmpty: поле списка не должно быть пустым.

@NotBlank: строковое поле не должно быть пустой строкой (т. е. оно должно содержать хотя бы один символ).

@Min и **@Max**: числовое поле допустимо только тогда, когда его значение выше или ниже определенного значения.

@Pattern: строковое поле допустимо только тогда, когда оно соответствует определенному регулярному выражению.

@Email: строковое поле должно быть действительным адресом электронной почты.

@VALIDATED AND @VALID

Во многих случаях Spring выполняет проверку за нас. Нам даже не нужно самим создавать объект валидатора. Вместо этого мы можем сообщить Spring, что мы хотим проверить определенный объект. Это работает с использованием аннотаций **@Validated** и **@Valid**

Аннотация **@Validated** - это аннотация на уровне класса, которую мы можем использовать, чтобы указать Spring проверять параметры, которые передаются в метод аннотированного класса.

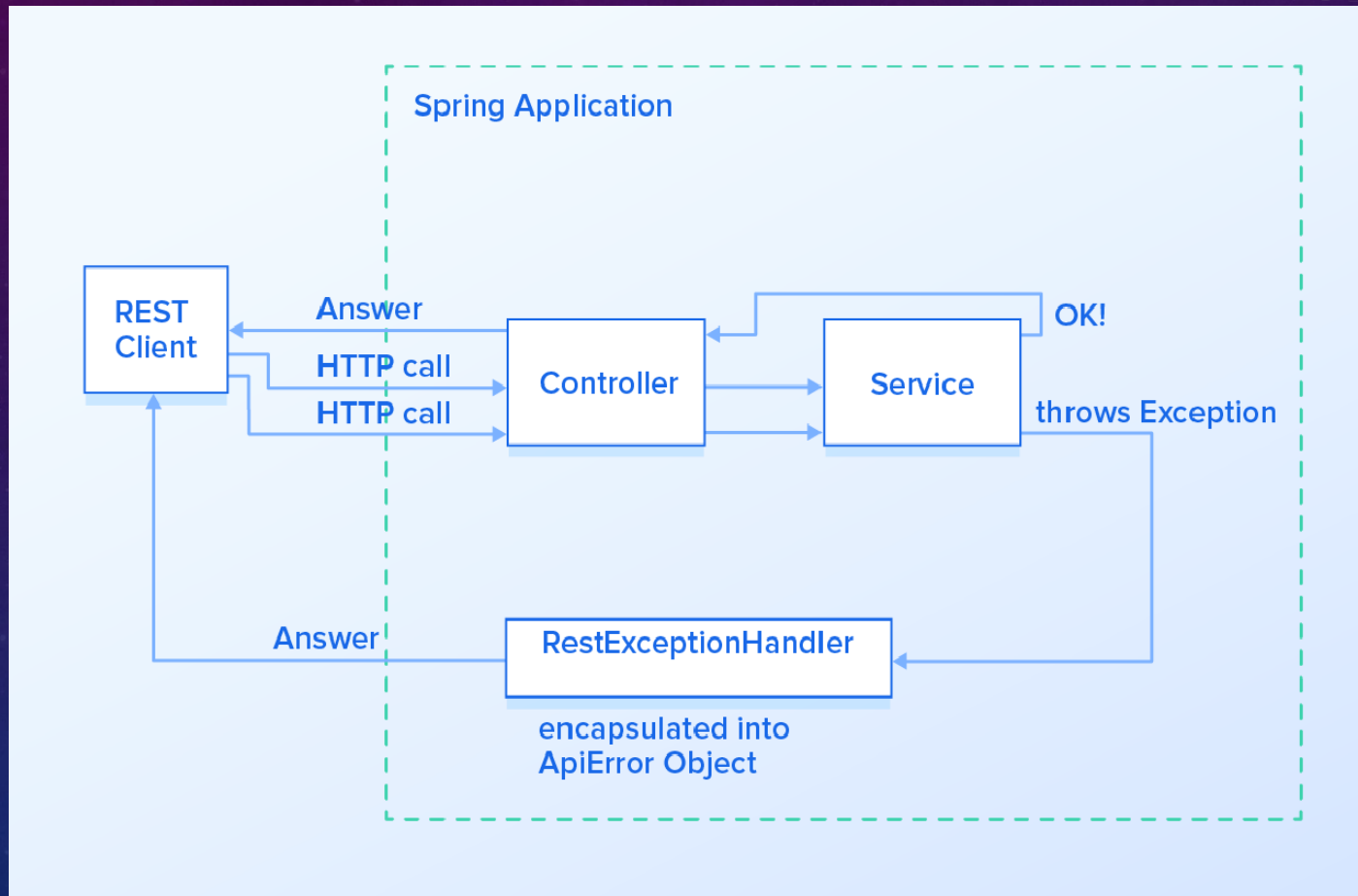
Мы можем поместить **@Valid** аннотацию в параметры и поля метода, чтобы сообщить Spring, что мы хотим, чтобы параметр или поле метода были проверены.

SPRING BOOT ERROR HANDLER

RestController— это базовая аннотация для классов, обрабатывающих операции REST.

ExceptionHandler— это аннотация Spring, предоставляющая механизм обработки исключений, возникающих во время выполнения обработчиков (операций контроллера). Эта аннотация, если она используется в методах классов контроллера, будет служить точкой входа для обработки исключений, создаваемых только внутри этого контроллера.

HANDLING EXCEPTIONS



JAVA PROFESSIONAL

SPRING BOOT: LOGGING AND AOP BASICS

WHAT IS AOP

АОП — аспектно-ориентированное программирование — это парадигма, направленная на повышение модульности различных частей приложения за счет разделения сквозных задач. Для этого к уже существующему коду добавляется дополнительное поведение, без изменений в изначальном коде.

Обычный объектно-ориентированный подход не всегда может эффективно решить те или иные задачи. В такой момент на помощь приходит АОП, дающий нам дополнительные инструменты для постройки приложения. Эти дополнительные инструменты — это увеличение гибкости при разработке, благодаря которой появляется больше вариантов решения той или иной задачи.

APPLICATION OF AOP

Аспектно-ориентированное программирование предназначено для решения сквозных задач, которые могут представлять собой любой код, многократно повторяющийся разными методами, который нельзя полностью структурировать в отдельный модуль. С его помощью мы можем оставить это за пределами основного кода и определить его по вертикали.

Пример: безопасность, транзакции, логирование и т.д

BASIC CONCEPTS OF AOP

Совет (advice) — это дополнительная логика, код, который вызывается из точки соединения.

Виды советов:

1. Перед (Before)
2. После (After)
3. После возврата (After Returning)
4. После бросания (After Throwing)
5. Вокруг (Around)

BASIC CONCEPTS OF AOP

Точка соединения (join point) — точка в выполняемой программе (вызов метода, создание объекта, обращение к переменной), где следует применить совет. Иначе говоря, это некоторое регулярное выражение, с помощью которого и находятся места для внедрения кода (места для применения советов).

Срез (pointcut) — набор **точек соединения**. Срез определяет, подходит ли данная точка соединения к данному совету.

Аспект (aspect) — модуль или класс, реализующий сквозную функциональность. Аспект изменяет поведение остального кода, применяя совет в точках соединения, определенных некоторым срезом. Иными словами, это комбинация советов и точек соединения.

Внедрение (introduction) — изменение структуры класса и/или изменение иерархии наследования для добавления функциональности аспекта в инородный код.

Цель (target) — объект, к которому будут применяться советы.

BASIC CONCEPTS OF AOP

Плетение (weaving) — это процесс связывания аспектов с другими объектами для создания рекомендуемых прокси-объектов. Это можно сделать во время компиляции, загрузки или во время выполнения.

Три вида плетения:

- **плетение во время компиляции** — если у вас есть исходный код аспекта и код, в котором вы используете аспекты, вы можете скомпилировать исходный код и аспект напрямую с помощью компилятора AspectJ;
- **посткомпиляционное плетение** (бинарное плетение) — если вы не можете или не хотите использовать преобразования исходного кода для вплетения аспектов в код, вы можете взять уже скомпилированные классы или jar-файлы и внедрить аспекты;
- **плетение во время загрузки** — это просто бинарное плетение, отложенное до момента, когда загрузчик классов загрузит файл класса и определит класс для JVM.
- Для поддержки этого требуется один или несколько «загрузчиков классов плетения». Они либо явно предоставляются средой выполнения, либо активируются с помощью «агента плетения».

ASPECTJ

AspectJ — зрелый AOP фреймворк с огромным комьюнити, множеством публикаций про него, хорошей документацией, достаточно стабильный, интегрированный в разнообразные системы сборки, интеграция в Spring, с хорошей поддержкой в IDE.

LOGGING

Логирование - это процесс записи каких-либо событий, которые происходят в коде.

Каждая запись лога содержит дату-время, уровень события, сообщение.

Есть несколько уровней записи. Основные — это info, debug, error.

- **INFO** — обычно это информационные сообщения. Например, записи в базу данных и так далее.
- **DEBUG** — более подробно описывает события конкретного момента.
- **ERROR** — ошибки. Например, когда мы оборачиваем что-то в try-catch, в блоке catch подставляется `e.printStackTrace()`. Он выводит запись только в консоль. С помощью логера можно мониторить эту запись.
- **WARN** — предупреждения.

JAVA PROFESSIONAL

SPRING BOOT: SCHEDULED JOBS

@SCHEDULED ANNOTATION

@Scheduled аннотация используется для планирования задач. Информация о триггере должна быть предоставлена вместе с этой аннотацией. Он принимает один атрибут из `cron`, `fixedDelay` или `fixedRate` для указания расписания выполнения в разных форматах.

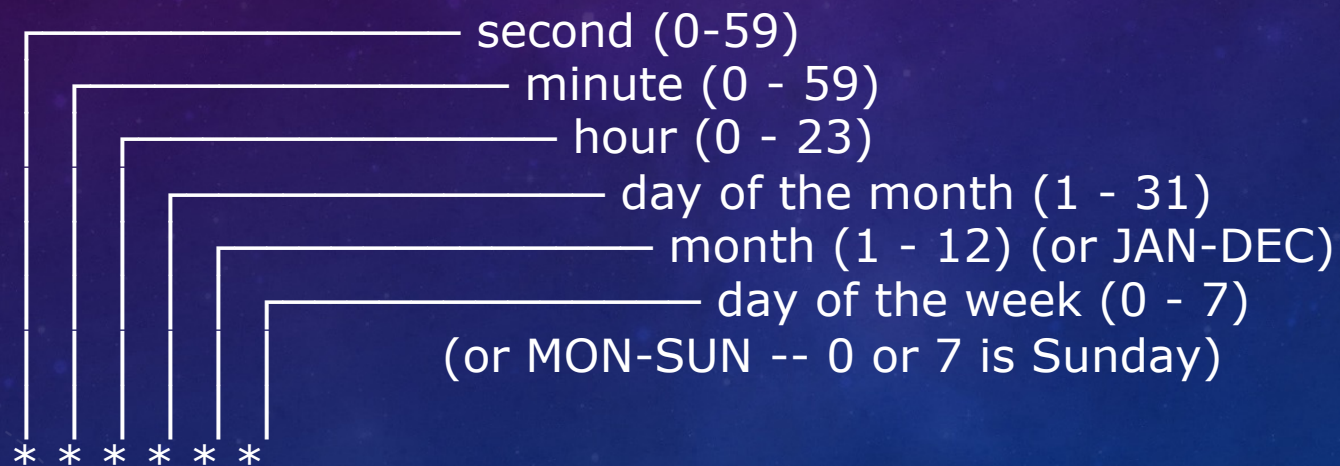
FIXED DELAY OR FIXED RATE

Мы используем `fixedDelay` атрибут, чтобы настроить выполнение задания после фиксированной задержки, он означает фиксированный интервал между концом предыдущего задания и началом нового задания. **Новое задание всегда будет ждать завершения предыдущего задания.** Его следует использовать в ситуациях, когда вызовы методов должны происходить последовательно.

Мы используем атрибут `fixedRate`, чтобы указать интервал для выполнения задания через фиксированный интервал времени. Его следует использовать в ситуациях, когда вызовы методов независимы. **Время выполнения метода не учитывается при решении, когда начинать следующее задание.**

CRON EXPRESSIONS

Выражение cron представляет собой строку из шести-семи полей, разделенных пробелом, для представления триггеров на секунду, минуту, час, день месяца, месяц, день недели и, необязательно, год. Однако выражение cron в Spring Scheduler состоит из шести полей, как показано ниже:



CRON EXPRESSIONS

Например, выражение cron: 0 15 10 * * * запускается в 10:15 каждый день (каждую 0-ю секунду, 15-ю минуту, 10-й час, каждый день). * указывает, что выражение cron соответствует всем значениям поля. Например, * в поле минут означает каждую минуту.

Такие выражения, как 0 0 * * * *, трудно читать. Чтобы улучшить читаемость, Spring поддерживает макросы для представления часто используемых последовательностей.

Пример:

@Scheduled(cron = "@hourly")

CRON EXPRESSIONS MACROS

Spring предоставляет следующие макросы:

@hourly,

@yearly,

@monthly,

@weekly, а также

@daily

RUNNING TASKS IN PARALLEL

По умолчанию Spring использует для запуска задач локальный однопоточный планировщик. В результате, даже если у нас есть несколько методов `@Scheduled`, каждый из них должен ждать, пока поток завершит выполнение предыдущей задачи.

Если наши задачи действительно независимы, их удобнее запускать параллельно. Для этого нам нужно предоставить `TaskScheduler`, который лучше соответствует нашим потребностям.

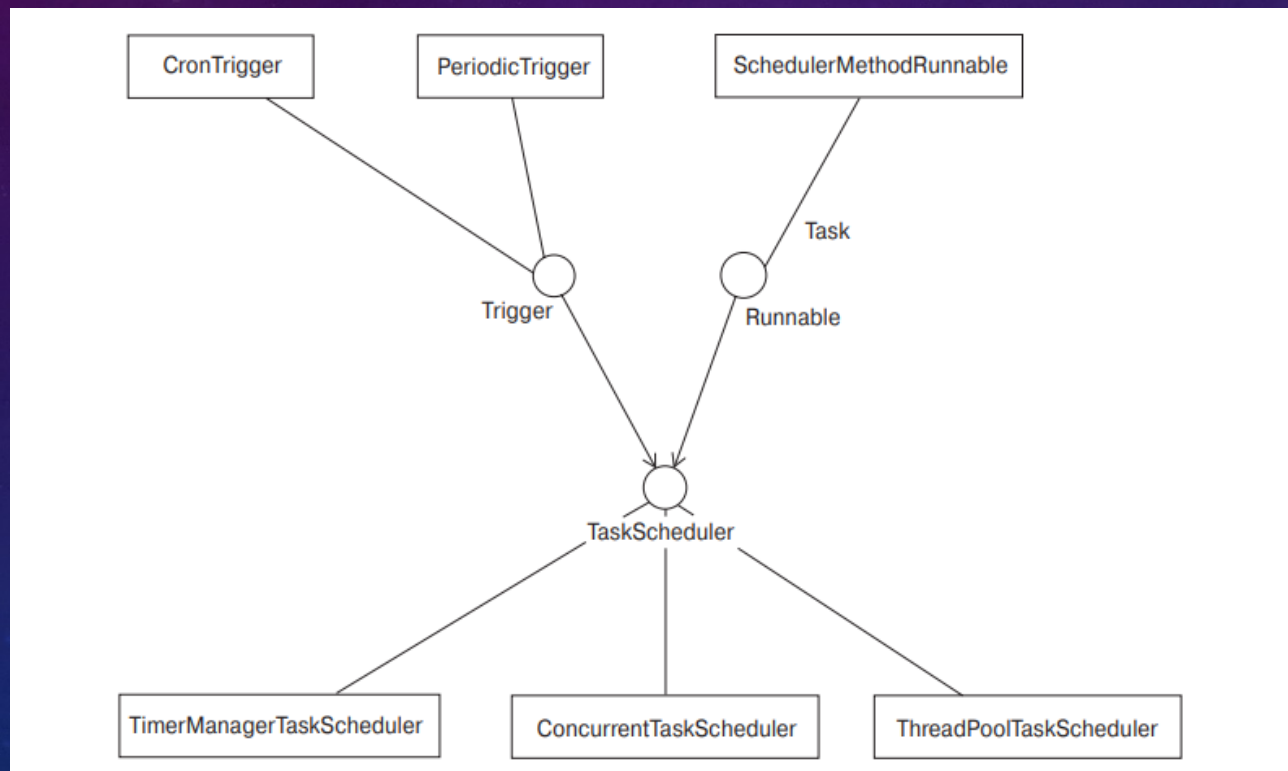
INTRODUCTION TO TASKSCHEDULER ABSTRACTION

В Spring-абстракции TaskScheduler имеются три главных участника:

- **Интерфейс Trigger.** В Spring доступны две реализации Trigger. Класс CronTrigger поддерживает запуск на основе выражения cron, а класс PeriodicTrigger — запуск на основе начальной задержки и затем фиксированного интервала.
- **Задача.** Задача — это порция бизнес-логики, запуск которой необходимо запланировать. В Spring задача может быть указана как метод внутри любого бина Spring.
- **Интерфейс TaskScheduler.** В Spring доступны три класса реализации интерфейса TaskScheduler:
 1. TimerManagerTaskScheduler
 2. ConcurrentTaskScheduler
 3. ThreadPoolTaskScheduler

INTRODUCTION TO TASKSCHEDULER ABSTRACTION

Планировать задачи с применением абстракции TaskScheduler из Spring можно двумя способами. Один из них предусматривает использование пространства имен `task` в XML-конфигурации Spring, а другой — аннотаций.



JAVA PROFESSIONAL

SPRING BOOT: SECURITY

INTRODUCTION

Spring Security — среда для аутентификации и авторизации пользователей. Фреймворк применяется для защиты приложений на Spring. В нем представлены базовые инструменты безопасности, которые без труда расширяются для решения разных задач.

Самым фундаментальным объектом является **SecurityContextHolder**. В нем хранится информация о текущем контексте безопасности приложения, который включает в себя подробную информацию о пользователе, работающим с приложением.

AUTHENTICATION

Аутентифика́ция — процедура проверки подлинности, например: проверка подлинности пользователя путём сравнения введённого им пароля с паролем, сохранённым в базе данных пользовательских логинов.

AUTHORIZATION

Авторизация - это предоставление определённому лицу или группе лиц прав на выполнение определённых действий, а также процесс проверки данных прав при попытке выполнения этих действий.

В более простых приложениях аутентификации может быть достаточно: как только пользователь проходит аутентификацию, он может получить доступ ко всем частям приложения.

Но у большинства приложений есть концепция разрешений (или ролей). Представьте: клиенты, имеющие доступ к общедоступному интерфейсу вашего интернет-магазина, и администраторы, имеющие доступ к отдельной области администрирования.

Оба типа пользователей должны войти в систему, но сам факт аутентификации ничего не говорит о том, что им разрешено делать в вашей системе. Следовательно, вам также необходимо проверить разрешения аутентифицированного пользователя, т.е. вам необходимо авторизовать пользователя.

OAuth2

OAuth2 — это протокол авторизации, который позволяет предоставить третьей стороне ограниченный доступ к защищенным ресурсам пользователя без необходимости передавать ей (третьей стороне) логин и пароль.

OAuth2 определяет 4 роли:

- Владелец ресурса
- Ресурсный сервер
- Сервер авторизации
- Клиент (приложение)

EXPECTED PROTOCOL FLOW

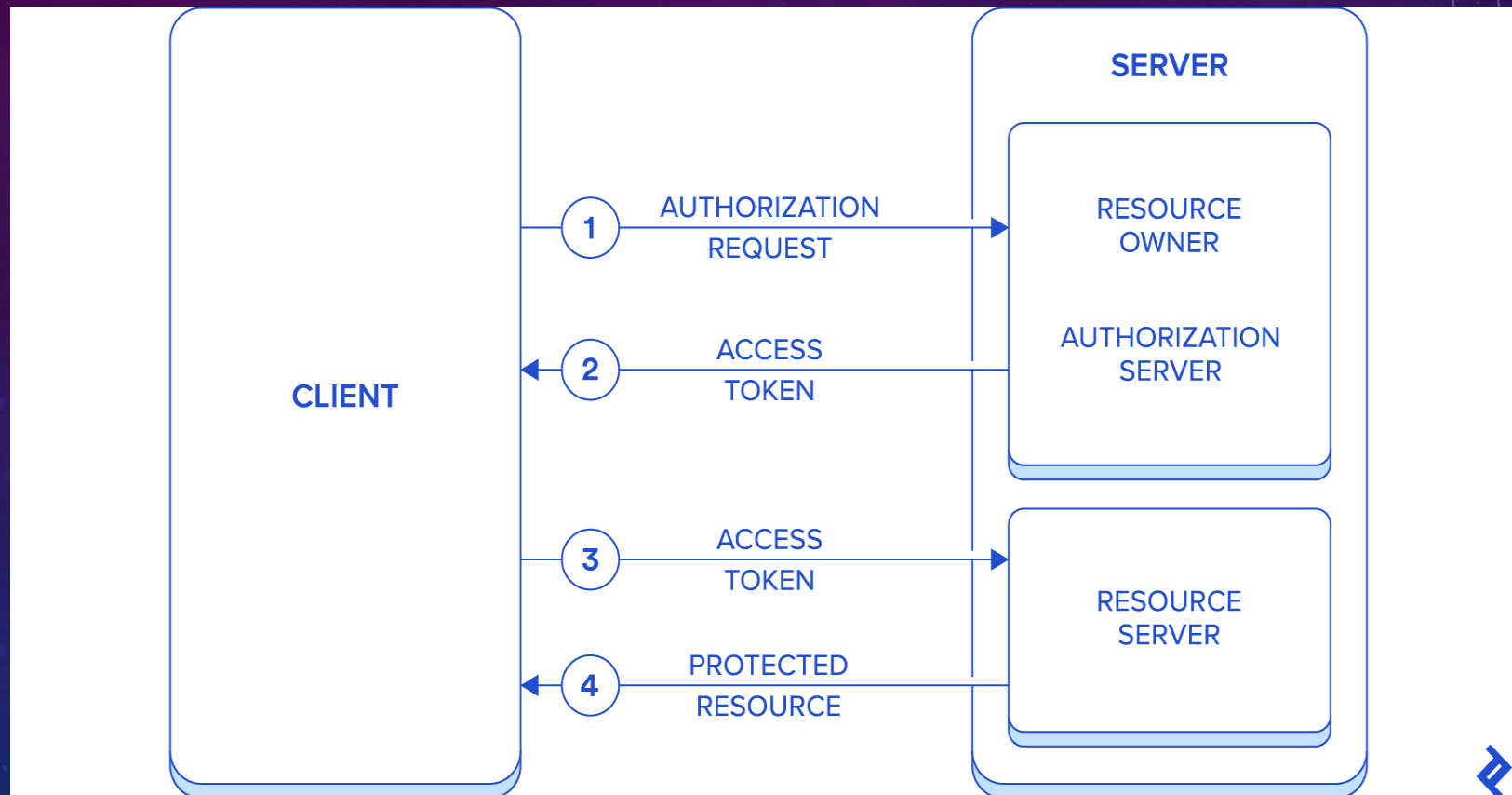
Хотя одной из основных особенностей OAuth2 является введение уровня авторизации, чтобы отделить процесс авторизации от владельцев ресурсов, для простоты результатом статьи является сборка единого приложения, олицетворяющего всех **владельцев ресурсов**, **сервер авторизации** и **роли сервера ресурсов**.

EXPECTED PROTOCOL FLOW

Упрощенный поток:

1. Запрос авторизации отправляется от клиента к серверу (действующему как владелец ресурса) с использованием предоставления авторизации пароля.
2. Токен доступа возвращается клиенту (вместе с токеном обновления)
3. Затем токен доступа отправляется от клиента к серверу (действующему в качестве сервера ресурсов) при каждом запросе на доступ к защищенному ресурсу.
4. Сервер отвечает требуемыми защищенными ресурсами

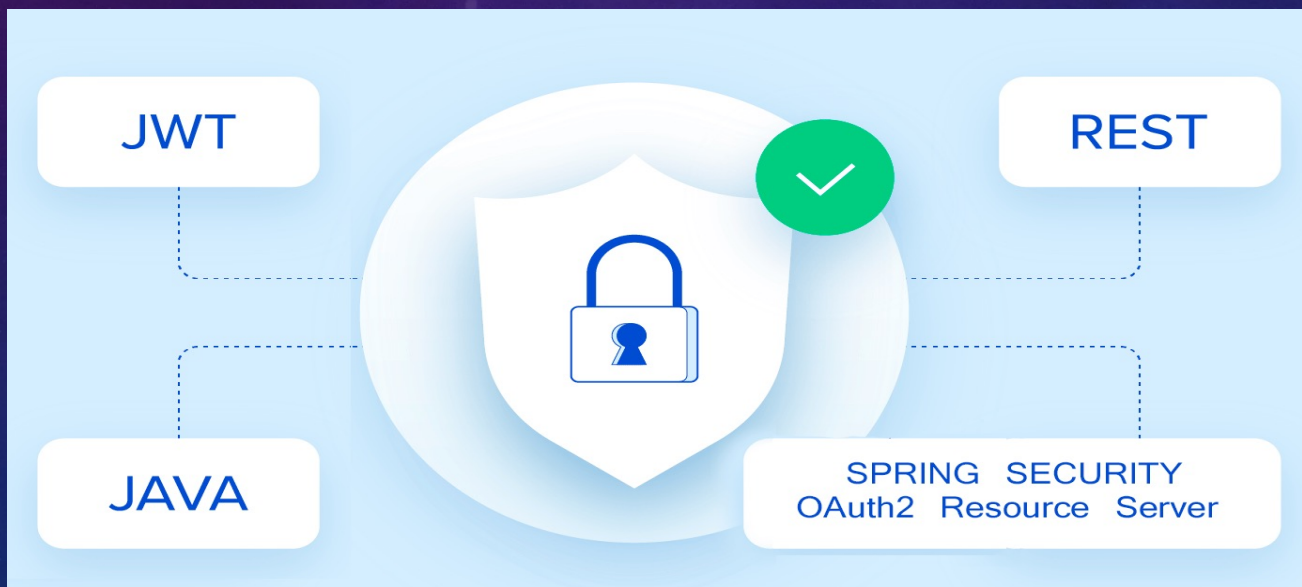
EXPECTED PROTOCOL FLOW



JWT TOKEN

JWT(JSON Web Token)

Токен – это просто строка, которая генерируется по запросу пользователя, который хочет в дальнейшем вызывать защищенные ресурсы. Пользователь регистрируется в системе, потом делает запрос на генерацию токена. Это похоже на пропуск входа для клиента, который сервер авторизации проверяет перед предоставлением доступа к защищенным ресурсам, таким как API или конечные точки HTTP.



JAVA PROFESSIONAL

SPRING BOOT: TRANSACTIONS

WHAT IS TRANSACTION?

Транзакцией называется набор связанных операций, все из которых должны быть выполнены корректно без ошибок. Если при выполнении одной из операций возникла ошибка, все остальные должны быть отменены. Такой механизм распространен при работе с БД.

Транзакция позволит нам объединить определенные операции таким образом, что по итогу мы либо запишем все изменения, либо ничего. То есть объединить несколько различных действий в атомарную операцию. Как пример - банковские операции.

EXAMPLE

Представим, что мы создали банковское приложение. Клиент Андрей переводит своему другу Михаилу 50 евро. Для выполнения этой транзакции нам потребуется три действия:

1. Списать деньги с баланса Андрея;
2. Записать операцию перевода от Андрея к Михаилу;
3. Добавить денег на баланс Михаилу;

Допустим, мы успешно списали деньги с баланса Андрея, но потом произошла ошибка. В ходе транзакции у Андрея списались деньги, но и до Михаила деньги не дошли. Как раз такие проблемы решают транзакции.

ACID TRANSACTION PROPERTIES

Транзакции обладают свойствами, которые называют ACID:

- **Атомарность (Atomicity)** гарантирует, что никакая транзакция не будет зафиксирована в системе частично. Будут либо выполнены все операции, либо ни одной.
- **Согласованность (Consistency)**. Выполненная транзакция, сохраняет согласованность базы данных.
- **Изолированность (Isolation)**. Во время выполнения транзакции параллельные транзакции не должны оказывать влияние на её результат.
- **Устойчивость (Durability)**. Независимо от проблем с оборудованием, изменения, сделанные успешно завершённой транзакцией, должны остаться сохранёнными после возвращения системы в работу.

TYPES OF TRANSACTION MANAGEMENT IN SPRING

Существует 2 вида управления транзакциями:

- **Программный**

Этот способ более сложный для чтения и поддержки, чем декларативный, но даёт большую гибкость.

- **Декларативный**

В случае декларативного управления транзакциями мы отделяем бизнес-логику от управления транзакциями. Мы используем либо аннотации, либо XML-файл для конфигурации управления транзакциями.

JAVA PROFESSIONAL

SPRING BOOT: DEPLOY. HEROKU.

SERVICE-ORIENTED ARCHITECTURES

Сервис-ориентированные архитектуры являются сущностью облачных вычислений и могут быть разделены по своему назначению.

Наиболее распространенными категориями являются «инфраструктура как услуга» (**IaaS**), «платформа как услуга» (**PaaS**) и «программное обеспечение как услуга» (**SaaS**).

IAAS, PAAS, SAAS

- 1. IaaS (Infrastructure as a Service)** – инфраструктура как услуга. Т.е. аренда вычислительных ресурсов (процессор CPU, оперативная память RAM, пространство на диске HDD). Из этих ресурсов собирают «виртуальный сервер», на который можно установить любое необходимое ПО. Это базовый, основной сервис в системе облачных услуг.
- 2. PaaS (Platform as a Service)** – платформа как услуга. Это уже готовая платформа с определенными настройками под разные задачи. Т.е. это IaaS + настроенное ПО. Здесь провайдер берет на себя всю настройку и конфигурацию оборудования, устанавливает необходимые программы, а вы загружаете туда свои данные и можете приступить к работе. Например, нам не нужно строить базу данных с нуля, это предлагается как уже готовый сервис (платформа), который предоставляет провайдер. Мы просто загружаем туда данные и можете начать работать.

IAAS, PAAS, SAAS

3.SaaS (Software as a Service) – программное обеспечение как сервис. Это полностью готовое решение, которое сразу же можно использовать. Примерами реализации услуг по принципу SaaS являются конструкторы сайтов, почтовые сервисы, различные CRM-системы, 1С в облаке, планировщик Google и т.д.

При реализации услуги SaaS провайдер берет на себя все обязательства по настройке сервисов, клиент не участвует в данных процессах, а лишь пользуется готовым продуктом с имеющимися настройками. Провайдер услуги осуществляет лицензирование, апгрейд, техподдержку и прочие настройки.

WHAT IS HEROKU?

Heroku – облачная мультязычная платформа как услуга (PaaS), которая позволяет разработчикам максимально быстро развертывать, масштабировать и управлять приложениями.

Эта платформа предлагает поддержку широкого спектра языков программирования, таких как Java, Ruby, PHP, Node.js, Python, Scala и Clojure. Heroku запускает приложения через виртуальные контейнеры, больше известные как Dynos.

WHAT IS THE HEROKU PLATFORM FOR?

Heroku нужна:

- для размещения приложений и веб-сервисов;
- упрощения и ускорения цикла разработки;
- снижения потребности в сложной работе с сервером;
- работы с нагруженными приложениями;
- быстрого масштабирования проектов.