



Spring Framework

Author: Pavel Šeda



Audience

- Beginners to Spring framework, who have logical, and analytical problem-solving skills and who want to begin with learning Spring framework and its modules
- Students who want to prepare for:
 - Spring Professional Certification



Course Objectives

1. Foundational Spring and Jakarta/Java EE
 - a. Review IDEs (Netbeans, Eclipse/Spring Tool Suite, IntelliJ IDEA)
 - b. Review of application servers
 - c. Spring vs Jakarta/Java EE
 - d. IoC Container
 - e. Metadata configurations (Java Config, XML Config, Annotations)
 - f. Spring bean lifecycle
 - g. Spring bean scopes
 - h. Proxy objects
 - i. Reviewing of monolithic, n-tier, and microservices architectures



Course Objectives

2. Spring and Spring Boot
 - a. Spring modules
 - b. Spring
 - c. Spring Boot
 - d. Initializing a Spring and Spring Boot application
 - e. Spring Boot bundles
 - f. Spring Boot autoconfiguration



Course Objectives

3. Working with configuration properties
 - a. Spring environment abstraction
 - b. Fine-tuning autoconfiguration
 - i. Configuration of data source
 - ii. Property file vs Yaml
 - c. External vs Internal config file
 - d. Configuration with Spring profiles



Course Objectives

4. Database Access
 - a. Java Database Connectivity (JDBC)
 - b. Spring JDBC Template
 - c. JPA basics:
 - i. Entities, Entity relationships,
 - ii. Common annotations (@Entity, @Table, @Column, @Id, etc.)
 - iii. Fetch types (Lazy vs Eager)
 - iv. EntityManager
 - d. Spring Data
 - i. Paging and sorting
 - e. Transactions



Course Objectives

- 5. Developing Service Layer
 - a. Reviewing problem statement
 - b. Consuming another REST service (synchronously vs asynchronously)
 - i. RestTemplate
 - ii. WebResource
 - iii. Declarative REST Feign Client
 - c. Consuming another SOAP service
 - i. Implementing SOAP Clients (wsimport)
 - ii. Dispatcher Client



Course Objectives

6. Spring Model View Controller (MVC)
 - a. Servlets
 - i. Dispatcher Servlet
 - b. Java Server Pages (JSP), Java Server Faces (JSF)
 - c. MVC Design Pattern
 - d. Common annotations:
 - i. @Controller, @ModelAttribute, @RequestMapping, @SessionAttributes
 - e. Validation (javax.validation package)
 - f. Embedded application servers (Tomcat, Jetty)



Course Objectives

7. Spring REST Services

- a. Rest architecture
- b. Common annotations:
 - i. `@RestController`, `@PathVariable`, `@RequestParam`, `@RequestBody`
- c. Data Transfer Object (DTO) classes
- d. Jackson
- e. Exception handling (`@RestControllerAdvice`)
- f. Swagger Documentation
- g. Testing Tools
 - i. Postman, SoapUI, Browser plugins



Course Objectives

8. Aspect Oriented Programming (AOP)
 - a. The reason for AOP
 - b. Joinpoint
 - c. Pointcut
 - d. Advice
 - e. Spring AOP vs AspectJ



Course Objectives

- 9. Spring Schedulers, Async Tasks, Caching
 - a. Scheduling using cron
 - b. Enabling async tasks and common mistakes
 - c. Caching
 - i. Caching problematics
 - ii. Cache evict
 - iii. Special databases for caching



Course Objectives

10. Spring Tests

- a. Database for testing
- b. Unit tests
- c. Mockito
- d. Integration Tests



Course Objectives

11. Spring Security

- a. Configuration of Spring Security
- b. Securing web requests
- c. Storing password in Database
- d. OAuth2 + OpenID Connect



Course Objectives

12. Spring Services Deployment

- a. Eureka
- b. Spring Boot Actuator
- c. Spring Boot Admin Server
- d. Deploying Jar vs War files
- e. ELK stack for logging



Introductions

Please briefly introduce yourself:

- Name
- Title or position
- Company
- Experience with Java programming and Java applications
- Reasons for attending



Section: Foundational Spring and Jakarta EE



Foundational Spring and Jakarta EE

- Review IDEs (Netbeans, Eclipse/Spring Tool Suite, IntelliJ IDEA)
- Review of application servers
- Spring vs Jakarta/Java EE
- IoC Container
- Proxy objects
- Extensible Markup Language (XML) vs Java Config
- Spring vs Spring Boot
- Reviewing of monolithic, n-tier, and microservices architectures
- Initializing a Spring application



Why Use IDE in Java?

- Code completion
- Automatic code generation
- Syntax checking
- Great support for refactoring features
- Useful plugins (SonarLint, Database tools, ...)
- Integrated debugging
- Navigating to members by treating them as hyperlinks
- Warning as you type (i.e. some errors do not even require a compile cycle)
- Possibility to be lazy



Integrated Development Environments (IDE)

- Netbeans
 - Created in 1996
 - Created by students project called Xelfi
 - Created in Czech Republic by students from Faculty of Mathematics and Physics at Charles University, later led by Roman Staněk
 - In 1999 acquired by Sun Microsystems (700 millions of dollars)
 - In 2010 Sun Microsystems was acquired by Oracle => Netbeans is Oracle IDE now
 - Advantages:
 - (+) It is free in full version
 - (+) It integrates latest trends in Java language, e.g., the modularization
 - Disadvantages
 - (-) It is slow
 - (-) It does not contain much plugins compared to other IDEs



Integrated Development Environments (IDE)

- Eclipse
 - Created in 1998
 - Created by IBM company to “eclipse” Netbeans
 - Open Source Release in 2001
 - Fear from industry that it is too much controlled by IBM
 - In 2004 moved to newly created Eclipse Foundation to lead and develop the Eclipse community
 - Consortium of software vendors (IBM/Red Hat, Oracle, Fujitsu, SAP SE, Microsoft, ...)
 - Advantages:
 - (+) It is free in full version
 - (+) A lot of plugins included, could be used by many languages, work spaces, sweet UI
 - (+) Commonly used to develop company IDE (Spring Tool Suite, JBoss Developer Studio)
 - Disadvantages
 - (-) Sometimes plugins hell, does not feel context as IntelliJ IDEA



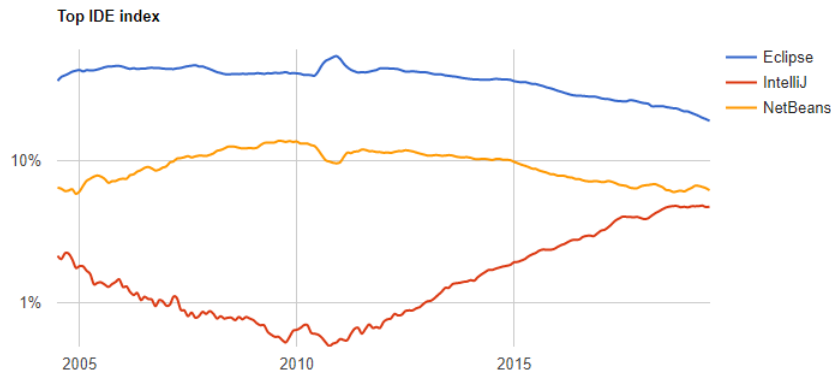
Integrated Development Environments (IDE)

- IntelliJ IDEA
 - Created in 2001
 - Russian company, headquarters is in St. Petersburg but large development center in Prague
 - Community vs Ultimate edition (Community is free, Ultimate is paid)
 - Ultimate edition price (2019): 12 290 Kč = ~482 € per year
 - In 2010 Sun Microsystems was acquired by Oracle => Netbeans is Oracle IDE now
 - Advantages:
 - (+) Feels context
 - (+) String checking for, e.g., JPQL queries, the best refactoring, fast
 - Disadvantages
 - (-) It is not free in full version
 - (-) It can be challenging for computers with RAM less than 2GB (old notebooks etc.)

Integrated Development Environments (IDE)

- Final thoughts:
 - <https://pypl.github.io/IDE.html>
- In this course we will use:
 - IntelliJ IDEA Ultimate

Worldwide, Visual Studio is the most popular IDE, Android Studio grew the most in the last 5 years (18.4%) and Eclipse lost the most (-18.2%)





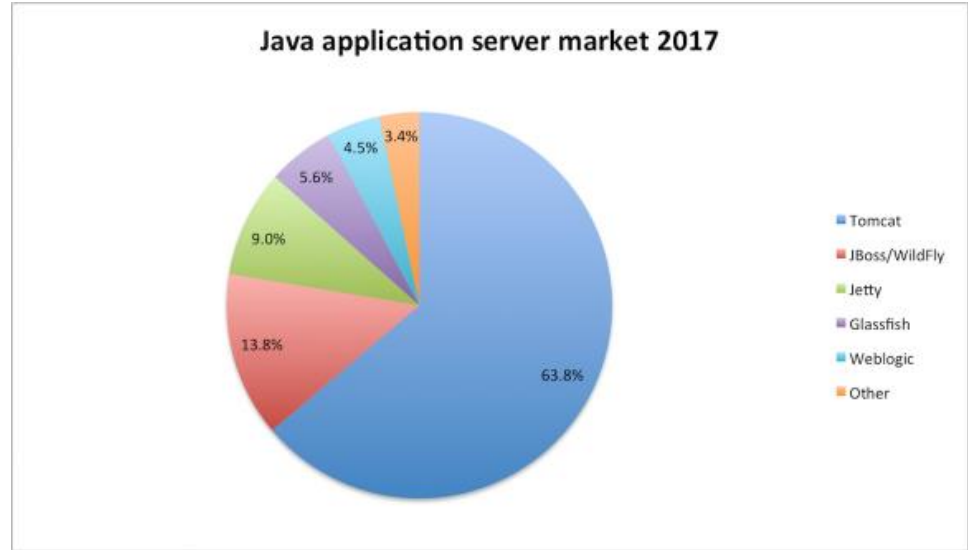
Review of Application Servers

- What is application server?
 - Software framework that consists web server connectors, computer programming languages, runtime libraries, database connectors, and the administration code needed to deploy, configure, manage, and connect these components on a web host
- Web Servers:
 - Tomcat
 - Jetty
- Application Servers:
 - Glassfish (reference Java EE implementation)
 - Oracle Weblogic
 - IBM WebSphere
 - Wildfly
 - Apache Geronimo

Web Servers	Application Servers
Servlets, JSP, Web Socket	Servlets, JSP, Web Socket
Expression Language, JTA	Expression Language, JTA
	Batch, CDI, EJB, JPA, JMS
	JAX-RS, JAX-WS, JavaMail
	JSON-P, Bean Validation

Review of Application Servers - Popularity

- Currently, Tomcat, WildFly and Jetty are the most popular (in 2017)
- The reason is probably the following:
 - They are easy to configure and manage
 - Tomcat and Jetty are well suited with Spring framework
- Web servers are commonly used these days as embedded servers inside .jar file





Review of Application Servers - Selection

- How to choose web/application server?
 - With Spring basically the most common is to use Tomcat (default option as embedded server inside .jar)
 - With Jakarta/Java EE applications the most common is to use WildFly because it is free and quite easy to configure
- What aspects are important:
 - Number of concurrent users the system should support
 - IDE tooling and GUI support
 - Security and authentication
 - Technical support and documentation
 - How they implement Jakarta/Java EE specification
 - E.g., Glassfish implements JPA with EclipseLink and WildFly implements JPA with Hibernate framework
 - What extensions from Jakarta/Java EE specifications they are supporting, e.g. Weblogic supports MakeConnection for JAX-WS services



Review of Application Servers

- Embedded vs external java web servers
 - Application with embedded server looks like a regular java program. You just launch jar file and that's it.
 - Regular web application is usually a war archive which needs to be deployed to some server
 - With embedded server your application is packaged with the server of choice and responsible for server start-up and management
-
- | | |
|---|---|
| <ul style="list-style-type: none">● Embedded pros:<ul style="list-style-type: none">○ Single object to be deployed○ You can easily test against server versions just like any other dependency● Embedded cons:<ul style="list-style-type: none">○ Dependency bloat, as you have to include all the dependencies of the web server | <ul style="list-style-type: none">● External servers pros:<ul style="list-style-type: none">○ Potentially more flexible application architecture● External servers cons:<ul style="list-style-type: none">○ Deployment complexity○ More difficult development |
|---|---|

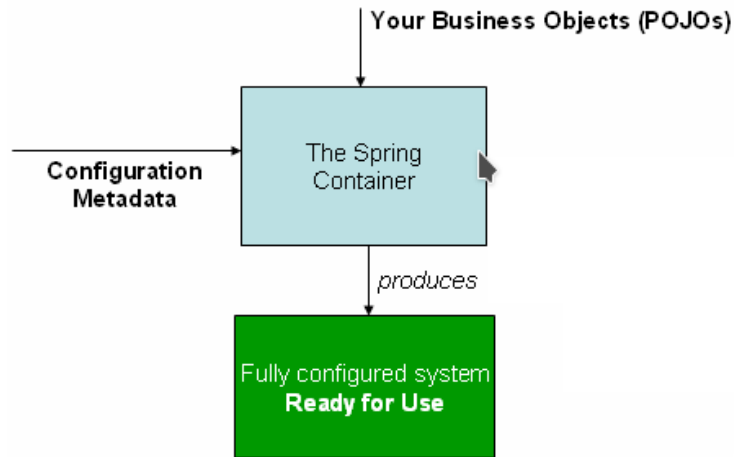


Spring Framework vs Jakarta/Java EE

- Jakarta/Java EE
 - Is a specification for creating enterprise level Java applications
 - Basically includes a set of interfaces which are implemented by application server incorporates JDBC for databases, JNDI for registries, JTA for exchanges, JMS for informing, etc.
 - In 2018 moved from Oracle to Eclipse Foundation (Java EE renamed to Jakarta EE)
- Spring framework
 - Was created in October 2002 by Rod Johnson who wrote his book 'Expert One-on-One J2EE Design and Development'
 - It is complementary to Jakarta EE, it integrates with carefully selected individual specifications from the EE umbrella: Servlet API, WebSocket API, Concurrency Utilities, JSON Binding API, Bean Validation, JPA, JMS, JTA
 - Contains a large set of modules: <https://spring.io/projects>
- In personal opinion, Spring (Boot) is nowadays the winner for cloud-based microservices
 - It is also easier to develop particular things as Security
 - Also, the middleware (web/application server) management is usually easier

Inversion of Control

- Spring container uses the design pattern of Inversion of Control (IoC) also known as dependency injection (DI)
- It is a process whereby objects define their dependencies, that is, the other objects they work with, only through constructor arguments, arguments to a factory method, or properties that are set on the object instance after it is constructed or returned from a factory method
- The container then injects those dependencies when it creates the bean. This process is fundamentally the inverse, hence the name Inversion of Control (IoC), of the bean itself controlling the instantiation or location of its dependencies by using direct construction of classes, or a mechanism such as the Service Locator pattern.





Metadata Configurations

- There exist three ways of metadata configurations:
 1. XML
 2. Annotations
 3. Code driven (Java Config)



Metadata Configurations - XML

- Configured by XML
- Different modules (different namespaces)
- Benefits
 - Pure declarative approach
 - Allows to change system configuration without changing code (very useful for customizations)
- Weaknesses
 - Not transparent
 - Lots of configuration is needed
 - Hard to maintain
 - Problem with refactoring



Metadata Configurations - XML (Examples)

- The example of defining Spring bean (managed class by Spring container) using XML config

```
<bean id="personService" class ="com.sedaq.service.PersonService"/>
```

- The id attribute is a string that you use to identify the individual bean definition
 - Must be unique
- The class attribute defines the type of the bean and uses the fully qualified class name
- Bean naming convention is standard Java convention for instance field names

```
<bean id="petStore"  
    class="org.springframework.samples.jpetsy.store.services.PetStoreServiceImpl">  
    <property name="accountDao" ref="accountDao"/>  
    <property name="itemDao" ref="itemDao"/>  
    <!-- additional collaborators and configuration for this bean go here -->  
</bean>
```



Metadata Configurations - XML (Examples)

- Composing XML-based configuration metadata

```
<beans>
  <import resource="services.xml"/>
  <import resource="resources/messageSource.xml"/>
  <import resource="/resources/themeSource.xml"/>
  <bean id="bean1" class="..." />
  <bean id="bean2" class="..." />
</beans>
```

- In the preceding example, external bean definitions are loaded from three files:
 - services.xml, messageSource.xml, themeSource.xml



Metadata Configurations - Annotations

- Configured by annotations
- Benefits:
 - Less configuration needed
 - More clear and transparent
 - No problem with refactoring
- Weaknesses:
 - Less flexibility
- Can be combined with xml configuration: (<context:annotation-config/>, <context:component-scan />)

```
@Configuration
public class RepositoryConfig {
    private @Autowired DataSource dataSource;
    public @Bean AccountRepository accountRepository() {
        return new JdbcAccountRepository(dataSource);
    }
}
```



Metadata Configurations - Annotations

- Proprietary
 - @Component, @Service, @Repository
 - @Autowired, @Required
- JSR-330:
 - @Named
 - @Inject, @Qualifier
- Other:
 - @Resource, @PersistenceContext, @PersistenceUnit

```
@Component  
public class MyClass{  
  
}
```



Metadata Configurations - Code Driven

- Benefits
 - Almost any code could be evaluated during initialization
 - No problem with refactoring
 - Almost everything is configured in Java classes
- Weaknesses:
 - Pure imperative approach
 - Configuration is hardcoded into the class
- @Configuration, @Bean, @ComponentScan, @PropertySource, @Import
- Each @Bean method is evaluated just once!

```
@Configuration
public class RepositoryConfig {
    private @Autowired DataSource dataSource;
    public @Bean AccountRepository accountRepository() {
        return new JdbcAccountRepository(dataSource);
    }
}
```



Move to Code Driven With Legacy XML Config

- It is possible to use `@ImportResource` to import XML configuration we do not know how to do in Java Config and other things just configure in Java classes

```
@Configuration
@ImportResource("classpath:applicationContext.xml")
@ComponentScan("com.sedaq")
public class Application{

    public static void main(String[] args){
        AnnotationConfigApplicationContext applicationContext =
            new AnnotationConfigApplicationContext(Application.class);
    }
}
```



Metadata Configurations - How to Select?

- XML configuration has no restrictions but is more difficult to create and maintain (nowadays, not usual)
- In pure Spring annotations and XML configuration was the most usual one
- In Spring Boot annotations + code driven are the most usual one
- The trend is to move to Spring Boot, so the annotations + code driven are nowadays the preferred one
 - The reason is that Java developers are more familiar with Java than with difficult XML configurations



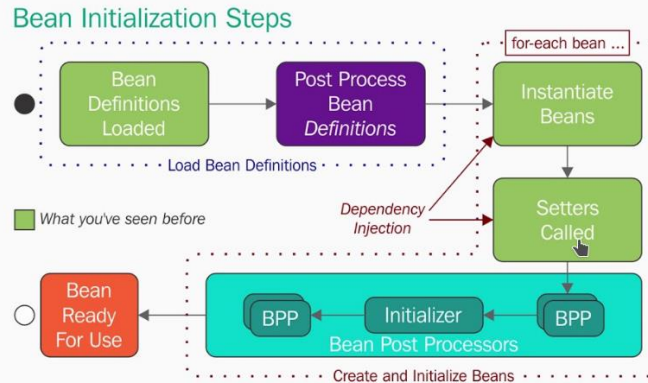
Constructor, Setter or Property Based Injection

- Constructor injection is the best practice:
 - In object-oriented programming and object must be in a valid state after construction and every method invocation changes the state to another valid state
- Overriding: Setter injection overrides the constructor injection. If we use both constructor and setter injection, IoC container will use the setter injection
- Prefer constructor injection to make code cleaner

Spring Bean Lifecycle

- If the Spring context is created all the declared beans are initialized (it is called eager loading)
- If the Spring context is shut down all the beans are destroyed from memory

Bean Lifecycle



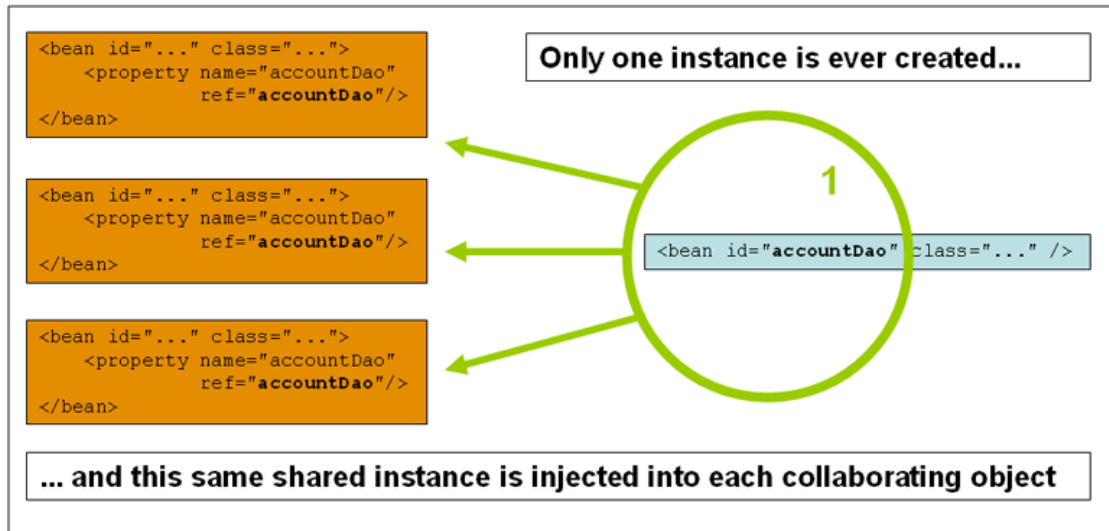
Spring Bean Scopes



Scope	Description
singleton	(Default) Scopes a single bean definition to a single object instance per Spring IoC container.
prototype	Scopes a single bean definition to any number of object instances.
request	Scopes a single bean definition to the lifecycle of a single HTTP request; that is, each HTTP request has its own instance of a bean created off the back of a single bean definition. Only valid in the context of a web-aware Spring ApplicationContext.
session	Scopes a single bean definition to the lifecycle of an HTTP Session. Only valid in the context of a web-aware Spring ApplicationContext.
application	Scopes a single bean definition to the lifecycle of a ServletContext. Only valid in the context of a web-aware Spring ApplicationContext.

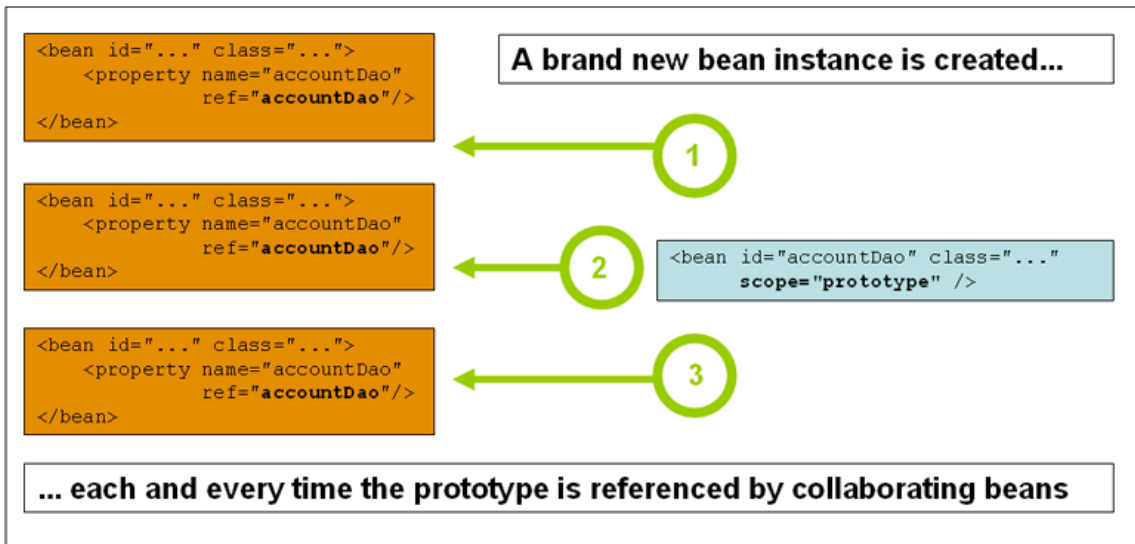
Spring Bean Scopes - Singleton

- Only one shared instance of a singleton bean is managed, and all requests for beans with an id or ids matching that bean definition result in that one specific bean instance being returned by the Spring container
- Spring singleton is best described as per container and per bean



Spring Bean Scopes - Prototype

- The non-singleton, prototype scope of bean deployment results in the *creation of a new bean instance* every time a request for that specific bean is made
- As a rule, use the *prototype* scope for all *stateful* beans and the *singleton* scope for *stateless* beans



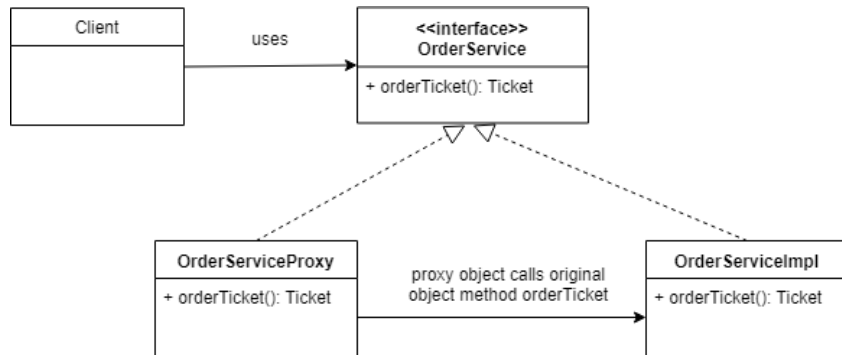


Spring Bean Scopes - Real World Scenarios

- **Singleton:** Connection to a database
- **Prototype:** Declare configured form elements (a textbox configured to validate names, e-mail addresses for example) and get “living” instances of them for every form being created
- **Request:** information that should only be valid on one page like the result of a search or the confirmation of an order. The bean will be valid until the page is reloaded
- **Session:** To hold authentication information getting invalidated when the session is closed (by timeout or logout). You can store other user information that you do not want to reload with every request here as well

Proxy Objects

- Proxy design pattern falls under the structural design pattern category and it is one of the most frequently used pattern in software development
- This pattern helps to control the usage and access behaviour of connected resources
- The proxy is the object that is being called by the client to access the real object behind the scene
- Gang of Four (GoF) defines it as:
 - “Provide a surrogate or placeholder for another object to control access to it”



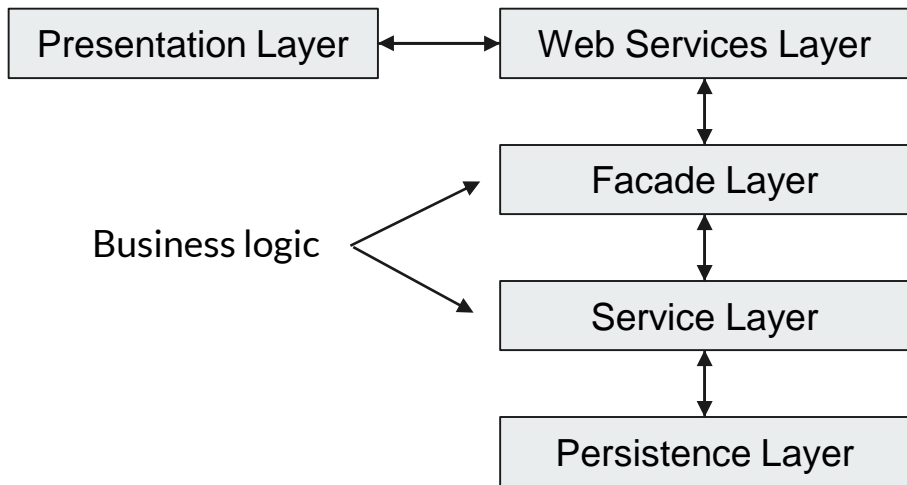


Reviewing of Architecture Styles (Monolithic, ...)

- Monolithic means that all the code is composed in one piece
- Monolithic software is designed to be self-contained; components of the program are interconnected and interdependent rather than loosely coupled
- Benefits:
 - Easier to design
 - For smaller applications it is preferred way
 - It usually has better throughput than modular approaches, such as the microservice architecture
 - Easier transaction management (distributed transactions are quite a hell)
- Cons
 - If any program component must be updated, the whole application has to be rewritten
 - Difficult for continuous delivery and deployment
 - Compiling and building large projects takes a lot of time
 - Developers are not able to create their own applications, they are just learning the project as it is

Reviewing of Architecture Styles (n-tier, ...)

- N-tier applications are usually divided into a few layers, these includes:



- Benefits:
 - Easy to manage: You can manage each tier separately, adding or modifying each tier without affecting the other tiers
 - More efficient development: You can split developer teams, easier to find particular logic in the code
- Disadvantage:
 - The performance could be slower



Reviewing of Architecture Styles (Microservices, ...)

- Used by Netflix, Amazon, eBay
- Microservice architecture is basically a set of loosely coupled, collaborating services
- Benefits:
 - Each service is highly maintainable and testable - enable rapid and frequent development and deployment
 - Loosely coupled with other services - enables a team to work independently the majority of time on their service(s) without being impacted by changes to other services and without affecting other services
 - Independently deployable - enables a team to deploy their service without having to coordinate with other teams
 - Capable of being developed by a small team - essential for high productivity by avoiding the high communication head of large teams
- Drawbacks:
 - Additional complexity with creating a distributed systems (dealing with partial failures, distributed transactions)
 - Testing the interactions between services is more difficult
 - Developers need to learn a huge set of difficult principles
 - Network bandwidth is higher (microservices could be deployed in other servers; multiple



Section: Spring and Spring Boot



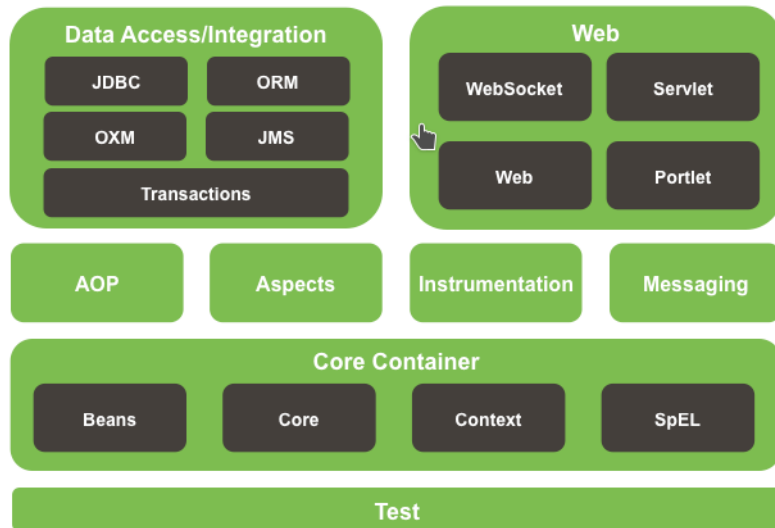
Spring and Spring Boot

- Spring modules
- Spring
- Spring Boot autoconfiguration
- Spring Boot parent project
- Initializing a Spring application

Spring Modules



Spring Framework Runtime





Spring Modules

- Spring modules could be separated into several parts:
 - **Core Container** - contains the most important parts as IoC container
 - **Data Access/Integration** - contains easier working with database using JdbcTemplate (wrapper for plain JDBC), or Spring Data which is wrapper for JPA
 - **Web** - which is used for easier web applications and web services (REST services) development
 - **Test** - the packages for implementing unit and integration tests
 - **AOP, Aspects, Instrumentation, Messaging**



Spring Applications Deployment

- Nowadays, Spring applications are widely used in the industry for cloud-based applications
- The usage of Spring framework is almost everywhere:
 - Banking systems
 - Streaming platforms
 - Applications of government agendas
 - Telecommunication systems (especially module Spring Integration)



Spring is Not Only Framework

- In Spring a lot of subprojects were created:
 - <https://spring.io/projects>
 - **Spring Data** - easier working with several databases
 - **Spring Security** - the complete solution for security scenarios (OpenID Connect, OAuth2, authentication, authorization, etc.)
 - **Spring Batch** - data management
 - **Spring LDAP** - support for LDAP storages including transactions; LdapTemplate (similar to JdbcTemplate)
 - **Spring Web Services** - support for SOAP-based web services
 - **Spring Cloud** - support for microservice architectures (Eureka, Hystrix, Zuul, etc.)
 - **Spring Integration** - useful for telecommunication companies to implement protocols like MQTT, web sockets and so on
 - **Spring Web Flux** - The sweet spot for Spring Web Flow are stateful web applications with controlled navigation such as checking in for a flight, applying for a loan, shopping cart checkout, or even adding a confirmation step to a form



Support for Spring in IDEs

- SpringSource Tool Suite (STS) - It is the extension of Eclipse IDE supported by Pivotal company (the owner of Spring framework)
- Spring IDE - the lightweight version of STS IDE
- IntelliJ IDEA
- Netbeans IDE

Creating a Simple Spring Application

- The minimum required dependencies:

```
<dependencies>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
  </dependency>
</dependencies>
```

The core library, necessary for creating Spring IoC etc.

Basic library of Spring, suitable for “hello world” applications

Creating a Simple Spring Application

- Starting up Spring container
 - Using XML plus annotations
config
("applicationContext.xml")
 - Placed in the root of classpath, e.g., in resources folder

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

    <context:annotation-config></context:annotation-config>
    <context:component-scan base-package="com.sedaq"/>

    <!-- <bean id="userRepository" class="com.sedaq.repository.UserRepository"/> -->
</beans>
```


Creating a Simple Spring Application

- Creating and starting Spring container
- Based on ClassPathXmlApplicationContext:

Loading XML config on classpath

```
import com.sedaq.model.Person;
import com.sedaq.repository.UserRepository;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class AppXmlAppContext {

    public static void main(String[] args) {
        ClassPathXmlApplicationContext applicationContext =
            new ClassPathXmlApplicationContext( configLocation: "applicationContext.xml");
        UserRepository userRepository = applicationContext.getBean(UserRepository.class);
        // call methods on userRepository
        Person person = userRepository.getPersonById(1L);
        System.out.println(person);
        applicationContext.close();
    }
}
```



Creating a Simple Spring Application

- Creating and starting Spring container
- Based on AnnotationConfigApplicationContext:

```
@Configuration
@ImportResource("classpath:applicationContext.xml")
@ComponentScan("com.sedag")
public class App {

    public static void main(String[] args) {
        AnnotationConfigApplicationContext applicationContext = new AnnotationConfigApplicationContext(App.class);
        UserRepository userRepository = applicationContext.getBean(UserRepository.class);
        // call methods on userRepository
        Person person = userRepository.getPersonById(1L);
        System.out.println(person);
        applicationContext.close();
    }
}
```

← Loading "legacy" XML config



Creating a Simple Spring Boot Application

- Maven dependencies:

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.1.6.RELEASE</version>
  <relativePath/> <!-- lookup parent from repository -->
</parent>
```

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>

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



Creating a Simple Spring Boot Application

- Creating and starting Spring container IoC
- As you can see, with Spring Boot it is quite easy
- ... and it's only the beginning of this framework in framework..

```
@SpringBootApplication
public class App {

    public static void main(String[] args) {
        ApplicationContext applicationContext = SpringApplication.run(App.class, args);
        UserRepository userRepository = applicationContext.getBean(UserRepository.class);
        userRepository.getPersonById(1L);
    }
}
```

Spring Boot Bundles

- We have only two dependencies in Maven and so much external libraries were downloaded...
- What happened?

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>

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

```
External Libraries
> < 11 > C:\Program Files\Java\jdk-11
> Maven: ch.qos.logback:logback-classic:1.2.3
> Maven: ch.qos.logback:logback-core:1.2.3
> Maven: com.jayway.jsonpath:json-path:2.4.0
> Maven: com.vaadin.external.google:android-json:0.0.20131108.vaadin1
> Maven: javax.annotation:javax.annotation-api:1.3.2
> Maven: junit:junit:4.12
> Maven: net.bytebuddy:byte-buddy:1.9.13
> Maven: net.bytebuddy:byte-buddy-agent:1.9.13
> Maven: net.minidev:accessors-smart:1.2
> Maven: net.minidev:json-smart:2.3
> Maven: org.apache.logging.log4j:log4j-api:2.11.2
> Maven: org.apache.logging.log4j:log4j-to-slf4j:2.11.2
> Maven: org.assertj:assertj-core:3.11.1
> Maven: org.hamcrest:hamcrest-core:1.3
> Maven: org.hamcrest:hamcrest-library:1.3
> Maven: org.mockito:mockito-core:2.23.4
> Maven: org.objenesis:objenesis:2.6
> Maven: org.ow2.asm:asm:5.0.4
> Maven: org.skyscreamer:jsonassert:1.5.0
> Maven: org.slf4j:jul-to-slf4j:1.7.26
> Maven: org.slf4j:slf4j-api:1.7.26
> Maven: org.springframework.boot:spring-boot:2.1.6.RELEASE
> Maven: org.springframework.boot:spring-boot-autoconfigure:2.1.6.RELEASE
> Maven: org.springframework.boot:spring-boot-starter:2.1.6.RELEASE
> Maven: org.springframework.boot:spring-boot-starter-logging:2.1.6.RELEASE
> Maven: org.springframework.boot:spring-boot-starter-test:2.1.6.RELEASE
> Maven: org.springframework.boot:spring-boot-test:2.1.6.RELEASE
> Maven: org.springframework.boot:spring-boot-test-autoconfigure:2.1.6.RELEASE
> Maven: org.springframework:spring-aop:5.1.8.RELEASE
> Maven: org.springframework:spring-beans:5.1.8.RELEASE
> Maven: org.springframework:spring-context:5.1.8.RELEASE
> Maven: org.springframework:spring-core:5.1.8.RELEASE
> Maven: org.springframework:spring-expression:5.1.8.RELEASE
> Maven: org.springframework:spring-jcl:5.1.8.RELEASE
> Maven: org.springframework:spring-test:5.1.8.RELEASE
> Maven: org.xmlunit:xmlunit-core:2.6.2
> Maven: org.yaml:snakeyaml:1.23
```



Why Use Spring Boot?

- To ease the Java-based applications development, unit and integration tests process
- To reduce development time by providing some defaults
- To increase productivity
- Uses best practices in the project creation
- No XML, configured by annotations and Java config
- Autoconfiguration is amazing
- Is Spring Boot too much magic?
 - Craig Walls statement: “When I drive my car, it’s unnecessary for me to fully understand how the engine works or individual components that the engine is made up from. Certainly, there are some people who enjoy that kind of detail, but for me it’s sufficient to turn the key, pump the gas, and steer my vehicle from one place to another. I take advantage of the abstraction (ignition, steering wheel, pedals) trusting that they properly coordinate with the underlying mechanics of the automobile.”



Spring Boot Executable .jar File (The Default)

- By default Spring Boot uses as final executable file the .jar file instead of .war
- To create such a class it is necessary to have the following class

```
@SpringBootApplication
public class App {

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

}
```

- By default Spring Boot uses as final executable file the .jar file instead of .war
- To create such a class it is necessary to have the following class

- To enable what auto-configurations are enabled start the application with --debug
- By default Spring Boot uses embedded Tomcat



Spring Boot Executable .war File

- To deploy Spring Boot application to external application server (not embedded)

```
<packaging>war</packaging>
```

- Remove 'spring-boot-maven-plugin' in build part
- Add the following dependency:

```
<dependency>  
  <artifactId>spring-boot-starter-tomcat</artifactId>  
  <groupId>org.springframework.boot</groupId>  
  <scope>provided</scope>  
</dependency>
```




Spring Boot Executable .war File

- Create the following class:

```
@SpringBootApplication
public class App extends SpringBootServletInitializer {

    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder application){
        return application.sources(App.class);
    }
}
```

The implementation of 'WebApplicationInitializer'



The class with @SpringBootApplication annotation



- Now it is configured to be deployable as .war file to arbitrary Jakarta/Java EE server (WildFly, Websphere, Weblogic, ...)
- Also, it could be run in an embedded server using, e.g., Tomcat or Jetty maven plugin



Section: Working with Configuration Properties

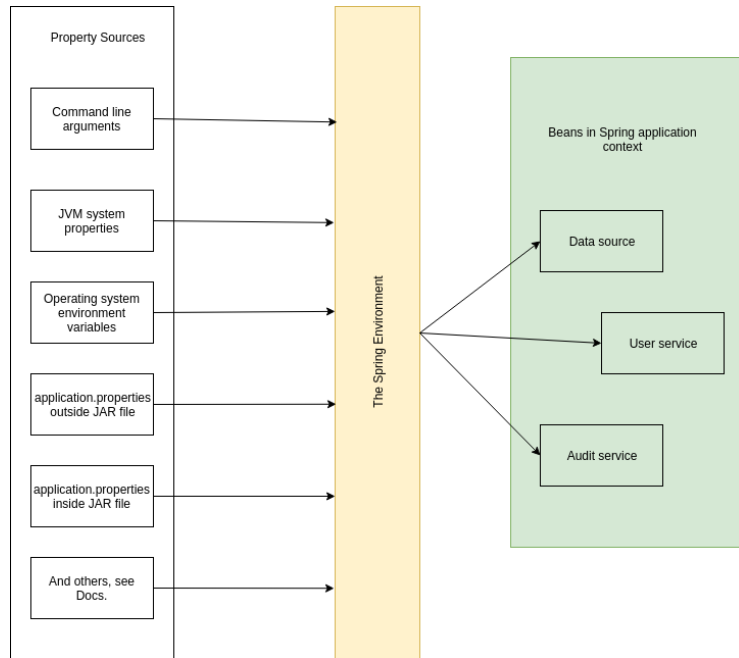


Working with Configuration Properties

- Working with configuration properties
 - a. Spring environment abstraction
 - b. Fine-tuning autoconfiguration
 - i. Configuration of data source
 - ii. Property file vs Yaml
 - c. External vs Internal config file
 - d. Configuration with Spring profiles

Spring Environment Abstraction

- The Spring environment is set through variety of possible places
- From property sources
 - application.properties
 - application.yml
 - etc.
- These could be inside or outside the application
- What is the configuration order of these properties?





Spring Environment Abstraction

- Setting the properties in application.properties file

```
server.port=9090
```
- Setting the properties in application.yml file (in my opinion easier to read large files)

```
server:  
  port: 9090
```
- Setting the server port as command line argument

```
$ java -jar myApp.jar --server.port=9090
```
- Setting the server port as operating system environment variable

```
$ export SERVER_PORT=9090
```

 - ❑ Notice that the environment variable setting is slightly different but Spring deal with it OK



application.properties

- By default Spring Boot lookup for application.properties file which could be in the following directories:
 - Classpath: /src/main/resources
 - The package: /src/main/resources/config
 - In the actual project directory
 - In subdirectory 'config' which must be in the actual subdirectory of the project
- In that file the arbitrary settings could be set, e.g., the setting of server.port or the setting of database connection, etc.
- Additionally, the properties could be set outside the application
- Please always comment all the properties in the property file
- The common application properties for Spring Boot applications listed in the following link:
<https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html>

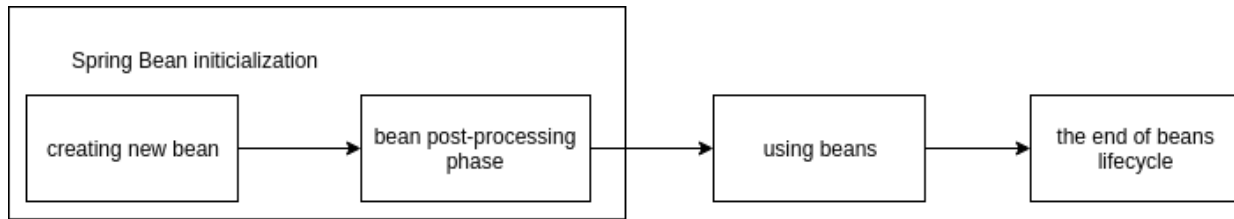


application.yml

- In addition to application.properties, you could use application.yml file
- The placement and rules are the same as for application.properties
- The difference is only in the file format of settings
- For larger settings, it is preferable due to easier readability

Configuration of Own Properties

- Sometimes you need to add additional properties which are not predefined by default as server.port, e.g., the property “contact.email”
- These properties could be set using context “property placeholder”
- If we look at the lifecycle of Spring context then:



- Beans are instantiated in the correct order always. Sometimes only please pay attention for cyclic dependencies in constructors, this could reach to the classical chicken-egg problem
- In post-processing phase, the “property-placeholder” is used to fully configure the newly created bean, based on the values from the property files



Creating Property Placeholder

- Sometimes you need to add additional properties which are not predefined by default as server.port, e.g., the property “contact.email”

@Configuration

public class PropertyPlaceholder {

// To resolve \${} in @Value

@Bean

public static PropertySourcesPlaceholderConfigurer propertyConfigInDev() {

PropertySourcesPlaceholderConfigurer confPropertyPlaceholder = **new** PropertySourcesPlaceholderConfigurer();

confPropertyPlaceholder.setIgnoreUnresolvablePlaceholders(**true**);

return confPropertyPlaceholder;

}

}



The Usage of Own Properties

- In application.properties:

`jdbc.url=jdbc:postgresql://localhost/rest-training`

- In configuration class:

```
@Configuration
@PropertySource("classpath:application.properties")
public class PersistenceConfiguration {
    @Value("${jdbc.url}")
    private String jdbcUrl;
    @Bean
    public HikariConfig hikariConfig() {
        HikariConfig hikariConfig = new HikariConfig();
        hikariConfig.setJdbcUrl(jdbcUrl);
        return hikariConfig;
    }
}
```

← Loading configuration property from file



Reading More Complex Properties

- In application.properties:

```
sedaq.oidc.issuers=google,facebook
```

- In configuration class:

```
@PropertySource("classpath:application.properties")  
public class ResourceServerConfiguration {  
  
    @Value("#{'${sedaq.oidc.issuers}'.split(',')}")  
    private List<String> issuers;  
  
}
```

← Loading configuration property as an
collection of values



External Configuration Properties

- To set the external file in configuration properties, it is necessary to set @PropertySource, e.g., as follows:

```
@PropertySource("file:${path.to.config.file}")
```

- This basically means that when you are running the application you need to provide additional parameter -Dpath.to.config.file to specify where the external configuration file is

```
$ /usr/bin/java -Dpath.to.config.file=/home/seda/workspace/properties/training/training-project.properties -jar /home/seda/workspace/sedaq-training/sedaq-rest-training/target/sedaq-rest-training-1.0.4.jar
```



@ConfigurationProperties (only Spring Boot)

- Sometimes it could be useful to load specific configuration properties to a class that will be later injected
 - to avoid @Value("\${some.property}) on a lot of places in the code

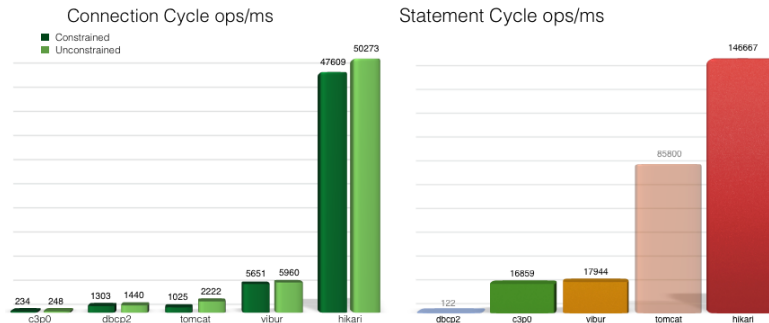
```
@Component
@ConfigurationProperties(prefix = "application.properties")
public class DatabaseProps {
    private String jdbcUrl;
    public String getJdbcUrl() {
        return jdbcUrl;
    }
}
```

- That class will be then injected into another classes which will use these properties

Configuration of Data Source

- Do not use DriverManager in the production ready project!
- Use some implementation of DataSource
 - The reason is that the DataSource uses connection pooling
 - The establishing JDBC connections is resource-expensive
 - It is preferable to use some pool of connections and reuse them whenever it is possible
- The comparison of DataSource implementation
- Previously, Spring Boot uses tomcat connection pooling, nowadays it uses Hikari

```
<dependency>  
  <groupId>com.zaxxer</groupId>  
  <artifactId>HikariCP</artifactId>  
  <version>3.3.1</version>  
</dependency>
```





Configuration of Data Source


@Configuration

```
public class PersistenceConfiguration {
```

@Bean

```
public HikariConfig hikariConfig() {  
    HikariConfig hikariConfig = new HikariConfig();  
    hikariConfig.setJdbcUrl("jdbc:postgresql://localhost/rest-training");  
    hikariConfig.setUsername("postgres");  
    hikariConfig.setPassword("postgres");  
    hikariConfig.addDataSourceProperty("cachePrepStmts", "true");  
    return hikariConfig;  
}
```

Setting the connection properties as url, username, password, etc.



@Bean

```
public DataSource dataSource() {  
    HikariDataSource hikariDataSource = new HikariDataSource(hikariConfig());  
    return hikariDataSource;  
}
```

Creating DataSource instance





Spring Profiles

- When the application is deployed to different run-time environments, usually some configuration details differs
- The details of database connection as username or password, it is probably not the same in the development environment as on the production environment
 - One way could be to use environment variables as the properties, e.g.,:

```
$ export SPRING_DATASOURCE_URL=jdbc:postgresql://localhost/rest-training
```
- Although this will work it is cumbersome to specify a lot of properties that way
- Instead of it, prefer using Spring Profiles



Spring Profiles - Configuration

- Basically there are two ways how to configure profiles:
 - Configure multiple property files, e.g.,
`application-dev.properties`
`application-prod.properties`
 - This file name should follow the naming convention `application-{profile name}.properties`
 - In Yaml configuration files it is also possible to add several spring profiles to one file separated by triple hyphens `--`
`-`

- Well, but how to active these profiles?
 - Configure it in the property as follows:
`spring.profiles.active=prod`

- Or as an environment variable:

```
$ export SPRING_PROFILES_ACTIVE=prod
```



Spring Profiles - Configuration

- Command line arguments to set a profile:

```
java -jar myApp.jar --spring.profiles.active=prod
```

- Well, it is possible to activate more than one profile?
 - Yes!
 - Just specify it as:

```
spring.profiles.active=prod,audit,some-other-profile
```

- Which way to prefer?
- In my opinion, it is preferable to use environment variables because you specify, e.g., the PROD property for all the applications deployed to the production server



Spring Profiles - Conditionally Creating Beans

- This @Profile annotation could be placed on the whole class or on bean creation methods, e.g.,

```
@Bean @Profile("dev") public HikariConfig hikariConfig(){...}
```

@Configuration

@Profile("PROD")

public class PersistenceConfigurationProd {

@Bean

public HikariConfig hikariConfig() {

HikariConfig hikariConfig = new HikariConfig();

hikariConfig.setJdbcUrl("jdbc:postgresql://localhost/rest-training");

//other properties

return hikariConfig;

}

}

@Configuration

@Profile("DEV")

public class PersistenceConfigurationProd {

@Bean

public HikariConfig hikariConfig() {

HikariConfig hikariConfig = new HikariConfig();

hikariConfig.setJdbcUrl("some-in-memory-db");

//other properties

return hikariConfig;

}

}



Spring @Conditional - Since Spring 4

- A Spring application context contains an object graph that makes up all the beans that our application needs at runtime
- Spring's @Conditional annotation allows us to define condition under which a certain bean is included into that object graph
- It is more flexible as Spring Profiles
- @Conditional,
- @ConditionalOnProperty,
- @ConditionalOnExpression,
- @ConditionalOnBean(OtherModule.class),
- @ConditionalOnResource(resources = "/logback.xml")



Spring @Conditional - Creating Custom Condition

- A Spring application context contains an object graph that makes up all the beans that our application needs at runtime

```
public class OnUnixCondition implements Condition {  
    @Override  
    public boolean matches(ConditionContext context,  
        AnnotatedTypeMetadata metadata) {  
        return System.getProperty("os.name").toLowerCase().contains("win");  
    }  
}
```

- The usage of such condition is as follows:

```
@Bean  
@Conditional(OnUnixCondition.class)  
UnixBean unixBean() {  
    return new UnixBean();  
}
```

What is Spring Boot Autoconfiguration?

- As was mentioned in the previous slides Spring Boot makes autoconfiguration for your specific settings based on the property file

```
spring.datasource.url=jdbc:postgresql://localhost/rest-training
spring.datasource.username=postgres
spring.datasource.password=postgres
```



- Spring Boot basically creates a `@Configuration` classes based on the given properties in the property file

```
@Configuration
@PropertySource("classpath:application.properties")
public class PersistenceConfigurationDev {

    @Bean public HikariConfig hikariConfig() {
        HikariConfig hikariConfig = new HikariConfig();
        hikariConfig.setJdbcUrl("jdbc:postgresql://localhost/rest-training");
        hikariConfig.setUsername("postgres");
        hikariConfig.setPassword("postgres");
        return hikariConfig;
    }

    @Bean public DataSource dataSource() {
        HikariDataSource hikariDataSource = new HikariDataSource(hikariConfig());
        return hikariDataSource;
    }
}
```



Section: Database Access



Database Access

- Database Access
 - a. Java Database Connectivity (JDBC)
 - b. Spring JDBC Template
 - c. JPA basics:
 - i. Entities, Entity relationships,
 - ii. Common annotations (@Entity, @Table, @Column, @Id, etc.)
 - iii. Fetch types (Lazy vs Eager)
 - iv. EntityManager
 - d. Spring Data
 - i. Paging and sorting
 - e. Transactions

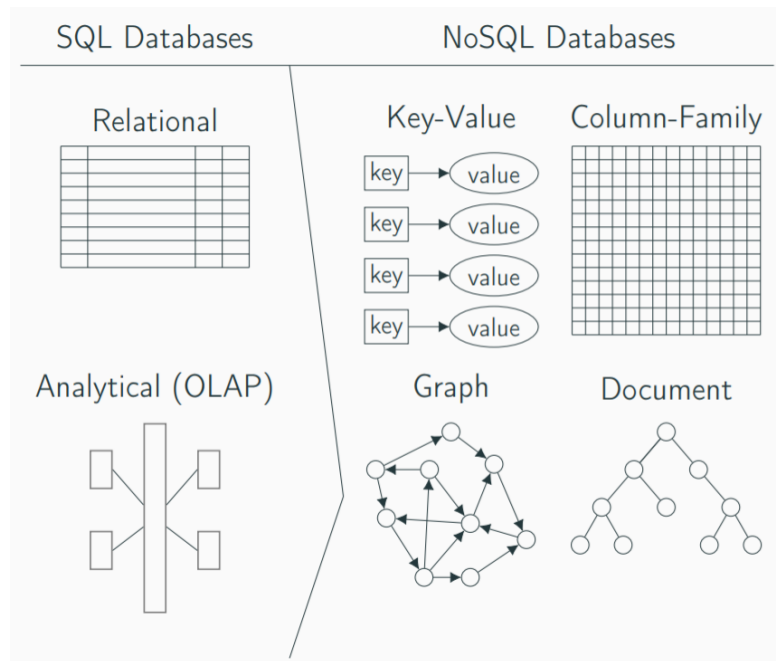


Database Access - Types of Databases

- Relational databases
 - Primarily used for storing and manipulation with structured data
 - Based on the relational data model created by Frank F. Codd in 1970 in IBM labs
 - Usually follows high level of consistency through Atomicity Consistency Isolation Durability (ACID)
 - The database design commonly follows the principle of normal forms (1NF, 2NF, 3NF, ...)
 - The examples: Oracle DB, IBM DB2, PostgreSQL, MySQL, Microsoft SQL Server, ...
- NoSQL databases
 - Usually used for storing semi-structured or unstructured data
 - Usually managed by a Consistency Availability Partition tolerance (CAP) or a Basically Available Soft-State Eventual consistency (BASE)
 - Data consistency is not crucial... good for marketing and behaviour data
 - The examples: MongoDB, Cassandra, Redis, ...
- <https://db-engines.com/en/ranking>

Database Access - NoSQL Databases

- NoSQL databases are divided into four main categories:
 - Key-Value -> Redis, ...
 - Column-Family -> Cassandra, ...
 - Graph -> Neo4J, ...
 - Document -> MongoDB, ...
- For these types of databases special query methods are essential
- How to work with them in Spring?
- -> Spring Data contains support for several NoSQL databases, e.g., MongoDB, Redis, Cassandra, Neo4J

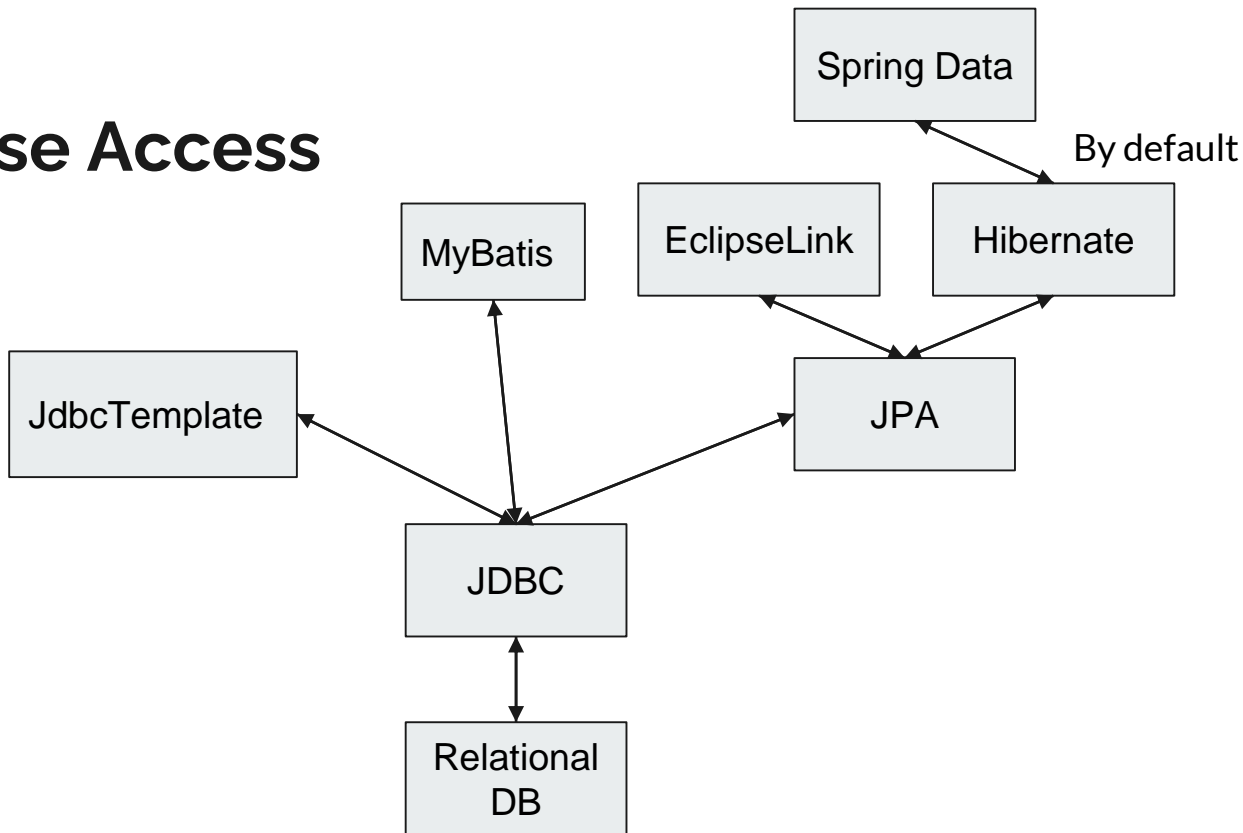




Database Access - Relational Databases

- Relational databases are designed using normal forms with the goal to reduce data redundancy and improve data integrity.
 - 1NF: To satisfy 1NF, we need to ensure that the values in each column of a table are atomic
 - 2NF:
 - Must be in 1NF
 - It does not have any non-prime attribute that is functionally dependent on any proper subset of any candidate key of the relation. A non-prime attribute of a relation is an attribute that is not a part of any candidate key of the relation.
 - 3NF:
 - Must be in 2NF
 - No non-prime (non-key) attribute is transitively dependent on any key i.e. no non-prime attribute depends on other non-prime attributes. All the non-prime attributes must depend only on the keys.
- Sometimes for analytical purposes it is suitable to do database denormalization

Database Access





Database Access

- JDBC
 - Accesses data as rows and columns
- JPA
 - Accesses data through Java objects using a concept called object-relational mapping (ORM). The idea is that you don't have to write as much code, and you get your data in Java objects.
- Each principle has it's cons and pros

Identifying the Structure of Relational Database

Table also called **Relation**

© guru99.com

CustomerID	CustomerName	Status
1	Google	Active
2	Amazon	Active
3	Apple	Inactive

Primary Key

Domain
Ex: NOT NULL

Tuple OR Row
Total # of rows is **Cardinality**

Column OR Attributes
Total # of column is **Degree**



Relational Database - Concepts

- **Attribute:** Each column in a Table. Attributes are the properties which define a relation. e.g., Student_Rollno, NAME,etc.
- **Tables** – In the Relational model the, relations are saved in the table format. It is stored along with its entities. A table has two properties rows and columns. Rows represent records and columns represent attributes.
- **Tuple** – It is nothing but a single row of a table, which contains a single record.
- **Relation Schema:** A relation schema represents the name of the relation with its attributes.
- **Degree:** The total number of attributes which in the relation is called the degree of the relation.
- **Cardinality:** Total number of rows present in the Table.
- **Column:** The column represents the set of values for a specific attribute.
- **Relation instance** – Relation instance is a finite set of tuples in the RDBMS system. Relation instances never have duplicate tuples.
- **Relation key** - Every row has one, two or multiple attributes, which is called relation key.
- **Attribute domain** – Every attribute has some pre-defined value and scope which is known as attribute domain



Relational Model - Advantages

- **Simplicity:** A relational data model is simpler than the hierarchical and network model.
- **Structural Independence:** The relational database is only concerned with data and not with a structure. This can improve the performance of the model.
- **Easy to use:** The relational model is easy as tables consisting of rows and columns is quite natural and simple to understand
- **Query capability:** It makes possible for a high-level query language like SQL to avoid complex database navigation.
- **Data independence:** The structure of a database can be changed without having to change any application.
- **Scalable:** Regarding a number of records, or rows, and the number of fields, a database should be enlarged to enhance its usability.



Relational Model - Disadvantages

- Few relational databases have limits on field lengths which can't be exceeded.
- Relational databases can sometimes become complex as the amount of data grows, and the relations between pieces of data become more complicated.
- Complex relational database systems may lead to isolated databases where the information cannot be shared from one system to another.
- Difficult working with BLOB data



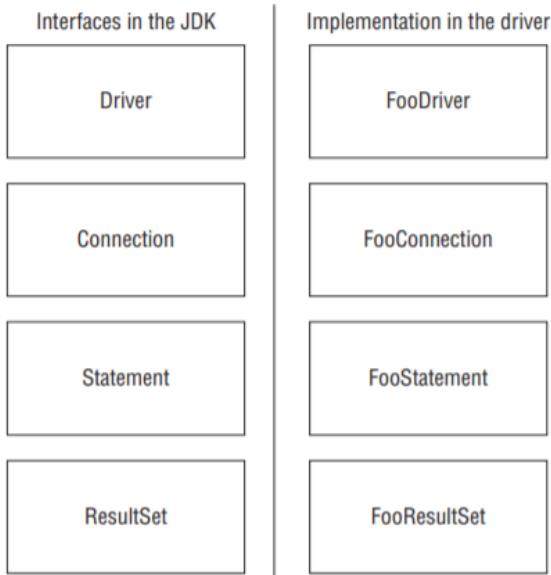
Writing Basic SQL Statements

- **INSERT:** Add a new row to the table
 - **SELECT:** Retrieve data from the table
 - **UPDATE:** Change zero or more rows in the table
 - **DELETE:** Remove zero or more rows from the table
-
- The example:
 - `SELECT id, email FROM person`



Java Database Connectivity (JDBC)

- Introducing the interfaces of JDBC
- What these drivers do?
- **Driver:** Knows how to get a connection to the database
- **Connection:** Knows how to communicate with the database
- **Statement:** Knows how to run the SQL
- **ResultSet:** Knows what was returned by a SELECT query





Java Database Connectivity (JDBC)

- Databases have the driver which implements these interfaces
- For example PostgreSQL provides the driver through the following maven dependency

```
<dependency>  
  <groupId>org.postgresql</groupId>  
  <artifactId>postgresql</artifactId>  
  <version>42.2.6</version>  
</dependency>
```

- For that reason, since every database access approach is build on top of JDBC, it is essential to add the driver for the selected database into your classpath



JDBC - Getting Connection

- Connection to a database is obtained in two possible ways
- **DriverManager**
 - NEVER USE! DriverManager in production ready code!
- **DataSource**
 - Always prefer selected DataSource implementation
 - DataSource maintains a connection pool so that you can keep reusing the same connection rather than needing to get a new one each time. Even the JavaDoc says DataSource is preferred over DriverManager



JDBC - The Example Query

- Pure JDBC approach is often not selected due to implementation difficulty where a lot of code is necessary to implement for a simple queries

```
public List<PersonDTO> getAllPersons() {  
    try (Connection conn = DBConnectionHikariDataSource.getConnection();  
        Statement statement = conn.createStatement();  
        ResultSet rs = statement.executeQuery("SELECT * FROM person");) {  
        return mapPersonTableToDTO(rs);  
    } catch (SQLException e) {  
        e.printStackTrace();  
    }  
    return Collections.emptyList();  
}
```

- For real-world scenarios please prefer PreparedStatement to prevent SQL Injection attacks

JDBC - Going Through the ResultSet

- The calling to `rs.next()` is essential for getting the cursor to the right place

Initial position	→	id integer	name character varying(255)	num_acres numeric
rs.next() true	→	1	African Elephant	7.5
rs.next() true	→	2	Zebra	1.2
rs.next() false	→			



JDBC - Retrieving Columns From ResultSet

- The database columns could be retrieved using:

- Order of a column

```
if(rs.next()) System.out.println(rs.getInt(1));
```

- Column name

```
if(rs.next()) System.out.println(rs.getInt("id"));
```

- I prefer the variant of the column name but the choice is up to you



JDBC - Executing CRUD operations

Method	DELETE	INSERT	SELECT	UPDATE
<code>stmt.execute()</code>	Yes	Yes	Yes	Yes
<code>stmt.executeQuery()</code>	No	No	Yes	No
<code>stmt.executeUpdate()</code>	Yes	Yes	No	Yes



Spring Variant of JDBC -> JdbcTemplate

- Does the previous code snippets seem too large and difficult for you?
- Did you think more about how to implement this kind of code instead of thinking about the business logic behind SQL?
- Spring provides JdbcTemplate for you, it is quite easy, lightweight and very powerful way to retrieve aggregate or analytical data



Spring JDBC vs JdbcTemplate - The Example

More focus on code



```
public List<PersonDTO> getAllPersons() {  
    try (Connection conn = DataSource.getConnection();  
         Statement statement = conn.createStatement();  
         ResultSet rs = statement  
             .executeQuery("SELECT * FROM person");) {  
        return mapPersonTableToDTO(rs);  
    } catch (SQLException e) {  
        e.printStackTrace();  
    }  
    return Collections.emptyList();  
}
```

More focus on SQL



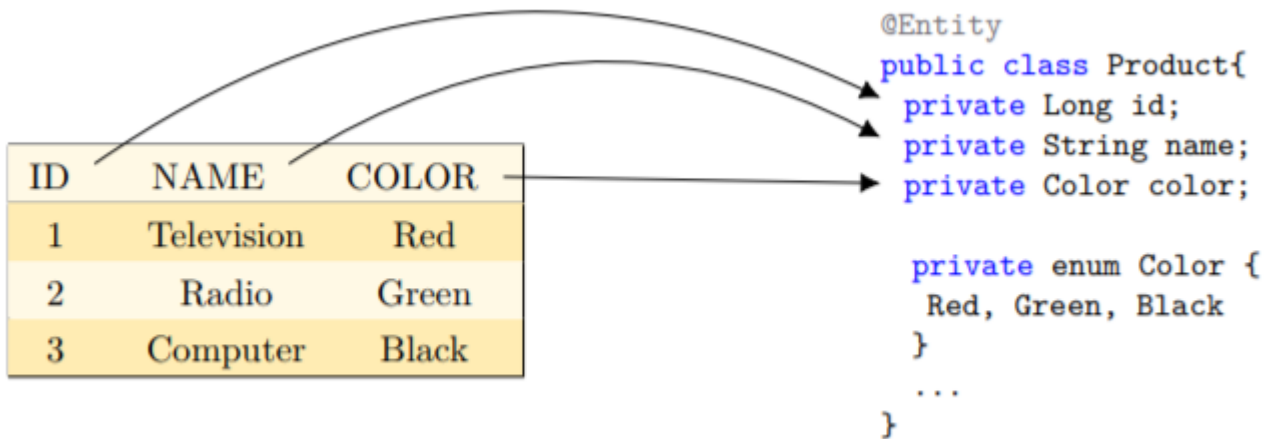
```
public List<PersonDto> getAllPersons() {  
    return jdbcTemplate  
        .query("SELECT * FROM person", new PersonMapper());  
}
```



Java Persistence API (JPA)

- Replaces EJB 2.1 entity beans with non EJB-entity classes
- It is specification for Object Relational Mapping (ORM)
- It does not require Java/Jakarta EE servers (Websphere, Weblogic, Tomcat, ...)
- Can be used with:
 - Container-managed persistence
 - Application-managed persistence
- JPA provides and object representation of a data
 - Entity classes are mapped to a relational database
- Several frameworks are implementing this specification
 - EclipseLink (the reference implementation), Hibernate (by default used by Spring), TopLink etc.

JPA





JPA Entity

- Plain Old Java Object (POJO) class
- This class must contain non-argument constructor and @Entity annotation
- Is serializable, can be used as detached object
- Entities can be queried
- It can be uniquely identified by a primary key
- It manages persistent data in concern with a JPA entity manager



Entity Class Requirements

- The class must be annotated with the `javax.persistence.Entity` annotation
- The class must have a public or protected, no-argument constructor. The class may have other constructors
- The class must not be declared final. No methods or persistent instance variables must be declared final
- If an entity instance is passed by value as a detached object, such as through a session bean's remote business interface, the class must implement the `Serializable` interface.
- Entities may extend both entity and non-entity classes, and non-entity classes may extend entity classes
- Persistent instance variables must be declared private, protected, or package-private and can be accessed directly only by the entity class's methods. Clients must access the entity's state through accessor or business methods



JPA - Creating an Person Entity

`@Entity`

Defines Entity

```
public class Person implements Serializable {
```

`@Id`

Primary Key

```
@GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
@Column(name = "id", updatable = false, nullable = false)
```

```
private Long id;
```

`@Column(nullable = true)`

Columns Definitions

```
private LocalDate birthday;
```

```
public Person() {
```

```
    // hibernate requires non-args constructor
```

```
}
```

```
}
```




Entity Class - Primary Keys

- Every JPA entity must have a primary key
- It is used to distinguish one entity instance from another
 - It is used by EntityManager to identify the object
 - The @Id annotation is used to identify the entity primary key
 - Commonly, primary key is set as Long or Integer, but it can be a class that corresponds to several database columns (composite primary keys)

@Id private Long id;

- Primary keys could be auto-generated (the most common variant)



Entity Class - Primary Keys - Generation Strategies

- JPA provides four generation strategies
 - AUTO
 - IDENTITY
 - SEQUENCE
 - TABLE
- The strategy is specified using @GeneratedValue annotation

```
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
private Long id;
```



Generation Strategy - AUTO

- The *GenerationType.AUTO* is the default generation type and lets the persistence provider choose the generation strategy.
- If you use Hibernate as your persistence provider, it selects a generation strategy based on the database specific dialect. For most popular databases, it selects *GenerationType.SEQUENCE*

```
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
private Long id;
```



Generation Strategy - IDENTITY

- The *GenerationType.IDENTITY* is the easiest to use but not the best one from a performance point of view
- It relies on an auto-incremented database column and lets the database generate a new value with each insert operation
- From a database point of view, this is very efficient because the auto-increment columns are highly optimized, and it doesn't require any additional statements

```
@Id @GeneratedValue(strategy = GenerationType.AUTO)
private Long id;
```

- This approach has a significant drawback if you use Hibernate. Hibernate requires a primary key value for each managed entity and therefore has to perform the insert statement immediately
- This prevents it from using different optimization techniques like JDBC batching.



Generation Strategy - SEQUENCE

- It requires additional select statements to get the next value from a database sequence. But this has no performance impact for most applications
- And if your application has to persist a huge number of new entities, you can use some [Hibernate specific optimizations](#) to reduce the number of statements.

```
@Id @GeneratedValue(strategy = GenerationType.Sequence)
@SequenceGenerator(name="person_generator", sequenceName = "person_seq", allocationSize=50)
private Long id;
```



Generation Strategy - TABLE

- The *GenerationType.TABLE* gets only rarely used nowadays
- It simulates a sequence by storing and updating its current value in a database table which requires the use of pessimistic locks which put all transactions into a sequential order
- This slows down your application, and you should, therefore, prefer the *GenerationType.SEQUENCE*, if your database supports sequences, which most popular databases do



Generation Strategy - SUMMARY

JPA offers 4 different ways to generate primary key values:

1. AUTO: Hibernate selects the generation strategy based on the used dialect,
2. IDENTITY: Hibernate relies on an auto-incremented database column to generate the primary key,
3. SEQUENCE: Hibernate requests the primary key value from a database sequence,
4. TABLE: Hibernate uses a database table to simulate a sequence.



Overriding Default Mapping

- When you design JPA entities you probably reach to state that you need to rename the table or column names
- It is done by @Table annotation on a class and @Column annotation on an field

```
@Entity
@Table(name = "contact")
public class Contact implements Serializable {
    ...
    @Column(name = "contact_type")
    private String contactType;
}
```

- If you need to escape the names you could use the following: `@Table(name = "\"contact\"")`



Transient fields

- If you need to define fields which should not be persisted you could annotate them @Transient

@Transient

```
private float cartTotal;
```

- This field would not be persisted by persistent provider but still it could be serialized (the difference between @Transient and Java SE transient keywords)



Persistent Data Types

Properties can be of the following data types:

- Java primitive types and Java wrappers, including:
 - Integer, Boolean, Float, and Double
- Serializable types, including:
 - String, BigDecimal, and BigInteger
- Arrays of bytes and characters, including:
 - byte[] and Byte[], char[] and Character[]
- Temporal types, including (but not limited to):
 - java.util.Date, java.util.Calendar, java.sql.Date, and java.sql.Timestamp
 - Since Hibernate 5 it is supported to use LocalDateTime, .. and the other classes from package java.time
 - In Hibernate 4 you must apply converters for that purpose
- Enums and collections
- Other entities



Persistent Fields Versus Persistent Properties

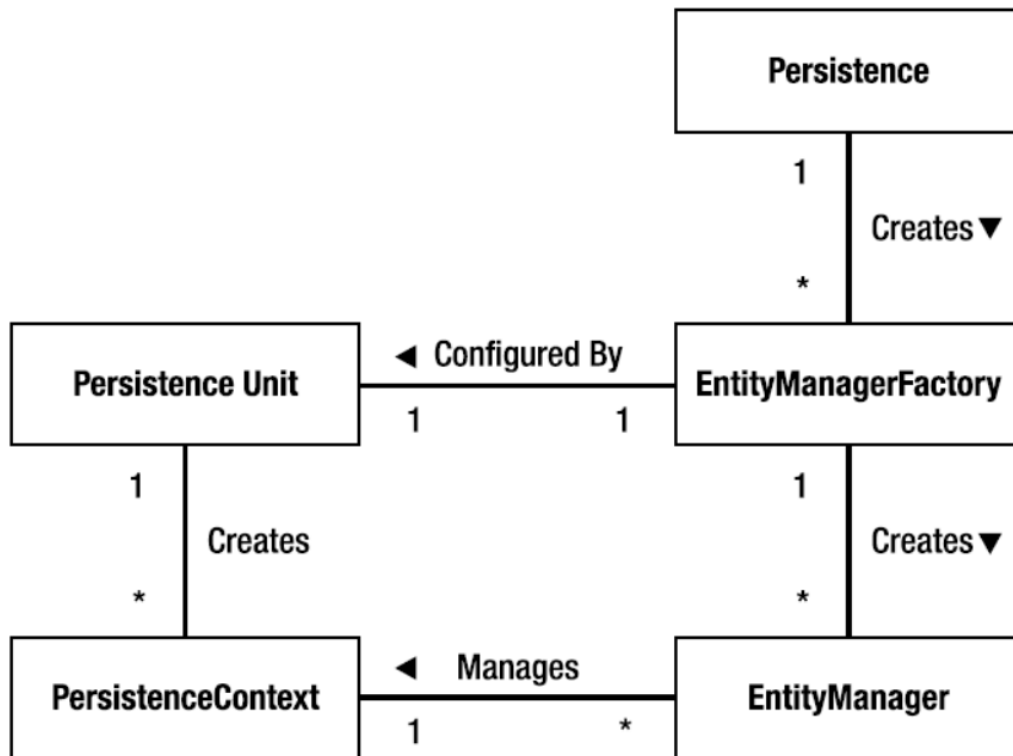
- In JPA, the state is retrieved from fields or from properties
- The entity state is synchronized with the database
- The base rules for field access is:
 - It should not be read from client directly (it should be private)
 - Unless they are annotated with `@Transient`, all the fields are persistent
- The base rules for property access is:
 - Access should be public or protected
 - Persistent annotation could be placed only on getter
 - It must follow the JavaBeans naming convention
- What access to choose?
 - It depends if we need additional process, If we need additional processing then property-based access is preferred, in other way the field access is preferred
 - If we annotate both field and property, on one of them should be `@Transient` annotation



EntityManager

- EntityManager is a part of the Java Persistence API
- Chiefly, it implements the programming interfaces and lifecycle rules defined by the JPA specification.
- Moreover, we can access the Persistence Context, by using the APIs in *EntityManager*.
- Entities to that EntityManager holds the references are called managed entities
- A set of entities within an entity manager at any given time is called its persistent context

Persistence



Persistence

- Persistence is configured in persistence.xml file placed in src/main/resources/META-INF folder

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
    http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd" version="2.1">
  <persistence-unit name="myPersistenceUnit" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
    <properties>
      <property name="hibernate.connection.url" value="jdbc:derby:memory:testdb" />
    ...
    </properties>
  </persistence-unit>
</persistence>
```

RESOURCE_LOCAL
OR JTA

- If you use Hibernate you could configure it in hibernate.cfg.xml file



Obtaining EntityManager

- In Java SE, you need to go through EntityManagerFactory
- In Java/Jakarta EE you could obtain entity manager by injection:

```
@PersistenceContext  
private EntityManager em;
```

- This is the use-case of injecting default entity manager
- If we want to inject some other defined entity manager we could specify

```
@PersistenceContext(name="myPersistenceUnit")  
private EntityManager em;
```



EntityManager Base Queries - Find

- Finding an entity:

```
Person person = em.find(Person.class, 14);
```

- It finds a person with ID 14
- The person object is now managed object by entity manager
- If no object is found the null is returned



EntityManager Base Queries - Update

- The entity is firstly find to make that entity managed and then any change on managed entity is stored

```
Person = em.find(Person.class, 14); ← Manage this entity  
person.setSalary(person.getSalary()+500); ← Changes are synchronized with  
DB in commit
```

- If we are working with unmanaged entity we could call merge firstly

```
person.setSalary(person.getSalary() + 1000);  
Person personCopy = em.merge(person);
```

- Alternative is to use merge method, this updates unmanaged entity and returns managed entity



EntityManager Base Queries - Delete

- First make entity managed and then remove the entity

```
Person person = em.find(Person.class, 14);  
em.remove(person) ;
```

- If the find method returns null then remove method will throw an IllegalArgumentException



EntityManager Base Queries - SELECT

- Select queries are using JPQL dialect as follows:

```
TypedQuery<Person> persons =  
    em.createQuery("SELECT p FROM Person p", Person.class);  
List<Person> personsList= persons.getResultList();
```

- This dialect is quite similar to classical SQL, but instead of querying to table and column names you provide the name of Entity and the name of entity fields



JPA Database Schema Modelling

- Until now, we used only single entity classes
- In practice, databases have a lot of tables
- For that reason, JPA provides the possibility to design relations between entities
- The relations are basically:
 - Uni-directional (the relation is presented only at its owning side)
 - Bi-directional (the relation is presented in both sides)
- These relations could be one-to-one, one-to-many, many-to-one, many-to-many

JPA Database Schema Modelling - One-to-One

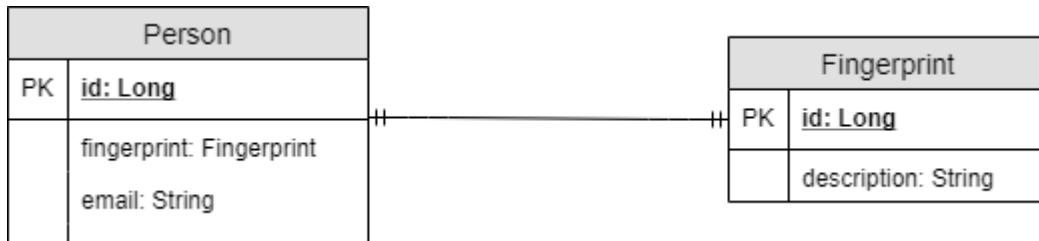
- Each person has directly one fingerprint and each fingerprint is assigned to directly one person
 - “Hopefully”
- Person is the owning side
- Owning side means that it holds the Foreign Key (FK)



One-to-One Relation Using uni-directional

```
@Entity
public class Person {
    @Id
    private Long id;
    private String email;
    @OneToOne
    @JoinColumn(name="fingerprint_id")
    private Fingerprint fingerprint;
}
```

```
@Entity
public class Fingerprint {
    @Id
    private Long id;
    private String description;
}
```



One-to-One Relation Using bi-directional

```
@Entity
public class Person {
    @Id
    private Long id;
    private String email;
    @OneToOne
    @JoinColumn(name="fingerprint_id")
    private Fingerprint fingerprint;
}
```

Person	
PK	<u>id: Long</u>
	fingerprint: Fingerprint
	email: String

```
@Entity
public class Fingerprint {
    @Id
    private Long id;
    private String description;
    @OneToOne(mappedBy="fingerprint")
    private Person person;
}
```

Fingerprint	
PK	<u>id: Long</u>
	description: String

Name of attribute in reference entity

This is called the inverse side

One-to-Many Relation Using uni-directional

- The one side is always holding the FK



One-to-Many Relation Using uni-directional

```
@Entity
public class Person {
    @Id
    private Long id;
    private String email;
    private String name;
}
```

```
@Entity
public class Order {
    @Id
    private Long id;
    @ManyToOne
    @JoinColumn(name = "person_id")
    private Person person;
}
```



One-to-Many Relation Using bi-directional

```
@Entity
public class Person {
    @Id
    private Long id;
    private String email;
    private String name;
    @OneToMany(mappedBy = "person")
    private Set<Order> orders;
}
```

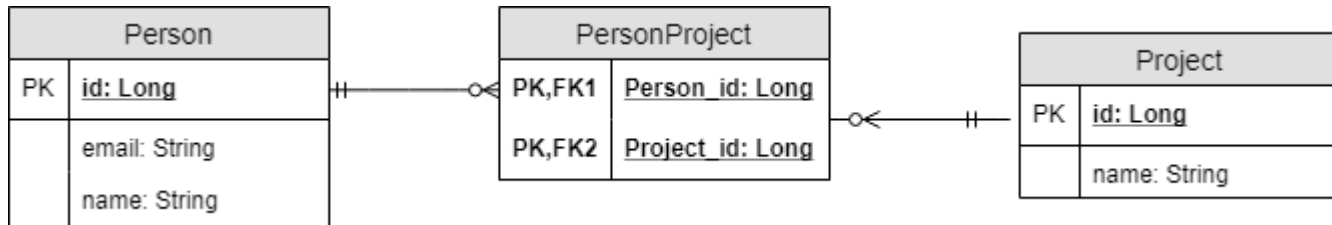
```
@Entity
public class Order {
    @Id
    private Long id;
    @ManyToOne
    @JoinColumn(name = "person_id")
    private Person person;
}
```



Many-to-Many Relation Using bi-directional

```
@Entity
public class Person {
    @Id
    private Long id;
    @ManyToMany
    @JoinTable(name="PersonProject",
        joinColumns=@JoinColumn(name="Person_id"),
        inverseJoinColumns=@JoinColumn(name="Project_id"))
    private Set<Project> projects;
}
```

```
@Entity
public class Project {
    @Id
    private Long id;
    @ManyToMany(mappedBy = "projects")
    private Set<Person> persons;
}
```





What Collections You Should Use?

- Prefer using Set for many-to-many associations
- Set is the most similar to the relational abstract model
- In some versions of Hibernate (e.g., in Hibernate 4) the List for collections generates additional queries



FetchTypes

- The *FetchType* defines when JPA gets the related entities from the database, and it is one of the crucial elements for a fast persistence tier
- In general, you want to fetch the entities you use in your business tier as efficiently as possible. But that's not that easy.
- You either get all relationships with one query or you fetch only the root entity and initialize the relationships as soon as you need them.



FetchType - LAZY

- Fetch type LAZY means that when you retrieve person entity no additional query for orders is processed
- If you press getOrders() then the additional query for retrieving orders for that person is executed

```
@Entity
public class Person {
    @Id
    private Long id;
    private String email;
    private String name;
    @OneToMany(mappedBy = "person", fetch = FetchType.LAZY)
    private Set<Order> orders;
}
```



FetchType - EAGER

- Fetch type EAGER means that the related entity is loaded into memory at build-time
- This is quite dangerous in case you have a lot of data..
- It could be used only in cases when you always want to work with all the data from related entity

```
@Entity
public class Person {
    @Id
    private Long id;
    private String email;
    private String name;
    @OneToMany(mappedBy = "person", fetch = FetchType.EAGER)
    private Set<Order> orders;
}
```



Default FetchType

- In JPA 2.0:
 - OneToMany: LAZY
 - ManyToOne: EAGER
 - ManyToMany: LAZY
 - OneToOne: EAGER



FetchType LAZY N+1 Problem

- Consider the following scenario?
- How many SQL commands are executed?

```
@Entity
public class Person {
    @Id
    private Long id;
    private String email;
    private String name;
    @OneToMany(mappedBy = "person", fetch = FetchType.LAZY)
    private Set<Order> orders;

    public Set<Order> getOrders(){ return orders;}
}
```

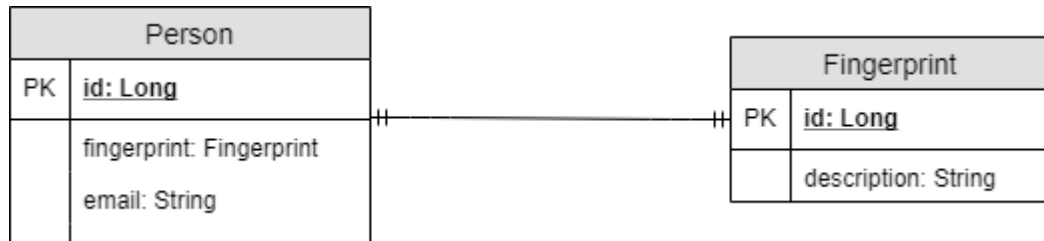
```
TypedQuery<Person> persons = em.query("SELECT p FROM
Person p", Person.class);
List<Person> personsList = em.getResultList();
personsList.forEach(person ->{
    System.out.println(person.getOrders());
});
```

Cascading

- Removing an entity in a relationship

```
public void removeFingerPrint(Long personId) {  
    Person person = em.find(Person.class, personId);  
    FingerPrint fingerPrint = person.getFingerPrint();  
    em.remove(fingerPrint);  
}
```

- This will lead to the database error foreign key violation
- To fix it we must explicitly remove fingerPrint from and Person before the commit



```
public void removeFingerPrint(Long personId) {  
    Person person = em.find(Person.class, personId);  
    FingerPrint fingerPrint = person.getFingerPrint();  
    person.setFingerPrint(null);  
    em.remove(fingerPrint);  
}
```



Cascading

- Entities with other entities could be managed by cascading operations

```
@Entity
public class Person {
    @Id
    private Long id;
    private String email;
    @OneToOne(cascade = CascadeType.REMOVE)
    @JoinColumn(name="fingerprint_id")
    private Fingerprint fingerprint;
}
```

Cascade mode attributes include:

- **PERSIST**: Cascade persist operations from source to target.
- **MERGE**: Cascade merge operations.
- **REMOVE**: Cascade remove operations.
- **REFRESH**: Cascade refresh operations.
- **DETACH**: Cascade detach operations.
- **ALL**: Cascade all entity state change operations (persist, merge, remove, refresh, detach) from source to target.
- Multiple options can be specified in a comma-separated list in braces:
- `@OneToOne(cascade = {CascadeType.MERGE, CascadeType.PERSIST})`



orphanRemoval()

- orphanRemoval has nothing to do with ON DELETE CASCADE
- It is entirely ORM-specific thing
- It marks “child” entity to be removed when it’s no longer referenced from the “parent” entity, e.g., when you remove the child entity from the corresponding collection of the parent entity
- ON DELETE CASCADE is a database-specific thing, it deletes the “child” row in the database when the “parent” row is deleted



Java Persistence Query Language (JPQL)

- JPQL is JPA query language for querying the database
- The benefit is that it is converted into database specific dialect, thus migration between databases is usually easier than in the case of pure JDBC (avoid using nativeQuery as much as possible)

```
SELECT p FROM Person p;
```

```
SELECT p FROM Person p WHERE p.salary > 2000;
```

```
SELECT COUNT(p) FROM Person p;
```



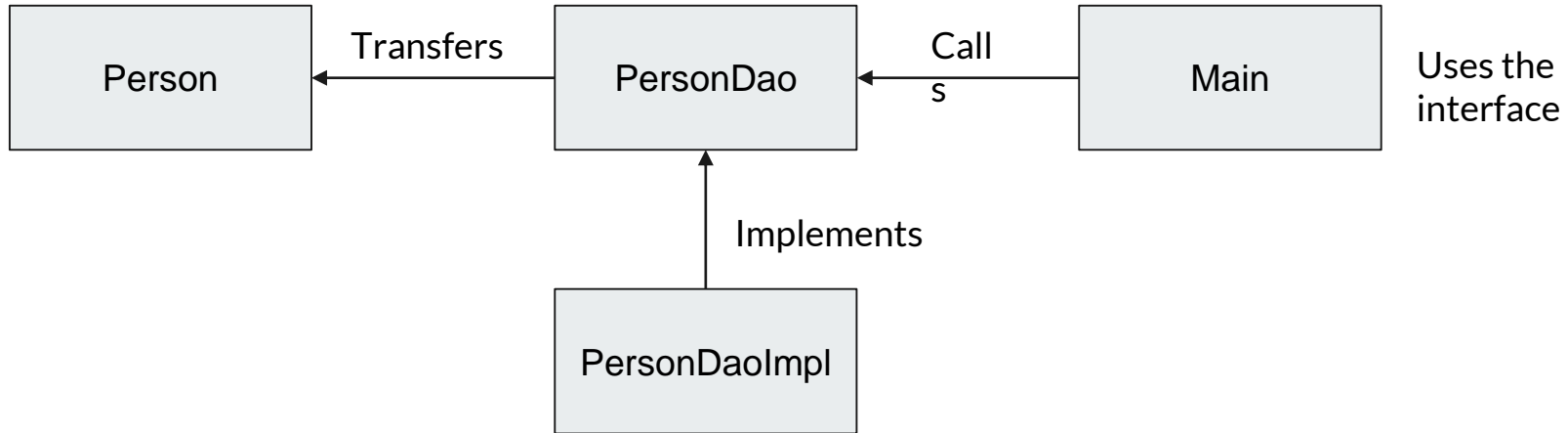
JOIN Between Entities

- In JPA it is essential to know JOIN FETCH
- If you want to load related entity data please write JOIN FETCH queries:

```
SELECT p FROM Person p JOIN FETCH Address
```

- This will significantly reduce the number of queries generated by JPA implementation framework

How We Design JPA Applications?




Here we inject EntityManager and call queries on it



And Finally... Spring Data !

- Spring Data is a wrapper over JPA
- No More DAO Implementations necessary !!
- Spring Data takes this simplification one step forward and **makes it possible to remove the DAO implementations entirely**
- The interface of the DAO is now the only artifact that we need to explicitly define
- In Spring Data it is called Repository



Spring Data - Repository

- This is the only thing you need to configure to have basic CRUD operations on Person entity available...

@Repository

```
public interface PersonRepository extends JpaRepository<Person, Long> {  
    Person findByName(String name);  
}
```

- Based on this interface the implementation class is generated



Spring Data - Automatic Custom Query

- The query is generated by method name

```
@Repository
public interface PersonRepository extends JpaRepository<Person, Long> {
    Person findByName(String name);
}
```



Spring Data - Manual Custom Query - Using Parameters

```
@Repository
public interface PersonRepository extends JpaRepository<Person, Long> {
    @Query("SELECT p FROM Person p WHERE p.name = ?1")
    Person findByName(String name);
}
```



Spring Data - Manual Custom Query

- The query is generated by method name

```
@Repository
public interface PersonRepository extends JpaRepository<Person, Long> {
    @Query("SELECT p FROM Person p WHERE p.name = :name")
    Person findByName(@Param("name") String name);
}
```




Spring Data - DELETE/UPDATE

- DELETE or UPDATE queries have to configure also @Modifying annotation

@Modifying

@Query("DELETE FROM Person p WHERE p.id = :personId")

void deletePersonById(@Param("personId") Long personId);




Spring Data - Pagination

- Once we have our repository **extending** from *PagingAndSortingRepository*, we just need to:
 - Create or obtain a *PageRequest* object, which is an implementation of the *Pageable* interface
 - Pass the *PageRequest* object as an argument to the repository method we intend to use
- We can create a *PageRequest* object by passing in the requested page number and the page size. Here, **the page counts starts at zero**:

```
Pageable firstPageWithTwoElements = PageRequest.of(0, 2);  
Pageable secondPageWithFiveElements = PageRequest.of(1, 5);
```
- The example in Spring Data repository:


```
Page<Person> findAll(Pageable pageable);
```



Spring Data - Pagination

- Specifying own query with Pageable method argument..
- In that case, it is necessary to define countQuery:

```
@Query(value = "SELECT p FROM Person p WHERE p.id = :personId",  
        countQuery = "SELECT COUNT(p) FROM Person p WHERE p.id = :personId")  
Page<Person> findPersonById(@Param("personId") Long personId, Pageable pageable);
```



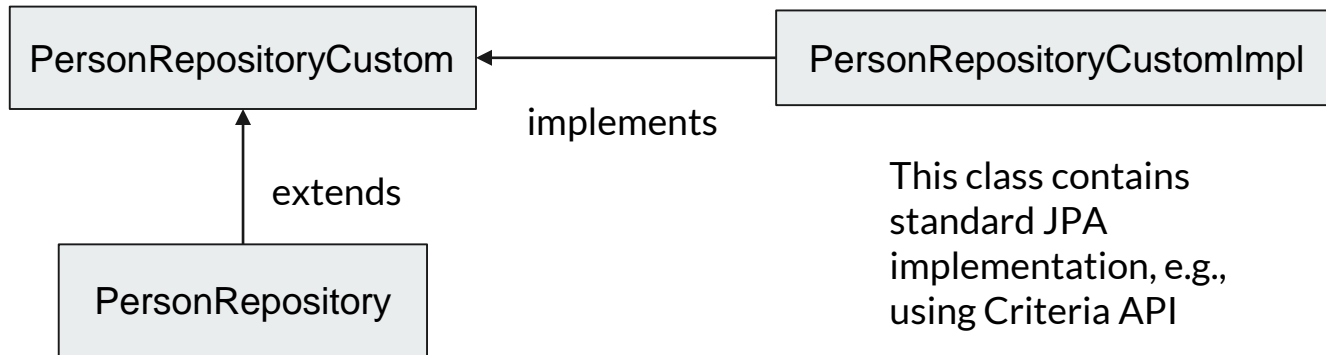
Spring Data - Pagination

- Specifying own query with Pageable method argument..
- In that case, it is necessary to define countQuery:

```
@Query(value = "SELECT p FROM Person p WHERE p.id = :personId",  
        countQuery = "SELECT COUNT(p) FROM Person p WHERE p.id = :personId")  
Page<Person> findPersonById(@Param("personId") Long personId, Pageable pageable);
```


Spring Data - Dynamic Queries

- Not everything could be defined in @Query
- Sometimes we need to write standard queries using EntityManager instance





Spring Data - EntityGraph

- **JPA 2.1 has introduced the Entity Graph feature as a more sophisticated method of dealing with performance loading.**
- It allows defining a template by grouping the related persistence fields which we want to retrieve and lets us choose the graph type at runtime.

```
@EntityGraph(attributePaths = {"address", ""})  
Optional<Person> findById(Long id);
```

- <https://www.baeldung.com/jpa-entity-graph>



Spring Data - Named Queries

- A named query is declared on an entity by using the `@NamedQuery` annotation

```
@Entity
```

```
@NamedQuery(name="Person.findByEmail",
```

```
    query="SELECT p FROM Person p WHERE p.email = :email")
```

```
public class Person {
```

```
    //...
```

```
}
```

- A named query is created with the `createNamedQuery` method



Query Best Practices

- Use named queries whenever possible
 - Named queries are compiled and highly optimized by providers
 - Dynamic queries will always be less efficient than named queries
- A named query is created with the `createNamedQuery` method
- Always load to `PersistenceContext` all the data you need in transaction in as less queries as possible



Spring Data - Configuration

- The only necessary configuration is configuring `@EnableJpaRepositories`

```
@EnableJpaRepositories(basePackages = "org.sedaq.persistence.repository")
public class PersistenceConfig {
    ...
}
```

- In property file:
spring.datasource.url=jdbc:h2:mem:db;DB_CLOSE_DELAY=-1
spring.datasource.username=sa
spring.datasource.password=sa



JPA - What We Did Not Cover

- Advanced Modelling (Inheritance, ...)
- @ElementCollection, @Embeddable, Map
- Criteria API
- Advanced JPA features
- StoredProcedures
- Caching
- Indexes



Section: Developing Service Layer



Section: Developing Service Layer

- a. Reviewing problem statement
- b. Consuming another REST service (synchronously vs asynchronously)
 - i. RestTemplate
 - ii. WebResource
 - iii. Declarative REST Feign Client
- c. Consuming another SOAP service
 - i. Implementing SOAP Clients (wsimport)
 - ii. Dispatcher Client

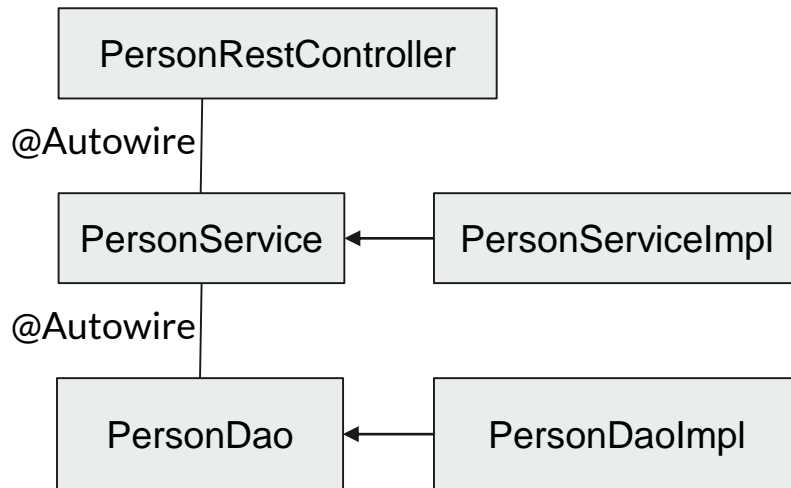


Developing Service Layer

- Service layer is used for developing business logic
- It contains mapping from entity classes to DTO classes
- It contains calls to another microservices
- It demarcates transactions
- Authorization checks

Developing Service Layer


- N-tier architecture





Developing Service Layer

Service layer classes should be annotated with @Service annotation



```
@Service
@Transactional
public class PersonService {
    private PersonRepository personRepository;

    @Autowired
    public PersonService(PersonRepository personRepository) {
        this.personRepository = personRepository;
    }

    public Optional<Person> findById(Long id) {
        try {
            return personRepository.findById(id);
        } catch (HibernateException ex) {
            throw new ServiceLayerException(ex);
        }
    }
}
```

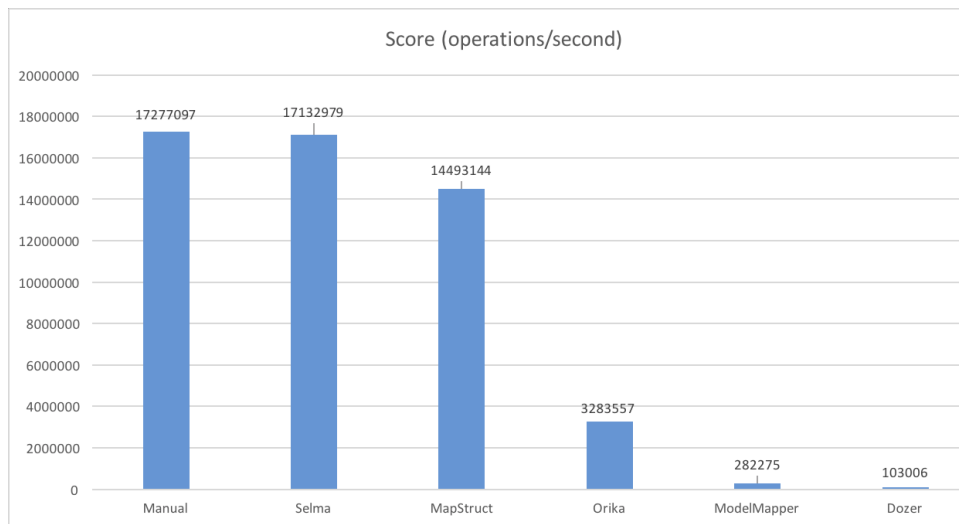


What is DTO Class? Why Do We Need It?

- A Data Transfer Object (DTO) is an object that is used to encapsulate data, and send it from one subsystem of an application to another.
- DTOs are most commonly used by the Services layer in an N-Tier application to transfer data between itself and the UI layer. The main benefit here is that it reduces the amount of data that needs to be sent across the wire in distributed applications. They also make great models in the MVC pattern
- How to map entities to DTO classes?
 - Manually? -> the best performance but too difficult to implement..
 - Libraries? Yes!
 - In the past, mappers as Dozer or ModelMapper were used, these uses reflection, it is slow, but easy to implement
 - Nowadays mappers as MapStruct or Selma are popular, these generated some-like manual implementation

Mappers Comparison

- Great article about mappers performance comparison is like always on baeldung..
- <https://www.baeldung.com/java-performance-mapping-frameworks>





Consuming Another REST Service

- RestTemplate

```
HttpEntity<String> entity = new HttpEntity<>(createHttpHeaders("user", "userPwd"));
ResponseEntity<PersonDTO> response = restTemplate
    .exchange(REST_URI + "/" + id, HttpMethod.GET, entity, PersonDTO.class);
if (response.getStatusCode().isError()) {
    // log user not found
}
return response.getBody();
```

- RestTemplate has several useful methods:

- getObject
- postForObject



Service Layer - Transactions

- Container-Managed Transactions are placed above methods to demarcate the transaction scope
- ACID
 - Atomicity - each transaction is treated as a single “unit”
 - Consistency - ensures that a transaction can only bring the database from one valid state to another
 - Isolation - ensures that concurrent execution of transactions leaves the database in the same state that would have been obtained if the transactions were executed sequentially
 - Durability - once a transaction has been committed, it will remain committed even in the case of a system failure
- Transactions usually uses pessimistic locking (database table or row is locked until the transaction finishes the execution)



Transactions - Container's Behaviour

Transactions propagation:

- **REQUIRED:**
 - The method becomes part of the caller's transaction. If the caller does not have a transaction, the method runs in its own transaction
- **REQUIRES_NEW:**
 - The method always runs in its own transaction. Any existing transaction is suspended
- **NOT_SUPPORTED:**
 - The method never runs in a transaction. Any existing transaction is suspended
- **SUPPORTS:**
 - The method becomes part of the caller's transaction if there is one. If the caller does not have a transaction, the method does not run in a transaction
- **MANDATORY:**
 - It is an error to call this method outside of a transaction
- **NEVER**
 - It is an error to call this method in a transaction



Transactions Scopes and Entity Synchronization

- Entity components must be synchronized with the underlying database at least once per transaction.
- Entity classes do not declare transactions. They inherit the transaction of the calling component.
- Entity classes can use optimistic locking. When a transaction commits entity data, the transaction can fail to commit because the data was modified by a concurrently running transaction



Own Annotations - For Read Only Operations

```
@Transactional(rollbackFor = Exception.class, readOnly = true)
@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Documented
public @interface TransactionalRO {
    @AliasFor("transactionManager")
    String value() default "";
    @AliasFor("value")
    String transactionManager() default "";
    Propagation propagation() default Propagation.REQUIRED;
    Isolation isolation() default Isolation.DEFAULT;
    int timeout() default -1;
}
```



Own Annotations - For Write Only Operations

```
@Transactional(rollbackFor = Exception.class)
@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Documented
public @interface TransactionalWO {
    @AliasFor("transactionManager")
    String value() default "";
    @AliasFor("value")
    String transactionManager() default "";
    Propagation propagation() default Propagation.REQUIRED;
    Isolation isolation() default Isolation.DEFAULT;
    int timeout() default -1;
}
```



Consuming Another SOAP Web Service

- Generating SOAP Client using WSIMPORT, this is supported by Netbeans, Eclipse, IntelliJ IDEA



Section: Spring Model View Controller (MVC)



Spring Model View Controller (MVC)

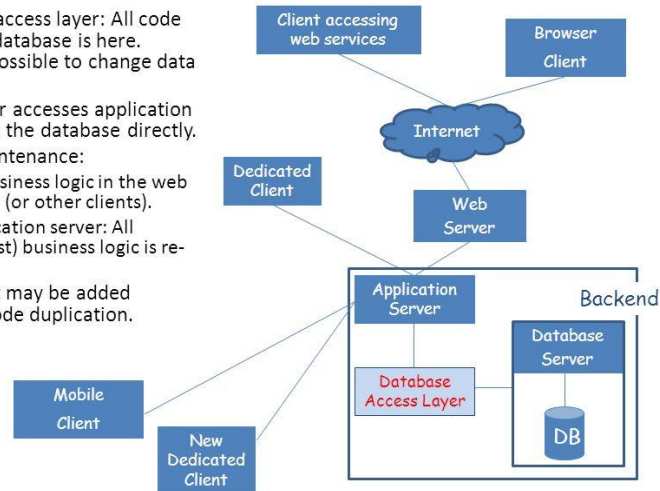
Spring Model View Controller (MVC)

- a. Servlets
 - i. Dispatcher Servlet
- b. Java Server Pages (JSP), Java Server Faces (JSF)
- c. MVC Design Pattern
- d. Common annotations:
 - i. @Controller, @ModelAttribute, @RequestMapping, @SessionAttributes
- e. Validation (javax.validation package)
- f. Embedded application servers (Tomcat, Jetty)

Layers in Multi-tier Application

N-tier (multi-tier) Architecture

- Database access layer: All code to access database is here. Makes it possible to change data store.
- Web server accesses application layer – not the database directly.
- Easier maintenance:
 - No business logic in the web server (or other clients).
 - Application server: All (almost) business logic is re-used.
- New client may be added without code duplication.





SaaS Cloud

- web applications are Software-as-a-Service type of cloud service
- provide on-demand access to software
- device independence – PC, notebook, tablet,
- smartphone, smart TV, ...
- web mail, messaging, office suites, media libraries,
- communication tools, business sw ...
- Gmail, Facebook, Google Drive, Dropbox, Spotify,
- Flickr, YouTube, WebEx, NetSuite ...



Deployment

- SaaS services can be deployed
 - Into Platform-as-a-Service (PaaS) cloud
 - Microsoft Azure, Amazon Elastic Beanstalk, IBM Cloud, RedHat OpenShift, Heroku, ...
 - into Infrastructure-as-a-Service (IaaS) cloud
 - Google Computing Engine, Amazon Elastic Compute Cloud, Microsoft Azure, ...
 - Locally
- software is provided as
 - downloadable executable code (i.e. JavaScript, Android app)
 - callable API on provider's servers (e.g. Google Calendar API)



Client Side Technologies

- HTML, forms, CSS
 - Cookies
 - JavaScript, Document Object Model, AJAX
 - HTML5 features - <canvas>, <video>, web storage, web sockets, file API, geolocation API, device orientation, media capture
 - Scalable Vector Graphics (SVG)
-
- Dead technologies: Java applets, Flash
 - Soon will be dead: JSP, JSF

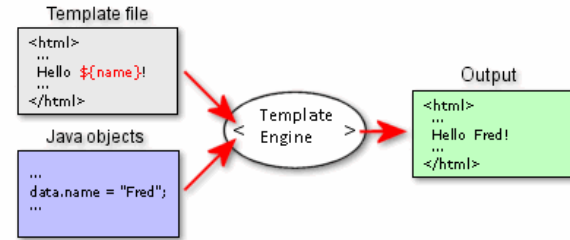


Server Side Technologies

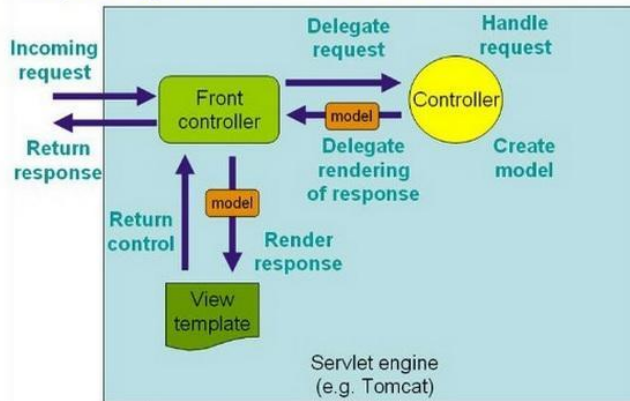
- Python, C#, PHP, Ruby on Rails, NodeJS, ...
- Java web containers (Java EE implementations) - Apache Tomcat, Jetty, JBoss, IBM WebSphere, ...)
- Servlet API for handling HTTP
- Java Server Pages (JSP) for page templates
- Frameworks on top of Servlet API

Java Server Side Frameworks

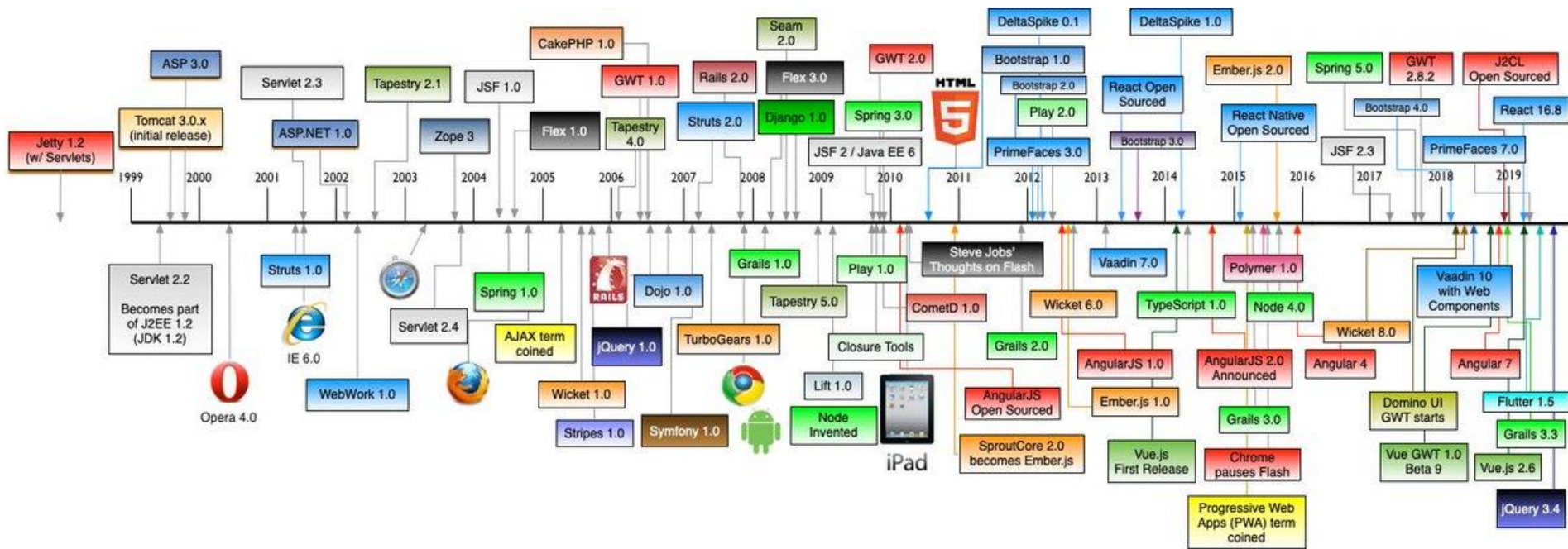
- Page templates
 - JSP, tag libraries, JSTL
 - Velocity
 - Freemaker
- Model-View-Controller
 - Spring MVC
 - Stripes
 - Apache Struts



Spring MVC Workflow



Web Frameworks History





Servlet API

- **Servlets**: for managing HTTP requests
- **Filters**: for modifying requests and responses
- **Listeners**: for handling events (e.g., app start)
- **HttpSession**: for maintaining state
- **RequestDispatcher** and **attributes** for cooperation among multiple servlets



Servlet API

```
@WebServlet("/persons/*")
public class PersonServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        request.getRequestDispatcher("/person.jsp").forward(request, response);
    }

    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.sendRedirect(request.getContextPath() + "/persons");
    }
}
```



Filter

```
@WebFilter("/persons")
public class PersonFilter implements Filter {
    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain filterChain)
        throws IOException, ServletException {
        doSomething(servletRequest, servletResponse);
        filterChain.doFilter(servletRequest, servletResponse);
    }

    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
    }

    @Override
    public void destroy() {
    }
}
```




Listeners

- ServletContextListener, ServletRequestListener, HttpSessionListener, ...

```
@WebListener
public class PersonContextListener implements ServletContextListener {

    @Override
    public void contextInitialized(ServletContextEvent sce) {
        ServletContext ctx = sce.getServletContext();
        ctx.setAttribute("common_object", getCommonObject());
    }

    @Override
    public void contextDestroyed(ServletContextEvent sce) {

    }
}
```



HttpSession

- Keeps a Map<String, Object> on server
- Assigned to a particular browser
- Maintained using cookie or URL rewriting
- Timeout after 30 minutes since last request

```
@WebServlet("/auth/*")
public class ServletAuthenticationChecker extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        HttpSession session = request.getSession(true);
        Boolean authenticated = (Boolean) session.getAttribute("authenticated");
        if (!authenticated) { response.sendError(HttpServletResponse.SC_UNAUTHORIZED); }
    }
    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException { }
}
```



WAR - Web Archive

- A servlet container can have multiple web applications running, mapped by different context path (first part of URL)
- Each web app were usually deployed in a *.war file
- It is a ZIP archive containing:
 - WEB-INF/classes/**/*.class -> classes
 - WEB-INF/lib/*.jar -> libraries
 - WEB-INF/web.xml -> deployment descriptor
 - WEB-INF/tags/*.tag -> custom JSP tags
 - Directly accessible files like *.jsp, *.png, *.css, *.js



Java Server Pages

- An HTML file with special directives, converted to a servlet on each change
- Scriptlets, EL language, tags, tag libraries

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<html>
    <%= pageContext.findAttribute("common_object") %>
    <% out.print(pageContext.findAttribute("a")); %>
    <c:out value="${a}" escapeXml="false"/>
</body>
</html>
```



JSP Expression Language (EL)

- Expressions inside of `${}`
- Value expressions:
 - `${attribute.property}`
 - `${attribute['property']}`
- Operators: `+`, `-`, `*`, `/`, `div`, `%`, `mod`, `and`, `or`, `not`, `==`, `!=`, `<`, `>`, `<=`, `>=`, `empty`
- Functions defined by tag libraries
 - `${fn:length(orderitems)}`



Own JSP Tags

- Defined in *.tag files with syntax similar to *.jsp
- Can be used for common page layout
- Attributes can be: simple, dynamic, fragment
 - a.tag example:

```
<%@ tag pageEncoding="utf-8" trimDirectiveWhitespaces="true" dynamic-attributes="attr" %>
<%@ attribute name="href" required="true" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<c:url value='${href}' var="url" scope="page"/>
<a href="
```



Common Page Layout Using *.tag

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib tagdir="/WEB-INF/tags" prefix="my"%>
<%@ taglib prefix="sec"
      uri="http://www.springframework.org/security/tags"%>
<my:pagetemplate title="Language School students">
    <jsp:attribute name="body">
        // some table showing persons
    </jsp:attribute>
</my:pagetemplate>
```



Java Standard Tag Library (JSTL)

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>

<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form"%>


<c:set var="a" value="123" scope="request"/>
<c:out value="${a}"/>
<c:if test="${a>1}"> a is bigger than 1 </c:if>
<c:forEach items="${cars}" var="car" varStatus="i"> ${i.count}: ${car.id} </c:forEach>
<c:choose>
    <c:when test="${a<0}"> a is negative </c:when>
    <c:when test="${a==0}"> a is zero</c:when>
    <c:otherwise> a is positive</c:otherwise>
</c:choose>
```




The Problems of Today's Web Design

- Wide range of screen sizes from 3" phones to 30" desktop monitors
- Wide range of pixel densities (80 ppi - 560 ppi)
- Touch screens do not have "mouse over" events
- Devices change orientation (portrait / landscape)



Responsive Web Design

- Web design that adapts to screen size and pixel density
- CSS media queries
 - @media screen and (min-width: 400px) {...}
- CSS pixels versus hardware pixels
 - CSS pixels are 96ppi at 28" distance (1px=0.26mm)
 - Hardware pixels described in CSS by device-pixel-ratio
 - device-pixel-ratio: 2 – iPhone4, iPad3
 - device-pixel-ratio: 3 - Galaxy S4, LG G3, HTC One
 - device-pixel-ratio: 4 - Galaxy Note Edge, Xiaomi Mi3
- Images should be served in HW pixel resolutions

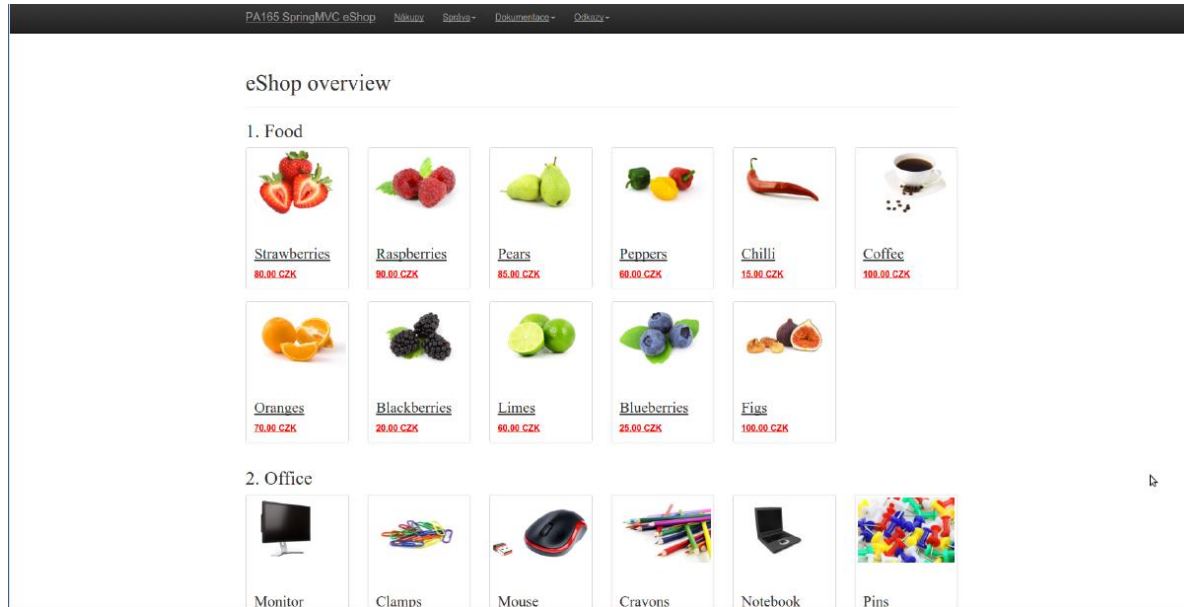


Bootstrap

- Originally created by Twitter
- It is CSS framework for responsive web design (nowadays, the de-facto standard)
- Navigation menu collapses on small screens
- 12-column grid for positioning
- 4 screen sizes: extra small, small, medium, large
- CSS classes for rows and columns

```
<div class="row">  
  <div class="col-xs-6 col-sm-8 col-md-9 col-lg-10"></div>  
  <div class="col-xs-6 col-sm-4 col-md-3 col-lg-2">  
    <div class="panel panel-default".../>  
  </div>  
</div>
```

Desktop 24" 1920x1080 90ppi



Tablet 10" 1920x1200 224ppi

PA165 SpringMVC eShop

Nákupy

Správa ▾

Dokumentace ▾

Odkazy ▾

eShop overview

1. Food



Strawberries

80.00 CZK



Raspberries

90.00 CZK



Pears

85.00 CZK



Peppers

60.00 CZK



Chilli

15.00 CZK



Coffee

100.00 CZK



Oranges

70.00 CZK



Blackberries

20.00 CZK



Limes

60.00 CZK



Blueberries

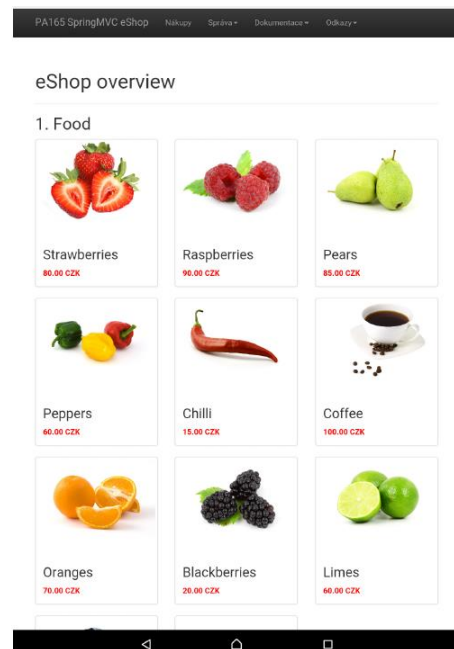
25.00 CZK



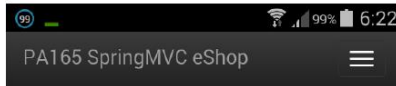
Figs

100.00 CZK

The same 10" in Portrait Mode



4.3" 540x960 256 ppi



eShop overview

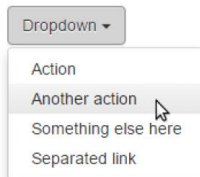
1. Food





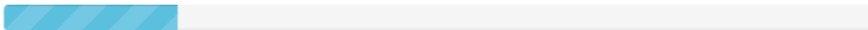
Bootstrap Additional Features

- Vector icons
- Support for screen readers
- Drop-down menus
- Buttons and button groups
- Badges
- Alerts
- Progress bars



Well done! You successfully read this important alert message.

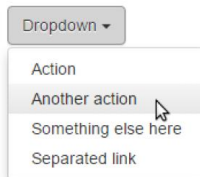
Heads up! This alert needs your attention, but it's not super important.





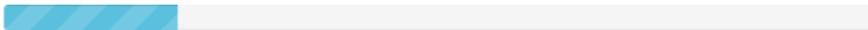
Bootstrap Additional Features

- Vector icons
- Support for screen readers
- Drop-down menus
- Buttons and button groups
- Badges
- Alerts
- Progress bars



Well done! You successfully read this important alert message.

Heads up! This alert needs your attention, but it's not super important.





Material Design

- Developed in 2014 by Google, **Material Design** is a “design language”. It is based on the “card” motifs that were first introduced in Google Now, and has expanded on that by including responsive transitions and animations, effects such as lighting and shadows and grid-based layouts
- Being a design language, Material Design defines a set of guidelines, which showcase how to best design a website. It tells you what buttons are for use and which ones you should use, how to animate or move it, as well as where and how it should be placed, etc.
- Providers:
 - Google material design: <https://material.io/design/>
 - IBM material design: <https://www.carbondesignsystem.com/>



Material Design vs Bootstrap

- **Philosophy and Purpose**

- While both of them are used for the purpose of web development, **Material Design** is more **oriented towards the way that a website or application will look**. This is evident in the way it is used, its design guidelines and “rules” and the numerous templates and components it comes with, which are focused on design
- **Bootstrap**, on the other hand, is mainly focused on easily creating responsive websites and web applications, which are functional and of high quality when it comes to user experience

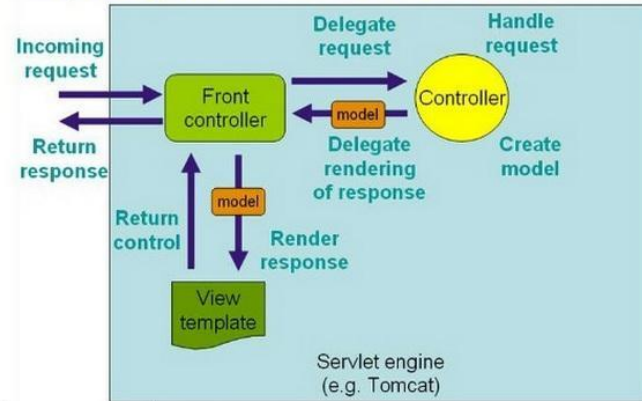
- **Design Process and Components**

- **Material Design comes with numerous components** that provide a base design, which then can be worked on by the developers themselves. It strictly follows the patterns of Material Design concept, which provides the necessary information on how to use each component
- Bootstrap is an open-source framework, also referred to as a UI library, which can use **Grunt** to create and run its designing processes

Spring MVC

- One of many Spring libraries, optional
- Model View Controller architecture
- Request-driven framework

Spring MVC Workflow





Spring MVC Initialization

- By hand:
 - Initialize a new `DispatcherServlet` instance with `WebApplicationContext`
 - Add the servlet instance to your web app
- Automatically:
 - Extend `AbstractAnnotationConfigDispatcherServletInitializer` and implements its methods `getRootConfigClasses()` and `getServletMappings()`
- In both cases, provide:
 - A class annotated with `@EnableWebMvc` that configures Spring MVC
 - A class annotated with `@Configuration` that configures Spring beans
 - Can be just a single class



Spring MVC Configuration

- A class with `@EnableWebMvc` or XML based
- Should provide:
 - `ViewResolver` for resolving views, e.g., JSPs
 - `MessageSource` for localized messages
 - `Validator` for validating data in beans
- Can enable default servlet for static files



Spring MVC Initialization - Automatically

```
public class StartInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {  
    @Override  
    protected Filter[] getServletFilters() {  
        CharacterEncodingFilter encodingFilter = new CharacterEncodingFilter();  
        encodingFilter.setEncoding("UTF-8");  
        return new Filter[] { encodingFilter };  
    }  
    @Override  
    protected Class<?>[] getRootConfigClasses() { return new Class<?>[] { SpringMVCConfig.class };}  
    @Override  
    protected String[] getServletMappings() { return new String[] { "/" };}  
    @Override  
    protected Class<?>[] getServletConfigClasses() { return null; }  
}
```



Spring MVC Configuration

```
@EnableWebMvc
@Configuration
@ComponentScan(basePackages = { "org.sedaq.mvc.controllers" })
public class SpringMVCConfig extends WebMvcConfigurerAdapter {

    ...

}
```




Spring MVC Configuration - ViewResolver

```
@Bean
public ViewResolver viewResolver() {
    logger.debug("registering JSP in /WEB-INF/jsp/ as views");
    InternalResourceViewResolver viewResolver = new InternalResourceViewResolver();
    viewResolver.setContentType("text/html; charset=UTF-8");
    viewResolver.setPrefix("/WEB-INF/jsp/");
    viewResolver.setSuffix(".jsp");
    return viewResolver;
}

@Override // Maps the main page to a specific view
public void addViewControllers(ViewControllerRegistry registry) {
    logger.debug("mapping URL / to login view");
    registry.addViewController("/").setViewName("login");
}
```



Spring MVC Configuration - MessageSource

```
@Bean
public MessageSource messageSource() {
    logger.debug("registering ResourceBundle 'Texts' for messages");
    ResourceBundleMessageSource messageSource = new ResourceBundleMessageSource();
    messageSource.setBasename(TEXTS);
    messageSource.setDefaultEncoding("UTF-8");
    return messageSource;
}
```



Spring MVC Configuration - Validator/Static Files

```
@Bean
public Validator validator() {
    logger.debug("registering JSR-303 validator");
    return new LocalValidatorFactoryBean();
}

/**
 * Enables default Tomcat servlet that serves static files.
 */
@Override
public void configureDefaultServletHandling(DefaultServletHandlerConfigurer configurer) {
    logger.debug("enabling default servlet for static files");
    configurer.enable();
}
```



Spring MVC Controllers

```
@Controller
@RequestMapping("/person")
public class PersonController {

    @Autowired
    private PersonService personService;

    @RequestMapping("/foo")
    public String foo(@RequestParam int a, Model model){
        // pass data as request attributes to views
        model.addAttribute("b", a+1);
        // ViewResolver resolves to /WEB-INF/jsp/foo.jsp
        return "foo";
    }
}
```



Spring MVC Controllers

- Any class annotated with `@Controller`
- Mapping of methods to URLs is set by `@RequestMapping`, can have common prefix for the whole class
- Dependencies are injected using `@Autowired`
- Can return `String`, which is resolved by `ViewResolver` (provided by `@EnableWebMvc`) to view, usually a JSP page
- Data are passed through instance of `Model`
- Method parameters specify inputs
- Automatic type conversion for request params and path



Spring MVC Controllers - Method Parameters

```
@RequestMapping("/foo/{a}/{r1:[a-z]+}{r2:\\d+}")
public String foo2(
    @PathVariable int a,
    @PathVariable String r1,
    @RequestParam("b") long b,
    Locale locale,
    HttpMethod httpMethod,
    @RequestHeader("User-agent") String userAgent,
    @CookieValue("mycookie") Cookie mycookie,
    Model model,
    HttpServletRequest req,
    HttpServletResponse res
) {
    return "foo";
}
```



Spring MVC Controllers - Redirect

- Triggered by return value starting with “redir:”

```
@RequestMapping("/redir")
public String someRedirect(Locale locale, RedirectAttributes redirectAttributes){
    redirectAttributes.addAttribute("pid", 21);
    redirectAttributes.addAttribute("cid", 33);
    return "redirect:/product/{pid}/category/{cid}";
}
```

- RedirectAttributes
 - Attributes replace placeholders {attr} in URL
 - @PathVariable parameters automatically added as attributes
 - Provide so called flash attributes, which exist only during the first next request
- More complex URL building possible using UriComponentsBuilder class
- Redirect after POST! To avoid duplicate submissions



Spring MVC - Order Controller

```
@Controller
@RequestMapping("/order")
public class OrderController {
    @Autowired
    private OrderFacade orderFacade;

    @RequestMapping(value = "/view/{id}", method = RequestMethod.GET)
    public String view(@PathVariable long id, Model model) {
        model.addAttribute("order", orderFacade.getOrderWithId(id));
        return "order/view";
    }

    @RequestMapping(value =("/{id}", method = RequestMethod.DELETE)
    public String removeOrder(@PathVariable long id,
        RedirectAttributes redirectAttributes) {
        orderFacade.deleteOrder(id);
        redirectAttributes.addFlashAttribute("message", "Order was deleted.");
        return "redirect:/order/list";
    }
}
```




Spring MVC - Tag Library For Forms

- Binds form fields to bean properties
- Display error messages when validation fails
- Keeps values entered by user when validation fails

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form"%>

<form:form method="post" action="/order" modelAttribute="orderCreate">

    <form:label path="name">Name</form:label>
    <form:input path="name" />
    <form:errors path="name"/>
    <button type="submit">Create</button>
</form:form>
```



Spring MVC - @Valid and BindingResults

- The form is passed to the method as Java Bean class

```
@RequestMapping(method = RequestMethod.POST)
public String createOrder(@Valid @ModelAttribute OrderCreateDto orderCreateDto,
                          BindingResult bindingResult,
                          Model model,
                          RedirectAttributes redirectAttributes) {
    if (bindingResult.hasErrors()) {
        for (FieldError fe : bindingResult.getFieldErrors()) {
            model.addAttribute(fe.getField() + "_error", true);
        }
        return "product/new";
    }
    //create order
    Long id = orderFacade.createOrder(orderCreateDto);
    redirectAttributes.addFlashAttribute("alert_success", "Order was created.");
    redirectAttributes.addAttribute("id", id);
    return "redirect:/product/view/{id}";
}
```



Spring MVC - Input Data Validation

- JSR-303 “Bean Validation” provides annotations and validators for java bean properties
- Hibernate Validator is implementation of JSR-303
- @NotNull, @Max, @Min, @Size, @Future, @Past, @Pattern, ...
- A single definition of validation reused in various layers - e.g., persistence and web forms
- You can define your own annotations and provide its validator and localized error messages
- Class with @EnableWebMvc has to provide Validator instance (in Spring Boot provided automatically :))



Spring MVC - Input Data Validation

```
public class OrderCreateDto {  
  
    @NotNull  
    @Size(min = 3, max = 50)  
    private String name;  
    @NotEmpty  
    @Size(min = 3, max = 500)  
    private String description;  
    @NotNull  
    @Min(0)  
    private BigDecimal price;  
    @Email(message = "Email must be provided.")  
    private String contactEmail;  
    @Pattern(regexp = "\\d+")  
    private Long pin;  
}
```



Spring MVC - Summary

- Spring MVC uses controllers to process HTTP requests (it is based on Java EE Servlets API)
 - Send Model to views to display
 - Or send redirects (always after POST)
- Flash attributes for passing data through redirects
- Form tag library helps in form handling
- Request parameters may be bound to properties of a method parameter with @ModelAttribute
- JSR-303 Bean Validation is commonly used for input object validation



Section: Spring REST Services



Section: Spring REST Services

Spring REST Services

- a. Rest architecture
- b. Common annotations:
 - i. `@RestController`, `@PathVariable`, `@RequestParam`, `@RequestBody`
- c. Data Transfer Object (DTO) classes
- d. Jackson
- e. Exception handling (`@RestControllerAdvice`)
- f. Swagger Documentation
- g. Testing Tools
 - i. Postman, SoapUI, Browser plugins



REST Architecture

- Representational State Transfer (REST) is architectural style for distributed hypermedia systems
 - It is architecture! Not protocol, that is the difference between REST and SOAP
 - REST was defined in 2001 in Roy Fielding's doctoral dissertation thesis
 - http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
 - The original idea behind Representational State Transfer is to mimic the behaviour of Web applications : as a net of Web pages and links, resulting in the next page(state change)
 - REST was born in the context of HTTP, but it is not limited to that protocol



REST Constraints

1. **Uniform interface:** You MUST decide APIs interface for resources inside the system which are exposed to API consumers and follow religiously
2. **Client-server:** This essentially means that client application
3. **Stateless:** Inspired by HTTP, server will not store anything about latest HTTP request cl. made
4. **Cacheable:** In REST, caching shall be applied to resources when applicable
5. **Layered system:** A client communicates only with an adjacent layer. A Service should not expose any implementation details
6. **Code on demand** (optional): Most of the time you will return resources in JSON (or in the past, XML) form, but when you need you are free to return, e.g., UI widget, or .ZIP or whatever you will need



REST Resources

- They key abstraction of information in REST is a **resource**
 - A resource identification is through URI
 - Resource consist data and metadata (metadata are often in the form of HTTP headers (eTag, etc.))
 - A RESTful web service is designed by identifying the resources
- In hypermedia-based REST, the resources are interconnected by hyperlinks



REST Design

- The key abstraction of information in REST is a **resource**
 - A resource identification is through URI
 - Resources consist of data and metadata (metadata are often in the form of HTTP headers (eTag, etc.))
 - A RESTful web service is designed by identifying the resources
- In hypermedia-based REST, the resources are interconnected by hyperlinks

https : //localhost : 8080 / sedaq-ar-rest / api/v1 / games/25/customers
protocol host port project name versioning REST resource



REST Action Resources

- Sometimes, it is required to expose an operation in the API that inherently is non RESTful
- Consider the following situation:
 - <http://localhost/my-rest/api/v1/accounts/1>
 - <http://localhost/my-rest/api/v1/accounts/2>
 - We want to transfer money between account 1 and account 2 ...
- What to do?
 - The verb we need to do is transfer, so by this verb we could identify the needed action resource:
 - <http://localhost/my-rest/api/v1/transfers>



REST Data Input

- URL path parameter: <https://localhost/sedaq-ar-rest/api/v1/games/{id}>
 - Use Path variable typically for retrieving specific record by id from a collection of resources
 - Do not use <https://localhost/sedaq-ar-rest/api/v1/games?id=1>
- Query parameters:
 - Use query parameters to filter, sort, or limit collections
 - <https://localhost/sedaq-ar-rest/api/v1/games?name=pavel&email=seda@email.cz>
- Headers:
 - Send metadata or links through HTTP headers (Content-Type, Accept, Authentication, ...)



REST Methods

Method	Collection of resources	Single
Item		
GET	Get a list of all the resources	
	Retrieve data for resource with id 1	
PUT	Update the collection with a new one	Update the
resource with id 1		
POST	Create a new member resource	
	Create a sub-resource	
DELETE	Delete the whole collection	Delete the
resource with id 1		
HEAD	Retrieve metadata information	Retrieve
data for resource with id 1		

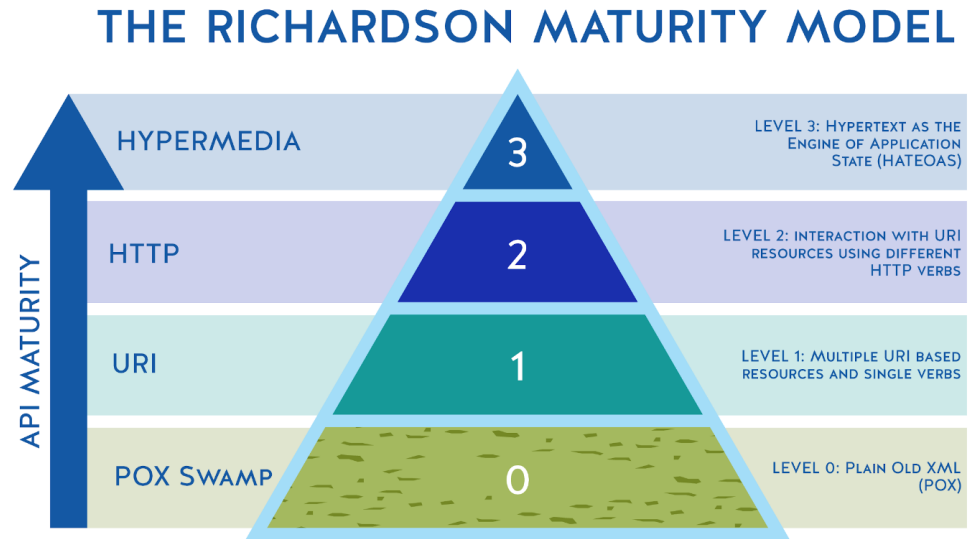


REST HATEOAS

- Hypermedia as the Engine of Application State (HATEOAS)
- With HATEOAS, a client interacts with a network application whose application servers provide information dynamically through [hypermedia](#)
- A REST client needs little to no prior knowledge about how to interact with an application or server beyond a generic understanding of hypermedia
- Based on Roy-Fieldings dissertation REST without Hypermedia is not a REST
 - But... This was not widely accepted by industry..
 - The idea is good, you need to know only the root page, e.g., <https://gopas.cz> and then navigate through hyperlinks to other pages
 - But.. It is quite difficult to implement on server and on client side
 - It is not supported (at least I know) by Swagger documentation

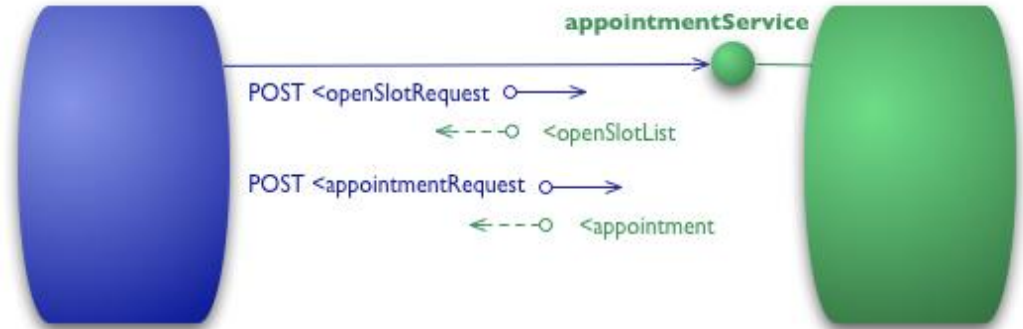
Richardson Maturity Model

- Explains different levels at which REST can be implemented
- For further, information see:
<https://martinfowler.com/articles/richardsonMaturityModel.html>



Level 0 - The SWAMP of POX*

- Looks more as Remote Procedure Call System
- We POST to and endpoint asking for different services
- There is no knowledge about resources, rather messages that are send to the endpoints (and back responses)

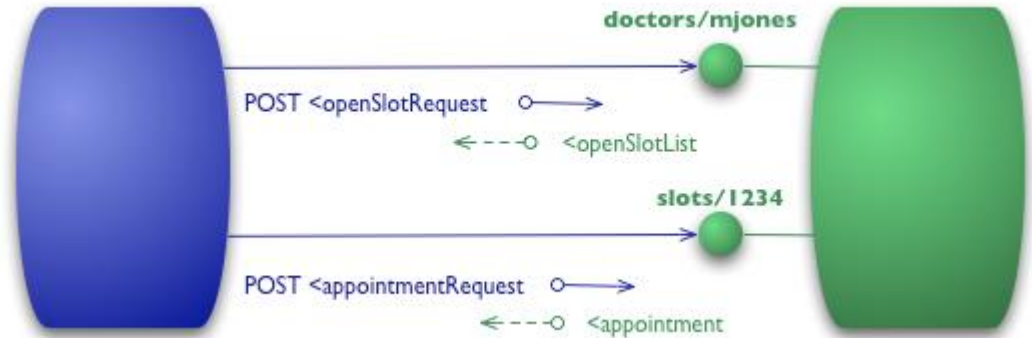


```
POST /appointmentService HTTP/1.1  
[various other headers]
```

```
<openSlotRequest date = "2010-01-04" doctor = "mjones"/>
```

Level 1 - Resources

- At this level we introduce Resources
- We contact resources, not endpoints
- Instead of passing parameters, now we contact the specific resource

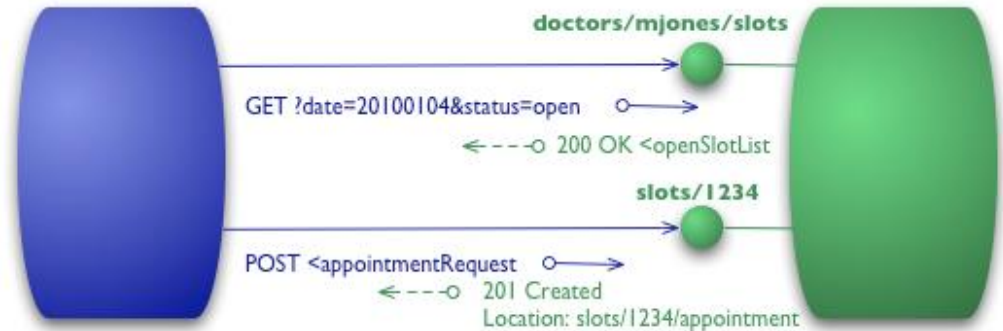


```
POST /doctors/mjones HTTP/1.1  
[various other headers]
```

```
<openSlotRequest date = "2010-01-04"/>
```

Level 2 - HTTP Verbs

- At this level we start using HTTP verbs
- We start differentiating between POST and GET
- We also start using HTTP response codes
- We start differentiating “safe” vs “unsafe” operations



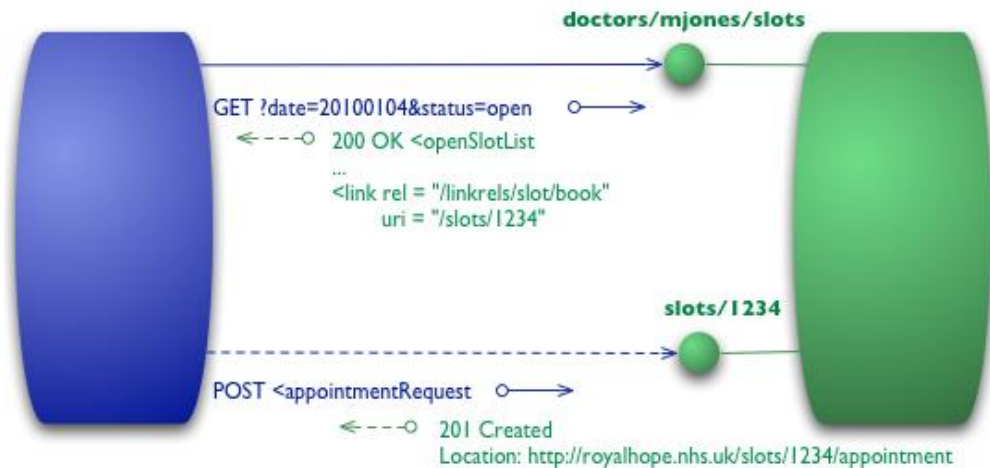
Level 3 - Hypermedia Controls

- We introduce HATEOAS

```
GET /doctors/mjones/slots?date=20100104&status=open HTTP/1.1
Host: royalhope.nhs.uk
```

```
HTTP/1.1 200 OK
[various headers]
```

```
<openSlotList>
  <slot id = "1234" doctor = "mjones" start = "1400" end = "1450">
    <link rel = "/linkrels/slot/book"
      uri = "/slots/1234"/>
  </slot>
  <slot id = "5678" doctor = "mjones" start = "1600" end = "1650">
    <link rel = "/linkrels/slot/book"
      uri = "/slots/5678"/>
  </slot>
</openSlotList>
```





Safety and Idempotence

- The term “**safe**” means that if a given method is called, the resource state on the server remains unchanged
- By specification, GET and HEAD should always be safe - clearly it is up to the developers not to violate this hidden specification
- PUT, DELETE are considered unsafe, while for POST generally depends
- The word “**idempotent**” means that, independently from how many times a given method is invoked, the end result is the same
- **GET** and **HEAD** are an example of an **idempotent** operation
- **PUT** is as well **idempotent**: If you add several times the same resource, it should be only inserted once
- **DELETE** is as well **idempotent**: issuing delete several times should yield the same result - the resource is gone (but what about DELETE /items/last ?)
- **POST** is generally **not** considered an **idempotent** operation



Safety and Idempotence

Method		Safety	Idempotence
GET		YES	YES
PUT		YES	NO
POST		NO	NO
DELETE	YES		NO
HEAD	YES	YES	



REST - versioning

- When to create version 2? api/v2? Retire version 1?
- Private internal services are easier to retire, public services less so
- No breaking changes: If possible, enhance your existing service by adding additional functionality (new URL or HTTP methods) but do not change anything that is in use by clients
- Parallel versions: Each version has a different root resource URL. /v1 and /v2/ are common
- Keep the version 1 URLs but use custom content types to switch to different behaviours

https : //localhost : 8080 / sedaq-ar-rest / api/v1 / games/25/customers
protocol host port project name versioning REST resource



REST - HTTP Status Codes

Code	Meaning	Description
200	Everything processes OK	OK
201	Resource is created	Created New
202	Accepted for processing but not yet completed	Accepted
301	Moved permanently	Requests should
400	Bad request	Bad
401	Unauthorized	



REST - Best Practices (1/3)

1. Have consistent usage of **resource names**, e.g., plural for resources -> **/users/1, orders/1**
2. **Use nouns not verbs in URIs**: Each resource is following structure as follows:

Resource	GET	POST	PUT	DELETE
~/games	Returns the list of games	Create a new game	Mass actualization of games	Delete all games
~/games/1	Return game with id 1	Unsupported method	Actualization of a specific game	Delete specific game

This structure shows that the REST should not contains resources as **~/getAllGames**



REST - Best Practices (2/3)

3. **HTTP header for format serialization:** Message format must be specified in HTTP header. **Content-Type** in HTTP header defines the request format and **Accept** defines the list of accepted response formats
4. **Allow possibility to filter, sort, selection of returned attributes and pagination:**
 - a. GET ~/games?sort=ASC
 - b. GET ~/games?type=action_games
 - c. GET ~/games?page=3
 - d. GET ~/games?fields=[id,name,title,description]
5. **API versioning:** Create API versioning in URI with the usage of ordinal numbers only! ~/api/v1
6. **Exception handling:** Without exception handling it is almost impossible to debug errors:
7. **Use sub-resources for relations:** If resource has a relation with another resource, it is suitable to use so-called sub-resource: ~/api/v1/games/34/customers (return all customers from the game with id 34)
8. **HTTP methods PUT, POST and PATCH should return resource representation**



REST - Best Practices (3/3)

9. Hyphens should be used instead of camelCase or underscore between words:

- a. CamelCase could bring problems if server does not recognize case sensitivity
- b. Underscore is bad because text searchers or editors usually underscore URIs, to provide higher visualization that this part of the text is clickable

10. Functions Create, Read, Update, DELETE (CRUD) should not be exposed in the URI:

11. Using HATEOAS:

- a. But as we discussed earlier, nowadays it is not necessary to use it

12. Additional best practices could be, e.g.,

- a. Using only lowercase letters in URI
- b. Not using file extension in the URI
- c. Not using symbol “/” as the last character in the URI and so on



REST - Documentation

- **Swagger:**
 - <https://swagger.io/>
 - Currently, the winner of REST documentation frameworks
- **Web Application Description Language (WADL):**
 - Was the REST variant of **SOAP WSDL**, not used much today
- **Apiary:**
 - <https://apiary.io/>
- **Spring REST Docs:**
 - Official Spring Project: <https://spring.io/projects/spring-restdocs>
- Based on the Fieldings words, documentation is **against** the principle of REST services (in the hypermedia context)
- In a lot of companies the biggest **PAIN** for developers:
 - REST services are not much documented, or there are a lot of mistakes in it..
 - It is wrong, take a time and document everything properly!



REST - In Spring Framework

REST - Spring REST Controller

```
@RestController
@RequestMapping(value = "/persons")
public class PersonRestController {
```

Resource base address, on class level use
`@RequestMapping`

path or value specify sub-resource path

```
@GetMapping(path =("/{id}", produces = {MediaType.APPLICATION_JSON_VALUE})
```

```
public ResponseEntity<Person> findPersonById(
```

```
    @PathVariable Long id,
```

```
    @RequestHeader HttpHeaders headers) {
```

```
    PersonDTO userResource = personFacade.findById(id);
```

```
    return new ResponseEntity<>(userResource, HttpStatus.OK);
```

```
}
```

```
}
```

Content-Type:
application/json

Use ResponseEntity objects



REST - Spring REST Controller

- DELETE:

```
@RestController
@RequestMapping(value = "/persons")
public class PersonRestController {

    @DeleteMapping(path =("/{id}")
    public void deletePersonById(@PathVariable Long id) {
        personFacade.deletePerson(id);
    }
}
```

@DeleteMapping, @GetMapping,
@PostMapping, @PutMapping
Are available since Spring 4.3
In the previous versions you have
to use @RequestMapping and
specify a HTTP method



REST - Spring REST Controller

- UPDATE:

```
@RestController
@RequestMapping(value = "/persons")

public class PersonRestController {

    @PutMapping(path =("/{id}", produces = {MediaType.APPLICATION_JSON_VALUE},
        consumes= {MediaType.APPLICATION_JSON_VALUE})
    public ResponseEntity<PersonUpdateDTO> updatePersonById(@PathVariable Long id,

        @RequestBody @Valid PersonUpdateDTO personUpdatedDTO) {

        PersonUpdateDTO personResource = personFacade.updatePerson(id, personUpdatedDTO);

        return ResponseEntity.ok(person);

    }

}
```




REST - Spring REST Controller

- CREATE:

```
@RestController
@RequestMapping(value = "/persons")

public class PersonRestController {

    @PostMapping(consumes = MediaType.APPLICATION_JSON_VALUE, produces = MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<PersonCreatedDTO> createPerson(
        @Valid @RequestBody PersonCreateDTO personCreatedDTO) {
        PersonCreateDTO person = personFacade.create(personCreatedDTO);
        return ResponseEntity.ok(person);
    }
}
```



REST - @RestController

- In the previous slides you may encounter that we use @RestController instead of @Controller from Spring-MVC
- @RestController is a stereotype annotation that combines @ResponseBody and @Controller
- More than that, it gives more meaning to your Controller and also may carry additional semantics in future releases of the framework
- In general, please use @RestController in your



REST - HTTP Response Using ResponseEntity

- **ResponseEntity:** is meant to represent the entire HTTP response
 - You can control anything that goes into it:
 - Status code
 - Headers
 - Body
 - Basically, ResponseEntity lets you do more
 - E.g., returning null or void results in a 204 (No Content) status.. But in some cases you want to declare it as HTTP 200 OK

REST - Serialization Configuration - Jackson

- The most common library for serialization in REST services is Jackson (also the default in Spring Boot)
- This configuration may include serialization of Dates, serialization of format (camelCase, snake_case, ...)

@Configuration

```
public class ObjectMapperConfiguration {
```

@Bean

```
public ObjectMapper objectMapper() {
```

```
    ObjectMapper objectMapper = new ObjectMapper();
```

```
    objectMapper.setPropertyNamingStrategy(PropertyNamingStrategy.SNAKE_CASE);
```

```
    objectMapper.registerModule(new JavaTimeModule());
```

```
    objectMapper.disable(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS);
```

```
    objectMapper.enable(SerializationFeature.INDENT_OUTPUT);
```

```
    return objectMapper;
```

```
}
```

```
}
```

Java POJO with camelCase letters
will be serialized into snake_case and
vice versa

Dates will be serialized
into one field

Will be serialized with
indented output



REST - Exception Handling

- Any unhandled exception will cause an HTTP 500 response (INTERNAL SERVER ERROR)
- To avoid returning HTTP 500 response for all exceptions you could annotate exceptions with `@ResponseStatus` to return appropriate HTTP error code and message

```
@ResponseStatus(value = HttpStatus.NOT_FOUND, reason = "The requested resource was not found")
```

```
public class ResourceNotFoundException extends RuntimeException {...}
```

- Methods annotated with `@ExceptionHandler` are handling exceptions
- It is not necessary to add `@ResponseStatus` to the Exceptions, it gives you more freedom in returning a custom error data structure

```
@ExceptionHandler @ResponseStatus(value = HttpStatus.NOT_FOUND, reason = "The requested resource was not found")
```

```
ApiErrorTraining handleResourceNotFound(ResourceNotFoundException ex) {  
    ApiErrorTraining apiError = new ApiErrorTraining();  
    apiError.setErrors(Arrays.asList("The requested resource was not found"));  
    return apiError;}  
}
```



REST - Global Exception Handler

- Another way is to have a global advice using `@RestControllerAdvice` that will manage exceptions for all the controllers
- Personally, I like that way

```
@RestControllerAdvice
```

```
public class CustomRestExceptionHandlerTraining extends ResponseEntityExceptionHandler {  
    ...  
    @ExceptionHandler({Exception.class})  
    public ResponseEntity<Object> handleAll(final Exception ex, final WebRequest request, HttpServletRequest req) {  
        final ApiErrorTraining apiError = new ApiErrorTraining  
            .ApiErrorBuilder(HttpStatus.INTERNAL_SERVER_ERROR,  
                ex.getLocalizedMessage()).setError("error").setPath(URLHELPER.getRequestUri(req)).build();  
        return new ResponseEntity<>(apiError, new HttpHeaders(), apiError.getStatus());  
    }  
}
```



REST - Swagger Documentation

- Moreover, every change in the API should be simultaneously described in the reference documentatie. Accomplishing this manually is a tedious exercise, so automation of the process was inevitable -> Swagger
- Swagger is annotation-based documentation
- Add Maven dependency:

```
<dependency>  
  <groupId>io.springfox</groupId>  
  <artifactId>springfox-swagger2</artifactId>  
  <version>${swagger.version}</version>  
</dependency>
```



REST - Swagger Spring Configuration

`@Configuration`

`@EnableSwagger2`



Enabled through this annotation

```
public class SwaggerConfig {
```

```
    @Bean
```

```
    public Docket api() {
```

```
        return new Docket(DocumentationType.SWAGGER_2)
```

```
            .groupName("public-api")
```

```
            .apiInfo(apiInfo()).useDefaultResponseMessages(false)
```

```
            .select()
```

```
            .apis(RequestHandlerSelectors.any())
```

```
            .paths(PathSelectors.any())
```

```
            .build();
```

```
    }
```




REST - Verification That It Works

- To verify that Springfox is working, you can visit the following URL in your browser

<http://localhost:8080/sedaq-rest/api/v2/api-docs>

- The result is a JSON response with a large number of key-value pairs, which is not very human-readable
- Fortunately, Swagger provides **Swagger UI** for this purpose



REST - Swagger UI


- Swagger UI is a built-in solution which makes user interaction with the Swagger-generated API documentation much easier
- To use Swagger UI, one additional Maven dependency is required:

```
<dependency>  
  <groupId>io.springfox</groupId>  
  <artifactId>springfox-swagger-ui</artifactId>  
  <version>${swagger.version}</version>  
</dependency>
```

- Now you can test it in your browser by visiting *<http://localhost:8080/your-app-root/swagger-ui.html>*



REST - Swagger UI

 **swagger**

Select a spec: public-api

REST API documentation

[Base URL: localhost:8080/sedaq-rest-training/api/v1]
<http://localhost:8080/sedaq-rest-training/api/v1/v2/api-docs?group=public-api>

Developed By Pavel Seda
[Terms of service](#)

basic-error-controller Basic Error Controller >

meeting-rest-controller Meeting Rest Controller >

operation-handler Operation Handler >

person-rest-controller Person Rest Controller >

web-mvc-links-handler Web Mvc Links Handler >

Models >



REST - Swagger UI

GET

/persons/{id} Get Person by Id.

Cancel

Parameters

Name	Description
Person Id <small>integer(\$int64)</small> <small>(path)</small>	id <div>Person Id - id</div>
fields <small>string</small> <small>(query)</small>	Fields which should be returned in REST API response <div>fields - Fields which should be returned in RE</div>

Execute

Responses

Response content type application/json

Code	Description
200	<div>OK</div> <div>Example Value Model</div> <div><pre>{ "age": 98, "birthday": "2018-11-21", "email": "pavel.seda@gmail.cz", "first_name": "Pavel", "id_person": 1, "nickname": "SedaQ", "surname": "Seda" }</pre></div>
404	<div>The requested resource was not found.</div>



REST - Configuration of Controllers for Swagger

- At class level use @Api annotation

```
@Api(value = "/persons", consumes = MediaType.APPLICATION_JSON_VALUE,  
      produces = MediaType.APPLICATION_JSON_VALUE)  
  
@RestController  
  
@RequestMapping(value = "/persons")  
  
public class PersonRestController {  
  
    ...  
  
}
```



REST - Configuration of Controllers for Swagger

- At method level use @ApiOperation, @ApiResponses, @ApiParam annotations

```
@ApiOperation(httpMethod = "GET", value = "Get Person by Id.",
    response = PersonDTO.class, nickname = "findPersonById",
    produces = MediaType.APPLICATION_JSON_VALUE)

@ApiResponses(value = {@ApiResponse(code = 404, message = "The requested resource was not found.")})

@GetMapping(path =("/{id}", produces = {MediaType.APPLICATION_JSON_VALUE})

public ResponseEntity<Object> findPersonById(

    @ApiParam(name = "Person Id") @PathVariable Long id) {

    ...

}
```



REST - Configuration of DTOs for Swagger

- At DTO classes use `@ApiModel` at class level and `@ApiModelProperty` at fields level

```
@ApiModel(value = "PersonDTO", description = "Information about person.")

public class PersonDTO {

    @ApiModelProperty(value = "Person ID.", example = "1")

    private Long idPerson;

    @ApiModelProperty(value = "Person email.", example = "pavelseda@email.cz")

    private String email;

}
```



REST - Swagger in PDF?

- Could be configured in Maven plugins using:
 - swagger-maven-plugin
 - swagger2markup-maven-plugin
 - asciidoctor-maven-plugin
- For more information see:
 - <https://github.com/SedaQ/rest-training/blob/master/rest/pom.xml>

1. Overview

API Reference Description.

1.1. Version information

Version : v1

1.2. Contact information

Contact : Pavel Seda

Contact Email : pavelseda@email.cz

1.3. URI scheme

Host : localhost:8080

BasePath : /rest-training/api/v1

Schemes : HTTP, HTTPS

1.4. Tags

- meetings
- persons

2. Paths

2.1. Get All Meetings.

GET /meetings

2.1.1. Parameters

Type	Name	Description	Schema	Default
Query	fields <i>optional</i>	Fields which should be returned in REST API response	string	
Body	body <i>optional</i>	Parameters for QueryDSL filtering	< string, < string > array > map	
Body	body <i>optional</i>		< string, < string > array > map	



REST - Testing Tools

- Postman
 - The most common for testing REST services
 - User friendly but contains less features as SoapUI
- SoapUI
 - The most common for testing SOAP services
 - Contains a lot of features, but less user friendly as Postman
- Browser plugins
 - **Chrome:** Restlet Client <https://chrome.google.com/webstore/detail/restlet-client-rest-api-t/aejoelaoggembcahagimdiliamlcdmfm>
 - **Firefox:** Advanced Rest Client <https://chrome.google.com/webstore/detail/advanced-rest-client/hgmloofddfdnphfgcellkdfbfbjeloo>

REST - Testing Tools - Postman

The screenshot displays the Postman REST client interface. At the top, the request method is set to **GET** and the URL is `http://localhost:8080/sedaq-res-...`. The environment is set to **No Environment**. The request URL is `http://localhost:8080/sedaq-rest-training/api/v1/persons`. The request body is empty. The **Headers** tab is selected, showing one header: **Content-Type** with the value `application/json`. The **Body** tab is also visible, showing the response status **200 OK**, time **713 ms**, and size **3.09 KB**. The response body is displayed in the **Body** tab, showing a JSON array of two person objects. The response is formatted as **JSON**.

GET `http://localhost:8080/sedaq-res-...` **No Environment**

`http://localhost:8080/sedaq-rest-training/api/v1/persons`

GET `http://localhost:8080/sedaq-rest-training/api/v1/persons` **Send** **Save**

Params Authorization **Headers (8)** Body Pre-request Script Tests **Cookies Code Comments (0)**

▼ Headers (1)

KEY	VALUE	DESCRIPTION	...	Bulk Edit	Presets ▼
<input checked="" type="checkbox"/> Content-Type	application/json				
Key	Value	Description			

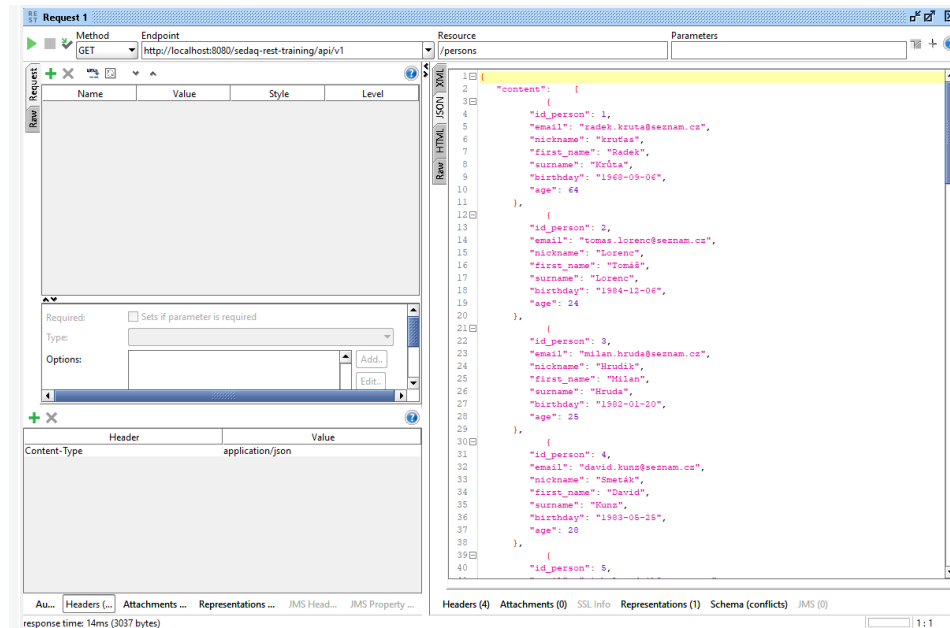
► Temporary Headers (7) ⓘ

Body **Cookies Headers (3) Test Results** Status: **200 OK** Time: **713 ms** Size: **3.09 KB** **Download**

Pretty Raw Preview **JSON**

```
1 {
2   "content": [
3     {
4       "id_person": 1,
5       "email": "radek.kruta@seznam.cz",
6       "nickname": "krutas",
7       "first_name": "Radek",
8       "surname": "Krůta",
9       "birthday": "1968-09-06",
10      "age": 64
11    },
12    {
13      "id_person": 2,
14      "email": "tomas.lorenc@seznam.cz",
15      "nickname": "Lorenc",
16      "first_name": "Tomáš",
17      "surname": "Lorenc",
18      "birthday": "1984-12-06",
```

REST - Testing Tools - SoapUI





Section: Aspect Oriented Programming

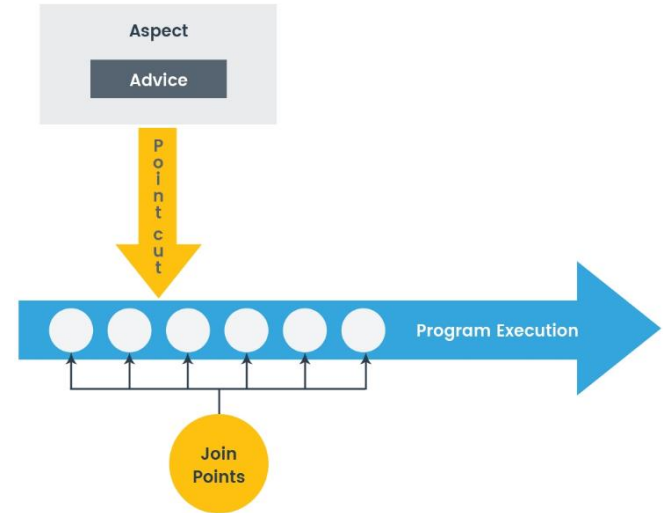


Aspect Oriented Programming

- Aspect Oriented Programming (AOP) is a programming paradigm that aims to increase modularity by allowing the separation of cross-cutting concerns.
- It does so by adding additional behaviour to existing code without modification of the code itself
- Instead, we can declare this new code and these new behaviours separately
- Usage:
 - Transaction management
 - Logging or auditing
 - Exception handling or translation
 - Validation
 - Authorization

Aspect Oriented Programming - terminology

- Business Object:
 - Class that has some business logic
- Aspect
 - An aspect is modularization of a concern that cuts across multiple classes
- Jointpoint
 - A joinpoint is a point during the execution of a program, such as execution of a method or the handling of an exception
- Pointcut
 - A pointcut is a predicate that helps match an Advice to be applied by an Aspect at a particular joinpoint
- Advice
 - An advice is an action taken by an aspect at particular joinpoint






Aspect Oriented Programming - Dependencies

- To enable AOP in Spring Boot just add the following dependency:

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

- This will add the following library to your project:

▶  Maven: org.aspectj:aspectjweaver:1.9.4



Aspect Oriented Programming - Pointcut

- Pointcuts are usually set at one class to avoid duplicating or bad AOP design
- In Pointcut annotation you could use the following Pointcut designators (PCD):
 - execution (the primary Spring PCD)
 - “target(org.sedaq.service.UserService)” says that it will be applied to all classes that implements UserRepository
 - @annotation(“org.sedaq.annotations.TrackTime”)
 - ...
- Basically, it is possible to specify pointcuts to every part of code you could think about

```
import org.aspectj.lang.annotation.Pointcut;
```

```
public class CommonPointcutsConfig {
```

```
@Pointcut("execution(* org.sedaq.persistence.repository.*(..))")
```

```
public void dataLayerExecutionLoggingDebug() {  
}
```

```
@Pointcut("target(org.sedaq.service.UserService)")
```

```
public void serviceLayerExecutionLoggingInfo() {  
}
```




Aspect Oriented Programming - Advice

- Advice is an action taken by an aspect at a particular join point
- The main purpose of aspects is to support cross-cutting concerns, such as logging, profiling, caching, and transaction management.
- Different types of advice include “around,” “before” and “after” advice
 - `@Before` - It is executed before the join point. It does not prevent the continued execution of the method it advises unless an exception is thrown
 - `@After` - It is executed after a matched method's execution, whether or not an exception was thrown (similar to finally block in some ways), additional types of advice includes `@AfterThrowing` or `@AfterReturning`
 - `@Around` - Surrounds a join point such as a method invocation -> the most powerful kind of advice



Aspect Oriented Programming - Advice (The Ex.)

- The example using AOP for logging each method in debug mode on a specific API layer

`@Aspect @Component`

```
public class LoggingAspect {  
    @Before("org.sedaq.aop.CommonPointcutConfig.dataLayerExecutionLoggingDebug()")  
    public void dataLayerExecutionLoggingDebug(JoinPoint joinPoint) {  
        logJoinPoint(joinPoint);  
    }  
    private void logJoinPoint(JoinPoint joinPoint, Instant time) {  
        StringBuilder builder = printMethodWithParameters(joinPoint);  
        LoggerFactory.getLogger(joinPoint.getSignature().getDeclaringTypeName()).debug(builder.toString());  
    }  
}
```

- Annotation class with `@Aspect` and `@Component` (to be scanned by Spring Container)



Aspect Oriented Programming - Advice (The Ex.)

- Why is @Around so powerful?

```
@Around("@annotation(org.sedaq.annotations.aop.TrackTime)")
```

```
public Object trackTimeAround(ProceedingJoinPoint joinPoint) throws Throwable {  
    Instant startTime = Instant.now();  
    logJoinPoint(joinPoint, startTime);  
    Object objectToReturn = joinPoint.proceed();  
    logJoinPointEnd(joinPoint, Instant.now());  
    return objectToReturn;  
}
```

=> because we could invoke code
before and after joinpoint as we want

- This example shows using proprietary @annotation class TrackTime



Aspect Oriented Programming - Custom Annot.

- It is the same as creation of any other custom annotation..
- This is later applied on some business logic method and specified in a Pointcut Config class

```
@Target(ElementType.METHOD)  
@Retention(RetentionPolicy.RUNTIME)  
@Documented  
@Inherited  
public @interface TrackTime {  
  
}
```



Aspect Oriented Programming - Conclusion

- Do not be afraid of using AOP
- It is one of the key components of the Spring Framework
- It is especially useful in large projects for defining common logic for, e.g., logging or transaction management
- Warning: Pay special attention when implementing Aspects on logging to handle exceptions properly (in some cases you could reach the state that you log same exceptions several times)



Section: Spring Schedulers, Async Tasks, Caching



Spring Schedulers, Async Tasks, Caching

- Spring Schedulers, Async Tasks, Caching
 - a. Scheduling using cron
 - b. Enabling async tasks and common mistakes
 - c. Caching
 - i. Caching problematics
 - ii. Cache evict
 - iii. Special databases for caching



Schedulers

- Using schedulers is essential part of each larger project
- Spring @Scheduled is based on java.util.concurrent package from Java SE
- java.util.concurrent.ScheduledExecutorService

```
ScheduledExecutorService scheduledExecutorService = Executors.newScheduledThreadPool(5);  
scheduledExecutorService.scheduleAtFixedRate(() -> System.out.println("Test "), 0, 5, TimeUnit.SECONDS);
```

- Explore Java SE 8 concurrent package, implement few basic schedulers using Java SE concurrent package
 - a. Sometimes it could be used when frameworks do not provide sufficient solution



Schedulers - Enable Scheduling in Spring

- It is quite easy..

```
import org.springframework.context.annotation.Configuration;
import org.springframework.scheduling.annotation.EnableScheduling;

@Configuration
@EnableScheduling
public class SchedulersConfig {

    // some code

}
```



Schedulers - The Rules For @Scheduled Methods

- Methods with @Scheduled annotation have to follow the following rules:
 - A method should have void return type
 - A method should not accept any parameters



Schedulers - fixedDelay

- In this case, the time between the end of the execution and start of the next execution is fixed by given delay
- The task always wait until the previous one is finished

```
@Scheduled(fixedDelay = 1000)
public void removeInactiveUsers() {
    // remove users
}
```



Schedulers - fixedRate

- This is used when each task execution is independent
- But.. pay special attention that tasks do not run in parallel by default..
 - a. Even if we use fixedRate, the next task would not be invoked until the previous one is done

```
@Scheduled(fixedRate = 1000)
public void removeInactiveUsers() {
    // remove users
}
```



Schedulers - fixedRate - Parallel Execution

- If we need parallel execution of tasks, we need to @EnableAsync and add @Async annotation over @Scheduled method
- Now, this task will be invoked each second even if the previous one did not finished

```
@Configuration
@EnableScheduling
@EnableAsync
public class SchedulersConfig {

    @Async
    @Scheduled(fixedDelay = 1000)
    public void removeInactiveUsers() throws InterruptedException {
        // remove users
    }
}
```



Schedulers - `fixedRate` vs `fixedDelay`

- The **`fixedDelay`** property makes sure that there is a delay of `n` millisecond between the finish time of an execution of a task and the start time of the next execution of the task
 - a. This property is specifically useful when we need to make sure that only one instance of the task runs all the time. For dependent jobs, it is quite helpful
- The **`fixedRate`** property runs the scheduled task at every `n` millisecond. It doesn't check for any previous executions of the task
 - a. This is useful when all executions of the task are independent. If we don't expect to exceed the size of the memory and the thread pool, *fixedRate* should be quite handy



Schedulers - Cron Expressions

- In some cases, delays and rates are not sufficient and we need to invoke some method, e.g., at midnight
- For that cases, we could use cron expression

```
/**  
 * Valid formats for cron expressions - https://stackoverflow.com/a/45126855  
 * "0 0 0 * * *" - means every day at midnight  
 **/  
@Scheduled(cron = "0 0 0 * * *", zone = "UTC")  
public void removeExpiredGroups() {  
    groupRepository.deleteExpiredIDMGroups();  
}
```



Schedulers - Parametrizing The Schedule

- Usually, the hardcoded strings in most cases are not preferred
- For those cases, `@Scheduled` annotations provide these options:

```
@Scheduled(fixedDelayString = "${remove.inactive.users.fixed.delay.in.milliseconds}")  
public void removeInactiveUsers(){ //remove users}
```

```
@Scheduled(fixedRateString = "${remove.inactive.users.fixed.rate.in.milliseconds}")  
public void removeInactiveUsers(){// remove users}
```

```
@Scheduled(cron = "${remove.inactive.users.cron}")  
public void removeInactiveUsers(){// remove users}
```




Schedulers - Cron Expressions Formats

- These are valid formats for cron expressions:
 - a. `0 0 * * * *` = the top of every hour of every day.
 - b. `*/10 * * * * *` = every ten seconds.
 - c. `0 0 8-10 * * *` = 8, 9 and 10 o'clock of every day.
 - d. `0 0 6,19 * * *` = 6:00 AM and 7:00 PM every day.
 - e. `0 0/30 8-10 * * *` = 8:00, 8:30, 9:00, 9:30, 10:00 and 10:30 every day.
 - f. `0 0 9-17 * * MON-FRI` = on the hour nine-to-five weekdays
 - g. `0 0 0 25 12 ?` = every Christmas Day at midnight
- The pattern is: `second, minute, hour, day, month, weekday`



Async Tasks

- Asynchronous execution support in Spring -> @Async annotation
- Simply put annotation above method of a bean with @Async It will make it execute in a separate thread, i.e., the caller will not wait for the completion of the called method
- Enable @Async support:

```
@Configuration
@EnableAsync
public class EnableAsyncConfig {...}
```

- **annotation** – by default, @EnableAsync detects Spring's @Async annotation
- **mode** – indicates the type of *advice* that should be used – JDK proxy-based or AspectJ weaving
- **proxyTargetClass** – indicates the type of *proxy* that should be used – CGLIB or JDK; this attribute has effect only if the **mode** is set to *AdviceMode.PROXY*
- **order** – sets the order in which *AsyncAnnotationBeanPostProcessor* should be applied; by default, it runs last, just so that it can take into account all existing proxies



Async Tasks - Methods

- @Async has two limitations, it must be applied to public methods only
 - The reason is simple, the method needs to be public so that it can be proxied
- Self-invocation - calling the async method from within the same class - won't work
 - This is based on the fact of proxy objects because it would not bypasses the proxy

```
@Async
public void asyncWithVoidReturnType() {
    System.out.println(
        "Execute method asynchronously. "
        + Thread.currentThread().getName());
}
```

```
@Async
public Future<String> asyncMethodWithReturnType() {
    System.out.println("Execute method asynchronously - "
        + Thread.currentThread().getName());
    try {
        Thread.sleep(5000);
        return new AsyncResult<String>("hello world !!!!");
    } catch (InterruptedException e) {
    }
    return null;
}
```



Async Tasks - The Executor

- By default, Spring uses a SimpleAsyncTaskExecutor to actually run these methods asynchronously
- The defaults can be overridden at two levels – at the application level or at the individual method level.

```
@Configuration
@EnableAsync
public class EnableAsyncConfig {
    @Bean(name = "myThreadPoolTaskExecutor")
    public Executor threadPoolTaskExecutor() {
        return new ThreadPoolTaskExecutor();
    }
}
```

The usage →

```
@Async("myThreadPoolTaskExecutor")
public void asyncMethodWithVoidReturnType() {
    System.out.println("Execute method asynchronously. "
        + Thread.currentThread().getName());
}
```

← Configuration



Async Tasks - The Executor - Application Level

- If we need to configure Async tasks at application level that means for all @Async methods
- It could be done using overriding getAsyncExecutor method in AsyncConfigurer interface

```
@Config
@EnableAsync
public class SpringAsyncConfigClassLevel implements AsyncConfigurer {
    @Override
    public Executor getAsyncExecutor() {
        return new ThreadPoolTaskExecutor();
    }
}
```



Async Tasks - Common Usage

- Common usage for @Async method is when calling another microservices which takes some time to return values
- In that scenario, the common use-case is the following:
 - 1. Client calls the server
 - 2. The server returns HTTP 202 Accepted with the ID of the task in the queue, where the results will be accessible later
 - Here, the server is calling with @Async method another microservices



Async Tasks - Security Context

- When calling @Async method, the security context will be lost.
- The original thread is null (it no longer lives due to its life-cycle in application server)
- The thread in @Async method is still alive but losses security context
- To share security context between these threads it is necessary to set the following bean:

```
@Bean
public MethodInvokingFactoryBean methodInvokingFactoryBean() {
    MethodInvokingFactoryBean methodInvokingFactoryBean = new MethodInvokingFactoryBean();
    methodInvokingFactoryBean.setTargetClass(SecurityContextHolder.class);
    methodInvokingFactoryBean.setTargetMethod("setStrategyName");
    methodInvokingFactoryBean.setArguments((Object[]) new String[]{SecurityContextHolder.MODE_INHERITABLETHREADLOCAL});
    return methodInvokingFactoryBean;
}
```



Spring Cache

- Caching is simply the process of storing data in a cache
- A cache is a temporary storage area
- The result is stored in that cache and in the next call it is returned from the cache
- In Spring boot add the following dependency:

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




Spring Cache - Configuration

- When using Spring Boot, the mere presence of the starter package on the classpath alongside the `EnableCaching` annotation would register the same `ConcurrentMapCacheManager`

```
@Component
public class CacheConfiguration implements CacheManagerCustomizer<ConcurrentMapCacheManager> {
    @Override
    public void customize(ConcurrentMapCacheManager cacheManager) {
        cacheManager.setCacheNames(List.of("persons"));
    }
}
```



Spring Cache - Use Catching With Annotations

- The simplest way to enable caching behaviour for a method is to demarcate it with `@Cacheable` and parameterize it with the name of the cache where the results would be stored

```
@Cacheable(value = "persons", key = "#id")
public Optional<Person> findById(Long id) {
    return personRepository.findById(id);
}
```

- The `findById` method will first check the cache persons before actually invoking the method and then caching the result



Spring Cache - CacheEvict

- Why we would not make all the methods @Cacheable?
 - The problem is size.. We do not want to populate the cache with values that we do not need often
 - Caches can grow quite large, quite fast and we could be holding on to a lot of unused data
 - Data will be stored in its cache until the deletion of that cache
 - => Possibility to create method like this which would be called periodically through @Scheduled to save reload the actual content from DB

```
@Scheduled(cron = "0 0 0 * * *", zone = "UTC")  
@CacheEvict(value = "persons", allEntries = true)  
public void reloadPersons() {...}
```

- Also, it is necessary to set @CacheEvict on **save** plus **delete** operations



Spring Cache - CachePut

- @CachePut on save/update operation:

```
@CachePut(value = "persons", key = "#person.idPerson", unless = "#result==null")
public Person createPerson(Person person) {
    return personRepository.save(person);
}
```

- @CacheEvict on delete operation:

```
@CacheEvict(value = "persons", key="#person.idPerson")
public void deletePerson(Person person) {
    personRepository.delete(person);
}
```



EHCache Support in Spring

- EHCache is one of the most used databases for caching: <https://db-engines.com/en/ranking>

Add maven dependency in Spring boot:

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-context-support</artifactId>
  <version>5.1.3.RELEASE</version>
  <scope>compile</scope>
</dependency>
<dependency> <!-- EHCache -->
  <groupId>net.sf.ehcache</groupId>
  <artifactId>ehcache</artifactId>
  <version>2.10.2.21</version>
</dependency>
```



EHCache Support in Spring

- Ye, simply as that you could move from ConcurrentHashMap to EHCache

```
@EnableCaching
@Configuration
public class EHCacheConfig {
    @Bean
    public CacheManager cacheManager() { //A EhCache based Cache manager
        return new EhCacheCacheManager(ehCacheCacheManager().getObject());
    }
    @Bean
    public EhCacheManagerFactoryBean ehCacheCacheManager() {
        EhCacheManagerFactoryBean factory = new EhCacheManagerFactoryBean();
        factory.setConfigLocation(new ClassPathResource("ehcache.xml"));
        factory.setShared(true);
        return factory;
    }
}
```

- For ehcache.xml example config look at: <http://websystique.com/spring/spring-4-cacheable-cacheput-cacheevict-caching-cacheconfig-enablecaching-tutorial/>



Section: Spring Tests



Spring Tests

- a. Database for testing
- b. Unit tests
- c. Mockito
- d. Integration Tests



Spring - Maven Dependencies

- To run tests in Spring Boot it is necessary to add spring-boot-starter-test dependency
- And also, it is suitable to add some database which has in-memory mode as H2
- Why tests are run against DB in in-memory mode as H2? And not against, e.g., PostgreSQL?

```
<!-- db for testing -->
```

```
<dependency>  
  <groupId>com.h2database</groupId>  
  <artifactId>h2</artifactId>  
  <version>${h2.version}</version>  
  <scope>test</scope>  
</dependency>
```

```
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-test</artifactId>  
  <scope>test</scope>  
</dependency>
```



Spring - H2 Configuration

- Place under `src/main/test/resources` application.properties file

For description of each field check: <https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html>

```
spring.datasource.url=jdbc:h2:mem:testdb;DB_CLOSE_DELAY=-1;DB_CLOSE_ON_EXIT=FALSE
spring.datasource.username=sa
spring.datasource.password=
spring.datasource.driverClassName=org.h2.Driver
spring.jpa.hibernate.ddl-auto=create-drop
spring.h2.console.enabled=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.H2Dialect
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.properties.hibernate.show_sql=false
```



Spring - Persistence Configuration

- Create special config class for tests configuration

```
@Configuration
@ComponentScan(basePackages = {"com.sedaq.training.persistence.model",
                                "com.sedaq.training.persistence.repository"})
@EntityScan(basePackages = "com.sedaq.training.persistence.model")
@EnableJpaRepositories(basePackages = "com.sedaq.training.persistence.repository")
public class PersistenceConfigTest {
}
```



Spring - Test Configuration (Persistence Layer)

- Run the tests with Spring JUnit support:

```
@RunWith(SpringRunner.class) @DataJpaTest
@Import(PersistenceConfigTest.class)
public class PersonRepositoryTest {

    @Autowired private TestEntityManager entityManager;

    @Autowired private PersonRepository personRepository;

    @SpringBootApplication static class TestConfiguration{}

    private Person person;

    @Before

    public void setUp() {

        person = new Person();

        person.setEmail("pavelseda@email.cz"); ...

    }
```

- Writing individual tests:

```
@Test

public void findPersonByIdTest() {

    Long id = (Long)
entityManager.persistAndGetId(person);

    Optional<Person> optionalPerson =
personRepository.findById(id);

    assertTrue(optionalPerson.isPresent());

    assertEquals(person, optionalPerson.get());

}
```



Spring - Test Configuration (Service Layer)

- Configure service layer:

```
@RunWith(SpringRunner.class)
public class PersonServiceTest {

    @Mock
    private PersonRepository personRepository;

    private PersonService personService;

    private Person person;

    @Before
    public void init() {

        MockitoAnnotations.initMocks(this);

        personService = new PersonService(personRepository);

        person = new Person(); // additional settings
    }
```

- Writing simple test using Mocks:

```
@Test
public void getPersonById() {
    given(personRepository.findById(any(Long.class)))
        .willReturn(Optional.of(person));

    Optional<Person> personResponse =
        personService.findById(1L);

    assertEquals(personResponse.get(), person);
}
```



Spring - Test Configuration (Rest Layer)

- Configure REST layer Config class:

```
@Configuration
@ComponentScan(basePackages = {"com.sedaq.training"})
@EntityScan(basePackages = "com.sedaq.training.persistence.model")
@EnableJpaRepositories(basePackages = "com.sedaq.training.persistence.repository")
public class RestConfigTest {
    ...
    @Bean
    public HttpServletRequest httpServletRequest() {
        return new HttpServletRequestWrapper(new Request(new Connector()));
    }
}
```



Spring - Test Configuration (REST Layer)

- Configure REST layer:

```
@RunWith(SpringRunner.class)
public class PersonRestControllerTest {
    private PersonRestController personRestController;
    @Mock private PersonFacade personFacade;
    @Mock private ObjectMapper objectMapper;
    private MockMvc mockMvc;
    private PersonDTO personDto;
    @Before public void init() {
        MockitoAnnotations.initMocks(this);
        personRestController = new PersonRestController(personFacade, objectMapper);
        this.mockMvc = MockMvcBuilders.standaloneSetup(personRestController)
            .setCustomArgumentResolvers(new PageableHandlerMethodArgumentResolver(),
                new QuerydslPredicateArgumentResolver(new
QuerydslBindingsFactory(SimpleEntityPathResolver.INSTANCE, Optional.empty())))
            .setMessageConverters(new MappingJackson2HttpMessageConverter()).build();
        ObjectMapper obj = new ObjectMapper();
        obj.setPropertyNamingStrategy(PropertyNamingStrategy.SNAKE_CASE);
        given(objectMapper.getSerializationConfig()).willReturn(obj.getSerializationConfig());
        PersonDTO personDTO = new PersonDTO(); //settings for personDTO
    }
}
```



Spring - Test Configuration (REST Layer)

- Creating simple REST layer test:

```
@Test
public void findPersonById() throws Exception {
    given(personFacade.findById(any(Long.class))).willReturn(personDto);
    String personValue = convertObjectToJsonBytes(personDto);
    given(objectMapper.writeValueAsString(any(Object.class))).willReturn(personValue);
    MockHttpServletResponse result = mockMvc.perform(get("/persons" + "/" + id, 11))
        .andExpect(status().isOk())
        .andExpect(content().contentTypeCompatibleWith(MediaType.APPLICATION_JSON))
        .andReturn()
        .getResponse();
    assertEquals(convertObjectToJsonBytes(convertObjectToJsonBytes(personDto)),
        result.getContentAsString());
}
```




Spring - Test Configuration (Integration Tests)

- Integration testing is the phase in software testing in which individual software modules are combined and tested as a group
 - In simple words, you will not mock anything..
 - It is suitable to write integration tests for most critical services
 - It takes some time to configure and write it..



Section: Spring Security



Spring Security

- a. Configuration of Spring Security
- b. Securing web requests
- c. Storing password in Database
- d. OAuth2 + OpenID Connect



Spring Security

- Authentication
 - Is the process of determining whether someone or something is, in fact, who or what it declares itself to be
 - Verified by, e.g., HTTP Basic form (username, password combination)
- Authorization
 - Determining if authenticated entity has permitted access to a protected resource or system
- Confidentiality
 - Is a set of rules or a promise that limits access or places restrictions on certain types of information
- Data integrity
 - Received data has not been modified, destroyed or lost

Solved by
SSL





Login Type

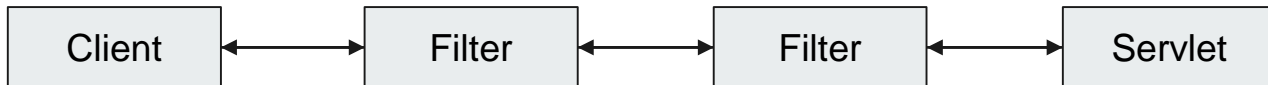
- We must determine what type of authentication we will be using
- There exists 4 main types of authentication:
 - **BASIC:** Transmits data unencrypted
 - **FORM:** Similar to BASIC but allows user to add input data (username, password) to the formular
 - **DIGEST:** Sends data encrypted
 - **CLIENT-CERT:** Authentication through client certificate. It is quite safe type of authentication, but the client must have a certificate on his device

Spring Boot Security

- Add the following maven dependency:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
</dependency>
```

- Spring Security in the web tier (for UIs and HTTP back ends) is based on Servlet Filters, so it is helpful to look at the role of Filters generally first (<https://spring.io/guides/topicals/spring-security-architecture>)





Spring Boot Security

- The client sends a request to the app, and the container decides which filters and which servlet apply to it based on the path of the request URI
- At most one servlet can handle a single request, but filters form a chain, so they are ordered, and in fact a filter can veto the rest of the chain if it wants to handle the request itself
- A filter can also modify the request and/or the response used in the downstream filters and servlet



Spring Boot - @EnableWebSecurity

- To configure Spring Security create config class with @EnableWebSecurity annotation

```
@Configuration
@ComponentScan("com.sedaq.training.security")
@EnableWebSecurity

public class RestSecurityConfiguration extends WebSecurityConfigurerAdapter {

    @Override

    protected void configure(AuthenticationManagerBuilder auth) throws Exception {

        auth.authenticationProvider(authProvider);

    }

    ...

}
```




Spring Boot - Specify Authentication Rules

- Configure `HttpSecurity` for certain URLs or Resources

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    Http
        .authorizeRequests()
        .antMatchers("/v2/api-docs", "/swagger-resources", "/swagger-ui.html", "/webjars/**")
        .permitAll()
        .and()
        .authorizeRequests()
        .antMatchers("/**")
        .authenticated()
        .and()
        .httpBasic();}
```



Spring - Authorization Checks

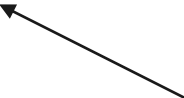
- Spring provides several annotations to check if a user has given roles:
 - `@Secured`, e.g., `@Secured("ROLE_USER")`
 - The `@Secured` annotation doesn't support Spring Expression Language (SpEL)
 - `@RolesAllowed`, e.g., `@RolesAllowed("ROLE_USER")`
 - The `@RoleAllowed` annotation is the JSR-250's equivalent annotation of the `@Secured` annotation
 - Similar to `@Secured`, does not support SpEL
 - `@PreAuthorize`, e.g., `@PreAuthorize("hasRole('ROLE_VIEWER')")`
 - Supports SpEL, the strongest variant, I prefer this one



Spring - Authorization Checks - Own Annotation

- It is common to create own stereotypes for security checks, e.g., for user it would look like:

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@PreAuthorize("hasRole('USER')")
@Documented
@Inherited
public @interface IsUser {
}
```



Defines that it must be
authenticated user with role
USER



Enforce HTTPs

- We must determine what type of authentication we will be using
- There exists 4 main types of authentication:
 - **BASIC:** Transmits data unencrypted
 - **FORM:** Similar to BASIC but allows user to add input data (username, password) to the formular
 - **DIGEST:** Sends data encrypted
 - **CLIENT-CERT:** Authentication through client certificate. It is quite safe type of authentication, but the client must have a certificate on his device



Create Self Signed Certificate

- **Certification authority (CA)** - is a trusted entity that manages and issues security certificates and public keys that are used for secure communication in a public network.
- **Self-signed Certificates** - certificates not issued by known CA but rather by the server hosting the certificate are called self-signed. These are often used in internal development environments that are not customer facing. The root certificates for these will be absent in the browser's certificate store.
- **KeyStore** - a keystore is a file which stores private key entries, certificates with public keys or just secret keys that we may use for various cryptographic purposes. It stores each by an alias for ease of lookup. Keystore is mostly owned by server-side and presents its corresponding public key and certificate to the client. Correspondingly, if the client also needs to authenticate itself – a situation called mutual authentication – then the client also has a keystore and also presents its public key and certificate.
- **TrustStore** - it is a file that contains the root certificates for Certificate Authorities (CA) that issue certificates. Basically, it is a file with a list of certificates which client trust. If a client talks to a Java-based server over HTTPS, the server will look up the associated key from its keystore and present the public key and certificate to the client. The client then looks up the associated certificate in its truststore. If the certificate or Certificate Authorities presented by the external server is not in its truststore, it'll get an `SSLHandshakeException` and the connection won't be set up successfully.



During the SSL handshake

During the SSL handshake:

1. A client tries to access https://
2. And thus, server-side responds by providing an SSL certificate (which is stored in its keystore)
3. Now, the client receives the SSL certificate and verifies it via truststore (i.e the client's truststore already has a predefined set of certificates which it trusts.). It's like: Can I trust this server? Is this the same server whom I am trying to talk to? No middle man attacks?
4. Once, the client verifies that it is talking to a server which it trusts, then SSL communication can happen over a shared secret key.

Note: I am not talking here anything about client authentication on the server side. If a server wants to do a client authentication too, then the server also maintains a trustStore to verify the client.



Java KeyStore (JKS)

- A JKS is an encrypted security file used to store a set of cryptographic keys or certificates in the binary format, and it requires a password to be opened. It is a usual way to hold certificates for use in Java code. Extension of the files is *.jks or *.pkcs12.
- What Are the Tools Used to Manipulate KeyStores?
 - For JKS, we can use the Java keytool utility, which comes inbuilt with the JDK, but we can also use the openssl utility.



Generating JKS

- To generate JKS in PKCS12 type, we use keytool utility:

```
keytool -genkeypair -alias {alias name} -keyalg RSA -keysize 2048 -storetype PKCS12 -keystore {filename of keystore}.p12 -dname "CN=domainname,OU=JavaDivision, O=SedaQ, L=Brno, ST=Czech Republic, C=CZ" -validity 3650 -ext SAN=DNS:localhost,IP:127.0.0.1
```

E.g.,

```
keytool -genkeypair -alias sedaq-keystore -keyalg RSA -keysize 2048 -storetype PKCS12 -keystore sedaq-keystore.p12 -dname "CN=domainname,OU=JavaDivision, O=SedaQ, L=Brno, ST=Czech Republic, C=CZ" -validity 3650 -ext SAN=DNS:localhost,IP:127.0.0.1
```




Generating JKS

- It creates *.p12 file from which then we create *.crt file using:

```
keytool -export -keystore {filename of keystore}.p12 -alias {alias name of crt} -file {filename of certificate}.crt
```

e.g.,

```
keytool -export -keystore sedaq-keystore.p12 -alias sedaq-keystore -file sedaq-keystore.crt
```



Generating JKS

- Basically, we extract the certificate from the keystore file. To import our certificate to CA which is also called truststore, use command:

```
sudo keytool -importcert -trustcacerts -file {path to certificate} -alias {alias of certificate} -  
keystore {path to trust store}
```

e.g.,

```
sudo keytool -importcert -trustcacerts -file /home/seda/SSL/sedaq-keystore.crt -alias sedaq-  
keystore -keystore /usr/lib/jvm/jdk-11/lib/security/cacerts
```



Generating JKS

- For Java applications, default truststore is file cacerts which is bundled with the JDK/JRE and is located in \$JAVA_HOME/lib/security/cacerts. After that we are able to check if our certificate is present in truststore file using:

`keytool -list -v -keystore {path to truststore} -alias {alias name of certificate}`

- For keytool options visit: <https://docs.oracle.com/javase/7/docs/technotes/tools/windows/keytool.html>



Spring Boot Configuration File With SSL

HTTPS and CA

security.require-ssl={ssl requirement}

server.ssl.key-store-type={the format used for the KeyStore}

server.ssl.key-store={path to KeyStore}

server.ssl.key-store-password={password used when generate KeyStore}

server.ssl.key-alias={alias of KeyStore}

server.ssl.trust-store={path to TrustStore} ## path to TrustStore, e.g., default for Java app is in JDK \$JAVA_HOME/lib/security/cacerts

server.ssl.trust-store-password={password to TrustStore}

server.ssl.trust-store-type={the format used for the TrustStore}

the requirement for ssl, e.g., true

the format used for the KeyStore, e.g. PKCS12

path to KeyStore, e.g., /etc/ssl/sedq-keystore.p12

password used when generate KeyStore , e.g., changeit

alias of KeyStore, e.g., sedq-keystore

password to TrustStore, e.g., default for cacerts is changeit

the format used for the TrustStore, e.g. JKS



Spring Boot Configuration of RestTemplate

```
@Value("${server.ssl.trust-store}")
private String trustStore;
@Value("${server.ssl.trust-store-password}")
private String trustStorePassword;

@Bean
public RestTemplate restTemplate() throws Exception{
    SSLContext sslContext = new SSLContextBuilder()
        .loadTrustMaterial(new File(trustStore), trustStorePassword.toCharArray())
        .setProtocol("TLSv1.2")
        .build();

    SSLConnectionSocketFactory socketFactory = new SSLConnectionSocketFactory(sslContext);
    HttpClient httpClient = HttpClients.custom()
        .setSSLSocketFactory(socketFactory)
        .build();

    HttpComponentsClientHttpRequestFactory factory = new HttpComponentsClientHttpRequestFactory(httpClient);
    RestTemplate restTemplate = new RestTemplate(factory);
    restTemplate.getMessageConverters().add(0, new StringHttpMessageConverter(Charset.forName("UTF-8")));
    return restTemplate;
}
```



OAuth2 + OpenID Connect

- OpenID Connect (OIDC) is extension of OAuth2 protocol (authorization protocol) with authentication and API for retrieving information about user
- OAuth 2 is an authorization protocol owned by the user resources on the resource server, empowers a foreign application, to treat resources on his behalf
- OIDC is quite similar with SAML2, but:
 - User is selecting which personal data application could access
 - Applications are not restricted to web
 - Not necessary to exchange metadata between IdP and SP
- https://dior.ics.muni.cz/~makub/oidc/OpenID_Connect_Martin_Kuba.pdf

OAuth2

- Defined in RFC 6749 in 2012
- Used by companies as Google, Facebook, Microsoft, Twitter, LinkedIn, GitHub, ...
- Is designed for secure delegation approach, but was used from the outset for federated login





OAuth2 - Terminology

- **Resource owner** - user
- **Resource Server** - server managing user data, allows certain operations over them, right for certain operations is called scopes
- **Client** - application, which wants to access to certain operations with user data (read, write, delete, ...)
- **Authorization server** - server which authenticated use, ask which scopes they want to allow for certain client, provides **access token**



OAuth2 - The Example

- **Resource owner** - me
- **Resource Server** - Google YouTube API: <https://developers.google.com/youtube/v3/>
- **Scopes**
 - Read and write - <https://www.googleapis.com/auth/youtube>
 - Only read - <https://www.googleapis.com/auth/youtube.readonly>
- **Client** - application “Business YouTube” for Android app from SedaQ Company
- **Authorization server** - <https://accounts.google.com>



Section: Spring Services Deployment



Spring Services Deployment

- a. Spring Boot Actuator
- b. Eureka
- c. Spring Boot Admin Server
- d. Deploying Jar vs War files
- e. ELK stack for logging



Deployment (Debian-based Systems) - systemd

```
[Unit]
Description=SedaQ Training service

[Service]
User=seda
ExecStart=/usr/bin/java -
Dpath.to.config.file=/home/seda/workspace/properties/training/training-project.properties -
jar /home/seda/workspace/sedaq-training/sedaq-rest-training/target/sedaq-rest-training-
1.0.0.jar
StandardOutput=syslog
StandardError=syslog
SyslogIdentifier=sedaq-training
Restart=no

[Install]
WantedBy=multi-user.target
```



Deployment (Debian-based Systems) - systemd

- Managing systemd services:
 - `$ sudo systemctl start sedaq-rest-training.service`
 - `$ sudo systemctl stop sedaq-rest-training.service`
 - `$ sudo systemctl restart sedaq-rest-training.service`
 - `$ sudo systemctl status sedaq-rest-training.service`