
# Spring Boot Framework Track for HZIV

ABIS Training & Consulting
www.abis.be
training@abis.be



Document number: 1618_03a.fm
11 December 2025

Address comments concerning the contents of this publication to:
ABIS Training & Consulting, Diestsevest 32 / 4b, B-3000 Leuven, Belgium
Tel.: (+32)-16-245610

# TABLE OF CONTENTS

# PREFACE

This course text is used as a practical workbook for mastering the Spring framework, covering aspects on application architecture, front-end and back-end development. This includes the creation of APIs and the integration with a database using JPA.

A good practical experience with Java SE is a must, to make full profit of this mastering course.

More information can be found in:

- **The other Abis Courses**
- https://docs.spring.io/spring-framework/reference/index.html
- https://docs.spring.io/spring-boot/index.html
- https://www.thymeleaf.org/doc/tutorials/3.1/usingthymeleaf.html
- https://www.baeldung.com
- https://chatgpt.com

# Introduction to Spring

**Objectives :**

- **Goals of the Spring Framework**
- **The Spring Framework Landscape**
- **Spring and/vs Spring Boot**
- **New in Spring Boot 3**

# Some history



- Rod Johnson (and Juergen Hoeller)
  book: Expert One-on-One J2EE Design and Development in October 2002
- reaction against complexity of EJBs at that time
- originally called Interface21 Framework
  Yann Caroff: “Spring” - association with nature
  + fresh start after winter
- first released under the Apache 2.0 license in June 2003
- versions

  1.0: March 2004

  2.0: October 2006; 2.5: November 2007

  3.0: December 2009; 3.1, 3.2: December 2012

  4.0: December 2013; 4.3: June 2016

  5.0: September 2017; 5.3.32: February 2024

  6.0: November 2022; 6.2.12: October 2025

  https://github.com/spring-projects/spring-framework/releases

# Design goals



- **MISSION: simplifying Java development of enterprise applications from front to back**
- **FOUR KEY STRATEGIES:**
  - ***Lightweight and minimally invasive development with POJOs***

    **Spring avoids littering application code with its API**
  - ***Loose coupling through dependency injection (inversion of control) and interface orientation***

    **Code becomes easier to understand, to test and to reuse. Whack-a-mole bug behaviour is avoided**
  - ***Declarative programming through aspects***

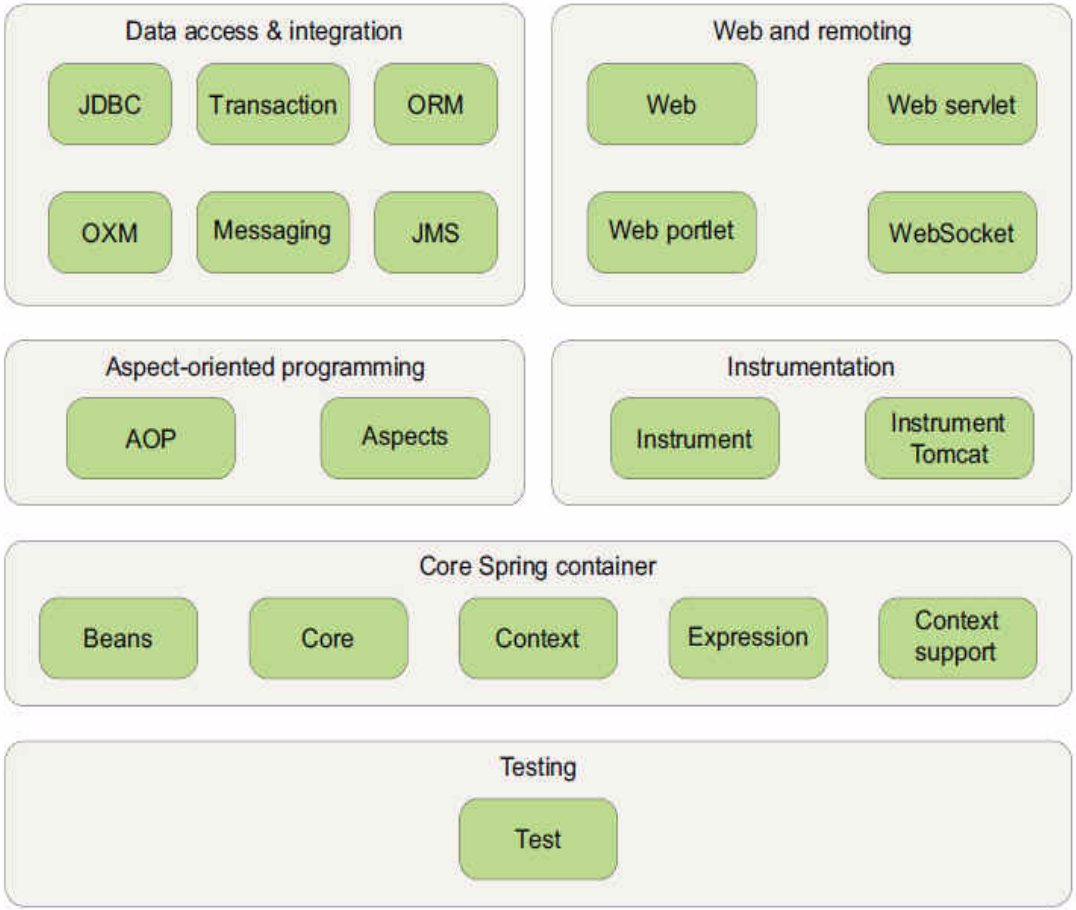
    **Separate out cross-cutting concerns like logging, transaction management and security**
  - ***Boilerplate reduction through aspects and templates***

    **Spring encapsulates commonly repeated code (DRY principle), so you can focus on the real business logic**

# The Spring Framework Landscape



- **core framework is composed of several distinct modules (20 Jar files), grouped in 6 categories of functionality**




- **free to choose the modules that suit your application**

# Spring Core Modules

- ***Core Spring Container***
  **create, configure and manage beans**
- ***Spring's AOP module***
  **decoupling application-wide concerns (transactions, security,...)**
- ***Data Access and Integration***
  **abstract away boilerplate code, provide hooks into popular ORM frameworks, JMS support, object-to-XML mapping**
- ***Web and Remoting***
  **MVC framework (servlet-based and portlet-based), support for RMI, Hessian and Burlap, JAX-WS, building REST APIs**
- ***Testing***
  **mock implementations for unit tests, support for integration testing**

# The Spring portfolio

- build upon the core framework and upon each other
- covers almost every facet of Java development
- more info can be found on: http://spring.io/projects
- *Spring Boot*
  - use automatic configuration techniques that can eliminate most (and in many cases, all) Spring configuration
  - helps to reduce the size of your Spring project build files
- *Spring Web Services*
  - offers contract-first model for building Web Services
- *Spring HATEOAS*
  - simplify creating REST applications that follow HATEOAS rules
- *Spring REST docs*
- *Spring Security*
  - declarative security mechanism based on AOP

# The Spring portfolio (cont.)

- *Spring Web Flow*
  - building conversational, flow-based web applications
- *Spring Integration*
  - interaction with other enterprise applications
- *Spring Batch*
  - to perform bulk operations on data
- *Spring Data*
  - more simplification of JPA
  - connecting to NoSQL databases
- *Spring Cloud*
  - provides tools for developers to quickly build some of the common patterns in distributed systems; useful for building and deploying microservices
- *Spring LDAP*
  - template-based approach to simplify applications using LDAP

## Spring and/vs Spring Boot



- Spring Boot = extension of the Spring framework which eliminates the boilerplate configurations required for setting up a Spring application
- based on convention-over-configuration principle
- advantages
  - easy to understand and develop spring applications
  - increases productivity
  - reduces development time
- no more XML configuration needed
- Spring Boot versions
  - 1.0 released in 2013 (Spring 4.0)
  - 2.0 released in 2018 (Spring 5.0)
  - 3.0 released in november 2022 (Spring 6.0, needs jdk17)
  - latest version: 3.5.7 (October 2025)

  https://github.com/spring-projects/spring-boot/releases

## Spring and/vs Spring Boot (cont.)

- **four main features:**
  - **Spring Boot Starters**
    aggregate common groupings of dependencies
  - **Auto Configuration**
    automatically configures beans needed by your application
  - **Command-line Interface (CLI)**
    simplifies even further, based on groovy
  - **Spring Boot Actuator**
    adds management features
- **integrated in STS/IntelliJ (Spring Initializr)**
- **running as Spring Boot App will bootstrap the embedded Tomcat server, deploy your app and map the URLs**

# Spring Starter Dependencies



- starters aggregate commonly needed dependencies
- based on the idea of transitive dependencies
- takes care of version compatibility
- more than 30 starters available
- for Spring Core, but also the portfolio projects
- examples
  - spring-boot-starter-web
  - spring-boot-starter-aop
  - spring-boot-starter-jdbc
  - spring-boot-starter-data-jpa
  - spring-boot-starter-ws
  - spring-boot-starter-rest (integrated in spring-boot-starter-web)

# Auto Configuration



- Spring Boot starters trigger auto configuration
- while adding dependencies on classpath (pom.xml), Spring makes assumptions about what configuration you will need
- cuts down the amount of configuration
- example:

  when adding spring-boot-starter-web, ViewResolver, resource handlers and message converters automatically added
- watch out for “side-effects”!

# New in Spring Boot 3 and Spring 6



- Java 17+

  records, sealed classes and interfaces, text blocks, switch expressions, pattern matching, ...

- Jakarta EE9

  package javax.* changed to jakarta.*

- reduced startup times, minimizing memory footprint
- support for RFC7807 (Problem Details Standard)
- Spring Security
  - simplified configuration with lambdas
  - use SecurityFilterChain instead of WebSecurityConfigurerAdapter
  - built-in support for OAuth 2.0
- Spring WebFlux: added PartEvent, better exception handling
- better testing support
- improved observability with Micrometer and Micrometer Tracing

# Getting Started with Spring Boot

**Objectives :**

- **Setting up a Spring Boot application**
- **Hello world**

# Philosophy



- **design:**
  - **Lasagna / ravioli code**
  - **Loose coupling**
  - **Coding to interfaces**
  - **Dependency Injection - Inversion of control (IoC)**

    objects do not create other objects on which they rely to do their work. Instead, they get the objects that they need from an outside source
- **DRY: avoid boilerplate code**

  **-> templates, configuration taken out of the code, aop**
- **easy to test**

  **-> due to design + extra features on top of JUnit**

# Ingredients



- Java Beans
  - reusable software components
  - classes conforming to a particular convention:
    default constructor, private variables with getters and setters, serializable
- Services - Repository/DA0
  - Service Layer decoupled from model
  - Repository: CRUD methods
- Configuration
  - XML (deprecated)
  - annotations
  - Java based
  - application.properties
- Container
  - ApplicationContext

# Tools



- IDE
  - IntelliJ
  - Eclipse + STS plugin
  - Spring Tool Suite (STS)
- Maven
  - build automation and software comprehension tool
  - uses Project Object Model (POM) to describe the software project being built, its dependencies on other external modules and components, and the build order
  - dynamically downloads Java libraries from one or more repositories https://central.sonatype.com
- Server
  - only application server needed
  - embedded Tomcat Server provided

# Hello World Example

## Recipe

- create a new Spring Initializr project/module
- make sure that the Spring Boot version in the pom.xml file is the correct one and add the needed dependencies
- Spring will add SpringHelloApplication and SpringHelloApplicationTests classes under the “base” package of the application
- add your own model and service classes under subpackages of the “base” package, only there Spring Boot will look for “registered” classes
- the Service layer should have an Interface + Implementation class(es), the class is “registered” with Spring via the @Component annotation
- use the SpringHelloApplicationTests class to write your Junit 5 test cases. “Wire” in the service INTERFACE in the test class (via @Autowired), and use that in your test. Spring Boot will “automatically” find the correct implementation class to use

# Maven Configuration - pom.xml



```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
          http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>3.5.7</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>

    <modelVersion>4.0.0</modelVersion>
    <groupId>be.abis</groupId>
    <artifactId>hellospring</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>
    <name>hellospring</name>

    <properties>
        <java.version>21</java.version>
    </properties>

    <!-- continued on next page-->
```

# Maven Configuration - pom.xml (cont.)

```
    <!-- continuation -->
    <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>
</project>
```

# Application Class



```
package be.abis.hellospring;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class HellospringApplication {

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

}
```

**This class will "start up" the application**
**+ will "register", instantiate and "wire" all necessary classes which it will look for in subpackages of the package where this class is in.**

# Model



```
package be.abis.hellospring.model;

public class Person {

    private String firstName;

    public Person(String firstName) {
        this.firstName  = firstName;
    }

    public String getFirstName() {
        return firstName;
    }

    public void getFirstName(String firstName) {
        this.firstName = firstName;
    }
}
```

# Service Layer



## Interface

```
package be.abis.hellospring.service;
import be.abis.hellospring.model.Person;
public interface HelloService {
    Person findPerson(int id);
    void sayHelloTo(Person person);
}
```

## Implementation

```
package be.abis.hellospring.service;
import be.abis.hellospring.model.Person;

@Component
public class AbisHelloService implements HelloService {
    public Person findPerson(int id) {
        return new Person("John");
    }
    public void sayHelloTo(Person person) {
        System.out.println("Welcome to Spring, " + person.getFirstName());
    }
}
```

# Test



```
import be.abis.hellospring.model.Person;
import be.abis.hellospring.service.HelloService;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import static org.junit.jupiter.api.Assertions.assertEquals

@SpringBootTest
public class HellospringApplicationTests {

    @Autowired
    private HelloService helloService;

    private Person person;

    @Test
    void contextLoads(){}     //will check whether all beans are registered correctly

    @BeforeEach
    void init(){ person  = new Person("John");}

    @Test
    void PersonNameIsJohn(){
        assertEquals("John", helloService.findPerson(1).getFirstName());
    }

    @Test
    void sayHelloToJohn(){
        helloService.sayHelloTo(person);
    }
}
```

**Getting Started with Spring Boot**

1. Philosophy
2. Ingredients
3. Tools
4. Hello World Example

# Maven

**Objectives :**

- **Core concepts**
- **Building a Java project**
- **Properties and resource filtering**
- **Building enterprise projects**
- **Support in IDE tools**

## Maven: build and dependency management tool for Java based application development

maven.apache.org

## Software project management framework/tool

- builds
- configuration management
- versioning
- project reports
- documentation
- ...

## Core concepts (cont.)

- Project Object Model POM
  - information about the project
  - configuration details (goals, dependencies, plugins, ...)
  - used to build (artifacts for) the project
- artifact
  - group id, version id, type
  - jar, war, ear, pom, ...
  - stored in repository
- repository
  - local vs remote
  - standardised directory structure and naming
- plugin
  - executed by Maven goals during build
- lifecycle

# POM

## Project Object Model

- information about the project and configuration details used by Maven to build the project
- project dependencies -> cornerstone of Maven !
- the plugins or goals that can be executed
- the build profiles
- (POM can inherit from another POM)
- ...

## Defined in pom.xml

## When executing goal, Maven looks for POM in the current directory. It reads POM, gets the needed configuration information, then executes the goal.

cf.: goal ~ ANT target or Gradle task

# POM example

```
<?xml version="1.0" encoding="utf-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                        http://maven.apache.org/maven-v4_0_0.xsd">

    <modelVersion>4.0.0</modelVersion>
    <groupId>be.abis.mvn</groupId>
    <artifactId>my-app</artifactId>
    <packaging>pom</packaging>
    <version>1.0-SNAPSHOT</version>
    <name>My first Maven POM</name>
</project>
```

# POM elements

- project: top-level element
- modelVersion: version of the maven object model
- groupId: unique identifier of the organization or group that created the project.

  based on the fully qualified domain name of your organization
- artifactId: unique base name of the primary artifact (typically JAR) being generated by this project

  secondary artifacts, like source bundles, also use the artifactId as part of their final name

  <artifactId>-<version>.<extension> (`myapp-1.0.jar`)
- version: of the generated artifact
  - SNAPSHOT indicates project in development

    <version>1.0-SNAPSHOT</version>
  - nothing indicates (stable) released project

    <version>1.0</version>
- name: display name used for the project.

# POM example (cont.)

- **properties: placeholders accessible anywhere within a POM**

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <java.version>21</java.version>
</properties>
```

- **dependencies: to other projects**

  **Maven downloads (JAR) to local repository, and links the (transitive) dependencies on compilation, as well as on other goals that require them**

```
<dependencies>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter</artifactId>
        <version>5.12.2</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

# Dependency scope

## determines when/where the dependency is available

- in the current build
- as a transitive dependency

## possible values

- compile
- runtime
- test
- provided
- system
- import

# More information about the dependencies

- use the maven-dependency-plugin

  mvn dependency:resolve

  shows a list of all artifacts that have been resolved

  mvn dependency:tree

  shows a tree of (transitive) dependencies

- via site generated report

  mvn site

# POM example (cont.)

- **build: declare the project's directory structure and manage plugins**

```
<build>
  <pluginManagement>
    <plugins>
      <plugin>
        <artifactId>maven-clean-plugin</artifactId>
        <version>3.1.0</version>
      </plugin>
    ........
    </plugins>
  </pluginManagement>
</build>

</project>
```

# Artifact



## Every artifact has

- group id
- artifact id
- version (1.0-SNAPSHOT, 2.1.0, ...)
- artifact type (pom, jar, war, ear, ...)
- (optionally) artifact classifier

## Artifacts are stored in a repository

# Repository



- standard directory layout

  <group id>/<artifact id>/<version>

  be/abis/my-app/1.0-SNAPSHOT

- standard naming conventions for artifacts

  <artifact-id>-<version>

  my-app-1.0-SNAPSHOT.jar

## Local repository

- created automatically in `<user_home>/.m2/repository`
- all dependencies are loaded from this repository first

## Remote repository

- used by Maven to download additional artifacts
- all downloads are copied to the local repo
- default central repo: https://mvnrepository.com
- (optionally) configure additional repos

# Plugin



## Maven is plugin execution framework, i.e. executes goals via plugins

- plugins have (collection of) goals
- goals are identified by `<plugin id>:<goal id>`
  clean:clean
  compiler:compile
- build your own plugins

(https://maven.apache.org/guides/mini/guide-configuring-plugins.html)

## Common plugins (`https://maven.apache.org/plugins`)

- maven-compiler-plugin: compiles Java sources
- maven-deploy-plugin: deploy an artifact into a remote repo
- maven-resources-plugin: copy resources to the output directory
- maven-surefire-plugin: runs JUnit (or TestNG) test
- ...

# Archetype

## Maven project templating toolkit

http://maven.apache.org/archetype/maven-archetype-plugin/usage.html

## plugin, used to generate standard projects, based on model

### mvn archetype:generate

### configuration via additional (interactive) parameters

- archetypeArtifactId: maven-archetype-quickstart
- groupId: be.abis.mvn  (based on organisation URL)
- artifactId: my-app  (~main directory)
- archetypeVersion: 1.4
- (Java) package for sources: be.abis.mvn

# Generate Maven project structure via archetype

```
mvn archetype:generate
    -DgroupId=be.abis.mvn
    -DartifactId=my-app
    -DarchetypeArtifactId=maven-archetype-quickstart
    -DarchetypeVersion=1.4
    -DinteractiveMode=false
```

## resulting directory structure (+ HelloWorld app !)

```
C:\....\my-app
pom.xml
+---src
   +---main
   ¦   +---java
   ¦       +---be
   ¦           +---abis
   ¦               +---mvn
   +---test
       +---java
           +---be
               +---abis
                   +---mvn
```

# Lifecycle



## build lifecycle consists of (ordered sequence of) phases

- validate, compile, test, package, integrationtest, verify, install, deploy, ...

- in every build phase, plugin goals are executed
- goals are bound to the lifecycle phases

## Example: for pom packaging...

install  install:install-file

**Maven will execute every phase in the sequence up to and including the one defined**

## Activation

- add a plugin
- configure in the POM

# Example of lifecycle phase

**mvn compile**

## Compile phase: the phases and goals that actually get executed are

- validate
- generate-sources
- process-sources
- generate-resources
- process-resources
- compile

## Maven lifecycle phases

- validate: is the project correct? all necessary information available?
- compile: the source code
- test: with unit testing framework e.g. JUnit
- package: into distributable format, e.g. JAR
- integration-test: process and deploy the package into environment where integration tests can be run
- verify: run any checks to verify the package is valid and meets quality criteria
- install: install the package into the local repository (for use as a dependency in other local projects)
- deploy: to the remote repository (for sharing with other developers and projects)

### Other

- clean: cleans up artifacts created by prior builds (target dir)
- site: generates site (HTML) documentation for this project

## Recommended directory structure 2.1

- sources: `/src/main/java`
- tests: `/src/test/java`
- target: `/target/classes`
- resources: `/src/main/resources`
- test resources: `/src/test/resources`
- test target: `/target/test-classes`

## Note: use archetype for preparation

mvn archetype:generate

select maven-quickstart-archetype

# Handling a Java project

## typical Maven phases/goals

- compile
- test (includes compile): JUnit based test files
- test-compile: only compiles to test target
- package: generate .jar in `/target` dir
- install: copy .jar to local repository

# Configuring plugins

## Example: Jacoco plugin (for checking test coverage)

```
<build>
    <plugins>
        <plugin>
            <groupId>org.jacoco</groupId>
            <artifactId>jacoco-maven-plugin</artifactId>
            <version>0.8.8</version>
            <executions>
                <execution>
                    <id>default-prepare-agent</id>
                    <goals>
                        <goal>prepare-agent</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

## Dependency, based on JUnit, in POM

```
<dependencies>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter</artifactId>
        <version>5.12.2</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

## Run tests

**mvn test**

## test files, based on (surefire plugin) JUnit:

- **/*Test.java
- **/Test*.java
- **/*TestCase.java

# Properties



## placeholders accessible anywhere within a POM

every tag inside the <properties> tag becomes a property

## Example

```
<properties>
    <java.version>21</java.version>
</properties>
```

to specify Java version 21 for compilation

## reference to the property inside the POM

${<property name>}

```
<version>${java.version}</version>
```

# Resource filtering



## allow filtering

```
<resources>
  <resource>
    <directory>src/main/resources</directory>
    <filtering>true</filtering>
  </resource>
</resources>
```

## properties ${ } will be replaced when handling resources

# Building enterprise projects



## add required dependencies for

- supporting frameworks (e.g. Hibernate, Spring, Vaadin, ...)
- application servers (e.g. JBoss EAP, Tomcat, Glassfish, Liberty, ...)
- custom libraries (utilities, security, logging, ...)

  use central maven repository: `https://mvnrepository.com`

## multi-module builds

- generation of multiple artifacts (JAR, WAR, EAR, ...)
- multi project development

## POM inheritance

- super POM
- explicit reference

# Multi module build



- project with packaging 'POM'
- for every module
  - subdirectory
  - <module> entry in the root project's POM
- referring to the subdirectory
  - modules can have dependencies on each other

Maven will determine the build order for the entire set of modules

Remarks:

- as from maven 4.1.0, modules are called subprojects instead
- <module(s)> is deprecated to be removed in the future, and should be replaced by <subproject(s)>

# Example: web project

## The root project directory contains

- directories: web and core
- root pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" ...>
...
    <name>webapp</name>
    <modules>
        <module>web</module>
        <module>core</module>
    </modules>
</project>
```

```
<project xmlns="http://maven.apache.org/POM/4.1.0" ...>
...
    <name>webapp</name>
    <subprojects>
        <subproject>web</subproject>
        <subproject>core</subproject>
    </subprojects>
</project>
```

# Example: web project (cont.)

## Subdirectory

- contains its own pom.xml
- suppose: web depends on core

```
<project>
    ...
    <packaging>war</packaging>
    <name>webapp :: web</name>
    ...
    <dependencies>
        <dependency>
            ...
            <artifactId>core</artifactId>
            <version>1.0-SNAPSHOT</version>
        </dependency>
    </dependencies>
</project>
```

# POM inheritance



## Common definitions in parent POM

### contains <parent/> element

## Module POM optionally inherits from parent POM

## POM without <parent> element, inherits from super POM

# Maven support in IDE tools 6

## Eclipse 6.1

### m2eclipse

https://www.eclipse.org/m2e/

# IntelliJ Idea



https://plugins.jetbrains.com/plugin/7179-maven-helper

- for new project
  - File -> New -> Project
  - check "Maven" on the left
  - (optionally) specify archetype
- from "normal" Java module
  - right click -> add framework support -> Maven
  - change the group id in the POM to the base package name
- new Maven module
  - Project -> New -> Module
  - check "Maven" on the left
  - next
  - add new module to NONE!!! and watch out with naming

# Maven in a Spring project



## for new Spring project

- File -> New -> Project
- select Spring Initializer project on the left
- check "Maven"
- (optionally) specify additional libraries or frameworks

# Spring Core

**Objectives :**

- **Container and Bean lifecycle**
- **Declaring, wiring, initializing and destroying Beans**
- **Using Annotations for Autodiscovering and Wiring Beans**
- **Java based Configuration**

# The Container

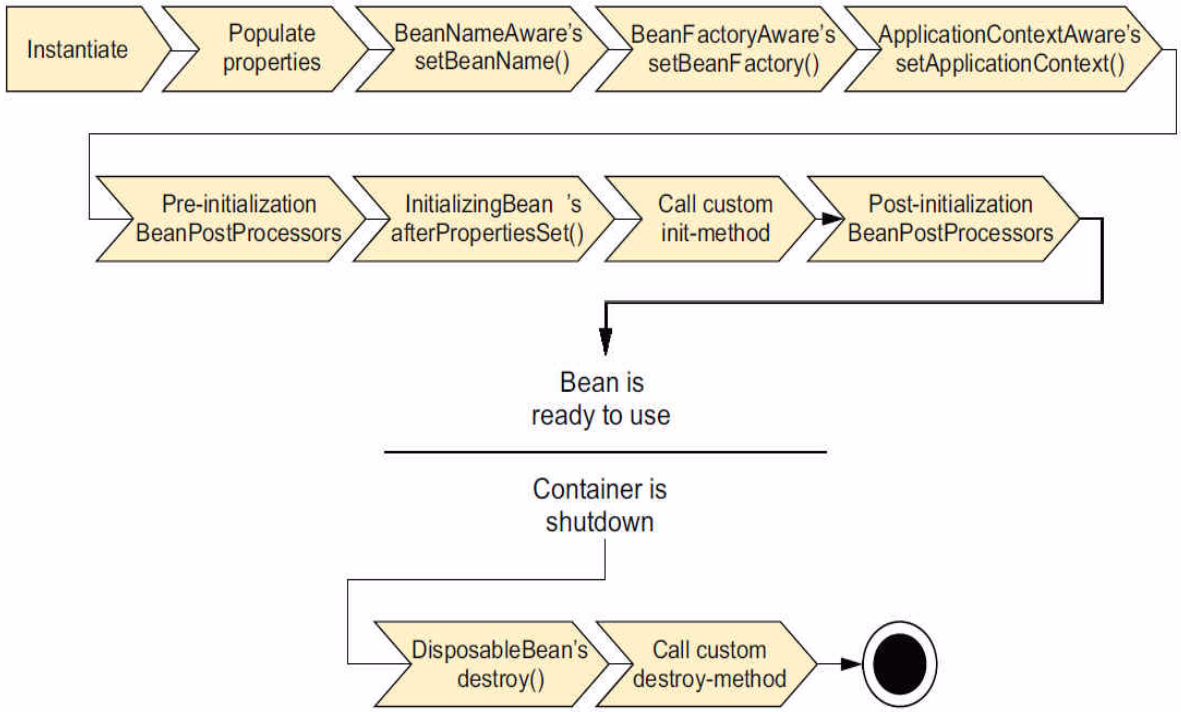


- **objects live within the Spring container**
- **container will create the objects, wire them together, configure them, and manage their complete lifecycle, using DI**
- **several container implementations, categorized in two types**
  - ***BeanFactory***
    - org.springframework.beans.factory.BeanFactory
    - basic support for DI, low-level
  - ***ApplicationContext***
    - org.springframework.context.ApplicationContext
    - extends BeanFactory
    - adds application framework services
    - different implementation classes present
- **Spring Boot Application automatically creates an ApplicationContext implementation behind the scenes**

# Bean Lifecycle



- Spring (Boot) by default uses eager loading when ApplicationContext is created (i.e. when running the application or test)
- Lifecycle:




- by default, beans will be singletons!

# Declaring and Wiring Beans



- beans that will be managed by Spring need to be configured/declared
- beans themselves always defined as interface + implementation
- historically configuration in XML, nowadays Java-based (Application class in Spring Boot)
- combine with annotations on classes for automatic bean discovery
- configuration loaded automatically when using JUnit in Spring

  **-> @SpringBootTest**
- different configurations can be combined if you want

# Java Based Configuration



- configuration of beans is done in a Java class
- Spring Boot already creates this class by default (...Application) in the base package of your application
- class is annotated with @SpringBootApplication (or @SpringBootTest)
- beans can be discovered via annotations in base package or its subpackages, or specifically defined in the configuration class via @Bean
- both techniques discussed above can be combined
  - autodiscovery of annotated classes for beans defined yourself
  - @Bean for 3rd party beans you want to use
    (more on this, see: 6 Explicit definition and wiring of beans in the Java Config class on page 83)

# Java Based Configuration - Example

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.web.client.RestTemplateBuilder;
import org.springframework.context.annotation.Bean;
import org.springframework.web.client.RestTemplate;
@SpringBootApplication
public class SpringRestTemplateClientApplication {
    public static void main(String[] args) {
        SpringApplication.run(SpringRestTemplateClientApplication.class, args);
    }

    @Bean
    public RestTemplate restTemplate(RestTemplateBuilder builder) {
        return builder.build();
    }
}
```

# Using Annotations for Autodiscovering and Wiring Beans



- explicit configuration of beans, whether Java Config or XML is very verbose
- use annotations on bean classes to make them autodiscoverable and to wire them together
- use one of the “core” stereotypes, or a more specialized one like @RestController (when using spring-boot-starter-web) for discovery
- wiring will be done via @Autowired
- approach has its pros and cons (compared to explicit configuration)
- could use CDI annotations @Named and @Inject instead, but not often done in Spring
- automatically configured in Spring Boot, scans current package and sub-packages
- if other base-package, add: @ComponentScan(basePackages="be.abis.otherpackage") on the configuration class

# Stereotypes



- **annotations from the org.springframework.stereotype package**
- **4 basic types of beans:**
  - **@Component**
    - most generic annotation
    - ensures that the class will be found during classpath scanning and that it will be registered in the context as a bean
  - **@Service**
    - beans in Service Layer (alias for @Component)
  - **@Repository**
    - for DAO or Repository classes
    - represent the database access layer in an application
    - has automatic persistence exception translation enabled
  - **@Controller**
    - tells the container that the class is an MVC controller
    - usually used together with the @RequestMapping annotation

# Meta-Annotations



- meta-annotation can combine annotations that are frequently used together
- meta-annotation can be applied to other annotations
- syntax:
  - state the Target of the annotation (Type, Method, Constructor)
  - add Retention Policy (annotation valid only for this class or for whole runtime)
  - add all annotations + all imports
  - public @interface MyAnnotation { } will define name of annotation
  - can have a body (containing annotation attributes)
- actually:

  @SpringBootApplication=

  @SpringBootConfiguration (special form of @Configuration)
  + @ComponentScan
  + @EnableAutoconfiguration

# Wiring Beans with Annotations



- **@Value to wire basic properties (String, int,...)**
- **@Autowired annotation used for wiring bean properties**
- **wire interface, not implementation class!**
- **can be put on constructor, setter, field and even methods**
- **alternatively you could use**
  - **@Inject (CDI, JSR 330)**
  - **@Resource**
    - Common Annotations for the JavaPlatform, JSR-250
    - to inject data sources, connectors, or any other desired resources available in the JNDI namespace
    - takes a name attribute: @Resource(name=”myBean”)
- **use @PostConstruct and @PreDestroy for init and destroy methods**

# Which Type of Dependency Injection to choose?



- constructor injection
  - for required dependencies
  - can be used for final fields
  - avoids circular dependencies
  - as of Spring 4.3: implicit constructor injection if only 1 constructor present!
- setter injection
  - for optional or changeable dependencies
  - class should still be able to function when omitted
- field injection
  - shorter, since no setter necessary
  - tightly coupled to DI container
  - classes cannot be instantiated (e.g. in unit tests) without reflection
  - to be avoided in most cases?

# Example



- **Coffee.java**

```
public class Coffee {
    private String flavour;

    public Coffee(String flavour) {this.flavour = flavour;}

    public String getFlavour() {return flavour; }
    public void getFlavour(String flavour) {this.flavour  = flavour; }
}
```

- **Person.java**

```
public class Person {
    private String firstName;
    private String coffeePreference;

    public Person(String firstName) {this.firstName = firstName;}
    public Person(String firstName, String coffeePreference) {
        this(firstName);
        this.coffeePreference=coffeePreference;
    }

    public String getFirstName() {return firstName;}
    public void setFirstName(String firstName) {this.firstName=firstName;}
    public String getCoffeePreference() {return coffeePreference;}
    public void setCoffeePreference(String coffeePreference) {
        this.coffeePreference=coffeePreference;
    }
}
```

# Example - Services - @Value

- **HelloService.java**

```
public interface HelloService {
    Person findPerson(int id);
    void sayHelloTo(Person person);
}
```

- **AbisHelloService.java**

```
@Component
public class AbisHelloService implements HelloService {

    private String helloMessage;

    public String getHelloMessage() {return helloMessage;}

    @Value("Welcome to Abis")
    public void setHelloMessage(String helloMessage) {
        this.helloMessage = helloMessage;
    }

    @Override
    public Person findPerson(int id) {
        return new Person("John");
    }

    @Override
    public void sayHelloTo(Person person) {
        System.out.println(helloMessage + ", " + person.getFirstName() + ".");
    }
}
```

# Example - Services - Init and Destroy

- **CoffeeService.java**

```
public interface CoffeeService {
    Coffee serveCoffeeToVisitor(Person person);
}
```

- **AbisCoffeeService.java**

```
@Service
public class AbisCoffeeService implements CoffeeService {

    @PostConstruct
    public void init(){
        System.out.println("preparing coffee");
    }

    @Override
    public Coffee serveCoffeeToVisitor(Person person) {
        return new Coffee(person.getCoffeePreference());
    }

    @PreDestroy
    public void destroy(){
        System.out.println("cleaning coffee things");
    }
}
```

# Example - Services - @Autowired

- **ReceptionService.java**

```
public interface ReceptionService {
    HelloService getHelloService();
    void welcomePersonAndGetCoffee(Person person);
}
```

- **AbisReceptionService.java**

```
@Service
public class AbisReceptionService implements ReceptionService {
     private HelloService helloService;
     @Autowired private CoffeeService coffeeservice;

     //@Autowired
     public AbisReceptionService(HelloService helloService){
         this.helloService=helloService;
     }

     public HelloService getHelloService() {return helloService; }
     public void setHelloService(HelloService helloService) {
         this.helloService=helloService;
     }

     public void welcomePersonAndGetCoffee(Person person) {
         helloService.sayHelloTo(person);
         Coffee coffee= coffeeservice.serveCoffeeToVisitor(person);
         System.out.println("Here is your "  + coffee.getFlavour() + " coffee.");
     }
}
```

# Example - Configuration and Tests

- **AnnotationsApplication.java**

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

- **AnnotationsApplicationTests.java**

```
@SpringBootTest
class AnnotationsApplicationTests {
    private Person person;
    @Autowired ReceptionService receptionService;

    @BeforeEach
    public void init(){ person=new Person("John","black");}

    @Test
    void personNameIsJohn(){
    assertEquals("John",
                    receptionService.getHelloService().findPerson(1).getFirstName());
    }

    @Test
    void welcomeJohn(){
        receptionService.welcomePersonAndGetCoffee(person);
    }
}
```

# Explicit definition and wiring of beans in the Java Config class



- define the beans via @Bean
  - method name (without the parentheses) = bean/instance name
  - use “new” to create instance as in plain java
  - all standard Java methods can be used
- wire beans together via setter injection or constructor injection, using method call to the previously defined beans
- Use @Bean(initMethod="init", destroyMethod="destroy") to call init and destroy methods. Make sure those are mentioned in the interface as well.
- use the bean in your test via @Autowired, or retrieve it explicitely:
  - retrieve the applicationContext by referencing the config class in your JUnit Test’s @BeforeAll method
  - retrieve bean from applicationContext
    - by referencing the Interface class
    - by bean name -> cast necessary!

# Example

- **model + service classes as mentioned in 5.5 Example on page 78, except for a small change in the CoffeeService interface:**

```
public interface CoffeeService {
    Coffee serveCoffeeToVisitor(Person person);
    void init();
    void destroy();
}
```

**Make sure to take out all Spring annotations (@Service, @Autowired,...)**

# Example (cont.)

- **configuration class will define all the beans explicitely**

```
@SpringBootApplication
public class JavaconfigApplication {

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

    @Bean
    public HelloService helloService(){
        AbisHelloService ahs = new AbisHelloService();
        ahs.setHelloMessage("Welcome to Abis");
        return ahs;
    }

    @Bean(initMethod="init" destroyMethod="destroy")
    public CoffeeService coffeeService(){
        return new AbisCoffeeService();
    }

    @Bean
    public ReceptionService receptionService(){
        ReceptionService rs = new AbisReceptionService(helloService());
        ((AbisReceptionService)rs).setCoffeeservice(coffeeService());
        return rs;
    }

}
```

# Example (cont.)

- **test**

```
class JavaconfigApplicationTests {

    private static ReceptionService receptionService;
    private Person person;

    @BeforeAll
    public static void retrieveBean(){
        ApplicationContext context =
            new AnnotationConfigApplicationContext(JavaconfigApplication.class);
        receptionService = (ReceptionService) context.getBean("receptionService");
    }

    @BeforeEach
    public void init(){ person = new Person("John","black");}

    @Test
    void personNameIsJohn(){
        assertEquals("John"
                receptionService.getHelloService().findPerson(1).getFirstName());
    }

    @Test
    void welcomeJohn(){
        receptionService.welcomePersonAndGetCoffee(person);
    }
}
```

# Mixing and Importing Configurations



- in case a Java config class gets too big, you can split it up and import one into another
- different types of configurations can also be mixed
  - use @Autowired for beans explicitely created in the Java config file
  - import an XML configuration file in a Java config file (for legacy code)
  - import a Java config file into the XML configuration
  - do a component scan from XML
  - ...
- watch out with order of precedence in case beans would be defined more than once!

# Import Java Config in other Java Config



- create a second java config file, annotated with @Configuration
- use @Import(SecondConfig.class) from the Spring Boot configuration file
- In case you need to refer to a bean defined in the imported file, pass it as an argument to the method
- example

```
@Configuration
public class SecondConfig {
    @Bean
    public HelloService helloService() {
        AbisHelloService ahs = new AbisHelloService();
        ahs.setHelloMessage("Welcome to Abis");
        return ahs;
    }
}
```

# Import Java Config in other Java Config - Example (cont.)

- **Spring Boot configuration file**

```
@SpringBootApplication
@Import(SecondConfig.class)
public class ImportJavaconfigApplication {
    public static void main(String[] args) {
        SpringApplication.run(ImportJavaconfigApplication.class,args);
    }
    @Bean(initMethod="init" destroyMethod="destroy"
        public CoffeeService coffeeService(){
            return new AbisCoffeeService();
    }

    @Bean
    public ReceptionService receptionService(HelloService helloService){
        ReceptionService rs = new AbisReceptionService(helloService);
        ((AbisReceptionService)rs).setCoffeeservice(coffeeService());
        return s;
    }
}
```

# Mix Explicit Java Config with ComponentScan



- beans explicitely defined in a Java config file, can be wired into other classes via @Autowired
- example

```
public class AbisCoffeeService implements CoffeeService {

    public void init(){
        System.out.println("preparing coffee");
    }

    @Override
    public Coffee serveCoffeeToVisitor(Person person) {
         return new Coffee(person.getCoffeePreference());
    }

    public void destroy(){
        System.out.println("cleaning coffee things");
    }
}
```

# Mix Explicit Java Config with ComponentScan - Example (cont.)

- **configuration**

```
@SpringBootApplication
public class JavaconfigcomponentscanApplication {

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

    @Bean(initMethod="init",destroyMethod="destroy")
    public CoffeeService coffeeService(){
        return new AbisCoffeeService();
    }

}
```

- **AbisReceptionService.java**

```
@Service
public class AbisReceptionService implements ReceptionService {
    private HelloService helloService;

    @Autowired
    private CoffeeService coffeeService;

    //other code like before
}
```

# Import XML Config in Java Config



- similar to importing java config into another java config
- @ImportResource to import the XML configuration file from the classpath
- example

myconfig.xml

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                  http://www.springframework.org/schema/beans/spring-beans.xsd">
    <bean id="coffeeService"
          class="be.abis.xmlconfig.service.AbisCoffeeService"
          init-method="init" destroy-method="destroy" />
</beans>
```

```
@SpringBootApplication
@ImportResource("classpath:myconfig.xml")
public class ImportxmlconfigApplication {

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

# Advanced Configuration Topics

**Objectives :**

- **Bean Scopes**
- **Addressing Ambiguity**
- **Environments and Profiles**
- **Conditional Beans**
- **Runtime Value Injection: Property Injection**

# Bean Scopes



- singleton
  - same instance of bean is returned each time
  - DEFAULT! -> service oriented nature of Spring!
- prototype
  - one instance of bean is created each time it is injected into or retrieved from the application context
- request/session
  - in web application, one instance for each HTTP request/session
- globalSession
  - in web application, one instance for each session, but only in portlet context
- application
  - in web application, one instance during life of a ServletContext
- websocket
  - one instance during life of a WebSocket

# Bean Scopes (cont.)

- configure via

  @Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)

  @Scope(WebApplicationContext.SCOPE_SESSION)

- example

```
@Component
@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
public class Person {

    private String firstName="John";
    private String coffeePreference="black";

    public Person(){}

    public Person(String firstName) {
        this.firstName=firstName;
    }

    public Person(String firstName, String coffeePreference) {
        this(firstName);
        this.coffeePreference=coffeePreference;
    }

    //getters and setters
}
```

# Bean Scopes - Example (cont.)

- **test**

```
@SpringBootTest
class BeanScopeTest {

    @Autowired private Person person1;
    @Autowired private Person person2;
    @Autowired private ReceptionService rs1;
    @Autowired private ReceptionService rs2;

    @Test
    void person1and2areDifferentObjects(){
        System.out.println("Person 1:"  + person1);
        System.out.println("Person 2:"  + person2);
        assertTrue(person1!=person2);
    }

    @Test
    void receptionServicesAreSameObject(){
        System.out.println("Service 1:"  + rs1);
        System.out.println("Service 2:" +  rs2);
        assertTrue(rs1==rs2);
    }

}
```

# Addressing Ambiguity



- @Autowired will by default use type inference to find implementation class
- if more implementation of the interface present, a **NoUniqueBeanDefinitionException** will be thrown
- solve with:
  - @Qualifier to distinguish between beans
  - @Primary to designate primary (“favourite”) bean

# @Qualifier Example

- **CampinaCoffeeService.java**

```
@Service
@Qualifier("campina")
public class CampinaCoffeeService implements CoffeeService {
    @Override
    public Coffee serveCoffeeToVisitor(Person person) {
        return new Coffee(person.getCoffeePreference() + " with extra milk");
    }
}
```

- **AbisCoffeeService.java**

```
@Service
@Qualifier("Abis")
public class AbisCoffeeService implements CoffeeService {

    //content: see earlier

}
```

# @Qualifier Example (cont.)

- **AbisReceptionService.java**

```
@Service
public class AbisReceptionService implements ReceptionService {
    private HelloService helloService;
    @Autowired
    @Qualifier("campina")
    private CoffeeService coffeeservice;
    // rest of content: see earlier
}
```

- **QualifierApplicationTests.java**

```
@SpringBootTest
class QualifierApplicationTests {
    private Person person;
    @Autowired ReceptionService receptionService;

    @BeforeEach
    public void init(){ person=new Person("John","black");}

    @Test
    void welcomeJohn(){
        receptionService.welcomePersonAndGetCoffee(person);
    }
}
```

# @Primary Example

- **CampinaCoffeeService.java**

```
@Service
public class CampinaCoffeeService implements CoffeeService {
    @Override
    public Coffee serveCoffeeToVisitor(Person person) {
        return new Coffee(person.getCoffeePreference() + " with extra milk");
    }
}
```

- **AbisCoffeeService.java**

```
@Service
@Primary
public class AbisCoffeeService implements CoffeeService {
    //content: see earlier
}
```

- **AbisReceptionService.java**

```
@Service
public class AbisReceptionService implements ReceptionService {
    private HelloService helloService;
    @Autowired
    private CoffeeService coffeeservice;
    // rest of content: see earlier
}
```

# Environments and Profiles



- environment = abstraction integrated in the container that models profiles and properties
- profile = named, logical group of bean definitions
- environment determines which profiles (if any) are currently active, and which profiles (if any) should be active by default
- use cases
  - working against an in-memory datasource in development vs looking up that same datasource from JNDI when in QA or production
  - registering monitoring infrastructure only when deploying an application into a performance environment
  - registering customized implementations of beans for customer A vs. customer B deployments
- application does not need to be rebuilt for each environment!

# Configuring and Activating Profiles

- Java Config (class or method level)
  @Profile("development”)
- default profile: @Profile(“default”)
- activation
  - NoSuchBeanDefinitionException thrown when profile not activated, unless default profile specified
  - programmatic, via Environment object
  - declarative, via spring.profiles.active property (command-line or in application.properties when using Spring Boot)
  - via @ActiveProfiles in Test
  - can have more than one profile active at once

# Profiles - Example

- **CampinaCoffeeService.java**

```
@Service
@Profile("test")
public class CampinaCoffeeService implements CoffeeService {
    @Override
    public Coffee serveCoffeeToVisitor(Person person) {
        return new Coffee(person.getCoffeePreference() + " with extra milk");
    }
}
```

- **AbisCoffeeService.java**

```
@Service
@Profile("production")
public class AbisCoffeeService implements CoffeeService {
    //content: see earlier
}
```

- **AbisReceptionService.java**

```
@Service
public class AbisReceptionService implements ReceptionService {
    private HelloService helloService;
    @Autowired
    private CoffeeService coffeeservice;
    // rest of content: see earlier
}
```

# Profiles - Example (cont.)

- **application.properties**

  ```
  spring.profiles.active=test
  ```

- **ProfilesApplicationTests.java**

```
@SpringBootTest
//@ActiveProfiles("production")    // will override the setting of application.properties
class ProfilesApplicationTests {
    private Person person;
    @Autowired ReceptionServices receptionService;

    @BeforeEach
    public void init(){ person=new Person("John","black");}

    @Test
    void welcomeJohn(){
        receptionService.welcomePersonAndGetCoffee(person);
    }
}
```

# Conditional Beans



- more flexible than profiles
- indicates certain condition that should be consulted before a bean is registered
- use cases
  - bean to be created only when some other bean is present
  - bean to be configured if and only if some library is available
  - bean to be created is dependent on specific environment variable
- @Conditional
  - condition based on method “matches” created in class that implements Condition
- @ConditionalOnProperty
  - provided by Spring Boot
  - condition based on property in application.properties
  - other options: ConditionalOn...
    Bean, Class, Jndi, Resource, MissingBean,Java,...

# Conditional - Example

- **AbisCondition.java**

```
public class AbisCondition implements Condition {
    @Override
    public boolean matches(ConditionContext ctx, AnnotatedTypeMetadata arg1) {
        return ctx.getEnvironment().getProperty("hello.company").matches("abis");
    }
}
```

- **OtherCompanyCondition.java**

```
public class OtherCompanyCondition implements Condition {
    @Override
    public boolean matches(ConditionContext ctx, AnnotatedTypeMetadata arg1) {
        return ctx.getEnvironment().getProperty("hello.company").matches("other");
    }
}
```

# Conditional - Example (cont.)

- **application.properties**

  **hello.company=other**

- **AbisHelloService.java**

```
@Service
@Conditional(AbisCondition.class)
public class AbisHelloService implements HelloService {

    private String helloMessage;

    public String getHelloMessage() {
        return helloMessage;
    }

    @Value("Welcome to Abis")
    public void setHelloMessage(String helloMessage) {
        this.helloMessage = helloMessage;
    }

    // and the other methods
}
```

# Conditional - Example (cont.)

- **OtherHelloService.java**

```
@Component
@Conditional(OtherCompanyCondition.class)
public class OtherHelloService implements HelloService{

    private String helloMessage;

    public String getHelloMessage() {
        return helloMessage;
    }

    @Value("Welcome at our company")
    public void setHelloMessage(String helloMessage) {
        this.helloMessage = helloMessage;
    }

    // and the other methods

}
```

# ConditionalOnProperty - Example

- **application.properties**

  **hello.company=abis**

- **AbisHelloService.java**

```
@Component
@ConditionalOnProperty(name="hello.company", havingValue="abis")
public class AbisHelloService implements HelloService {
    //content: see previous example
}
```

- **SomeOtherHelloService.java**

```
@Component
@ConditionalOnProperty(name="hello.company", havingValue="other")
public class SomeOtherHelloService implements HelloService{
    //content: see previous example
}
```

# Runtime Value Injection



- to avoid hard-coded values, determined at runtime instead
- two techniques
  - Injecting external values via property placeholders
  - Spring Expression Language

# Injecting external values



- add a new property to application.properties
- use ${myPropertyName} to call it
- in case you want to use a specific file for these properties:
  - create my.properties under the resources folder
  - add in your configuration class via @PropertySource("classpath:my.properties")
  - application.properties has precedence over this new file
- to programmatically resolve a property:
  - wire Environment env in your class
  - get properties via env.getProperty(“mykey”);

# Property from application.properties - Example

- **application.properties**

```
spring.profiles.active=production
hello.company=Abis
```

- **GeneralHelloService.java**

```
@Service
public class GeneralHelloService implements HelloService {

    private String helloMessage;

    public String getHelloMessage() {
        return helloMessage;
    }

    @Value("Welcome to ${hello.company}")
    public void setHelloMessage(String helloMessage) {
        this.helloMessage=helloMessage;
    }

    public Person findPerson(int id) {
        return new Person("John");
    }

    public void sayHelloTo(Person person) {
        System.out.println(helloMessage + ", " +  person.getFirstName());
    }
}
```

# Property from my.properties - Example

- **application.properties**

  ```
  spring.profiles.active=production
  ```

- **my.properties**

  ```
  hello.company=Abis
  ```

- **RuntimevalueinjectionApplication.java**

```
@SpringBootApplication
@PropertySource("classpath:my.properties")
public class RuntimevalueinjectionApplication {
    public static void main(String[] args) {
        SpringApplication.run(RuntimevalueinjectionApplication.class, args);
    }
}
```

# Property from code - Example

- **application.properties**

```
spring.profiles.active=production
hello.company=Abis
```

- **GeneralHelloService.java**

```
@Service
public class GeneralHelloService implements HelloService {

    @Autowired private Environment env;
    private String helloMessage;
    public String getHelloMessage() {
        return helloMessage;
    }
    public void setHelloMessage(String helloMessage) {
        this.helloMessage=helloMessage;
    }

    public Person findPerson(int id) {
        return new Person("John");
    }

    public void sayHelloTo(Person person) {
        System.out.println("Welcome at " + env.getProperty("hello.company")
                                + ", " + person.getFirstName());
    }
}
```

# Aspect-oriented Spring

**Objectives :**

- **What is AOP**
- **Terminology**
- **Configuring AOP in Spring**

# What is AOP



## General



- aspect-oriented programming
- separate cross-cutting concerns from the business logic

  logging, transactions, security
- declaratively define how and where functionality is applied
- cross-cutting concerns modularized into special classes

  -> aspects

# Terminology



- ***Aspect***

  **The general functionality that has to be handled**

- ***Advice***

  **The what and when of an aspect**

  **what: code to be executed**

  **when: before, after, after-returning, after-throwing, around**

- ***Join Points***

  **point in execution where aspect could be plugged in**

- ***Pointcuts***

  **matches the join points where the advice is applied**

- ***Weaving***

  **applying aspects via a proxied object**

  **can be done at compile-time, classload time or runtime**

# Spring's AOP support



- AOP frameworks differ in join point model, weaving approach,...
- Three major players
  - AspectJ
  - JBoss AOP
  - Spring AOP
- Spring AOP borrows a lot from AspectJ
- Advice is written in Java: Aspect is a class, advice are methods
- Spring advises objects at runtime (weaving is proxy-based)
  - only methods defined in the interface can be used
  - methods calling methods in the same class, won't be advised
- Spring only supports method join points
- Advice should not influence behaviour of original method!

# Writing pointcuts





- **AspectJ pointcut expression language is used**
- **Spring supports a subset of AspectJ pointcut designators**

  **args(), @args(), execution(), this(), target(), @target(), within(), @within() and @annotation()**

  **additional: bean()**
- **within()**

  **within(com.xyz.service.*) -> any method within the service package**

  **within(com.xyz.service..*) -> any method within the service package or a sub-package**
- **target()**

  **target(com.xyz.service.AccountService) -> any method where the target object implements the AccountService interface**
- **bean()**

  **bean(myHelloBean) -> any method on a bean named myHelloBean (id-based!)**

  **bean(*Service) -> any method on Spring beans having names that match the wildcard expression '*Service'**
- **args() -> to pass args to advice, used together with execution()**

  **args(name)**

# Writing pointcuts (cont.)



- **execution()**
  - **execution(modifiers-pattern? ret-type-pattern declaring-type-pattern? name-pattern(param-pattern) throws-pattern?)**
  - *** wildcard can be used for return type + as all or part of a name-pattern**
  - **param-pattern:**

    **() : method without arguments**

    **(..) : method with any number of arguments (zero or more)**

    **(*) : one parameter of any type**

    **(*,String): two parameters, first any type, second of type String**
  - **examples**

    **execution(public * *(..)) -> execution of any public method**

    **execution(* set*(..)) -> execution of any method with name beginning with set**

    **execution(* Instrument.play(..)) -> execution of play method in Instrument class**

    **execution(* com.xyz.service.*.*(..)) -> execution of any method in service package**

# Defining Aspects



- AspectJ's annotation-oriented model
- @Aspect makes class an aspect
- @Pointcut defines pointcut (combined with type: after, before,...)
- Maven Dependency

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

# Annotating Advices





- Types of advices:
  - @Before, @After
  - @AfterReturning, @AfterThrowing
  - @Around
- put just before the advice method, combined with a pointcut
- possibility to define named pointcuts that can be combined

```
public class MyPointcuts {
    @Pointcut("within(com.xyz.service.*"))
    public void inService(){}
    @Pointcut("execution(* perform*(..))")
    public void execPerform(){}
    @Pointcut(inService() && execPerform())
    public void execPerformInService(){}
}
```

  -> use via @Pointcut(MyPointCuts.execPerform())

- passing arguments: @AspectJ leans on Java syntax to determine details of parameters passed, arg-names not necessary

# Before and After(Returning/Throwing) Advice



- aspect is a pure Java class with (possibly) multiple methods (the advices)
- @Before: executed BEFORE executing the original method called by the PointCut
- @After: executed AFTER executing the original method called by the PointCut
- @AfterReturning: executed AFTER the original method called by the PointCut returned correctly
- @AfterThrowing: executed AFTER the original method called by the PointCut returned threw an Exception
- @AfterTrowing: possibility to work with / access the exception
- advice method's return-type is **void**
- (Proceeding)JoinPoint can be used to give information about the method/class that is under advice
- (Proceeding)JoinPoint has to be FIRST argument of advice!

# Around Advice



- single method for before and after advice
- if information needs to be shared
- return-type is **void or Object**
- Object -> if original method returns something, to fetch returned value
- Uses ProceedingJoinPoint instead of JoinPoint
- ProceedingJoinPoint pjp -> **(Object o =) pjp.proceed();** should be called explicitly

# Example



- **HelloService.java**

```
public interface HelloService {
    Person findPerson(int id);
    void sayHelloTo(Person person);
}
```

- **AbisHelloService.java**

```
@Service
public class AbisHelloService implements HelloService {

    private String helloMessage;

    public String getHelloMessage() {
        return helloMessage;
    }
    public void setHelloMessage(String helloMessage) {
        this.helloMessage=helloMessage;
    }

    public Person findPerson(int id) {
        return new Person("John");
    }

    public void sayHelloTo(Person person) {
        System.out.println("Welcome at Abis, " +  person.getFirstName());
    }
```

# }Example (cont.)

- **AbisReceptionService.java**

```
@Service
public class AbisReceptionService implements ReceptionService {

    private HelloService helloService;
    @Autowired CoffeeService coffeeService;

    //@Autowired
    public AbisReceptionService(HelloService helloService) {
        this.helloService = helloService;
    }

    //getters and setters

    @Override
    public void welcomePersonAndGetCoffee(Person person)
                                        throws PersonNotWelcomeException {
        if (!person.getFirstName().equals("Mr. Bean") {
            helloService.sayHelloTo(person);
            Coffee coffee = coffee.serviceserveCoffeeToVisitor(person);
            System.out.println("Here is your "  +coffee.getFlavour() + ".");
        } else {
            throw new PersonNotWelcomeException
                    (person.getFirstName()+", you are not welcome.");
        }
    }

}
```

# Example (cont.)

- **AopPointcuts.java**

```
public class AopPointcuts {

    @Pointcut("execution(* welcome*(..))")
    public void execWelcome(){}

    @Pointcut("execution(* be.abis.demo.service.*.get*(..))")
    public void execGet(){}

    @Pointcut(
        "execution(* be.abis.demo.service.*.find*(..))")
    public void findThings(){}

}
```

# Example (cont.)

- **MyAspect.java**

```
@Component
@Aspect
public class MyAspect {

    @Before("execution(public * be.abis.demo.service.*.welcome*(..))")
    public void beforeWelcome(){
        System.out.println("Receptionist stands up:");
    }

    @AfterReturning("AopPointcuts.execWelcome()")
    public void afterWelcomeOK(JoinPoint jp){
        System.out.println("Guest is guided to lounge.");
    }

    @AfterThrowing(pointcut="AopPointcuts.execWelcome()",throwing="exc")
    public void afterNotWelcome(GuestNotWelcomeException exc){
        System.out.println("You are escorted back to the door " +exc.getMessage());
    }

    @After("AopPointcuts.execGet()")
    public void afterGetting(JoinPoint jp){
        System.out.println("finished calling getter: " + jp.getSignature().getName());
    }


    //continued on next page!
```

# Example (cont.)

```
    // MyAspect.java continued
    @Around("AopPointcuts.findThings()")
    public Object timing(ProceedingJoinPoint pjp){
        Object o =null;
        try {
            long start = System.nanoTime();
            o = pjp.proceed();
            long end= System.nanoTime();
            System.out.println("time taken by " + pjp.getSignature().getName() +" : " +
                ((end-start)/1000000.0) + " milliseconds");
        } catch (Throwable e) {
            e.printStackTrace();
        }
        return o;
    }
}
```

# Example (cont.)

- **ReceptionTest.java**

```
@SpringBootTest
class ReceptionTest {
    private Person person;
    @Autowired ReceptionService receptionService
    @BeforeEach public void init() { person=new Person("John" ,"black coffee"); }

    @Test void personNameIsJohn(){
        assertEquals("John",
                receptionService.getHelloService().findPerson(1).getFirstName());
    }

    @Test void welcomeJohn() throws PersonNotWelcomeException {
        receptionService.welcomePersonAndGetCoffee(person;
    }

    @Test
    void welcomeMrBeanException()  {
        Person person2 = new Person("Mr. Bean", "none");
        assertThrows(PersonNotWelcomeException.class,
                () -> {receptionService.welcomePersonAndGetCoffee(person2);});
    }
}
```

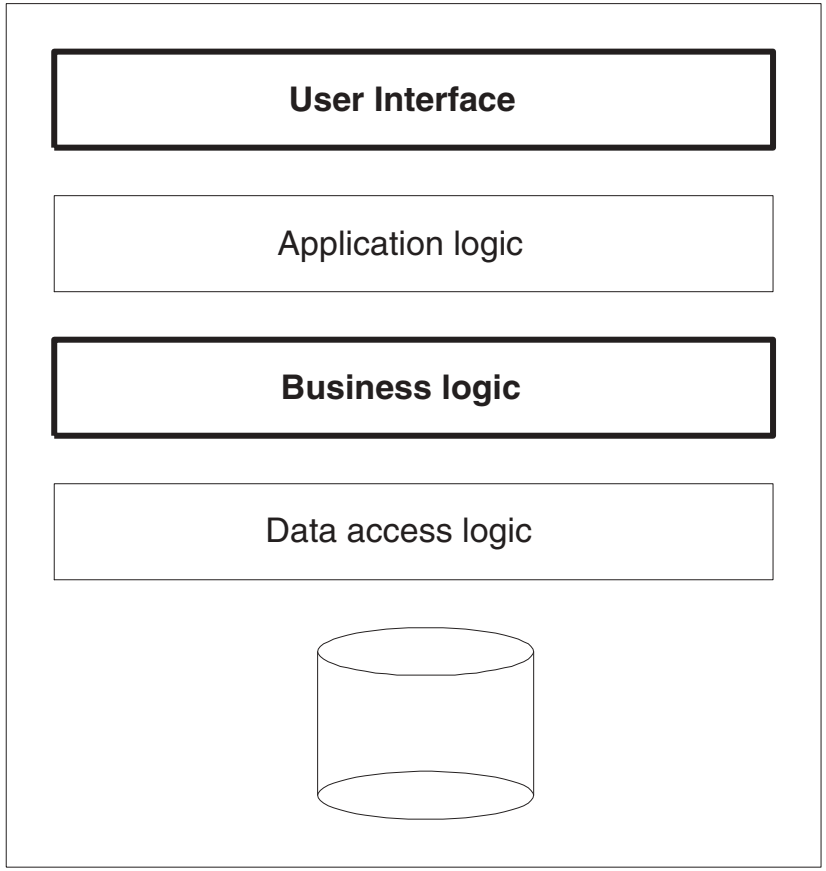
# Introduction to REST and JSON

**Objectives :**

- **Application Development and Distributed Computing**
- **RESTful Web Services / APIs**
- **JSON vs XML**

# Application Development and Distributed Computing 1

## Structuring an Application: Layers 1.1



# Design Goals



## Think further than just this 1 application you want to build

## Several interfaces on same business layer/domain model

## Creating modular applications

- no “spaghetti” code
- reuse of parts (even code written in other programming languages)

## Easy maintenance

## Extensibility

# Distributed Computing



**“Different components and objects making up an application can be located on different computers connected to a network”**

## Advantages:

- Reuse (multiple-client applications)
- Location transparency (standard mechanisms for lookup)
- Higher reliability and availability (clustering)
- Scalability (deploying on powerful servers)
- Higher performance (parallel processing)
- Extensibility (dynamic configuration)
- Higher productivity and lower development time (small programs)
- Interoperability and reduced vendor dependence

# Layers vs. Tiers

## Where does each layer run?

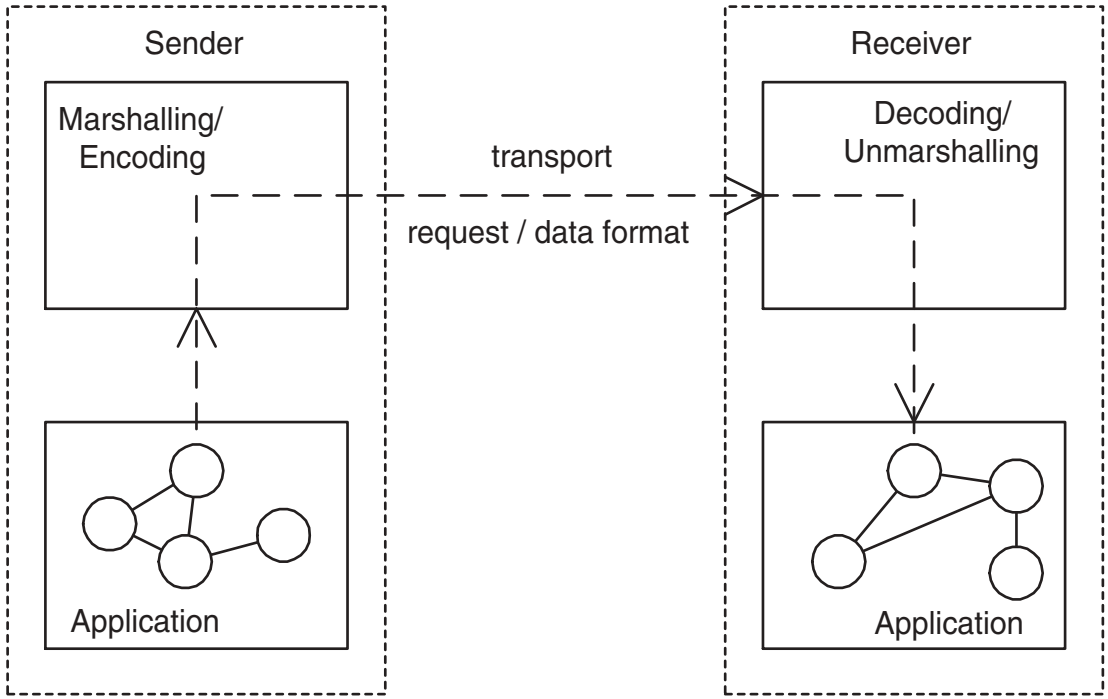
## Evolution:

- monolithic
- client / server: 2-tier, 3-tier,... (e.g. Java + Oracle server)
- n-tier, distributed components (Corba, Java RMI, MS DCOM)
- fully distributed, heterogeneous systems:
  - independent services
  - standard internet technology (web / cloud / mobile / IoT /...)

# Communicating Objects & Data Types








- **remote communication (remote procedure call / RPC)**
- **request / (complex) data structure / data types**
- **marshal / serialize / encode**

# Remote ‘Object’ Protocols

## CORBA (IIOP)

## MS DCOM

## Java RMI (Remote Method Invocation)

## WDDX (Web Distributed Data Exchange)

## XML-RPC (XML Remote Procedure Call)

## SOAP

## REST (Representational State Transfer)

## JSON-RPC

# RESTful Web Services / APIs



## Before: classic web services with SOAP and WSDL

- well structured & governable, but complex

## From +/- 2010: shift to more lightweight 'API'

- easier access to services outside the enterprise
- more agile, less ceremony
- often using REST and JSON vs. SOAP/XML
- more suited for consumer apps, new channels, mashups (Web 2.0) etc...

# What is REST?





## REST = Representational State Transfer

- ***Representational*** **- resources can be represented in virtually any form, including XML, JavaScript Object Notation (JSON), or even HTML, whatever form best suits the consumer of resources**
- ***State*** **- more concerned with the state of a resource than with the actions you can take against resource**
- ***Transfer*** **- transfer of resource data, in some representational form, from one application to another**

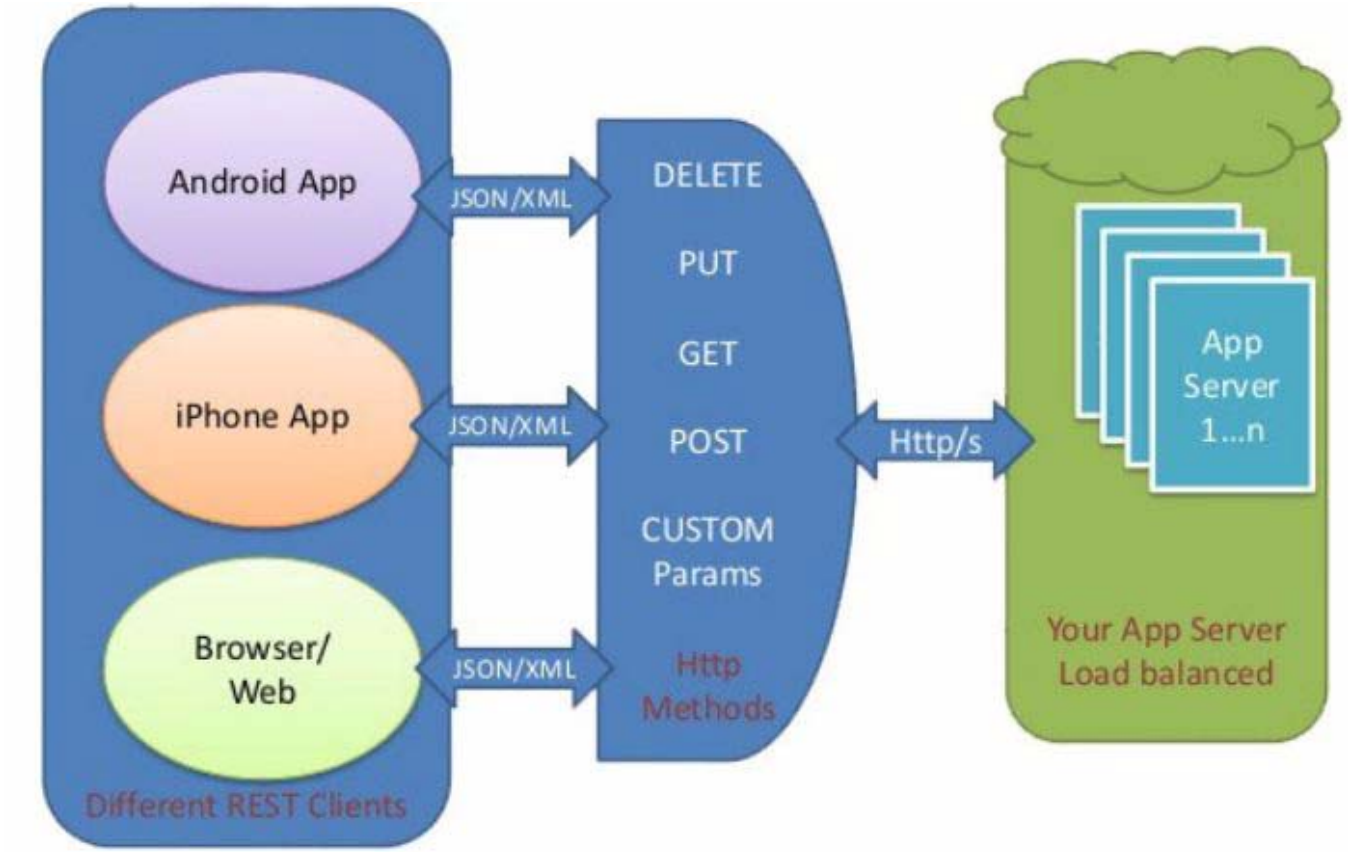
## A basic philosophy: “everything is a resource on the web”, should be addressable (URI)

**Example: http://twitter.com/users?screen_name=james**

## Uses HTTP to access resources:
## 4 basic operations (cf. CRUD): GET, PUT, POST, DELETE

## REST is an architectural style, not a protocol like SOAP

# REST API Architecture



# The REST Architectural Style



## REST is not an architecture, but a set of constraints that creates a software architectural style

## Following constraints define a RESTful system:

- Client-server: client and the server should be loosely coupled
- Stateless: there should be no need for the service to keep users' sessions
- Cacheable: network infrastructure should support a cache to avoid repeated round trips between the client and the server for retrieving the same resource
- Uniform interface: resource exposed must have a unique address and should be accessible through a generic interface
- Layered: server can have multiple layers for implementation
- Code on demand (optional): functionality of the client applications can be extended at runtime by allowing a code download from the server and executing the code

# Resources



**RESTful resource is anything that is addressable over the web**

**Resource is a logical, temporal mapping to a concept in the problem domain**

**Examples:**

- **a news story**
- **the temperature in Leuven at 4:00 p.m. EST**
- **a person working for a company**
- **list of participants to a course**

**URI in a RESTful web service is a hyperlink to a resource**

**URI is not meant to change over time as it may break the contract between a client and a server**

**Resource's mapping is unique, but different requests for a resource can return the same underlying binary representation from the server**

# HTTP Methods



## Map CRUD actions on the resources to the appropriate HTTP methods

| Data Action | HTTP equivalent |
|---|---|
| CREATE | POST |
| READ | GET |
| UPDATE | PUT (or PATCH) |
| DELETE | DELETE |

## Will be linked with @GetMapping, @PostMapping,... in the Spring controller classes

## Other methods: HEAD and OPTIONS

# The Representation of Resources





## Representation (of resource) =

- what is sent back and forth between clients and servers
- temporal state of the actual data located in some storage device at the time of a request
- binary stream together with its metadata that describes how the stream is to be consumed by the client
- can take various forms, e.g. image, text file, XML, JSON

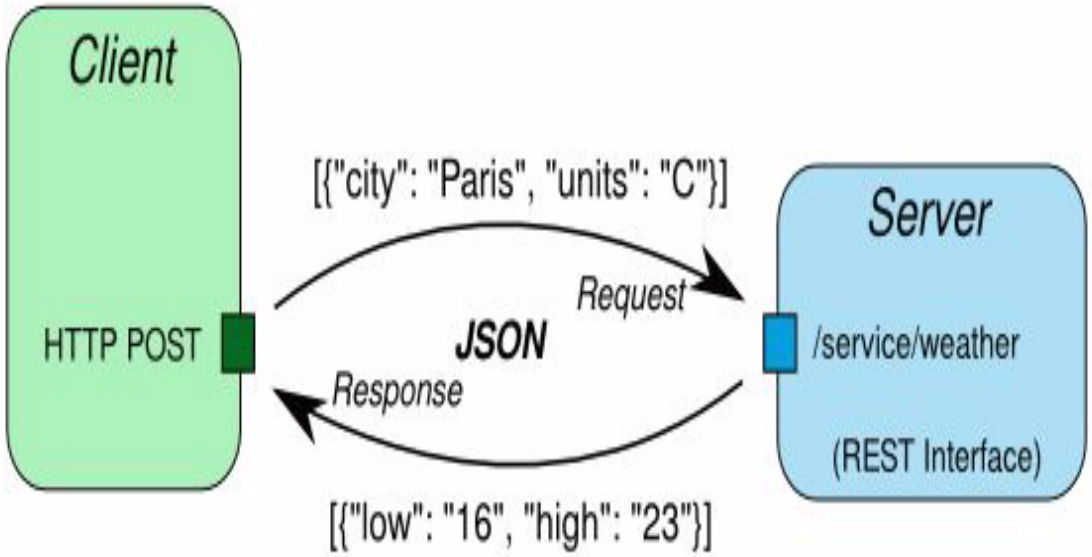
## All clients will use the same URI with appropriate accept header values for accessing the same resource in different representations

## Metadata can also contain extra information about the resource

- validation
- encryption information
- extra code to be executed at runtime

# Returning resource via HTTP request





# REST Resource Naming Best Practices



## Use nouns to represent resources

## Consistency is the key

## Never use CRUD function names in URIs

## Use query component to filter URI collection

# Use nouns to represent resources

## Resource is a thing (noun) instead of an action (verb)

## Four categories of resource archetypes

- **document**
  - **single resource inside resource collection**
  - **use “singular” name to denote document resource archetype**
  - **examples:**

    **http://api.example.com/device-management/managed-devices/{device-id}**
    **http://api.example.com/user-management/users/{id}**
    **http://api.example.com/user-management/users/admin**

# Use nouns to represent resources (cont.)



- collection
  - server-managed directory of resources
  - clients may propose new resources to be added to a collection, but is up to the collection to choose to create a new resource or not
  - Use “plural” name to denote collection resource archetype
  - examples:

    http://api.example.com/device-management/managed-devices

    http://api.example.com/user-management/users

    http://api.example.com/user-management/users/{id}/accounts

# Use nouns to represent resources (cont.)

- store
  - client-managed resource repository
  - lets an API client put resources in, get them, decide to delete
  - store never generates new URIs, each stored resource has a URI that was chosen by a client when it was initially put into the store
  - use “plural” name to denote store resource archetype
  - examples

    http://api.example.com/cart-management/users/{id}/carts
    http://api.example.com/song-management/users/{id}/playlists
- controller
  - models resource as a procedural concept
  - like executable functions, with parameters and return values
  - use “verb” to denote controller archetype
  - examples:

    http://api.example.com/cart-management/users/{id}/cart/checkout
    http://api.example.com/song-management/users/{id}/playlist/play

# Consistency is the key



## Use forward slash (/) to indicate a hierarchical relationship

- forward slash (/) character is used in the path portion of the URI to indicate a hierarchical relationship between resources
- examples:

  http://api.example.com/device-management
  http://api.example.com/device-management/managed-devices
  http://api.example.com/device-management/managed-devices/{id}
  http://api.example.com/device-management/managed-devices/{id}/scripts
  http://api.example.com/device-management/managed-devices/{id}/scripts/{id}

## Do not use trailing forward slash (/) in URIs

- adds no semantic value and may cause confusion
- better to drop them completely
- examples:

  NOT:
  http://api.example.com/device-management/managed-devices/

  BUT:
  http://api.example.com/device-management/managed-devices

# Consistency is the key (cont.)



## Use hyphens (-) to improve the readability of URIs

- easier for people to scan and interpret
- use to improve the readability of names in long path segments
- examples:

  **NOT:**
  **http://api.example.com/inventory-management/**
  **managedEntities/{id}/installScriptLocation**

  **BUT:**
  **http://api.example.com/inventory-management/**
  **managed-entities/{id}/install-script-location**

# Consistency is the key (cont.)



## Do not use underscores ( _ )

- possible that the underscore (_) character can get partially obscured or completely hidden in some browsers or screens
- use hyphens (-) instead
- examples

  **NOT:**
  **http://api.example.com/inventory_management/managed_entities/{id}/install_script_location**

  **BUT:**
  **http://api.example.com/inventory-management/managed-entities/{id}/install-script-location**

# Consistency is the key (cont.)



## Use lowercase letters in URIs

- lowercase letters should be consistently preferred in URI paths
- RFC 3986 defines URIs as case-sensitive except for the scheme and host components
- examples

  http://api.example.org/my-folder/my-doc //1

  HTTP://API.EXAMPLE.ORG/my-folder/my-doc //2

  http://api.example.org/My-Folder/my-doc //3

  in above examples, 1 and 2 are the same but 3 is not as it uses My-Folder in capital letters

# Consistency is the key (cont.)



## Do not use file extensions

- looks bad and does not add any advantage
- removing them decreases the length of URIs
- if you want to highlight the media type of API using file extension then rely on the media type communicated through the Content-Type header
- examples:

  NOT:
  http://api.example.com/device-management/managed-devices.xml

  BUT:
  http://api.example.com/device-management/managed-devices

# Never use CRUD function names in URIs

## URIs should not indicate that a CRUD function is performed

## URIs should be used to uniquely identify resources and not any action upon them

## HTTP request methods should be used to indicate which CRUD function is performed

## Examples:

**//get all devices**
**HTTP GET http://api.example.com/device-management/managed-devices**

**//create new device**
**HTTP POST http://api.example.com/device-management/managed-devices**

**//get device for given id**
**HTTP GET http://api.example.com/device-management/managed-devices/{id}**

**//update device for given id**
**HTTP PUT http://api.example.com/device-management/managed-devices/{id}**

**//delete device for given id**
**HTTP DELETE http://api.example.com/device-management/managed-devices/{id}**

# Use query component to filter URI collection

## In case you need a collection of resources sorted, filtered or limited based on some certain resource attribute(s)

## Do not create new APIs – rather enable in resource collection API

## Pass the input parameters as query parameters

## Examples

**http://api.example.com/device-management/managed-devices**

**http://api.example.com/device-management/managed-devices?region=USA**

**http://api.example.com/device-management/managed-devices?region=USA&brand=XYZ**

**http://api.example.com/device-management/managed-devices**
**?region=USA&brand=XYZ&sort=installation-date**

## XML 3.1

- XML = Extensible Markup Language
- a universal way of structuring text documents
- XML 1.0 (1998/2000)
- from World Wide Web Consortium (W3C)
- markup language, cf. SGML, HTML,...
- objectives:
  independent, interoperable, simple, human readable,...
- importance of XML
  - data definition (well formed)
  - data validation (valid)
  - data interchange
  - hierarchical
  - namespaces

# XML example

```
<?xml version="1.0" encoding="UTF-8"?>
<CourseParticipants>
   <Participant>
      <firstName>Maria</firstName>
      <lastName>Meuris</lastName>
      <age>52</age>
   </Participant>
   <Participant>
      <firstName>Bert</firstName>
      <lastName>Mak</lastName>
      <age>34</age>
   </Participant>
   <Participant>
      <firstName>Peter</firstName>
      <lastName>Buenk</lastName>
      <age>26</age>
   </Participant>
</CourseParticipants>
```

# JSON

## JavaScript Object Notation:

- ECMA international standard since 2013 (but already exists longer)
- open-standard file format
- serialization of data/objects (simpler than XML)
- also used outside JavaScript
- used in REST, JSON-RPC, Ajax,...

# JSON syntax

- name-value pairs, enclosed by curly braces ({}), to form objects
- name-value pairs are separated by commas (,)
- colon (:) between name and value
- name must be a string, and put between double quotes (")
- values must be one of the following data types:
  - a string
  - a number
  - an object (JSON object)
  - an array
  - a boolean
  - null
- curly braces ({}) also used to denote further nesting
- square brackets ([]) hold arrays

# JSON example

```
[
    {"firstName": "Maria","lastName": "Meuris", "age": 52},
    {"firstName": "Bert",  "lastName": "Mak", "age": 34},
    {"firstName": "Peter","lastName": "Buenk", "age": 26}
]
```

**Introduction to REST and JSON**

1. Application Development and Distributed Computing
2. RESTful Web Services / APIs
3. XML and/vs. JSON

# Spring REST basics

**Objectives :**

- **Spring's REST basics: Hello World REST API**
- **Testing APIs**

# Spring REST basics



- builds upon Spring MVC
- controllers can handle requests for all HTTP methods. Spring 3.2 and higher also supports the PATCH method
- **@PathVariable** annotation enables controllers to handle requests for parameterized URLs
- resources can be represented in a variety of ways using Spring views and view resolvers, including implementations for rendering model data as **XML, JSON (default), Atom, and RSS**
- view-based rendering can be bypassed altogether using the **@ResponseBody** annotation and various HttpMethodConverter implementations
- **@RequestBody** annotation, along with HttpMethodConverter implementations, can convert inbound HTTP data into Java objects passed in to a controller's handler methods
- applications can consume REST resources using **RestTemplate or WebClient**
- Spring Boot helps to simplify the configuration

# Configuration





- **Maven Dependencies**
  - **Spring MVC / REST + tomcat-starter + jackson-bind + spring-web**

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

- **Packaging**
  - **jar: embedded Tomcat Server will be used**
  - **war: can deploy on own server (but extra configuration needed)**

# Spring Boot auto-configuration

- only have to write the controllers, rest is handled automatically
- use application.properties (in src/main/resources) to set up the port and context root of your web application

```
server.port=8080
server.servlet.context-path=/helloapi
```

# Writing a Controller



## The Basics

- configuration is annotation-based (main one: @RestController)
- @RequestMapping maps to the url requested by the client
  - default HTTP method = GET
  - also possible to use @GetMapping instead
- wire in your Service class(es)
- request-handling method can have any name
- return will be object
  - text/plain when returning String
  - application/json when returning other object type

## Accessing Request Input

- use @RequestParam attribute in controller method
- pass in URL via localhost:8080/helloapi/welcome**?name=John**

# Hello World REST API





- **Controller**

```
@RestController
public class HelloRestController {

    @Autowired
    HelloService helloService;

    @GetMapping("/welcome")
    public String returnMessage(@RequestParam String name){
        Guest guest = new Guest(name);
        return helloService.sayHelloTo(guest);
    }

    @GetMapping("/welcomeGuest")
    public Guest returnGuest(@RequestParam int id){
        return helloService.findGuest(id);
    }
}
```

- **Test via:**
  - **localhost:8080/helloapi/welcome?name=John**
  - **localhost:8080/helloapi/welcomeGuest?id=2**

## SoapUI 2.1

**Open-source web service testing application for service-oriented architecture (SOA) and representational state transfer (REST)**

**Owned by SmartBear Software**

**Core features:**

- **web services inspection**
- **web services invoking**
- **web services development**
- **web services simulation and mocking**
- **web services functional, compliance and security testing**

**SoapUI Pro (commercial enterprise version) adds a number of productivity enhancements to the SoapUI core**

**https://www.soapui.org/downloads/soapui.html**

# Postman





## Postman company

## Originally only as Chrome plugin (deprecated), nowadays a standalone app

## More REST oriented, but SOAP can be tested as well (with some effort)

## Need to create a (free) account

## Postman Pro and Postman Enterprise add extra features

## https://www.getpostman.com/downloads/

# More REST API functionality

**Objectives :**

- **Defining REST resources with @RequestMapping**
- **Mapping the HTTP request methods**
- **Request parameters: @PathVariable, @RequestParam and @RequestBody**
- **More JSON mapping**
- **Producing and Consuming XML**

# Defining RESTful Resources



## Basics of @RequestMapping

- URI path to which a resource class or method will respond
- value is relative to URI of the server where REST resource is hosted
- annotation can be applied at both the class and the method levels

## Specifying @RequestMapping on a controller class or method

- path defined on class level is relative to the base path

```
@RestController
@RequestMapping("guests")
public class RestReceptionController {  }
```

- base path typically takes the following URI pattern:
  http://host:port/<context-root>
- path defined on method level is relative to the path defined at the class level

```
@RequestMapping(path = "/welcome")
public String returnMessage(@RequestParam("name") String name) { }
```

URI:

http://localhost:8081/springrest/guests/welcome?name=John

# Defining RESTful Resources (cont.)

## Specifying variables in the URI path template

- path template allows you to define variables that appear as placeholders in the URI
- define variable between curly braces ({ })
- variable will be linked with @PathVariable

```
@RequestMapping("{id}")
public Guest returnGuestById(@PathVariable("id") int id) {
    return receptionService.findGuest(id);
}
```

- variable will be replaced at runtime with the values set by the client

  http://localhost:8081/springrest/guests/**2**

# Mapping the HTTP Request Methods



## Two ways to pass HTTP method

- @RequestMapping(path="", method=RequestMethod.GET)
  (or POST, PUT, DELETE instead of GET)
- abbreviated syntax (since Spring 4.3)
  @GetMapping, @PostMapping, @PutMapping, @DeleteMapping

# @GetMapping



- for retrieving the resources referenced in the URI path
- often used together with the request parameters
- return Object
- Object will by default be transformed to JSON
- example:

```
@GetMapping("{id}")
public Guest returnGuestById(@PathVariable int id) {
    return receptionService.findGuest(id);
}
@GetMapping("")
public List<Guest> returnall(){
    return receptionService.findAllGuests();
}
@GetMapping("/query")
public Guest returnGuestByName(@RequestParam String name) {
    return receptionService.findGuest(name);
}
```

URI:

http://localhost:8081/springrest/guests/1
http://localhost:8081/springrest/guests
http://localhost:8081/springrest/guests/query?name=John

# @PostMapping



- to send data to the server (new object to be created)
- linked with @RequestBody
- transforms the incoming JSON to the Object type specified
- no-arg constructor needs to be present in model class!
- return void
- example

```
@PostMapping("")
public void addGuest(@RequestBody Guest guest){
    receptionService.addGuest(guest);
}
```

JSON passed:

```
{"id": 3, "name": "Bart", "age": 43 }
```

URI:

http://localhost:8081/springrest/guests

- cannot be tested any more by simply putting URI in browser!

# @PutMapping



- for updating the resource pointed to by the URI
- generally has a message body carrying the payload
- payload will be mapped to the method parameter as appropriate by the framework
- return void
- example:

```
@PutMapping("{id}")
public void changeGuest(@PathVariable int id, @RequestBody Guest guest) {
    receptionService.changeGuest(guest);
}
```

JSON passed:

```
{"id": 3, "name": "Bart", "age": 34 }
```

URI:

http://localhost:8081/springrest/guests/3

- remark: whole object passed!
- alternative: @PatchMapping

# @DeleteMapping



- to delete a resource
- pass the id of the resource
- return void
- example

```
@DeleteMapping("{id}")
public void deleteGuest(@PathVariable int id){
    receptionService.deleteGuest(id);
}
```

URI:

http://localhost:8081/springrest/guests/3

# Accessing Request Data



## Three possibilities to pass data in request

- @PathVariable
- @RequestParam
- @RequestBody

# @PathVariable



- will be added on the URL requested
  http://localhost:8081/springrest/guests/1
- passed via @RequestMapping variable
  @RequestMapping("{id}")
- pass to method via:
  public Guest findGuest(@PathVariable("id") int id)

# @RequestParam



- will be added after the ? in the URL

  http://localhost:8081/springrest/guests/query?name=John

- pass to method via:

```
public Guest returnGuestByName(@RequestParam("name") String name) {
    return receptionService.findGuest(name);
}
```

- parameter is required, unless a defaultValue is specified

```
@GetMapping(path = "/query")
public Guest returnGuestByName
        (@RequestParam(value="name", defaultValue="An") String name) {
    return receptionService.findGuest(name);
}
```

URI:

http://localhost:8081/springrest/guests/query

# @PathVariable vs. @RequestParam - When to use which?

## General remarks

- no official standard rules, but good practices
- @PathVariable and @RequestParam can be used together

## @PathVariable

- to link to a specific resource
- to denote hierarchy
- links can be easily bookmarked and cached
- parameters are mandatory

## @RequestParam

- for filtering, pagination
- for none-hierarchical data

# @RequestBody



- indicates a method parameter should be bound to the body of the request
- maps the HttpRequest body to an object, enabling automatic deserialization of the inbound HttpRequest body onto a Java object
- Spring automatically deserializes the JSON into a Java type assuming an appropriate one is specified
- by default, the type we annotate with the @RequestBody annotation must correspond to the JSON sent from our client-side
- other media-types can be specified as well via the consumes attribute on @RequestMapping

# Returning Responses



## @ResponseBody



- tells a controller that the object returned is automatically serialized into JSON and passed back into the HttpResponse object
- could be used on top of a controller method
- example

```
@GetMapping("{id}")
@ResponseBody
public Guest returnGuestById(@PathVariable("id") int id) {
    return receptionService.findGuest(id);
}
```

- actually not needed any more since

  @RestController = @Controller + @ResponseBody

- other media-types can be specified as well via the produces attribute on @RequestMapping

# More JSON Mapping



## 3 techniques to modify JSON outcome

- annotations
- adding wrapper elements for lists
- programmatic mapping with ObjectMapper class

## Based on Jackson library, which is automatically included in spring-boot-starter-web

# Jackson Annotations



## Library: com.fasterxml.jackson.annotation

## Different categories, depending on the goal of the annotation

- property naming and inclusion

  @JsonProperty, @JsonIgnore, @JsonIgnoreProperties

- deserialization and serialization details

  @JsonFormat, @JsonUnwrapped, @JsonPropertyOrder, @JsonRootName

- ...

## Placed on class or property level

## More info, see:

https://github.com/FasterXML/jackson-annotations/wiki/Jackson-Annotations
https://www.javadoc.io/doc/com.fasterxml.jackson.core/jackson-annotations

# Jackson Annotations - Example

```
@JsonRootName("guest")
@JsonPropertyOrder({"id","name","enroldate"})
@JsonIgnoreProperties(ignoreUnknown=true)
public class Guest {
    private int id;
    @JsonProperty("first")
    private String name;
    @JsonIgnore
    private int age;
    @JsonFormat(pattern="dd/MM/yyyy")
    private LocalDate enrolDate=LocalDate.now();
    public Guest(){}
    public Guest(int id, String name, int age) {
        this.id=id;
        this.name = name;
        this.age=age;
    }
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public int getAge() { return age; }
    public void setAge(int age) { this.age = age; }
    public LocalDate getEnrolDate() { return enrolDate; }
    public void setEnrolDate(LocalDate enrolDate) { this.enrolDate = enrolDate; }
}
```

# Adding a Wrapper Element around a List of Objects

5.2

## JSON list does not get an associated wrapper element by default

## Solution:

- create an extra class that contains a list of the object wanted
- annotate the class with @JsonRootName
- add some (de)serialization variables in application.properties
- return the wrapper object from the RestController

## This technique also needs to be applied when using JAXB

# Example

## Guests.java

```
@JsonRootName("guests")
public class Guests {
    private List<Guest> guests;
    public Guests() {}
    public Guests(List<Guest> guests) { this.guests = guests; }
    public List<Guest> getGuests() { return guests; }
    public void setGuests(List<Guest> guests) { this.guests = guests; }
}
```

## application.properties

```
spring.jackson.serialization.WRAP_ROOT_VALUE=true
spring.jackson.deserialization.UNWRAP_ROOT_VALUE=true
spring.jackson.serialization.indent-output=true
```

## RestReceptionController.java

```
@GetMapping(path="all")
public Guests returnallGuestsViaWrapper(){
    return new Guests(receptionService.findAllGuests());
}
```

# Jackson ObjectMapper





## Library: com.fasterxml.jackson.databind

## Used to (de)serialize JSON String to/from Object programmatically

## Methods:

- readValue()
- writeValue(), writeValueAsString()
- configure()  -> adds extra (de)serialization options

## Also possible to read into a list using

```
new TypeReference<List<Guest>>(){}
```

# ObjectMapper - Example

```
@JsonRootName("person")
@JsonPropertyOrder({"id","name","enroldate"})
@JsonIgnoreProperties(ignoreUnknown=true)
public class Guest {
    private int id;
    @JsonProperty("first name")
    private String name;
    @JsonIgnore
    private int age;
    @JsonFormat(pattern="dd/MM/yyyy")
    private LocalDate enrolDate=LocalDate.now();
    public Guest(){ }
    public Guest(int id, String name, int age) {
        this.setId(id);
        this.name = name;
        this.age=age;
    }
    //getters and setters
}
```

# ObjectMapper - Example (cont.)

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class Ch3ObjectMapperApplicationTests {
    @Autowired AbisReceptionService serv;
    @Test
    public void testMapperRead() {
        String jsonString =
            "{\"person\" : {\"id\" : 5,\"first name\" : \"John\",\"enrolDate\" : \"11/06/2019\"}}";
        ObjectMapper om = new ObjectMapper();
        om.configure(DeserializationFeature.UNWRAP_ROOT_VALUE, true);
        try {
            Guest g = om.readValue(jsonString,Guest.class);
            assertEquals("John",g.getName());
        } catch (IOException e) { e.printStackTrace(); }
    }
    @Test
    public void testMapperWrite() {
        Guest g = serv.findGuest(1);
        ObjectMapper om = new ObjectMapper();
        om.configure(SerializationFeature.WRAP_ROOT_VALUE, true);
        try {
            String s = om.writeValueAsString(g);
            System.out.println(s);
        } catch (JsonProcessingException e) { e.printStackTrace(); }
    }
}
```

# Producing and Consuming XML



- possible to specify which media types a resource can produce or consume by using the produces and consumes attributes on the @...Mapping annotations
- consumes mostly linked with @PostMapping and @PutMapping
- produces and consumes can also be placed at class level
- method-level attributes override the class-level attributes
- Maven dependency:

```
<dependency>
        <groupId>com.fasterxml.jackson.dataformat</groupId>
        <artifactId>jackson-dataformat-xml</artifactId>
</dependency>
```

- will use the Jackson mappings added before, but can give “strange” results
- extra mappings can be added to map java objects to the correct representation using JAXB

# Using JAXB for mapping XML

6.1

- Java Architecture for XML binding
- conversion of XML to and from Java types
- marshal/unmarshal XML content into/from Java representation
- access, update and validate Java representation against schema constraint
- javax.xml.bind package
- uses annotations to map Objects to XML elements
  - @XmlRootElement
  - @XmlAccessorType
  - @XmlElement
  - @XmlTransient
  - ...
- automatically supported in Spring REST

# JAXB Jackson Mapping

## Jackson library adds annotations on top of JAXB

- @JacksonXmlRootElement
- @JacksonXmlProperty
- @JacksonXmlWrapper
- ...

## JSON annotations are also supported for XML

# JAXB Mapping - Example

```
<?xml version="1.0" encoding="UTF-8"?>
<persons>
    <person>
        <firstname>Maria</firstname>
        <lastname>Meuris</lastname>
    </person>
    <person>
        <firstname>Bert</firstname>
        <lastname>Mak</lastname>
    </person>
    <person>
        <firstname>Peter</firstname>
        <lastname>Buenk</lastname>
    </person>
</persons>
```

# JAXB Mapping - Example (cont.)

```
@JacksonXmlRootElement(localName="persons")
public class Persons {
    @JacksonXmlProperty(localName="person")
    @JacksonXmlElementWrapper(useWrapping = false)
    private ArrayList<Person> persons= new ArrayList<>();
    public ArrayList<Person> getPersons()  {return persons;}
    public void setPersons(ArrayList<Person> persons)  {this.persons = persons;}
}

@JacksonXmlRootElement(localName="person")
public class Person {
    private String firstName;
    @JacksonXmlProperty(localName="lastname")
    private String lastName;
    public String getFirstName()  {return firstName;}
    public void setFirstName(String firstName)  {this.firstName = firstName;}
    public String getLastName()  {return lastName;}
    public void setLastName(String lastName) {this.lastName = lastName;}
    public String toString()  {return firstName+ " " + lastName;}
}
```

# Producing and Consuming XML - Example

- **Guest.java with Jackson JAXB annotations**

```
@JacksonXmlRootElement(localName="guest")
public class Guest {

    private int id;
    private String name;
    private int age;

    public Guest(){}

    public Guest(int id, String name, int age) {
        this.setId(id);
        this.name = name;
        this.age=age;
    }

    public int getId() {return id;}
    public void setId(int id) {this.id = id;}
    public String getName() {return name;}
    public void setName(String name) {this.name = name;}
    public int getAge() {return age;}
    public void setAge(int age) {this.age = age;}

}
```

# Producing and Consuming XML - Example (cont.)

- **Controller**

```
@RestController
@RequestMapping(path="guests", produces="application/json" )
public class RestReceptionController {
    @Autowired ReceptionService receptionService;

    @GetMapping(path = "/query")
    public Guest returnGuestByName(
        @RequestParam(value="name", defaultValue="An") String name) {
            return receptionService.findGuest(name);
    }
    @GetMapping(path = "{id}", produces=MediaType.APPLICATION_XML_VALUE)
    public Guest returnGuestById(@PathVariable("id") int id) {
        return receptionService.findGuest(id);
    }
    @PostMapping(path="",consumes=MediaType.APPLICATION_XML_VALUE)
    public void addGuest(@RequestBody Guest guest){
        receptionService.addGuest(guest);
    }
}
```

# Producing and Consuming XML - Example (cont.)

- **Test URIs**

  **http://localhost:8081/springrest/guests/query?name=John (will return JSON)**

  **http://localhost:8081/springrest/guests/1 (will return XML)**

  **can use accept header to state which media-type to return**

  **POST: http://localhost:8081/springrest/guests**

  **xml entered :**

```
<guest>
    <id>3</id>
    <name>James</name>
    <age>25</age>
</guest>
```

# WebFlux and WebClient

**Objectives :**

- **Spring WebFlux**
- **Using WebClient to access an API**

# WebFlux



- reactive, non-blocking web framework introduced as an alternative to Spring MVC
- based on Project Reactor (Mono, Flux)
- enables:
  - high concurrency with fewer threads
  - event-loop execution model
  - backpressure support via Reactive Streams
- works with:
  - Netty, Undertow, or Servlet 3.1+ containers
  - functional and annotation-based programming models

# WebFlux (ctd.)

- when use WebFlux:
  - high-throughput I/O applications
  - real-time streaming or SSE
  - services calling other reactive services
  - to avoid thread-per-request overhead
  - not always needed / less useful for CPU-bound or simple CRUD apps
- configuration:
  - pom.xml: replace spring-boot-starter-web with spring-boot-starter-webflux
  - application.properties: replace server.servlet.context-path with spring.webflux.base-path

## Mono<T> 2.1

https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Mono.html

- represents 0 or 1 value
- used for standard return types
- create a Mono object via

```
Mono<Guest> guestMono =
Mono.fromCallable(() -> guestRepository.findGuest(id))     //method call returning result
        .subscribeOn(Schedulers.boundedElastic())            //multithreaded scheduler
Mono.fromRunnable(() -> guestRepository.addGuest(guest))
        .subscribeOn(Schedulers.boundedElastic())            //returns Mono<Void>
```

# Flux<T>





https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html

- represents 0 to many values
- used for streams, lists, server-sent events
- create a Flux object via

```
Flux<Guest> guests =
    Mono.fromCallable(guestRepository::findAll)
            .subscribeOn(Schedulers.boundedElastic()).
            .flatMapMany(Flux::fromIterable);          // single blocking call gets elements
                                                       one by one and converts into a Flux<Guest>

Flux<Guest> guests =
    Flux.fromCallable(guestRepository::findAll)
            .subscribeOn(Schedulers.boundedElastic()).
            .flatMapIterable(list -> list);          // converts returned List<Guest> into a
                                                     Flux<Guest> that emits one element at a time
```

# Calling Reactive Methods





- when simply calling the method, nothing is executed yet!
- reactive methods return Publisher objects
- 2 ways of handling:
  - subscribe()
  - block()
- subscribe()
  - execution begins asynchronously
  - calling thread continues immediately
  - provide callbacks: onNext, onError, onComplete
  - example:

```
guestService.findGuest(2)
        .subscribe(guest -> System.out.println("Received: " + guest))
```

# Calling Reactive Methods (ctd.)



- **block()**
  - **waits synchronously for completion**
  - **rreezes the thread until the result arrives**
  - **breaks non-blocking WebFlux behavior**
  - **examples:**

    **Mono -> guestService.findGuest(2).block()**

    **Flux -> guestService.findAllGuests().collectList().block()**

- **In JUnit tests: always use block(), since any asynchronous call is probably not answered before the test method finishes**

# Example

## AbisGuestService.java

```
@Service
public class AbisGuestService implements GuestService {
    @Autowired private GuestRepository guestRepository;
    public Flux<Guest> getAllGuests() {
        return Mono.fromCallable(guestRepository::getAllGuests)
                        .subscribeOn(Schedulers.boundedElastic())
                        .flatMapMany(list -> Flux.fromIterable(list));
    }

    public Mono<Guest> findGuest(int id) {
        return Mono.fromCallable(() -> guestRepository.findGuest(id))
                        .subscribeOn(Schedulers.boundedElastic());
    }

    public Mono<Void> addGuest(Guest g){
         return Mono.fromRunnable(() ->  guestRepository.addPerson(g));
    }
}
```

# Example (ctd.)

## GuestController.java

```
@RestController
@RequestMapping("guests")
public class GuestController {

    @Autowired
    GuestService gs;

    @GetMapping("")
    public Flux<Guest> getAllGuests() {
        return gs.getAllPersons();
    }

    @GetMapping("{id}")
    public Mono<Guest> findGuest(@PathVariable int id) {
        return gs.findGuest(id);
    }

    @PostMapping("")
    public Mono<Void> addGuest(@RequestBody Guest g) {
        return  gs.addGuest(g);
    }
}
```

# Example (ctd.)

## GuestServiceTest.java

```
@SpringBootTest
class GuestServiceTest {
    @Autowired
    GuestService gs;
    @Test
    void guest1IsJohn() {
        assertEquals("John",gs.findGuest(1).block().getFirstName());
    }

    @Test
    public void numberOfGuestsIs4() {
        int size =gs.getAllGuests().collectList().block().size();
        assertEquals(4,size);
    }
    @Test
    public void addingNewGuestWorks(){
        Guest newGuest = new Guest(8,“An”,”latte”);
        gs.addGuest(newGuest).block();
    }
}
```

# WebClient



- modern non-blocking HTTP client introduced in Spring 5 as a replacement for RestTemplate
- part of Spring WebFlux, built on top of Project Reactor
- supports both:
  - synchronous (blocking) and
  - asynchronous (non-blocking, reactive) calls
- ideal for making REST API calls, especially when dealing with high concurrency or reactive applications
- needs spring-boot-starter-webflux dependency

# Creating a WebClient instance



- **basic instance**

```
WebClient webClient = WebClient.create("localhost:8080/guestapi");
```

- **builder with customization**

```
WebClient webClient = WebClient.builder()
        .baseUrl("ocalhost:8080/guestapi")
        .defaultHeader(HttpHeaders.CONTENT_TYPE,
                                    MediaType.APPLICATION_JSON_VALUE)
        .defaultHeader(HttpHeaders.AUTHORIZATION, "Bearer your-token")
    .build();
```

- **spring bean (which can then be autowired)**

```
@Bean
public WebClient webClient(WebClient.Builder builder) {
    return builder
        .baseUrl("https://api.example.com")
        .defaultHeader(HttpHeaders.CONTENT_TYPE,
                    MediaType.APPLICATION_JSON_VALUE)
        .build();
}
```

# Getting Data



- get() method will launch a request to the server
- uri() will pass the path as a String, or as a UriBuilder object when adding query parameters, headers,...
- retrieve() will fetch the result
- bodyToMono() or bodyToFlux() will convert the JSON into your object class
- use subscribe/block to handle the result afterwards
- examples:

```
Mono<User> userMono = webClient.get()
    .uri("/guests/{id}", 1)
     .retrieve()
     .bodyToMono(Guest.class);

List<User> users = webClient.get()
    .uri(uriBuilder -> uriBuilder.path("/guests")
                                 .queryParam("name", "John")
                                 .build())
    .retrieve()
    .bodyToFlux(User.class)
    .collectList()
    .block();
```

# Posting Data



- create a new instance of the type you want to post (could be retrieved via a web form)
- post() will launch the request
- set uri() as before
- use bodyValue() to pass the body of the request
- retrieve() and handle the result as before
- examples:

```
Guest newGuest = new Guest(8,“An”,”latte”);
webClient.post()
        .uri("/guests")
        .bodyValue(newUser)
        .retrieve()
        .bodyToMono(Void.class)
```

# Updating Data



- change the instance you want to update (via setter methods,...)
- call the **put()** or patch() method on WebClient
- pass body, retrieve result like before
- examples:

```
Guest guest = new Guest(8,“An”,”latte”);
guest.setCoffeePreference(”black”);
webClient.put()
    .uri("/users/{id}", 1)
    .bodyValue(guest)
    .retrieve()
    .bodyToMono(Void.class);
CoffeePreference cp = new CoffeePreference(”espresso”));
webClient.patch()
    .uri("/users/{id}/coffeepreference", 1)
    .bodyValue(cp)
    .retrieve()
    .bodyToMono(Void.class);
```

# Deleting Data



- call delete() on the webClient
- pass the id via the uri
- retrieve as before
- example

```
webClient.delete()
        .uri("/users/{id}", 1)
        .retrieve()
        .bodyToMono(Void.class);
```

# Exception Handling and Validation

**Objectives :**

- **WebFlux Exception Handling**
- **HTTP status codes**
- **ResponseEntity and ProblemDetail classes**
- **REST Exception Handling**
- **Validation**

# WebFlux Exception Handling



- exceptions will have to be passed as reactor errors
- unchecked (RuntimeException) that can carry other checked exceptions
- created with Mono.error or Flux.error

  Mono.error(new GuestNotFoundException("guest not found"))
- coming up:
  - throw new WebFlux exceptions
  - pass non-reactive exceptions as WebFlux exceptions
  - tesing WebFlux exceptions

# Creating new WebFlux exceptions

1.1

- Mono
  - original method is reactive

```
guestRepository.findGuest(id)
    .switchIfEmpty(Mono.error(new GuestNotFoundException("guest not found")));
```

  - original method is not reactive

```
Mono.justOrEmpty(guestRepository.findGuest(id))
    .switchIfEmpty(Mono.error(new GuestNotFoundException("guest not found")));
```

- Flux
  - original method is reactive

```
guestRepository.findGuests(coffeePreference)
    .switchIfEmpty(Flux.error
        new NoLikersFoundException("No persons like " + coffeePreference)
    ));
```

  - original method is not reactive

```
Mono.fromCallable(() -> guestRepository.findGuests(coffeePreference))
    .subscribeOn(Schedulers.boundedElastic())
    .flatMapMany(Flux::fromIterable)
    .switchIfEmpty(Flux.error
        new NoLikersFoundException("No persons like " + coffeePreference)
 ));
```

# Passing non-reactive exceptions



- use Mono.fromCallable() to pass the exception
- exception is automatically transformed into a reactive exception
- examples:

```
public Mono<Guest> findGuest(int id) {
    return Mono.fromCallable(() -> guestRepository.findGuest(id));
}


public Mono<Void> deleteGuest(int id) {
    return Mono.rfomCallable(() -> {
        guestRepository.deleteGuest(id);
        return null;
    });
}
```

# Testing WebFlux exceptions



- exception will be passed as a RuntimeException
- original thrown exception will also be passed inside, but can only be tested after capturing the original exception
- example:

```
Throwable ex = assertThrows(RuntimeException.class, ()-> gs.findGuest(500).block());
assertTrue(ex.getCause() instanceof GuestNotFoundException);
```

- extra test classes can be added via project reactor
  - add Maven dependency

```
<dependency>
    <groupId>io.projectreactor</groupId>
    <artifactId>reactor-test</artifactId>
    <scope>test</scope>
</dependency>
```

  - example

```
Mono<Guest> result = gs.findGuest(500);
StepVerifier.create(result)
    .expectErrorSatisfies(ex -> {
        assertTrue(ex instanceof GuestNotFoundException);
     })
     .verify();
```

# HTTP Status Codes





## Not Java of course but necessary to understand for

- exception handling in our services/clients
- notifying the client when a post/put/delete was processed
- debugging

## Different categories

- 1xx (informational): request was received, continuing process
- 2xx (successful): request was successfully received, understood, and accepted
- 3xx (redirection): further action needs to be taken in order to complete the request
- 4xx (client error): request contains bad syntax or cannot be fulfilled
- 5xx (server error): server failed to fulfil an apparently valid request

# HTTP Status Codes - 2xx Success





## 200: OK

- standard response for successful HTTP requests
- actual response will depend on the request method used
- in a GET request, response will contain an entity corresponding to the requested resource
- in a POST request, the response will contain an entity describing or containing the result of the action

## 201: Created

- request has been fulfilled, resulting in creation of a new resource

## 202: Accepted

- request accepted for processing, but processing has not been completed
- request might or might not be eventually acted upon, and may be disallowed when processing occurs

# HTTP Status Codes - 2xx Success (cont.)

## 203: Non-Authoritative Information

- server is a transforming proxy that received a 200 OK from its origin, but is returning a modified version of the origin's response

## 204: No Content

- request successfully processed, but no content is returned

## 205: Reset Content

- request successfully processed, but no content is returned
- unlike a 204 response, this response requires that the requester resets the document view

# HTTP Status Codes - 2xx Success (cont.)

## 206: Partial Content

- only part of the resource delivered due to arrange header sent by the client
- range header is used by HTTP clients to enable resuming of interrupted downloads, or split a download into multiple simultaneous streams

## 207: Multi-Status

- message body that follows is by default an XML message and can contain a number of separate response codes, depending on how many sub-requests were made

# HTTP Status Codes - 4xx: Client Errors



## 400: Bad Request

- server cannot or will not process the request due to an apparent client error (e.g., malformed request syntax, size too large)

## 401: Unauthorized

- similar to 403 Forbidden, but specifically for use when authentication is required and has failed or has not yet been provided
- response must include a WWW-Authenticate header field containing a challenge applicable to the requested resource
- semantically means "unauthenticated",i.e. the user does not have the necessary credentials

## 402: Payment Required (reserved for future use)

- original intention was that it might be used as part of some form of digital cash or micro payment scheme
- Google Developers API uses this status if a particular developer has exceeded the daily limit on requests

# HTTP Status Codes - 4xx: Client Errors (cont.)

## 403: Forbidden

- request was valid, but the server is refusing action
- user might not have the necessary permissions for a resource, or may need an account of some sort

## 404: Not Found

- requested resource could not be found but may be available in the future
- subsequent requests by the client are permissible

## 405: Method Not Allowed

- request method is not supported for the requested resource, e.g. a GET request on a form that requires data to be presented via POST, or a PUT request on a read-only resource

# HTTP Status Codes - 4xx: Client Errors (cont.)



## 406: Not Acceptable

- requested resource is capable of generating only content not acceptable according to the Accept headers sent in the request

## 407: Proxy Authentication Required

- client must first authenticate itself with the proxy

## 408: Request Timeout

- server timed out waiting for the request

## 409: Conflict

- request could not be processed because of conflict in the current state of the resource, such as an edit conflict between multiple simultaneous updates

# HTTP Status Codes - 4xx: Client Errors (cont.)



## 410: Gone

- resource requested is no longer available and will not be available again
- should be used when a resource has been intentionally removed and the resource should be purged

## 411: Length Required

- request did not specify the length of its content, which is required by the requested resource

## 412: Precondition Failed

- server does not meet one of the preconditions that the requester put on the request

## 413: Payload Too Large

- request is larger than the server is willing or able to process

# HTTP Status Codes - 4xx: Client Errors (cont.)

## 414: URI Too Long

- URI provided was too long for the server to process
- often result of too much data being encoded as a query-string

## 415: Unsupported Media Type

- request entity has a media type which the server or resource does not support

## 418: I'm a teapot

- defined in 1998 as one of the traditional IETF April Fools' jokes
- RFC specifies this code should be returned by teapots requested to brew coffee
- used as an Easter egg in some websites, including Google

# HTTP Status Codes - 4xx: Client Errors (cont.)

## 422: Unprocessable Entity

- request was well-formed but was unable to be processed due to semantic errors

## 429: Too Many Requests

- user has sent too many requests in a given amount of time
- intended for use with rate-limiting schemes

## 431: Request Header Fields Too Large

- server can’t process request because an individual header field, or all the header fields collectively, are too large

## 451: Unavailable For Legal Reasons

- server operator has received a legal demand to deny access to a resource or to a set of resources that includes the requested resource
- code 451 was chosen as a reference to the novel Fahrenheit 451

# HTTP Status Codes - 5xx: Server Errors



## 500: Internal Server Error

- generic error message, given when an unexpected condition was encountered and no more specific message is suitable

## 501: Not Implemented

- server either does not recognize the request method, or it lacks the ability to fulfil the request
- usually this implies future availability (e.g. new feature of a web service API)

## 502: Bad Gateway

- server was acting as a gateway or proxy and received an invalid response from the upstream server

## 503: Service Unavailable

- server is currently unavailable (because it is overloaded or down for maintenance)
- generally, this is a temporary state

# HTTP Status Codes - 5xx: Server Errors

## 504: Gateway Timeout

- server was acting as a gateway or proxy and did not receive a timely response from the upstream server

## 505: HTTP Version Not Supported

- server does not support the HTTP version used in the request

## 507: Insufficient Storage

- server is unable to store the representation needed to complete the request

## 508: Loop Detected

- server detected an infinite loop while processing the request

## 511: Network Authentication Required

- client needs to authenticate to gain network access
- intended for use by intercepting proxies used

# REST Exception Handling



- exceptions from API are not passed correctly by default
  - when no result found, empty page will be returned and response status is still 200 (OK) -> status should be 404 (?), and response body should carry an error message
  - exceptions thrown from the controller will give an Internal Server Error (500)
- Spring offers some options for dealing with exceptions
  - creation of Error objects that can carry the error messages
  - use ResponseEntity that carries Error object and HttpStatus
  - create a centralized exception handler (@RestControllerAdvice)

# The ResponseEntity class



- represents the whole HTTP response: status code, headers, and body
- can use it to fully configure the HTTP response
- ResponseEntity is a generic type, so we can use any type as the response body (even void)
- also possible to use builder to create the ResponseEntity object
- can be used in combination with WebFlux
- can be used for the translation into info messages and exceptions by the client
- used in @RestController methods

# ResponseEntity - Example

## RestReceptionController.java (API)

```
@GetMapping(path = "{id}")
public ResponseEntity<Guest> returnGuestById(@PathVariable("id") int id) {
    Guest g = receptionService.findGuest(id);
    ResponseEntity<Guest> re = ResponseEntity.ok().body(g);
    return re;
}
@GetMapping(path="", produces=MediaType.APPLICATION_XML_VALUE)
public ResponseEntity<?> returnall(){
    Guests g = new Guests();
    g.setGuests(receptionService.findAllGuests());
    HttpHeaders responseHeaders = new HttpHeaders();
    responseHeaders.add("content-type", MediaType.APPLICATION_XML_VALUE);
    return new ResponseEntity<Guests>(g,responseHeaders,HttpStatus.OK);
}
@PostMapping(path="",consumes=MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<Void> addGuest(@RequestBody Guest guest){
    receptionService.addGuest(guest);
    return ResponseEntity.noContent().build();
}
```

# ProblemDetail Class



- **not only pass HTTP status code when something goes wrong**
- **use object to pass more information about the error**
- **client can than interpret the message**
- **RFC 7807 provides a standard format for returning problem details from HTTP APIs, represented by the ProblemDetail class**
  - **error responses MUST use standard HTTP status codes in the 400 or 500 range to detail the general category of error**
  - **error responses will be of the content-type application/problem, appending a serialization format of either json or xml: application/problem+json, application/problem+xml**
  - **error responses will have each of the following keys:**
    - title (String): short, human-readable title for the general error
    - statuscode (int): HTTP status code of the error
    - detail (String): human-readable description of the specific error
    - type (String, optional): URL to a document describing the error
    - instance(String, optional):
      URI pointing to error log for that specific response

# Exception Handling with ResponseEntity



- ResponseEntity can be used to pass a HttpStatus.NOT_FOUND (404) (or another code) when some exception occurs
- ProblemDetail passed as the entity, client checks on the 4xx
- ProblemDetail can only be passed in JSON! Create a custom error class in case you need to return XML
- can used with or without WebFlux
- WebClient will check the error via the onStatus() method after doing the retrieve(), and can then pass the error further, or convert it to a custom exception again
- approach works, but all controller methods will have to return ResponseEntity polluting your code

# Exception Handling with ResponseEntity - API Example

## GuestNotFoundException.java

```
public class GuestNotFoundException extends Exception {
    public GuestNotFoundException(String message) {
        super(message);
    }
}
```

## AbisReceptionService.java

```
@Service
public class AbisReceptionService implements ReceptionService {
    @Autowired GuestRepository gr;
    @Override
    public Guest findGuest(int id) throws GuestNotFoundException {
        return gr.findGuest(id);
    }
    public Mono<Guest> findGuest(String name) {
        return Mono.fromCallable(() -> guestRepository.findGuest(name));
    }
}
```

# Exception Handling with ResponseEntity - API Example (cont.)

## RestReceptionController.java

```
public class RestReceptionController {
    @Autowired  ReceptionService receptionService;
    @GetMapping("{id}")
    public ResponseEntity<?> returnGuest(@PathVariable int id) {
        try {
            Guest g = receptionService.findGuest(id);
            return  ResponseEntity.ok().body(g);
        } catch (GuestNotFoundException e) {
            ProblemDetail problem = ProblemDetail.forStatus(HttpStatus.NOT_FOUND;
            problem.setTitle("Guest Not Found");
            problem.setDetail(e.getMessage());
            return ResponseEntity.status(HttpStatus.NOT_FOUND).body(problem);
        }
    }
    @GetMapping("/query")
    public Mono<ResponseEntity<Object>> returnGuest(@RequestParam String name) {
        receptionService.findGuest(name);
            .map(ResponseEntity::ok)
            .onErrorResume(GuestNotFoundException.class, ex -> {
                ProblemDetail problem = ProblemDetail.forStatus(HttpStatus.NOT_FOUND;
                problem.setTitle("Guest Not Found");
                problem.setDetail(e.getMessage());
                return Mono.just(ResponseEntity.status(HttpStatus.NOT_FOUND)
                                                .body(problem));
         });
}}
```

# Exception Handling with ResponseEntity - Client Example

## MyClientService.java

```
@Service
public class MyClientService {
    WebClient guestClient = WebClient.create("http://localhost:8081/guestapi");
    public Course findGuestById(int id) {
        return guestClient.get()
                .uri("/guest/{id}",id)
                .retrieve()
                .onStatus(status -> status.is4xxClientError(),
                        resp -> resp.bodyToMono(ProblemDetail.class)
                   .flatMap(pd -> Mono.error(new GuestNotFoundException(pd.getDetail()))))
                .bodyToMono(Course.class)
                .block();
    }
    public Mono<Course> findGuestByName(String name) {
        return guestClient.get()
                 .uri(uriBuilder -> uriBuilder.path("/guests/query")
                                    .queryParam("name", name).build())
                .retrieve()
                .onStatus(status -> status.is4xxClientError(),
                        resp -> resp.bodyToMono(ProblemDetail.class)
                   .flatMap(pd -> Mono.error(new GuestNotFoundException(pd.getDetail()))))
                 .bodyToMono(Guest.class);
        }
}
```

# Centralizing Exception Handling



- possible to centralize exception handling based on AOP advice
- create a class annotated with @RestControllerAdvice
- make class extend ResponseEntityExceptionHandler, to already treat the most common HTTP Exceptions (e.g. HttpRequestMethodNotSupportedException, HttpMediaTypeNotSupportedException)
- imported class ResponseEntityExceptionHandler is different in non-reactive and reactive version!
- implement a method with annotation @ExceptionHandler, which contains the code, which returns the ProblemDetail object like you created in the Controller before
- client stays the same

# Centralizing Exception Handling Non-Reactive - Example

## RestResponseEntityExceptionHandler.java

```
import org.springframework.web.servlet.mvc.method.annotation
                                                .ResponseEntityExceptionHandler;


@RestControllerAdvice
public class RestResponseEntityExceptionHandler
                        extends ResponseEntityExceptionHandler {
    @ExceptionHandler(value  = GuestNotFoundException.class)
    protected ProblemDetail handleGuestNotFound
                                (GuestNotFoundException gnfe, WebRequest request) {
        ProblemDetail problem = ProblemDetail.forStatus(HttpStatus.NOT_FOUND);
        problem.setTitle("Guest Not Found");
        problem.setDetail(gnfe.getMessage());
        return problem;
    }
}
```

## RestReceptionController.java

```
@GetMapping("{id}")
public Guest returnGuestById(@PathVariable("id") int id)
                                                    throws GuestNotFoundException {
    return receptionService.findGuest(id);
}
```

# Centralizing Exception Handling Reactive - Example

## RestResponseEntityExceptionHandler.java

```
import org.springframework.web.reactive.result.method.annotation
                                                    .ResponseEntityExceptionHandler;

@RestControllerAdvice
public class RestResponseEntityExceptionHandler
                        extends ResponseEntityExceptionHandler {

    @ExceptionHandler(value  = GuestNotFoundException.class)
    protected Mono<ProblemDetail> handleGuestNotFound
                            (GuestNotFoundException gnfe, ServerWebExchange exchange) {

        ProblemDetail problem = ProblemDetail.forStatus(HttpStatus.NOT_FOUND);
        problem.setTitle("Guest Not Found");
        problem.setDetail(gnfe.getMessage());
        return Mono.just(problem);

    }
}
```

## RestReceptionController.java

```
@GetMapping("/query")
public Mono<Guest> returnGuestById(@RequestParam String name){
    return receptionService.findGuest(name);
}
```

# Validation



- syntax validation can be done via the JSR-303 Bean Validation standard, add spring-boot-starter-validation to the pom.xml
- uses annotations on the model classes

  e.g. @NotNull, @NotEmpty, @Size, @Min, @Pattern, @Email, ...
- error messages can be added
- mostly used for validating POST and PUT requests
- use @Valid on the @RequestBody argument to trigger validation
- RFC 7807: add “errors” part to the ProblemDetail definition
- handling of validation can be added to your ExceptionHandler class
- validation errors will throw MethodArgumentNotValidException, add a handler for this in your exceptionhandler class
- Watch out: when using Webflux, the request will not be intercepted by the handler correctly. It will pass the BAD REQUEST, but not customized. You will need handling in the controller to achieve this!
- In the WebClient, you will need to fetch error 400 seperately from the other error codes.

# Validation API - Example

## Guest.java

```
public class Guest {
    private int id;
    @NotBlank(message="name must be filled in")
    private String name;
    @Min(value=2, message="age must be bigger than 2")
    private int age;
    public Guest(){}
    public Guest(int id, String name, int age) {
        this.setId(id);
        this.name = name;
        this.age=age;
    }
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public int getAge() { return age; }
    public void setAge(int age) { this.age = age; }
}
```

# Validation API - Example (cont.)

## RestReceptionController.java

```
@RestController
@RequestMapping(path="guests", produces="application/json" )
public class RestReceptionController {
    @Autowired
    ReceptionService receptionService;
    @PostMapping(path="",consumes=MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<Void> addGuest(@Valid @RequestBody Guest guest){
        receptionService.addGuest(guest);
    }
}
```

# Validation API - Example (cont.)

## RestResponseEntityExceptionHandler.java

```
@RestControllerAdvice
public class RestResponseEntityExceptionHandler
                    extends ResponseEntityExceptionHandler {
    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ProblemDetail handleValidationException
                            (MethodArgumentNotValidException ex) {

        var problem = ProblemDetail.forStatus(HttpStatus.BAD_REQUEST);
        problem.setTitle("Validation Failed");

        var errors = new HashMap<String, String>();
        ex.getBindingResult().getFieldErrors().forEach(error ->
                errors.put(error.getField(), error.getDefaultMessage())
        );

        problem.setProperty("errors", errors);
        return problem;
    }
}
```

# Validation API - WebFlux Example

```
@Autowired Validator validator;

@PostMapping("")
public Mono<ResponseEntity<ProblemDetail>> addGuest(@RequestBody Guest g) {
    Set<ConstraintViolation<Guest>> violations = validator.validate(g);
    if (!violations.isEmpty()) {
        Map<String, String> errors = violations.stream()
            .collect(Collectors.toMap(v -> v.getPropertyPath().toString(),
                                      ConstraintViolation::getMessage));
        ProblemDetail problem = ProblemDetail.forStatus(HttpStatus.BAD_REQUEST);
        problem.setTitle("Validation Failed");
        problem.setProperty("errors", errors);
        return Mono.just(ResponseEntity.badRequest().body(problem));
    }
    return gs.addGuest(g).thenReturn(ResponseEntity.noContent().build());
}
```

# Validation Client - Example

```
public Mono<Void> addGuest(Guest g) {
    return guestClient.post()
                .uri("/guests")
                .bodyValue(g)
                .retrieve()
                .onStatus(status -> status.is4xxClientError(),
                    resp -> resp.bodyToMono(ProblemDetail.class)
                        .flatMap(problem -> {
                            if (resp.statusCode() == HttpStatus.BAD_REQUEST ){
                                return Mono.error(new BadRequestException("bad request"));
                            } else {
                                return Mono.error(new RuntimeException("Other 4xx error"));
                            }
                        }))
            .bodyToMono(Void.class);
}
```

# Introduction to ORM and (Spring) JPA

**Objectives :**

- **Persisting objects**
- **The object-relational mismatch**
- **Features of persistency mechanisms**
- **(Spring) JPA**

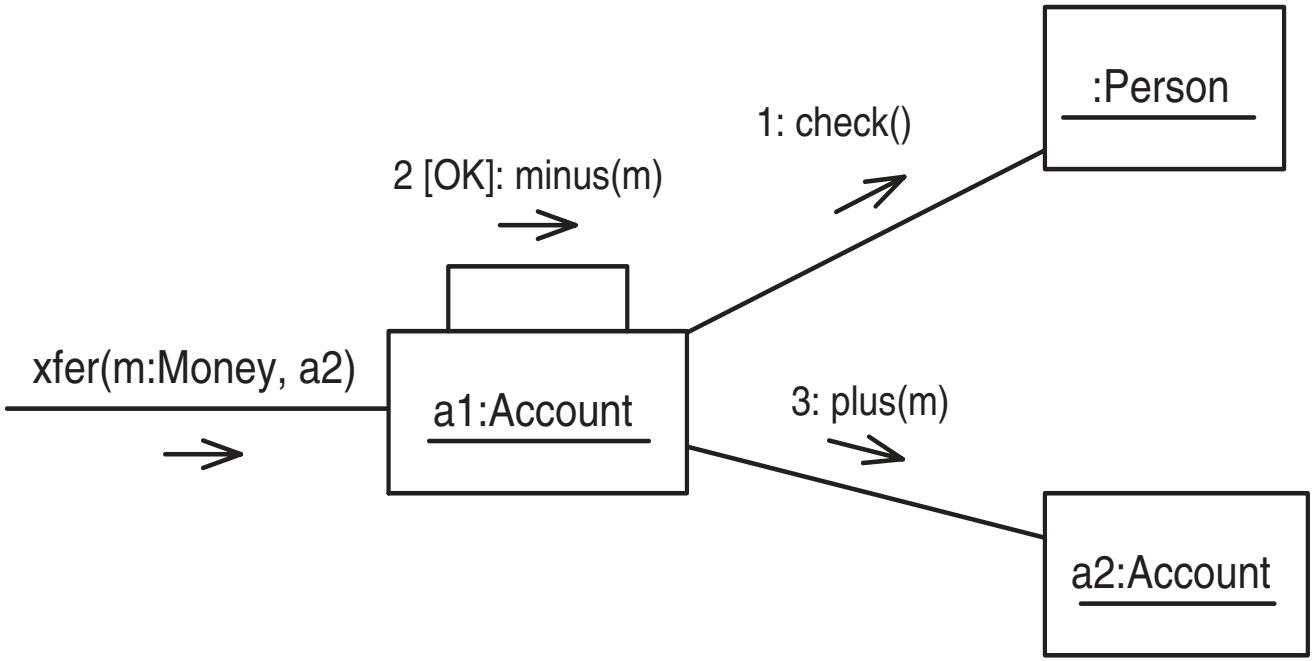
# Persisting objects



## OO-applications are composed of objects which

- consist of **data** and **behaviour**
- are connected to each other
- send messages to each other

## Example: transfer money between accounts



# Object life cycle

- objects are described in classes (domain model)
- classes are instantiated at runtime
- instances are populated with data
- these data must be preserved i.e. persisted/saved

# Object - relational mapping (ORM)



## O/R mismatch

- granularity
  - object attributes
- mapping of inheritance trees
  - inheritance is (mostly) not supported in RDBMSs
  - map tree to table(s)
  - polymorphic queries
- identification of entities
  - in OO: not directly supported
  - in RDBMS: through primary keys (PK)

# Object - relational mapping (cont.)

- relationship management between entities (associations)
  - in OO: (navigational) links between objects
  - in RDBMS: PK - FK relations
  - directional problem in 1-many (or many-many) relationship
- transaction management
  - in application (application environment) or by data store?
  - concurrency control
- synchronization
  - how to keep data in objects in sync with database and vice-versa
- performance
  - caching
  - lazy loading

# Object - relational mapping solutions

## How to specify O/R mapping

- hard coding of DB schema in Java
- meta-data definitions in XML
- annotations

## O/R mapping solutions (frameworks)

- do it yourself: extremely difficult and cost intensive
- use existing solutions:
  - JPA - Java Persistence Architecture
  - Java/Jakarta EE: Entity EJB (based on JPA)
  - Spring Data JPA (based on Hibernate)
  - alternatives: Hibernate, OJB, EclipseLink, Cayenne, MyBatis, ...

## based on **Unit Of Work** Enterprise Application Architecture pattern

https://java-source.net/open-source/persistence

## Features of persistence mechanisms



- **persist** the information (data) in the object model, i.e.
  - the data in the objects described as attributes in class model
  - links between objects described as relationships in class model
- **synchronization** between application and data
  - data in memory must be synchronized with data in data store
  - data in data store must be synchronized with data in memory
  - important if different applications access the same data
- **transactions**
  - set of actions that move data from one consistent state to another
  - key features: **A**tomicity, **C**onsistency, **I**solation, **D**urability
- **concurrency control**
  - different users/applications must reach same data at same time...
  - ...while keeping the data in a consistent state

# Features of persistence mechanisms (cont.)

- **query mechanism**
  - need for some mechanism to retrieve data in a selective way from the data store
- **identity support**
  - unique identification of object
  - avoid multiple copies of the same data
- **security**
  - unauthorized people must not see sensitive data
- **performance optimisation**
  - use of caching

# (Spring) JPA

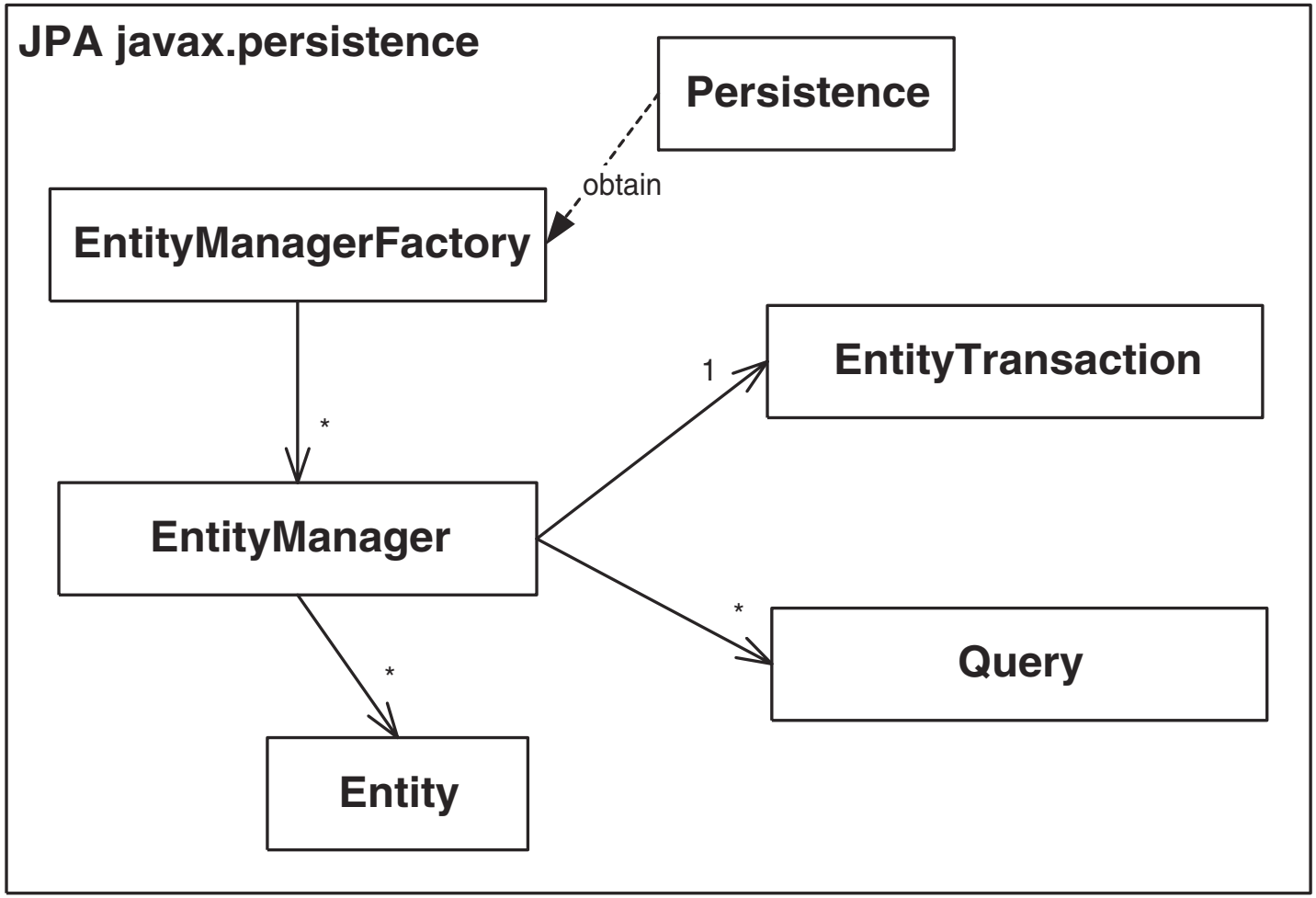


- **JPA: Java Persistence API (2006)**
  - **Java specification for accessing, persisting, and managing data between Java objects / classes and a RDBMS**
  - **defined as part of the EJB 3.0 specification as a replacement for the EJB 2 CMP Entity Beans specification**
  - **jakarta.persistence package (before javax.persistence)**
  - **specification, not implementation**
  - **implementations aka. JPA providers**

    EclipseLink, OpenJPA, **Hibernate,** Data Nucleus
- **Hibernate**
  - **started in 2001 as open source project**
  - **later further developed under JBoss (Red Hat)**
  - **used SessionFactory as basic interface**
  - **became certified implementation of JPA since version 3.5 (2010)**
  - **now mainly uses EntityManager as “interface”**

# JPA Architecture





# JPA API

**Introduction to ORM and (Spring) JPA**

1. Persisting objects
2. Object - relational mapping (ORM)
3. Features of persistence mechanisms
4. (Spring) JPA

## package jakarta.persistence

- Persistence: get EntityManagerFactory instances (vendor-neutral)
- EntityManagerFactory: factory for EntityManagers
- EntityManager: manages a set of persistent objects

  acts also as factory for Query instances
- Entity: persistent object that represents datastore records
- EntityTransaction: (associated with 1 EntityManager) group operations on persistent data into consistent units of work
- Query: interface to find persistent objects that meet certain criteria

  uses the Java Persistence Query Language (JPQL), the Criteria API and/or the native Structured Query Language (SQL)

# Spring's JPA integration



- **Spring Data JPA**
  - **powerful repository and custom object-mapping abstractions**
  - **Query DSL:**
    **dynamic query derivation from repository method names**
  - **implementation domain base classes providing basic properties**
  - **support for transparent auditing (created, last changed)**
  - **possibility to integrate custom repository code**
  - **easy configuration in Spring Boot**
  - **advanced integration with Spring MVC controllers**
  - **experimental support for cross-store persistence**

# Configuration

**Objectives :**

- **Configuring data sources**

# Spring JPA: Basic Configuration



- **Maven dependency**

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

- **Spring Boot will do autoconfiguration**
  - will try to use embedded database by default
  - when own DataSource defined, this is disabled
  - disable manually via:

```
@SpringBootApplication(exclude={DataSourceAutoConfiguration.class})
```

# Configuring a data source



## General



- data source defines driver, database url, username and password
- central configuration makes it easy to change later on
- driver depends on the database, can be chosen when setting up Spring Boot Application for several supported DBs

# Maven dependencies



## H2

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
</dependency>
```

## Oracle

```
<dependency>
    <groupId>com.oracle.database.jdbc</groupId>
    <artifactId>ojdbc11</artifactId>
</dependency>
```

## PostgreSQL

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

## SQL Server

```
<dependency>
    <groupId>com.microsoft.sqlserver</groupId>
    <artifactId>mssql-jdbc</artifactId>
</dependency>
```

## Embedded Data Source 3.1

- embedded database runs as part of your application instead of a separate database
- invisible to end-user
- perfect choice for development and test purposes
- test data is reset every time you restart your application
- pass script that is used to set up the DB
- products: Derby, H2,..
- configuration:

```
@Bean
@Profile("development")
public DataSource testDataSource(){
    return new EmbeddedDatabaseBuilder()
            .setType(EmbeddedDatabaseType.H2)
            .addScript("classpath:h2.sql")
            .build();
}
```

# Basic Data Source



- simplest possible, but limited data source
- provided by Spring (org.springframework.jdbc.datasource)
  - DriverManagerDataSource

    new connection every time one is requested
  - SingleConnectionDataSource

    returns same connection every time
- DBMS providers also have implementations available (e.g. OracleDataSource)
- no pooling possibilities!
- for small applications or at development phase

# Basic Data Source - Configuration

- **application.properties**

```
spring.datasource.url=jdbc:oracle:thin:@//delphi:1521/TSTA
spring.datasource.username=tu00010
spring.datasource.password=tu00010
spring.datasource.driver-class-name = oracle.jdbc.OracleDriver
```

- **Java config (deprecated)**

```
@Bean
public DataSource dataSource() throws SQLException {
    OracleDataSource ds = new OracleDataSource();
    ds.setURL(env.getProperty("spring.datasource.url"));
    ds.setUser(env.getProperty("spring.datasource.username"));
    ds.setPassword(env.getProperty("spring.datasource.password"));
    ds.setDriverType(env.getProperty("spring.datasource.driver-class-name"));
    return ds;
}
```

# Pooled Data Source



- directly configured in Spring
- provided by 3rd party (Apache Commons DBCP, c3po,Hikari,...)
- Hikari prefered (for Oracle/PostgreSQL/...), already provided in spring-boot-starter-data-jpa
- Maven dependencies for Apache Commons:
  - Apache Commons DBCP

```
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-dbcp2</artifactId>
</dependency>
```

RELIES ON:

  - Apache Commons Pool

```
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-pool2</artifactId>
</dependency>
```

# Pooled Data Source - Configuration

- **application.properties**

```
// basic connection settings like before
spring.datasource.hikari.minimumIdle=5
spring.datasource.hikari.maximumPoolSize=20
spring.datasource.hikari.idleTimeout=30000
spring.datasource.hikari.maxLifetime=2000000
spring.datasource.hikari.connectionTimeout=30000
```

- **Java Config (deprecated)**

```
@Bean
public DataSource prodDatasource() throws SQLException {
    HikariDataSource hds = new HikariDataSource();
    hds.setJdbcUrl(env.getProperty("spring.datasource.url"));
    hds.setUsername(env.getProperty("spring.datasource.username"));
    hds.setPassword(env.getProperty("spring.datasource.password"));
    hds.setDriverClassName
            (env.getProperty("spring.datasource.driver-class-name"));
    hds.setMinimumIdle
            (Integer.parseInt(env.getProperty("spring.datasource.hikari.minimumIdle"));
    hds.setMaximumPoolSize(Integer.parseInt
            (env.getProperty("spring.datasource.hikari.maximumPoolSize"));
    hds.setIdleTimeout
            (Long.parseLong(env.getProperty("spring.datasource.hikari.maxLifetime"));
    hds.setConnectionTimeout(Long.parseLong
            (env.getProperty("spring.datasource.hikari.connectionTimeout"));
    return hds;
}
```

# JNDI Data Source



- best to use in production
- data source managed by server (WebSphere, Tomcat,...) and retrieved via JNDI
- configuration
  - application.properties

```
spring.datasource.jndi-name=java:comp/env/jdbc/MyJndiName
```

  - Java Config (deprecated)

```
@Bean
public DataSource dataSource() throws NamingException {
    return (DataSource)  new JndiTemplate().lookup
                (env.getProperty("java:comp/env/jdbc/MyJndiName"));
}
```

# Simplifications from Spring Boot



- Spring Boot can autoconfigure datasources based on properties files
- don’t need to configure them any more and load resources!
- create application-**myprofilename**.properties files when profiles used
- use pre configured properties of Spring Boot
- driver will be deduced from database url
- when nothing configured, automatically embedded db used based on definitions in pom.xml
- EntityManager will be used behind the scenes

# Extra configuration options





## application.properties file can contain many extra properties

- DB initialization: spring.jpa.hibernate.ddl-auto
  - create: existing tables are first dropped, and then recreated
  - create-drop: tables are created, and dropped after application is finished
  - validate: validates whether the tables and columns exist; otherwise, it throws an exception
  - update:  object model created based on the mappings is compared with the existing schema, schema is updated according to the diff
  - none: no DDL is being done

    Spring Boot defaults the parameter value to create-drop if no schema manager has been detected, otherwise none for all other cases

# Extra configuration options (cont.)

- formatting/logging SQL
  - spring.jpa.show-sql = true
  - spring.jpa.properties.hibernate.format_sql = true
- standard logging
  - logging.level.org.hibernate.SQL=DEBUG
  - logging.level.org.hibernate.type.descriptor.sql.BasicBinder =TRACE

# Testing The Connection



- **DataSourceTest.java**

```
@SpringBootTest
public class DataSourceTest {

    @Autowired
    DataSource dataSource;

    @Test
    public void testConnectionViaDataSource() {

        try (Connection c = dataSource.getConnection()){
            System.out.println("connection succeeded via "
                                + c.getMetaData().getDatabaseProductName() + " . " );
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

# Basic OR Mapping and JPA repositories

**Objectives :**

- **Mapping of tables and properties**
- **Object identity**
- **JPA repositories and Query Methods**
- **Spring Data JUnit Testing**

# Mapping concepts



## What to map?

- map class to relational table
- map obj attribute / property to relational column + type conversion
- provide object identity
- map class association to relational FK
- inheritance/hierarchy mapping

## Mapping is done using (JPA) annotations (since Hibernate 3)

using configuration by exception,
the number of annotations required to do simple mappings
can be very limited

# Class to table mapping



## Use annotations on class level

- `@Entity` enables class to become persistent
- `@Table` associates with RDBMS table name (default = class name)

## Example

```
@Entity
@Table(name="GUESTS")
public class Guest {  }
```

## Define schema

- use schema attribute in table annotation

  `@Table(name="GUESTS", schema="SPRINGJPA")`
- default schema in application.properties

  **spring.jpa.properties.hibernate.default_schema** = springjpa

# Object identity



## Distinction between

- identical objects: `a == b`
- equal objects: `a.equals(b)`
- database identity: same primary key (PK) in database

## Distinction between

- natural keys (e.g. email)
- synthetic or surrogate keys (e.g. generated keys)

## Criteria to choose a primary key

- not null
- unique
- value never changes, immutable (i.e. no `set()` method provided)

# Object identity - mapping



## Annotation `@Id` to (placement is important for access type for all attributes)

- key field -> *field access type* (for all attributes!)
- get method (property) -> *property access type* (for all attributes!)

## If no key generator is provided

- application is responsible for providing/defining a value for the PK
- define constructor with argument for PK, e.g.

```
public Guest(String name) { this.name=name; }
```

## Unique key/identifier value can be generated: specify strategy -> `@GeneratedValue`

- AUTO: selects generation strategy based on the DB specific dialect
- IDENTITY: use identity columns (if supported in RDBMS)
- SEQUENCE: sequential unique key
- TABLE: key based on value in separate table

# Examples

- **Auto and Identity**

```
@GeneratedValue(strategy = GenerationType.AUTO)
@Column(name="gno")
private int id;
```

```
@GeneratedValue(strategy = GenerationType.IDENTITY)
@Column(name="gno")
private int id;
```

- **Sequence generator**

```
@SequenceGenerator(name="mySeqGen", sequenceName="guestsequence",
                                allocationSize=1)
@Id  @GeneratedValue (strategy=GenerationType.SEQUENCE,
                                  generator="mySeqGen")

@Column(name="gno")
private Long id;
```

- **Table generator (**

```
@TableGenerator(name="myTabGen", table="keytable",
                        pkColumnName="keygenName", pkColumnValue="myGen",
                        valueColumnName="newValue", initialValue=100,
                         allocationSize=10)

@Id @GeneratedValue (strategy=GenerationType.TABLE, generator="myTabGen")

@Column(name="gno")
private Long id;
```

# Composite primary key



- 2 techniques for creating a composite key
  - @IdClass
  - @EmbeddedId
- in both cases, an extra class is added for the Id “Object”
- class has to
  - implement Serializable (since possible session caching could occur)
  - override equals/hashCode (for testing key equality)
- usage in JpaRepository class will be slightly different for both techniques!

# Composite key via @IdClass

```
public class PersonKey implements Serializable {
    private int nr;
    private String email;
    public PersonKey() { }
    public PersonKey(int nr, String email) {
        this.nr=nr;
        this.email=email;
    }
    public boolean equals(Object o){
        Person p= (Person)o;
        return this.nr==p.nr && this.email.equals(p.email);
    }
    public int hashcode(){
        return Objects.hash(nr,email);
    }
}
@Entity
@IdClass(PersonKey.class)
public class Person {
    @Id
    private int nr;
    @Id
    private String email;
    ...
}
```

# Composite key via @IdClass (cont.)

```
public interface PersonJpaRepository extends JpaRepository<Person, PersonKey> {

    // Query DSL
    Person findByNrAndEmail(int nr, String email);

     //JPQL
    @Query("select p from Person p where p.nr=:nr and p.email = :email")
    Person findPersonById(@Param("nr") int nr, @Param("email") String email);

}
```

# Composite Key via @EmbeddedId

```
@Embeddable
public class PersonKey implements Serializable {
    private int nr;
    private String email;
    public PersonKey() { }
    public PersonKey(int nr, String email) {
        this.nr=nr;
        this.email=email;
    }
    public boolean equals(Object o){
        Person p= (Person)o;
        return this.nr==p.nr && this.name.equals(p.name);
    }
    public int hashcode(){
        return Objects.hash(nr,email);
    }
}
@Entity
public class Person {
    @EmbeddedId
    private PersonKey persKey;
    ...
}
```

# Composite Key via @EmbeddedId (cont.)

```
public interface PersonJpaRepository extends JpaRepository<Person, PersonKey> {

    // Query DSL
    Person findByPersKey_NrAndPersKey_Email(int nr, String email);

     //JPQL
    @Query("select p from Person p where p.persKey.nr=:nr and p.persKey.email = :email")
    Person findPersonById(@Param("nr") int nr, @Param("email") String email);

}
```

# Property mapping



- persistent field (direct field access), or
- persistent property (property access via get/set)

```
@Column (name="column_name")
```

- name of column in relational table (default = property name)
- private, protected,  package visibility for fields

## Example

```
@Column(name="ctitle")
private String title;

@Column(name="cdur")
public short getDuration() { return duration; }
```

**Configuration by exception:** i.e. default configuration/specification for each (non static non transient) field!

# Property mapping (cont.)

## Additional attributes on `@Column`

- length (optional): column length (**default 255 !**)
- updatable/insertable: whether or not the column will be part of the update/insert statement (default true)
- nullable: set the column as nullable (default true)
- unique: set a unique constraint on the column (default false)
- precision: column decimal precision (default 0)
- scale: column decimal scale if useful (default 0)
- table: define the targeted table (default primary table)
- columnDefinition: e.g. CHAR(45)

## Notes:

- `@Transient` -> field will not be persisted
- make use of Bean Validation annotations, e.g. `@NotNull`

# Type conversion - built-in Hibernate types



| Mapping type | Java type | ANSI Standard SQL type |
|---|---|---|
| big_decimal | java.math.BigDecimal | NUMERIC |
| big_integer | java.math.BigInteger | NUMERIC |
| boolean | boolean or java.lang.Boolean | BIT |
| byte | byte or java.lang.Byte | TINYINT |
| character | java.lang.Character | CHAR(1) |
| double | double or java.lang.Double | DOUBLE |
| float | float or java.lang.Float | FLOAT |
| integer | int or java.lang.Integer | INTEGER |
| long | long or java.lang.Long | BIGINT |
| short | short or java.lang.Short | SMALLINT |
| string | java.lang.String | VARCHAR |
| yes_no | boolean or java.lang.Boolean | CHAR(1) (‘Y’ or ‘N’) |
| true_false | boolean or java.lang.Boolean | CHAR(1) (‘T’ or ‘F’) |
| date | java.util.Date or java.sql.Date | DATE |
| time | java.util.Date or java.sql.Time | TIME |
| local date | java.time.LocalDate (Java 8) | DATE |
| local time | java.time.LocalTime (Java 8) | TIME |
| timestamp | java.util.Date or java.sql.Timestamp | TIMESTAMP |
| calendar | java.util.Calendar | TIMESTAMP |

# Type conversion - built-in Hibernate types (cont.)

| Mapping type | Java type | Standard SQL type |
|---|---|---|
| calendar_date | java.util.Calendar | DATE |
| locale | java.util.Locale | VARCHAR |
| currency | java.util.Currency | VARCHAR |
| timezone | java.util.TimeZone | VARCHAR |
| class | java.lang.Class | VARCHAR |
| binary | byte[] | VARBINARY (or BLOB) |
| blob | java.sql.Blob | BLOB |
| clob | java.sql.Clob | CLOB |
| serializable | java.io.Serializable implementor | VARBINARY (or BLOB) |
| text | java.lang.String | CLOB |

- type is derived by using reflection
- `@Temporal(TemporalType.DATE, TIME, or TIMESTAMP)` for date/time fields
- `@Lob` for large object (or serializable) properties
- `@Enumerated` for enums (EnumType.STRING or .ORDINAL)
- User-defined types can be specified with `@Type` annotation

# Property mapping - derived properties



## Property not mapped to database column

- annotation `@Formula`
- value of property calculated at runtime
- no column attribute (no insert or delete) -> **read-only (Select)**
- are allowed in formula (SQL fragment):
  - columns of database table
  - SQL functions
  - subselects

## Example

```
@Formula("cprice*1.21")
private float finalPrice;
```

# JPA repositories



## General

5.1

- in any reasonable-sized application, same methods (add, remove,...) will reappear (for different classes) over and over again
- only domain types will be different
- solution: only write repository interface that extends JpaRepository

```
public interface GuestJpaRepository extends JpaRepository<Guest,Long> {}
```

- above example will persist **Guest** objects with id of type **Long**
- JpaRepository contains 18 methods for performing common persistence operations, such as save(=merge!), delete, find by id,...
- extra methods can be added, which are translated to SQL
- implementation is generated at application startup
- configure via
  - Java Config

    ```
    @EnableJpaRepositories(basePackages="be.abis.ch8jparepo.dao")
    ```

  - autimatically configured in @SpringBootApplication

# Repository Interfaces



- CrudRepository<T,ID>
  - <S extends T> S save(S var1): saves/updates a given entity
  - Optional<T> findById(ID var1): retrieves an entity by its id
  - boolean existsById(ID var1): does entity with given id exists
  - Iterable<T> findAll(): all instances of the type
  - long count(): number of entities available
  - void deleteById(ID var1): deletes the entity with the given id
  - void delete(T var1): deletes a given entity
  - void deleteAll(): deletes all entities managed by the repository

# Repository Interfaces (cont.)

- PagingAndSortingRepository (extends CrudRepository)
  - Iterable<T> findAll(Sort var1): returns all entities sorted by the given options
  - Page<T> findAll(Pageable var1): returns a Page of entities meeting the paging restriction provided in the Pageable object
- JpaRepository (extends PagingAndSortingRepository)
  - List<T> findAll()
  - List<T> findAll(Sort var1)
  - void flush(): flushes all pending changes to the database
  - <S extends T> S saveAndFlush(S var1): saves an entity and flushes changes instantly
  - void deleteInBatch(Iterable<T> var1): deletes the given entities in a batch which means it will create a single query
  - void deleteAllInBatch(): deletes all entities in a batch call
  - T getById(ID var1): returns a **reference** to the entity with the given identifier

# Query Methods



- possibility to add methods to interface
- Spring can determine implementation based on the method name
- example

  ```
  List<Company> getCompanyByCountryOrderByName(String country);
  ```

- four verbs allowed in method name: get, read, find and count
- subject of repository is mostly optional
- syntax: query verb + subject (optional) + predicate + order by
- predicate
  - condition must reference one or more properties (combined with And and Or)
  - if no comparison operator, than equals is used
  - other comparison operators:

    IsAfter, IsGreaterThen, IsLessThan, IsBetween, IsNull, IsLike, Containing, IsNotIn, IsStartingWith, Contains, IsTrue,...
  - IgnoringCase can also be added

# Example

```
public interface GuestJpaRepository extends JpaRepository<Guest, Long> {

    Guest findById(long id);
    List<Guest> findByName(String name);
    List<Guest> findByNameStartsWith(String name);
    List<Guest> findByAge(int age);
    List<Guest> findByAgeIsLessThan(int age);
    List<Guest> findByNameStartsWithAndAgeIsLessThan(String name, int age);
    long countByName(String name);
}
```

# Spring Data JUnit Testing



- put @Transactional on test class or methods (org.springframework.transaction.annotation.Transactional)
- test-managed transactions will automatically be rollbacked
- test-managed transaction is one that is managed by TransactionalTestExecutionListener
- when running tests while starting up the Spring application context, a TransactionalTestExecutionListener is automatically configured
- @Transactional creates a new transaction that is then automatically rolled back after test completion
- can be changed, @Commit or @Rollback at the class or method level
- can use the static methods in TestTransaction, to start and end transactions, flag them for commit or rollback or check the transaction status
- Watch out: tests via RestTemplate or Postman are not automatically rolled back!

# Mapping Associations

**Objectives :**

- **Value types**
- **One class for two tables**
- **Mapping associations**
- **Cascading and fetching strategies**

# Value types

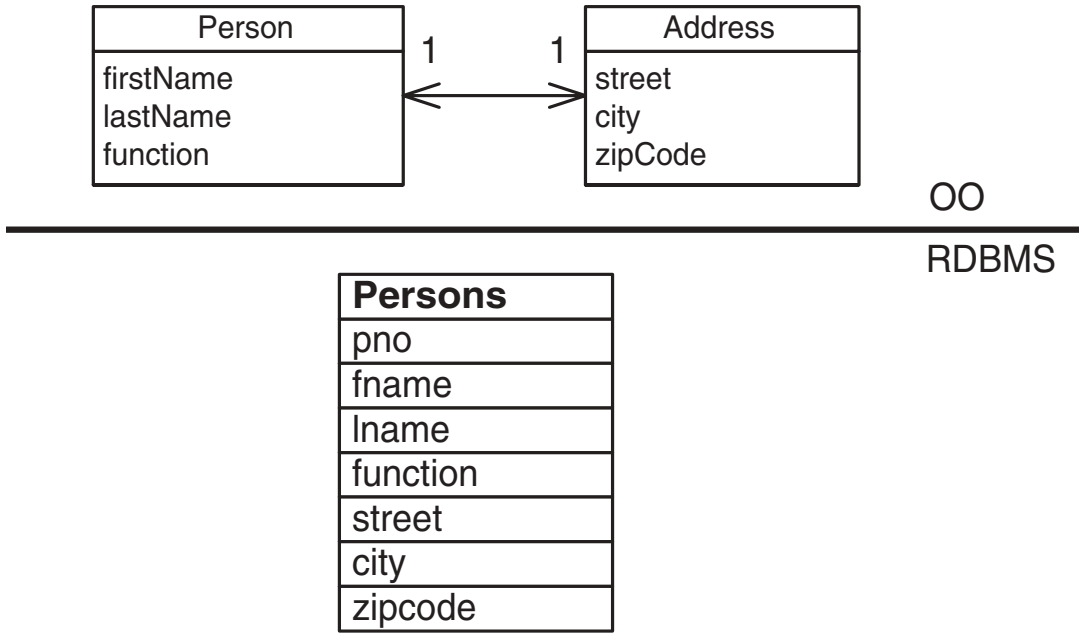




## Difference between

- **entity type: has its own database identity**
- **value type: depends on database identity of owning entity type**

  **2 (or more) classes map to 1 database table**



# Embedded objects

## Use `@Embedded` annotation (in 'parent' class) on contained property

```
@Entity
@Table(name="Persons")
public class Person {
    @Embedded
    private Address address;
    ...
}
```

## Use `@Embeddable` annotation on contained ('child') class

```
@Embeddable
public class Address {
    ...
    private String street;
    private String city;
}
```

## Notes:

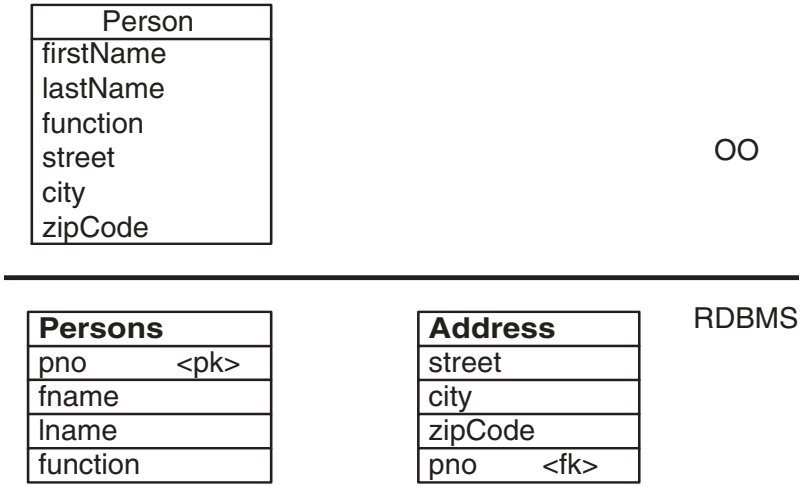
- shared references to Address object is not possible (composition)
- `@Embedded` is optional if `@Embeddable` is defined

# One class for two tables



## Use @SecondaryTable annotation

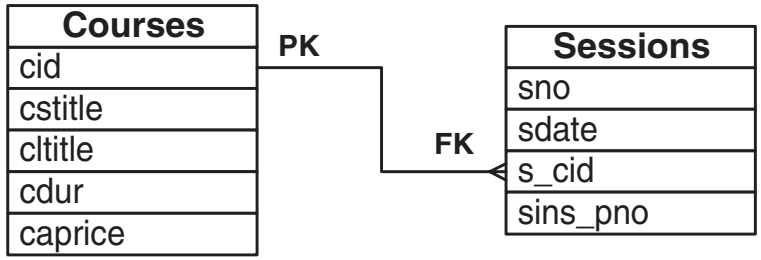
```
@Entity
@Table(name="Persons")
@SecondaryTable(name="Address",
pkJoinColumns = @PrimaryKeyJoinColumn
(name = "pno"))
public class Person {
    ...
    @Column(name="lname")
    private String lastName;
    ...
    @Column(table="Address", name="street")
    private String street;
    @Column(table="Address", name="city")
    private String city;
    ...
```



# Association mapping




- multiplicity (one-to-one, one-to-many, many-to-one, many-to-many)
- directionality (navigational access to data)
  - uni- or bidirectional
  - must be implemented in Java code
  - no managed associations in JPA -> owner class is responsible for updates
- PK - FK relationship (no direction) in RDBMS



# One-to-one mapping





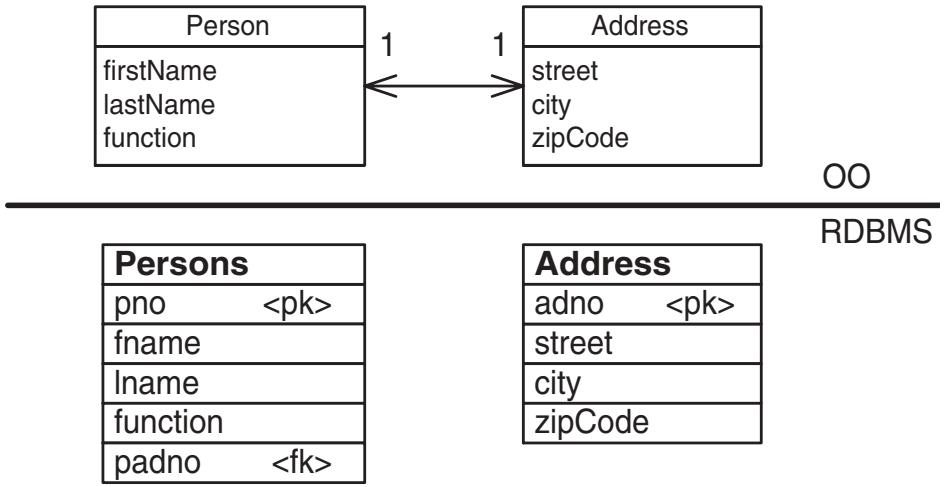
## Use when:

- value-type mapping (embedding) not possible
- entity mapping needed - each class mapped to a table

## Possibilities:

- foreign key association
- primary key association
- association table (rarely used)

# One-to-one mapping (cont.)




- FK in Persons table refers to PK of Address table
- FK (padno) must be unique -> add unique constraint in database
- for bidirectional mapping:
  - use also one-to-one mapping in Address mapping
  - class Person, containing FK, is owning side

# One-to-one mapping annotations

- `@OneToOne` annotation on association property
- naming of FK - PK columns
  - `@JoinColumn(name= ...)` on FK property
  - `@Column(name= ...)` on PK Id property

# One-to-one mapping - Example

```
@Entity
@Table(name="Persons")
public class Person {
    ...
    @OneToOne
    @JoinColumn(name="padno")
    private Address homeAddress;
    ...
}

@Entity
@Table(name="Address")
public class Address {
    ...
    @Id @GeneratedValue
    @Column(name="adno")
    private int addressNr;
    ...
}
```

# Bidirectional association

- *Owner (owning side)* class is responsible for the association column(s) update
- Specify additional property on the *target (inverse)* side
- Annotate with `@OneToOne` and attribute `mappedBy` at inverse side of the assocation (refer to property name in *owner* class)

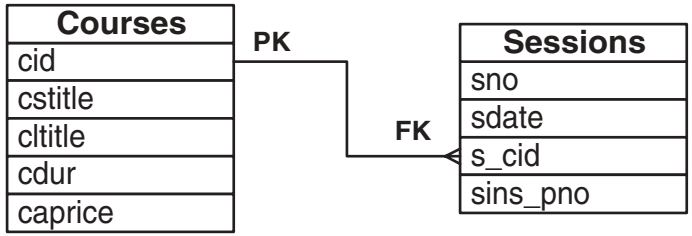
## Example

```
@Entity
@Table(name="Address")
public class Address {
    ...
    @OneToOne(mappedBy="homeAddress")
    private Person pers;
}
```

# Many-to-one mapping




## PK - FK relationship (no direction) in RDBMS




- Use `@ManyToOne` annotation on the Course object in the Session class

# Many-to-one mapping - Example

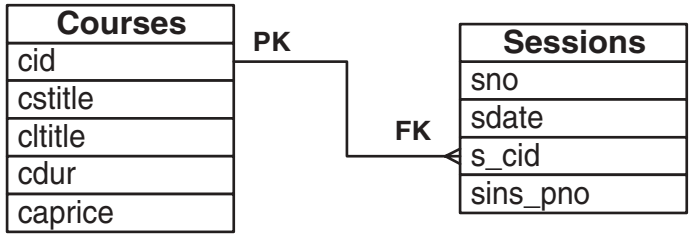
```
public class Course {
    ...
    @Id @GeneratedValue
    @Column(name="cid")
    private int courseId;
    private String coursetitle;
    ...
}

public class Session {
    ...
    @ManyToOne
    @JoinColumn(name="s_cid")
    private Course course;
    ...
}
```

# One-to-many mapping




# PK - FK relationship (no direction) in RDBMS




# Use `@OneToMany` annotation (on property with **interface** Set or List)

# One-to-many mapping (cont.)

- Define foreign key column via `mappedBy` attribute, and (optionally) target of association via `targetEntity` attribute

  (not required if generic type is specified)
- reverse direction: use `@ManyToOne` and `@JoinColumn` annotations
- 'many' side is owning side

## Example

```
public class Course {
...
    @OneToMany(targetEntity=Session.class, mappedBy="course")
    private Set<Session> courseSessions = new HashSet<Session>();
}

public class Session {
...
    @ManyToOne
    @JoinColumn(name="s_cid")
    private Course course;
}
```

# One-to-many mapping - remarks

- don’t forget to initialise the collection !
- override `equals()` and `hashCode()` for the entities used in a set identification based on “business key equality”
- control association inside application, e.g. via convenience method

```
public class Course{
...
    public void addSession(Session sess) {
        this.getCourseSessions().add(sess);
        sess.setCourse(this);
    }
    public void removeSession(Session sess) {
        this.getCourseSessions().remove(sess);
        sess.setCourse(null);
    }
```

# One-to-many mapping - remarks (cont.)

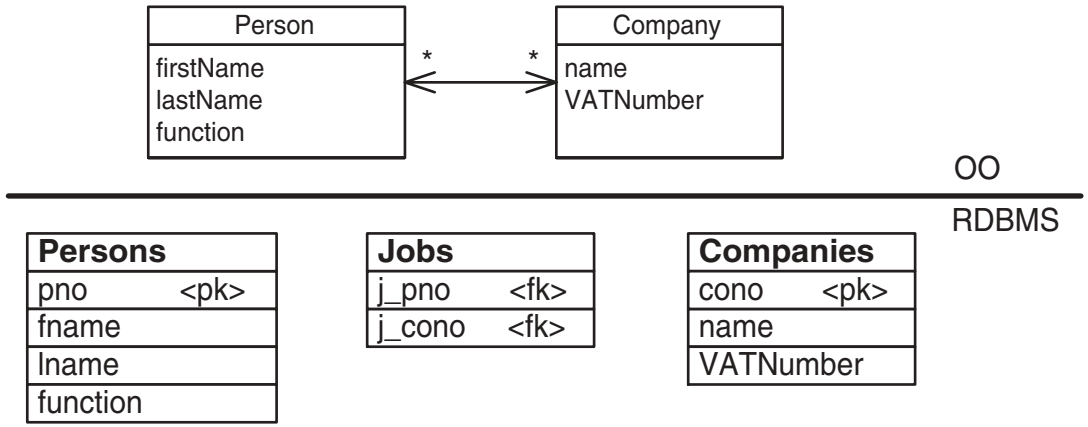
## If the many-collection is implemented as a java.util.Map, use the `@MapKey` annotation for the primary key.

## Example: person with multiple email addresses

```
@Entity
public class Person {
. . . .
@OneToMany
@MapKey(name="number")
private Map<String, Email> emails;
. . . .
}
```

# Many-to-many associations






## Can always be mapped with two many-to-one associations (recommended)

## Use `@ManyToMany` annotation

## Unidirectional and bidirectional associations possible

## In the database, one additional table is needed

(intermediate association class)

# Many-to-many mapping

## Mapping table generated for each direction by default

- use `mappedBy` attribute to point to same mapping table
- PK is combination of both foreign keys

## Definition of join table (and columns)

- `@JoinTable(name="...")`
- `@JoinColumns(joinColumns=@JoinColumn(name="..."), inverseJoinColumns=@JoinColumn(name="..."))`

## Bidirectional mapping

- repeat `@JoinTable` and switch join column names

# Example

```
public class Person {
    ...
    @ManyToMany
    @JoinTable(name = "jobs",
                    joinColumns = @JoinColumn(name = "j_pno"),
                    inverseJoinColumns = @JoinColumn(name = "j_cono"))
    private List<Company> listOfCompanies = new ArrayList<Company>();
    ...
}

public class Company {
    @ManyToMany(targetEntity=Person.class, mappedBy="listOfCompanies")
    private List<Person> listOfEmployees = new ArrayList<Person>();
    ...
}
```

# Additional options

4

## Cascade strategy

4.1

- how will cascading happen along the association path?
- default = no cascading !
- define `cascade=CascadeType.` (javax.persistence package)
  - PERSIST: save/persist corresponding record in other table
  - REMOVE: delete corresponding record in other table
  - MERGE: merge (after detach) corresponding record in other table
  - REFRESH: refresh corresponding record from other table
  - DETACH: detach corresponding record in other table
  - ALL: all the above options
- If cascade type does not correspond with database action:

  `TransientObjectException`

# Additional options (cont.)

## Fetching strategy

4.2

- how are associated records fetched?
- define `fetch=FetchType`.
  - EAGER: load corresponding record (default for one-to-one and many-to-one)
  - LAZY: do not load corresponding record (default for one-to-many and many-to-many)

    also called lazy initialisation

**Mapping Associations**

1. Value types
2. One class for two tables
3. Association mapping
4. Additional options

# Querying in Spring JPA

**Objectives :**

- **@Query**
- **JPQL**
- **Named queries**
- **Native queries**
- **Using DTOs and mappers**

# @Query



- @Query can be defined on repository methods to specify your own query, instead of using a generated one
- query can be written in JPQL or SQL

```
@Query("select c from Course c where c.title = 'Java' ")
List<Course> findJavaCourses();

@Query( value = "select * from courses where title = 'Java' ",
        nativeQuery=true)
List<Course> findJavaCoursesNative();
```

- positional or named parameters can be passed

```
@Query("select c from Course c where c.title = ?1 ")
List<Course> findCoursesWithTitle(String title);

@Query("select c from Course c where c.title = :title ")
List<Course> findCoursesWithTitle(@Param("title") String title);
```

# JPQL

## JPA Query Language (JPQL) characteristics

- use classes + properties  instead of  tables + columns
- polymorphic queries
- automatic join for associations
- full support of relational operations
- paging features

## Querying is done in an ‘object oriented’ way!

# Basic JPQL



**projection** SELECT specify selected data

**selection** FROM data source

**restriction** WHERE match records to criteria

# Selection: From



- **Defines the class name + an alias**

  **select cc from ConcreteCourse [as] cc**

- **JPQL is case sensitive for class names and properties**
- **if @Entity(name="abc") attribute is used on class, use abc in JPQL from clause**
- **alias will be used in where clause**

## JPQL: polymorphic queries

```
Select c from Course c
     //selects all ConcreteCourse and CourseGroup objects
Select o from java.lang.Object o
     //selects all persistent objects
Select s from java.io.Serializable s
     //selects all serializable objects
```

# Restriction: Where



- **Comparison operators**
  - **where column [ = | <> | < | > | <= | >= ] expression**
  - **use of NOT: negates the expression**

```
select c from Course c where c.title = 'Java'
select c from Course c where c.title <> 'Java'
select c from Course c where c.duration > 3
select c from Course c where c.duration <= 4
select c from Course c where not(c.title = 'Java')
```

- **(NOT) IN operator**

```
select c from Course c where c.title IN ('JAVAPROG','SPRINGJPA')
select c from Course c NOT IN ('JAVAPROG','SPRINGJPA')
```

- **(NOT) Between operator (border values are inclusive)**

```
select c from Course c where c.price between 200 and 300
```

# Restriction: Where (cont.)

- **(NOT) like operator**

```
select c from Course c where c.title like 'Java%'
select c from Course c where c.title like '%Java%'
select c from Course c where c.title not like '%Java%'
select c from Course c where lower(c.title) like '%java%'
```

- **Is (NOT) Null checking**

```
select c from Course c where c.title is null
select c from Course c where c.title is not null
```

- **AND, OR, NOT operators**

```
select c from Course c
where c.title not like '%java%'
   and (c.duration < 2 or c.duration > 4)
```

- **Collection operations**

```
select c from Company c, Person p
where c.employees is not empty

select c from Company c, Person p
where p member of c.employees

select c from Company c, Person p
where size(c.employees) > 100
```

- **functions : https://en.wikibooks.org/wiki/Java_Persistence/JPQL**

# Joins in JPQL



- inner join automatically generated when traversing entities

  select p from Person p where p.address.country = 'B'

- join can also be mentioned explicitly
  - “comma” notation, with join condition in where statement

    select d from Employee e, Department d where e.department = d
  - (inner) join

    select distinct p from Person p join p.enrolments e where e.pricePerDayPayed > 400
  - left join

    select p from Person p left join Company c where p.age>40
  - join fetch (to override FetchType.LAZY)

    select p from Person p join fetch p.companies c where p.age>40

  mainly used in combination with @OneToMany and @ManyToMany associations

  watch out: if no where condition or traversing entity used, a cartesian product is generated!

  select p from Person p join Address a

# Parameter binding in JPQL



- **Named parameters**
  - parameter name preceded by a ":"
  - in method, use @Param which states the name of the parameter

```
@Query("select c from Course c
        where c.duration = :dur and c.title like %:title%)
List<Course> getCourseByDurationAndTitle
      (@Param("dur") int duration, @Param("title") String title)
```

- **Positional parameters (deprecated!)**

```
@Query("select c from Course c
        where c.duration = ?1 and c.title like %?2%")
List<Course> getCourseByDurationAndTitle
                      (int duration, String title)
```

**Note: watch out for SQL injection !**

# Named queries



- Predefine commonly used queries in the persistent class
- Add `@NamedQuery` annotation before entity class
- add a method in the repository interface which has exactly the same name as the name of the named query
- name obligatory prefixed with the class name
- uses JPQL by default
- possible to add parameters
- also @NamedNativeQuery, which will use SQL instead
- Example

```
@NamedQuery(name="Course.findByDurAndTitleNQ",
                    query="select c from Course c
                                where c.duration = :dur and c.title like %:title%")
public class Course {
    ...
}
public interface CourseJpaRepository extends JpaRepository<Course,Integer> {
    List<Course> findByDurAndTitleNQ
        (@Param("dur") int duration, @Param("title") String title);
}
```

# Native queries



- use @Query(value="", nativeQuery=true)
- via `@NamedNativeQuery`
- specify result class in the repository method
- plain SQL that uses table names and columns
- JPA will not generate the query/queries in this case
- mostly used in case of limiting results when using joins
- gives you more control, but are DBMS specific
- Example

```
@Query(   value="select * from courses where cname like %:name% order by cname",
          nativeQuery=true)
List<Course> findCoursesOrderedByName(@Param("name") String name);
```

remark: it is possible that you will need to use a concat function when using a “like” combined with a parameter in native queries (certainly when used together with other functions):

```
select * from courses where cname like concat('%',:name,'%') order by cname
```

# Changing the “select” object



- returned entity will change when select doesn’t fetch the full object
  - aggregate functions -> numerical value returned
  - select specific columns or group by -> List<Object[ ]> returned
- use Data Transfer Objects (DTOs) and mappers to convert to/from the “original” entity
- mainly needed when using native queries with joins, or limiting the result shown via a REST API
- better for performance, but comes with the “cost” of mapping the result

# Aggregate functions



- **avg**

```
@Query("select avg(c.duration) from Course c")
Double getAverageDurationOfCourses();
```

- **min**

```
@Query("select min(c.duration) from Course c")
Integer getMinimumDurationOfCourses();
```

- **max**

```
@Query("select max(c.duration) from Course c")
Integer getMaximumDurationOfCourses();
```

- **sum**

```
@Query("select sum(c.duration) from Course c")
Integer getTotalDurationOfCourses();
```

- **count**

```
Integer countByDuration(int duration);
```

- **count distinct**

```
@Query("select count(distinct c.duration) from Course c")
Integer getNumberOfDifferentDurationOfCourses();
```

# Limiting/Changing selected columns

5.2

- Object[ ] or List<Object[ ]> returned instead of original object
- Examples

```
@Query("select c.title, c.duration from Course c where c.id=:id")
Object[ ] getCourseTitleAndDurationById(@Param("id") int id);

@Query("select c.title, count(c) from Course c group by c.domain having count(c)>2")
List<Object[ ]> getNumberOfCoursesPerDomain();

@Query(value="p.pfname, p.plname, co.coname
                    from persons p inner join companies c on p.pa_cono=c.cono",
            nativeQuery=true)
List<Object[ ]> getPersonsWithCompanyName();
```

- will need to be mapped to a DTO
- techniques:
  - custom Mapper class
  - @Named(Native)Query and @SqlResultSetMapping
  - Interface-based DTO projections
  - Class-based DTO projections

  Only the first 2 techniques can be used for native queries!

## Data Transfer Objects (DTOs) 6.1

- class that maps the result of a query
- provide a constructor with all arguments that will be passed
- used to limit the size of objects passed to the client (web or via JSON)
- will increase performance
- better design: separates entity layer and “view” layer objects
- will need transformation/mapping!
- Example

```
public class PersonDTO {
    private String firstName;
    private String lastName;
    private String companyName;
    public PersonDTO(String firstName, String lastName, String companyName){
        ...
    }
    ...
}
```

# Custom Mapper Class



- user-defined class that maps the Object[ ] to a PersonDTO object
- contains a static method PersonDTO toDTO(Object[ ])
- service will then call the mapper object, and returns the PersonDTO object
- all array elements will need to be converted from Object to their correct type!

# Custom Mapper Class - Example

```
public PersonJpaRepository extends JpaRepository<Person,Integer>{
    @Query(value="select p.pfname, p.plname, co.coname
                  from persons p inner join companies c on p.pa_cono=c.cono",
                nativeQuery=true)
    List<Object[ ]> getAllPersonsWithCompanyName();
}


public class PersonDTOMapper {
    public static PersonDTO toDTO(Object[ ] objArray){
        PersonDTO p = new PersonDTO(objArray[0].toString(),
                                    objArray[1].toString(),
                                    objArray[2].toString());
        return p;
    }
}


public class MyPersonService implements PersonService {
    @Autowired PersonJpaRepository pjr;
    public List<PersonDTO> findAllPersonsWithCompanyName(){
        List<PersonDTO> persons = new ArrayList<PersonDTO>();
        List<Object[ ]> resList = pjr.getAllPersonsWithCompanyName();
        for (Object[ ] objArray : resList){
            persons.add(PersonDTOMapper.toDTO(objArray));
        }
        return persons;
}
```

# @NamedNativeQuery and @SqlResultSetMapping



- use a named (native) query on the class
- @SqlResultSetMapping then used to map the columns
- List of DTO objects can then directly be returned from the repository
- Example

```
@NamedNativeQuery(name="Person.getAllPersonsWithCompanyNameNQ",
                  query="p.pfname first, p.plname last, co.coname comp
                  from persons p inner join companies c on p.pa_cono=c.cono",
                  resultSetMapping = "PersonDtoMapping")

@SqlResultSetMapping(name = "PersonDtoMapping",
                     classes = @ConstructorResult(targetClass = PersonDto.class,
                                                  columns = {@ColumnResult(name = "first"),
                                                             @ColumnResult(name = "last"),
                                                             @ColumnResult(name = "comp"})))
@Entity
public class Person {
    ...
}


public PersonJpaRepository extends JpaRepository<Person,Integer>{
    @Query(nativeQuery=true)
    List<PersonDTO> getAllPersonsWithCompanyNameNQ();
}
```

# Mapping JPQL query results



- Class-based DTO projections
  - use constructor call directly in the JPQL query
  - fully qualified name needed!

```
public PersonJpaRepository extends JpaRepository<Person,Integer>{
@Query
    (value="new be.abis.demo.dto.PersonDTO(p.firstName, p.lastName,
                        p.company.name) from Person p")
    List<PersonDTO> getAllPersonsWithCompanyName();
}
```

- Interface-based DTO projections
  - instead of using DTO class, use an interface with the columns in the exact order
  - watch out: easy to write, but bad performance!

```
public interface PersonDTO {
    String getFirstName();
    String getLastName();
    String getCompanyName();
}
public PersonJpaRepository extends JpaRepository<Person,Integer>{
    @Query(value="p.firstName, p.lastName,p.company.name from Person p")
    List<PersonDTO> getAllPersonsWithCompanyName();
}
```

**Querying in Spring JPA**

1. @Query
2. JPQL
3. Named queries
4. Native queries
5. Changing the “select” object
6. Using DTOs and mappers

# Transaction Management and Exception Handling

**Objectives :**

- **Exception Handling**
- **Modifying queries**
- **Transaction Management**
- **Declarative and programmatic transactions**

# Exception Handling



- Spring JPA predefines many (around 35 of them!) exceptions
- platform-agnostic
- super class = org.springframework.dao.DataAccessException
- descriptive names :
  IncorrectResultSizeDataAccessException,
  DataIntegrityViolationException,...
- all of them are RuntimeException -> unchecked
- still important to do correct exception handling!!!
- documentation:

  https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/dao/DataAccessException.html

# Modifying queries



- JPQL/native queries can be used for DML
- add @Modifying annotation on the repository method
- obligatory to add @Transactional when calling them!
- Examples

```
public PersonJpaRepository extends JpaRepository<Person,Integer>{
    @Modifying
    @Query(value="update persons set ppwd=:newpwd where pno=:id ,
                nativeQuery=true)
    Person updatePassword(@Param("newpwd") String newpwd, @Param("id") int id);
}
public class MyPersonService implements PersonService {
    @Autowired PersonJpaRepository pjr;
    @Transactional
    public void updatePassword(String newpwd, int id){
        pjr.updatePassword(newpwd,id);
    }
}
```

# Understanding transactions



- **transactions ensure all-or-nothing behaviour**
- **only make sense in a non-reactive environment (if really needed, can make use of a programmatic transaction in a reactive context)**
- **data never left in inconsistent state**
- **ACID test**
- **Hibernate/JPA imposes transaction management**
- **global vs local transactions**
  - **global transactions**
    - enable you to work with multiple transactional resources, typically relational databases and message queues
    - application server manages global transactions through JTA
  - **local transactions**
    - resource-specific, e.g. transaction linked to a JDBC connection
    - simpler, but no application server involved, so limited

# Spring's Transaction Philosophy



- consistent programming model in any environment (whether global or local)
- only need an application server's JTA capability if application needs to handle transactions across multiple resources
- Spring supports both programmatic and declarative transaction management
- abstract away actual transactional implementation from code
- transaction strategy based on PlatformTransactionManager
- TransactionException is a RuntimeException
- By default everything is autocommitted, unless @Transactional is used

# Transaction Managers



- different implementations of PlatformTransactionManager
- acts as facade to platform-specific transaction implementation
  - DataSourceTransactionManager (JDBC) -> dataSource
  - HibernateTransactionManager -> sessionFactory
  - JpaTransactionManager -> entityManager
  - JtaTransactionManager -> need dataSource via JNDI
- getTransaction() will return a TransactionStatus based on a TransactionDefinition
- Spring Boot automatically configures a JpaTransactionManager (if no JTA context found)
- use via
  - @Transactional (declarative transactions)
  - wired in transactionTemplate (programmatic transactions)

# Transaction Status



- might represent a new transaction, or can represent an existing transaction if a matching transaction exists in the current call stack
- associated with a thread of execution
- code can use this to retrieve status information, and to programmatically request a rollback
- provides access to savepoint management facilities
- methods
  - flush(): flush the underlying session to the datastore, if applicable
  - hasSavepoint(): does transaction internally have a savepoint
  - isCompleted(): is transaction completed (commit or rollback)
  - isNewTransaction(): is present transaction new or nested
  - isRollbackOnly(): is transaction marked as rollback-only
  - setRollbackOnly(): set the transaction rollback-only.

# Transaction Definition



- **isolation level**
  - **degree to which transaction is isolated from other transactions**
  - **possibilities:**
    - ISOLATION_READ_COMMITTED: dirty reads are prevented; non-repeatable reads and phantom reads can occur
    - ISOLATION_READ_UNCOMMITTED: dirty reads, non-repeatable reads and phantom reads can occur.
    - ISOLATION_REPEATABLE_READ: dirty reads and non-repeatable reads are prevented; phantom reads can occur.
    - ISOLATION_SERIALIZABLE: dirty reads, non-repeatable reads and phantom reads are prevented
    - (default!) ISOLATION_DEFAULT: use the default isolation level of the underlying datastore

# Transaction Definition (cont.)

- **propagation behaviour**
  - **defines the transaction scope**
  - **Spring offers all of the transaction propagation options familiar from EJB CMT**
  - **possibilities**
    - PROPAGATION_NESTED: execute within a nested transaction if a current transaction exists, PROPAGATION_REQUIRED like behaviour else
    - PROPAGATION_NEVER: do not support a current transaction; throw an exception if a current transaction exists.
    - PROPAGATION_NOT_SUPPORTED: do not support a current transaction; rather always execute non-transactionally
    - (default!) PROPAGATION_REQUIRED: support a current transaction; create a new one if none exists.
    - PROPAGATION_REQUIRES_NEW: create a new transaction, suspending the current transaction if one exists.
    - PROPAGATION_SUPPORTS: support a current transaction; execute non-transactionally if none exists.

# Transaction Definition (cont.)

- timeout
  - how long does transaction runs before timing out and being rolled back automatically
  - defined as int
  - default = - 1: use default timeout of the underlying transaction system
- read-only
  - defines whether underlying transaction is readonly
  - default = false
- rollback rules
  - defines for which Exceptions the transaction should be rolled back
  - default: RuntimeException and Error

# Declarative transactions



- implemented through Spring's AOP framework
- annotation based:
  - @Transactional (org.springframework.transaction.annotation)
  - attributes define the transaction definition properties
- only need to override non-default options
- method annotations override class annotations
- use in the Service Layer for DML methods
- for delete: use myRepo.flush() such that the exception is thrown within the transaction boundary
- in JUnit tests
  - automatic rollback when putting @Transactional on class/method, use @Commit on test to change this behaviour
  - Make sure that the @Transactional annotation on the test does not change the behaviour of the method in the Service layer!
  - put @Transactional only on "happy flow" tests, exceptions should be handled from the Service layer

# Example

- **AbisReceptionService.java**

```
@Service
public class AbisReceptionService implements ReceptionService {
    @Autowired
    private CompanyJpaRepository cjr;

    ...
    @Transactional(isolation = Isolation.SERIALIZABLE,
                        rollbackFor = CompanyException.class)
    public void doStuffWithCompanies() throws CompanyException{
        try {
            Company c = new Company
                    (1000, "My Company", "some street", "30", "3000", "Leuven", "B");
            cjr.save(c);
            c.setStreet("Diestsevest");
            cjr.save(c);
            System.out.println(cjr.findById(1000));
            cjr.deleteById(3);
            List<Company> companies = cjr.findAll();
            for (Company comp : companies) {System.out.println(comp);}
            cjr.flush();
        } catch (EmptyResultDataAccessException | DataIntegrityViolationException e){
                throw new CompanyException("oops");
        }
    }
}
```

# Example (cont.)

- **ReceptionTest.java**

```
@SpringBootTest
public class ReceptionTest {

    @Autowired
    private ReceptionService receptionService;

    @Test
    @Transactional
    public void testdoStuffWithCompanies() {
        assertDoesNotThrow(()->receptionService.doStuffWithCompanies());
    }

    @Test
    public void testdoStuffWithCompanies() {
        assertThrows(CompanyException.class,
                                ()->receptionService.doStuffWithCompanies();
    }


}
```

# Programming transactions



- control every step in the transaction yourself
- can be more fine-grained than declarative transactions
- wire in TransactionManager in your service class
- use this to create a TransactionStatus object
- in try-catch, call commit/rollback manually
- intrusive in your code!

# Example

```
@Service
public class GuestService implements Service {

    @Autowired
    private  PlatformTransactionManager txManager;
    public Mono<Void> changeCoffeePreference(Guest g) {
        return Mono.fromRunnable() -> {
            TransactionStatus tx = txManager.getTransaction
                                    (new DefaultTransactionDefinition());
            try {
                guestRepo.changeCoffeePreference(g,"latte");
                txManager.commit(tx);
            } catch (Exception e) {
                txManager.rollback(tx);
        }
        }).subscribeOn(Schedulers.boundedElastic()).then();
    }
}
```

**Transaction Management and Exception Handling**

1. Exception Handling
2. Modifying queries
3. Understanding transactions
4. Transaction Status
5. Transaction Definition
6. Declarative transactions
7. Programming transactions

# Advanced Topics

**Objectives :**

- **Mapping collections**
- **Hierarchy mapping**

# Mapping collections





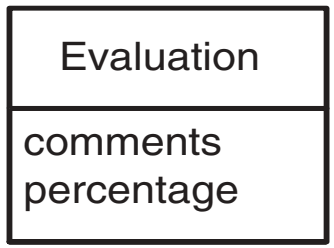
## Difference between:

- collections of entity types

  mapped with true entity relationships

  - `@OneToMany`: see 3.3 One-to-many mapping on page 311
  - `@ManyToMany`: see 3.4 Many-to-many associations on page 315
  - mapped with collections semantics: not recommended, requires separate collections table (not covered here)
- collections of value types
  - different possibilities: set, list, map -> in separate table
  - use `@ElementCollection` and `@CollectionTable`

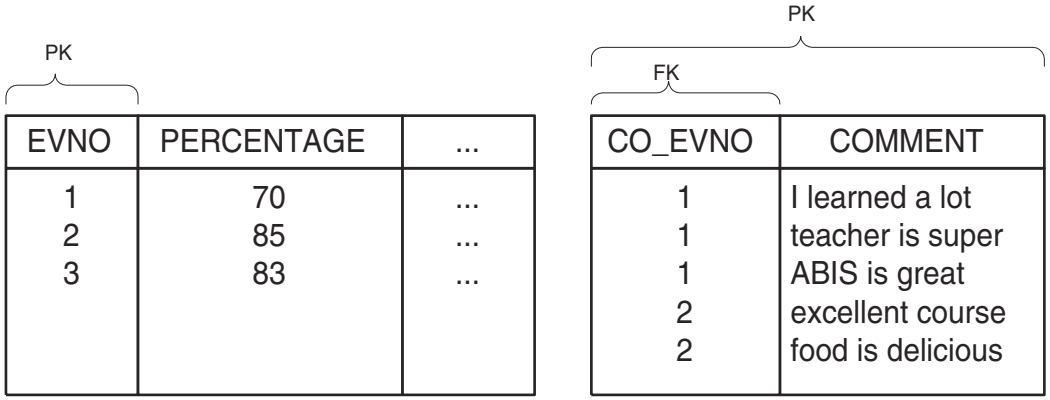
## Example



# Set








- **Java class is a `Set` implementation**
- **Set cannot contain duplicate elements -> two times same comment is impossible for same evaluation**

```
@Entity
public class Evaluation {

    ...
    @ElementCollection
    @CollectionTable(name="Comments",joinColumns=@JoinColumn(name="co_evno"))
    @Column(name="comment")
    private Set<String> comments;

    ...
}
```

# List





PK

| EVNO | PERCENTAGE | ... |
|---|---|---|
| 1 | 70 | ... |
| 2 | 85 | ... |
| 3 | 83 | ... |

PK

FK

| CO_EVNO | POSITION | COMMENT |
|---|---|---|
| 1 | 0 | I learned a lot |
| 1 | 1 | teacher is super |
| 1 | 2 | ABIS is great |
| 2 | 0 | I learned a lot |
| 2 | 1 | food is delicious |

- Java class is a `List` implementation (e.g. `ArrayList`)
- duplicates are allowed
- preserves order

```
@Entity
public class Evaluation {

    ...
    @ElementCollection
    @CollectionTable(name="Comments",joinColumns=@JoinColumn(name="co_evno"))
    @OrderColumn(name="position")
    @Column(name="comment")
    private List<String> comments;
    ...
}
```

# Map





PK

| EVNO | PERCENTAGE | ... |
|---|---|---|
| 1 | 70 | ... |
| 2 | 85 | ... |
| 3 | 83 | ... |

PK

FK

| CO_EVNO | CO_TYPE | COMMENT |
|---|---|---|
| 1 | duration | too long |
| 1 | presentation | teacher is super |
| 1 | food | delicious |
| 2 | duration | good |
| 2 | presentation | excellent |

- **Java class is a *unordered* `Map`**
- **Key/value pair**
  - **key: basic, embeddable, or entity type**
    - unique
    - hashCode() and equals()
  - **value:**
    - basic or embeddable type: use @ElementCollection
    - entity type: use @OneToMany or @ManyToMany

# Map - annotations



## Key column specification

- @MapKeyColumn(name=...)

  default name: attribute_KEY
- @MapKeyEnumerated(EnumType.STRING)

  use enumerated type key with string names
- @MapKeyTemporal(DATE | TIME | TIMESTAMP)

  use Date type as key
- @MapKeyJoinColumn(name=...)

  use entity as key

## Value column specification

- @Column(name=...)

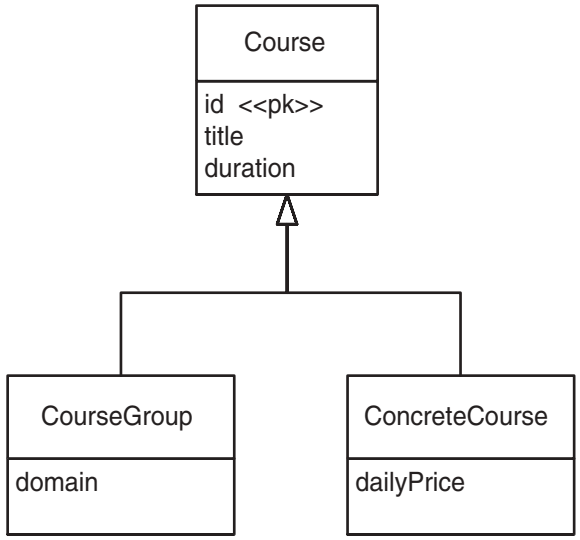
# Map - example

```
@Entity
public class Evaluation {
    ...
    @ElementCollection
    @CollectionTable(name="Comments")
    @MapKeyColumn(name = "co_type")
    @MapKeyEnumerated(EnumType.String)
    @Column(name = "comment")
    private Map<CommentType,String> comments;
    ...
}
public enum CommentType {
    DURATION,
    PRESENTATION,
    INFRASTRUCTURE,
    FOOD
}
```

# Hierarchy mapping



## Hierarchical relations between entities not always supported in a relational database




## Three alternatives:

## 1. table per class hierarchy (default)

## 2. table per concrete class

## 3. table per subclass (joined)

# Table per class hierarchy





## ‘Single table’ inheritance

| courses |
|---|
| id <<pk>> |
| title |
| duration |
| domain |
| dprice |
| ctype <<discriminator>> |

- **define `@Entity` on each class in hierarchy**
- **define `@Inheritance` on root class and use `InheritanceType.SINGLE_TABLE`**
- **define `@DiscriminatorColumn` to define column name and type**
- **define `@DiscriminatorValue` in each class to define column value**

  default value = class name

**notice discriminator (DTYPE) column in table**

internal column used by JPA

**-> no normalisation in RDBMS, nulls allowed in table!**

# Table per class hierarchy - example

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn (name="ctype", discriminatorType=DiscriminatorType.STRING)
@DiscriminatorValue("co")
@Table(name="Courses")
public class Course {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name="id")
    private int id;

    @Column(name="title")
    private String title;

    @Column(name="duration")
    private int duration;

    // DO NOT MENTION THE DISCRIMINATOR COLUMN AS PROPERTY !!!

    // constructors + getters/setters

    public String getCourseType(){
        return this.getClass().getAnnotation(DiscriminatorValue.class).value();
    }


}
```

# Table per class hierarchy - example (cont.)

```
@Entity
@DiscriminatorValue("cg")
public class CourseGroup extends Course {
    @Column(name="domain")
    private String domain;
    // constructors + getter/setter
}

@Entity
@DiscriminatorValue("cc")
public class ConcreteCourse extends Course {
    @Column(name="dprice")
    private double dailyPrice;
    // constructors + getter/setter
}

public interface CourseJpaRepository extends JpaRepository<Course,Integer> {
    Course findById(int id);
    Query(value = "select ctype from courses where id=:courseId", nativeQuery = true)
    String findCourseType(@Param("courseId") int courseId);
}
```

# Table per class hierarchy - example (cont.)

```
public interface CourseService {
    Course findCourseById(int id);
    Course addCourse(Course course);
    long count();
    String findCourseType(int courseId);
}


@Service
public class AbisCourseService {
    @Autowired
    CourseJpaRepository courseRepository;
    public Course findCourseById(int id){
        courseRepository.findById(id);
    }
    @Transactional
    public Course addCourse(Course course){
        return courseRepository.save(course);
    }
    public long count(){
        return courseRepository.count();
    }
    public String findCourseType(int id){
        return courseRepository.findCourseType(id);
    }
}
```

# Table per class hierarchy - example (cont.)

```
@SpringBootTest
public class CourseServiceTest {

    @Autowired
    CourseService courseService;

    @Test
    public void course1IsAconcreteCourse(){

        Course found = courseRepository.findCourseById(1);
        assertTrue(found instanceof ConcreteCourse);
    }

    @Test
    @Transactional
    public void addCourseGroupWorks(){

        CourseGroup groupToAdd = new CourseGroup("new title", 5, "Java");
        long coursesBefore = courseService.count();
        Course added = courseService.addCourse(groupToAdd);
        long coursesAfter = courseService.count();
        assertEquals(1, coursesAfter-coursesBefore);
        assertEquals("cg", added.getCourseType());
        assertEquals("cg", courseService.findCourseType(added.getId()));

}
```

# Table per concrete class





## 1. Union subclass

| coursegroups |
|---|
| id <<pk>> |
| title |
| duration |
| domain |

| concretecourses |
|---|
| id <<pk>> |
| title |
| duration |
| dprice |

- define `@Entity` (and `@Table`) on each class in hierarchy
- define `@Inheritance` on root class and use `InheritanceType.TABLE_PER_CLASS`
- parent class properties and id are inherited in child classes

## Notes:

- Duplicate columns in different tables!
- do not use AUTO nor IDENTITY generation for the key field
- no polymorphic queries

# Table per concrete class (union subclass) - example

```
@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public class Course {
    ...
}

@Entity
@Table(name="CourseGroup")
public class CourseGroup extends Course {
    ...
}

@Entity
@Table(name="ConcreteCourse")
public class ConcreteCourse extends Course {
    ...
}
```

## Table per concrete class

### 2. Implicit polymorphism

- parent class is not mapped (no `@Entity`). Use `@MappedSuperclass`
- parent properties are mapped/annotated
- define `@Entity` (and `@Table`) and map properties on each subclass in hierarchy
- parent class properties and id are inherited in child classes

### Notes:

- parent columns duplicated in different tables!
- do not use AUTO nor IDENTITY generation for the key field

# Table per concrete class (implicit polymorphism) - example

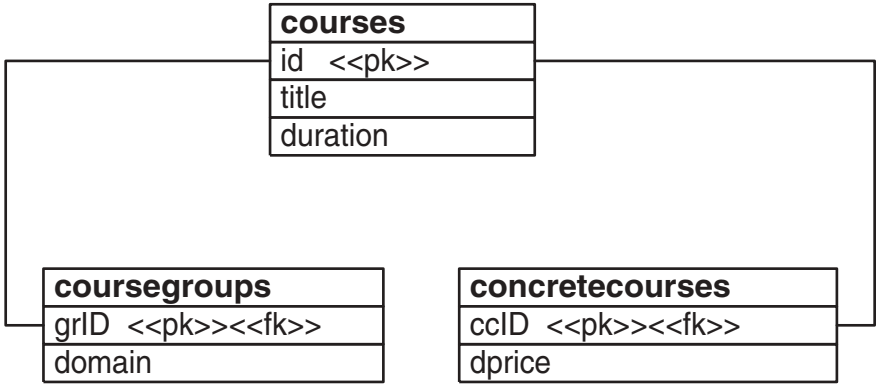
```
@MappedSuperclass
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public class Course {
    ...
}
@Entity
@Table(name="CourseGroup")
public class CourseGroup extends Course {
    ...
}

@Entity
@Table(name="ConcreteCourse")
public class ConcreteCourse extends Course {
    ...
}
```

# Table per subclass (joined)








- **define `@Entity` on each class in hierarchy**
- **define `@Inheritance` on root class and use `InheritanceType.JOINED`**
- **adds key in subclass**

  key can have the same name in all classes
  `@PrimaryKeyJoinColumn(name="...")`
- **create PK-FK relationships in database**

**JPA uses outer join to access data!**

# Table per subclass (join) - example

```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
@Table(name="Courses")
public class Course {
    @Id
    @Column(name="cid")
    private int courseNr;
    ...
}

@Entity
@PrimaryKeyJoinColumn(name="cid")
public class CourseGroup extends Course {
    ...
}
@Entity
@PrimaryKeyJoinColumn(name="cid")
public class ConcreteCourse extends Course {
    ...
}
```

# Choose a hierarchy mapping strategy





**No polymorphic queries or associations needed (e.g. if superclass is abstract)**

- **prefer table-per-concrete-class**

**Polymorphic queries or associations needed**

- **not too many subclasses and not too many attributes in subclasses**

  **-> table-per-class-hierarchy**

  **-> lot of nulls in table**

- **many subclasses or many attributes in subclasses**

  **-> table-per-subclass (join)**

**Note: it is possible to mix strategies (but DON'T use mixing)**

# Securing REST APIs

**Objectives :**

- **API keys**
- **HTTP basic and digest authentication**
- **Authorizing Access**
- **CORS**
- **OAuth 2.0 and OpenID**

# Securing APIs



## Security is an important part of any enterprise application

## Not all APIs are/should be public!

## Two essential elements

- **Authentication:**
  **verifying the identity of the user trying to access the API**
- **Authorization:**
  **verifying what an authenticated user is permitted to do in the API**

## Ways to secure your REST API

- **API keys**
- **HTTP basic / digest authentication**
- **OAuth 2.0**
- **Authorize the RESTful web service accesses**

# Application Security vs Server Security

## Application Security

- application has its own way to check authentication & authorization
- custom database
- some possibilities:
  - authentication via login screen
  - other web pages check whether login was done correctly
  - authorization via roles of users
- “custom” encoding mechanisms via (java) frameworks

## Server Security

- SERVER defines users and roles (possibly via LDAP)
- can be used for different applications at once
- application not called when authentication fails
- encryption via HTTPS, Digest authentication,...

# API Keys



**API key is a token that a client provides when making API calls**

**Supplied by the API provider (e.g. when creating an account)**

**An API Key has these properties:**

- **token, in the form of a relatively long random string (e.g. 32 characters)**
- **identifier, for storage and unique identification**
- **transmitted with the request (requires SSL for in production) in the HTTP header or via a cookie (convention: call it X-API-Key)**
- **known to the client**
- **can be validated by the server**
- **unique to a device or software (e.g. UUID)**
- **bound to a user if necessary**

# Using API keys - General Recipe



- create an ApiKeyStore in the API
  - store UUIDs (the keys) linked with a user id
  - UUID can be linked to user registration, or passed from a specific API call
- when passed from a specific controller API method towards the client
  - generate a UUID to serve as the api key
  - save the new key in the keystore
  - send the key as the X-API-key header on the response, or as a cookie
  - client fetches the key via the “exchangeToMono” method instead of “retrieve()”, to be able to also fetch the headers of the response
  - wrap the body and the apiKey together in an object to send to the Controller
  - in case of a web application, store the key/cookie in a session attribute

# Using API keys - General Recipe (ctd.)

- to send api key from a client to an api add
  **.header(“X-API-Key”,apiKey)** in the request
- in the controller of the called API
  - add **@RequestHeader("X-API-Key") String apiKey** in the method heading
  - check the key via the **isValid()** method of the ApiKeyStore class
  - throw an ApiKeyViolationException if wrong, otherwise call the service method to return the response

# API keys in the API - Example



## ApiKeyStore.java

```
public class ApiKeyStore {

    private static final Map<Integer, String> tore  new ConcurrentHashMap<>();

    public static void store(int personId, String apiKey) {
        toreput(personId,apiKey);
    }

    public static boolean isValid(int personId, String apiKey) {
        String storedKey = toreget(personId);
        return storedKey != null && storedKey.equals(apiKey);
    }

}
```

# API Keys in the API (ctd.)

## ApiKeyNotCorrectException.java

```
public class ApiKeyNotCorrectException extends Exception {
    public ApiKeyNotCorrectException(String message) {
        super(message);
    }
}
```

## RestResponseEntityExceptionHandler.java

```
@RestControllerAdvice
public class RestResponseEntityExceptionHandler
                                extends ResponseEntityExceptionHandler {

    @ExceptionHandler(value = ApiKeyNotCorrectException.class)
    protected Mono<ProblemDetail> handleApiKeyWrong
                (ApiKeyNotCorrectException aknce, ServerWebExchange exchange) {
        ProblemDetail problem = ProblemDetail.forStatus(HttpStatus.UNAUTHORIZED);
        problem.setTitle("API key problem");
        problem.setDetail(aknce.getMessage());
        return Mono.ustproblem);
}}
```

# API Keys in the API (ctd.)

## GuestController.java

```
@GetMapping("{id}")
public Mono<ResponseEntity<GuestDTO>> findGuest(@PathVariable int id) {
    return gs.findGuest(id)
             .map(guest -> {
                 String apiKey = UUID.randomUUID().toString();
                 ApiKeyStore.store(id, apiKey);
                 return ResponseEntity.ok()
                         .header("X-API-Key", apiKey)
                        .body(GuestMapper.toDTO(guest));
             });
}


@PutMapping("{id}")
public Mono<GuestDTO> changeCoffeePreference(@PathVariable("id") int id,
                 @RequestBody GuestForm guest,
                 @RequestHeader("X-API-Key") String apiKey) {


    if (!ApiKeyStore.isValid(id, apiKey)) {
        return Mono.error(new ApiKeyNotCorrectException("Invalid API key"));
    }
     Mono<Guest> g = gs.changeCoffeePreference
                              (GuestMapper.toGuest(guest),guest.getCoffeePreference());
    return g.map(GuestMapper::toDTO);
}
```

# API Keys in the Client - Example



## AbisGuestService.java

```
public Mono<GuestResult> findGuest(int id) {
    return guestClient.get()
            .uri("/guests/{id}")
            .exchangeToMono(response -> {
                String apiKey = response.headers().asHttpHeaders().getFirst("X-API-Key");
                return response.bodyToMono(GuestDTO.class)
                        .map(guest -> new GuestResult(guest, apiKey));
            });
}


public Mono<GuestDTO> changeCoffeePreference
            (int id, GuestForm g, String newPreference ,String apiKey) {
    g.setCoffeePreference(newPreference);
    return guestClient.put()
            .uri("/persons/"+id)
            .header("X-API-Key", apiKey)
            .bodyValue(g)
            .retrieve()
            .onStatus(
                    status -> status.value() == HttpStatus.UNAUTHORIZED.value(),
                    response -> response.bodyToMono(ProblemDetail.class)
                            .flatMap(pd -> Mono.error(
                                    new ApiKeyNotCorrectException(pd.getDetail()))))
            )
            .bodyToMono(GuestDTO.class);
}
```

# API Keys in the Client - Example (ctd.)

## GuestResult.java

```
public record GuestResult(GuestDTO guest, String apiKey) {}
```

## GuestController.java

```
@GetMapping("{id}")
public Mono<ResponseEntity<GuestDTO>> findByID(@PathVariable int id) {
    return gs.findGuest(id)
          .map(result -> ResponseEntity.ok()
                .header("X-API-Key", result.apiKey())
                .body(result.guest()));
}
@PutMapping("{id}")
public Mono<GuestDTO> changeCoffeePreference(@PathVariable("id") int id,
          @RequestBody GuestForm guest,
          @RequestHeader("X-API-Key") String apiKey)  {
    return gs.changeCoffeePreference(id,guest, guest.getCoffeePreference(),apiKey);
}
```

# API Keys in the Client - Example (ctd.)

## GuestServiceTest.java

```
@SpringBootTest
public class GuestServiceTest {
    @Autowired  GuestService gs;
    @Test
    public void changePreferenceOfGuest3WithCorrectKey()  {
        GuestResult result = gs.findGuest(3).block();
        String key = result.apiKey();
        GuestDTO found = result.guest();
        GuestForm g = new GuestForm("John","latte");
        GuestDTO updated =
                gs.changeCoffeePreference(g.getId(),g,g.getCoffeePreference(),key).block();
        assertEquals("JOHN",updated.getFirstName());
    }

    @Test
    public void changePreferenceOfGuest3WithWrongKey()  {
        GuestResult result = gs.findGuest(5).block();
        String key = result.apiKey();
        GuestDTO found = result.guest();
        GuestForm g = new GuestForm("John","latte");
        Throwable ex = assertThrows(RuntimeException.class,
            ()-> gs.changeCoffeePreference(3,g,g.getCoffeePreference(),key).block());
        assertTrue(ex.getCause() instanceof ApiKeyNotCorrectException);
}
```

# Introduction to Spring Security



- provide declarative security for Spring-based applications
- roots in Acegi Security
- tackles security from two angles
  - secure web requests via servlet filters
  - secure method invocations using AOP
- security namespace
- best put all configuration in separate file
- Maven Dependency

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

contains several modules like spring-security-config, spring-security-web, spring-aop and spring-web

# HTTP Basic and Digest Authentication

## BASIC

- send a Base64 encoded username and password as a pair in the HTTP authorization header
- username and password must be sent for every HTTP request made by the client
- chain of events:
  - client requests a URI
  - server replies with a 401 HTTP response
  - client receives response, scans through it, and prepares a new request with the necessary data needed to authenticate the user
  - new request from the client will contain the authorization header set to a Base64 encoded value of column delimited username and password string, <username>:<password>
  - server verifies credentials and replies with requested resource

# HTTP Basic and Digest Authentication (cont.)

- deficiencies:
  - unauthorized access could take place with cached credentials
  - no way for a web service to differentiate authorized requests from unauthorized ones
  - not enough for security because usernames and passwords are only encoded using Base64 encoding, which is easy to decipher
  - intent of Base64 is not to secure the name-value pair, but to uniformly encode characters when transferred over HTTP
  - not recommended to use basic authentication over HTTP for application accessed over the Internet
  - solve this potential security hole by using HTTPS

# HTTP Basic and Digest Authentication (cont.)

## DIGEST

- unlike basic authentication, password is not transmitted in clear text between the client and the server
- client sends a one way cryptographic hash of the username, password, and a few other security related fields using the MD5 message-digest hash algorithm
- when server receives the request, it regenerates the hashed value for all the fields as done by client and compares it with the one present in the request
- if the hashes match, request is treated as authenticated and valid
- remark: not all servers (like WAS) support digest authentication!

# Configuring Minimal Security



- **Java Config**

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {}
```

```
@Configuration
@EnableWebFluxSecurity
public class SecurityConfig {}
```

- **default configuration will**
  - demand that each HTTP request is authenticated
  - since no user store is defined, nobody can login yet
  - automatically reroute the user to a default login screen

    e.g. go to localhost:8081/security/guests in your browser
- **actions to be taken**
  - user store needs to be configured
  - specify which requests should and should not require authentication and what authorities are required
  - (provide a custom login screen)
- **Watch out: since Spring Boot 3: security config class cannot extend WebSecurityConfigurerAdapter anymore !!!**

# Creating a User Store



- Spring is capable of authenticating against virtually any data store
  - in-memory
  - relational database
  - LDAP
- also possible to create custom implementation
- since Spring 5: PasswordEncoder required
- in-memory
  - add PasswordEncoder
  - create Bean with type UserDetailsService which returns an InMemoryUserDetailsManager

# Example of an In-Memory User Store

- **SecurityConfig.java**

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public BCryptPasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }


    @Bean
    public UserDetailsService userDetailsService(PasswordEncoder encoder){
        UserDetails user = User.withUsername("user1")
                .password(encoder.encode("pwd1"))
                .roles("USER")
                .build();

        UserDetails admin = User.withUsername("user2")
                .password(encoder.encode("pwd2"))
                .roles("USER","ADMIN")
                .build();

        return new InMemoryUserDetailsManager(admin, user);
    }
}
```

# Intercepting Requests



- decide which URIs will require login, and which won't
- add a bean returning a SecurityFilterChain in SecurityConfig.java, taking in a HttpSecurity object as a parameter
- use authorizeHttpRequests() method
- uses ANT-style matching syntax
- shows 401 when not authorized
- disable csrf for the moment

# Intercepting Requests - Example

- access to /guests permitted for everyone (watch out, also POST possible for the moment)
- all subpaths of /guests (like /guests/1) will have to be authenticated

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http.csrf(AbstractHttpConfigurer::disable)
        .authorizeHttpRequests(authorizationManagerRequestMatcherRegistry ->
                authorizationManagerRequestMatcherRegistry
                        .requestMatchers("/guests").permitAll()
                        .requestMatchers("/guests/*").authenticated())
        .httpBasic(Customizer.withDefaults())
        .sessionManagement(httpSecuritySessionManagementConfigurer ->
            httpSecuritySessionManagementConfigurer
            .sessionCreationPolicy(SessionCreationPolicy.STATELESS));

    return http.build();
}
```

# HTTP Basic Authentication in the Client



- username and password will now have to be provided when calling the API for certain requests
- 2 ways to do this
  - add authentication in header of the request

```
guestClient.post()
    .uri("/guests")
    .headers(headers -> headers.setBasicAuth("user1", "pwd1"))
    .bodyValue(c)
    .retrieve()
```

  - pass authentication when creating the Webclient object

```
WebClient courseClient = WebClient.builder()
            .baseUrl("http://localhost:8080/guestapi")
            .defaultHeaders(h -> h.setBasicAuth("user1", "pwd1"))
            .build();
```

- make sure exception handling treats the 401 status
  -> watch out: status return empty error body, not ProblemDetail!

# HTTP Basic Authentication in the Client - Example

## AbisGuestService.java

```
@Service
public class AbisGuestService {
    WebClient guestClient = WebClient.builder()
                    .baseUrl("http://localhost:8080/guestapi")
                    .defaultHeaders(h -> h.setBasicAuth("user1", "pwd1"))
                    .build();

    public Mono<Course> addGuest(Guest g) {
        return  guestClient.post()
                     .uri("/guests")
                    .bodyValue(g)
                    .retrieve()
                     .onStatus(status -> status.is4xxClientError(),
                        resp -> resp.bodyToMono(ProblemDetail.class)
                                    .defaultIfEmpty(ProblemDetail.orStatusAndDetail
                                        HttpStatus.INTERNAL_SERVER_ERROR,
                                        "No ProblemDetail body in response"
                                    ))
                        .flatMap(problem -> {
                            if (resp.statusCode() == HttpStatus.BAD_REQUEST) {
                                return Mono.error(new BadRequestException("bad request"));
                            }  if (resp.statusCode() == HttpStatus.UNAUTHORIZED {
                                return Mono.rornew NotAuthenticatedException
                                                        ("you are not authenticated"));
                        } else {return Mono.error(new RuntimeException("Other 4xx error"));}
                    }))
                    .bodyToMono(Course.class);
}}
```

# Authorizing Access



- some actions will need extra authorization on top of authentication (e.g. post instead of get)
- authorization will be role-based (roles passed via SecurityConfig)
- Spring uses method level security to enforce authorization
- client will stay the same as with basic authentication
- don’t forget to catch 401 (Unauthorized) and 403 (Forbidden) now

# Securing Methods with Spring



- second line of defence next to Web Security
- secure bean methods
- 3 different kinds of security annotations
  - Spring Security's own @Secured
  - JSR-250's @RolesAllowed
  - fine-grained expression-driven annotations
- configuration:
  - add @EnableMethodSecurity(jsr250Enabled=true) on your security config class
  - Secure the method via @RolesAllowed

```
@PostMapping("")
@RolesAllowed("ADMIN")
public void addGuest(@Valid @RequestBody Guest guest){
    receptionService.addGuest(guest);
}
```

- HttpStatus.FORBIDDEN (403) thrown if access not allowed

# CORS



- CORS = Cross-Origin Resource Sharing
- browser security mechanism that restricts HTTP requests from one origin to resources of another origin
  - origin = Protocol + Domain + Port
  - prevents unauthorized cross-site calls
  - servers must explicitly allow cross-origin requests
- configuration
  - in the API (on the Controller)

```
@CrossOrigin(origins = "http://localhost:4200")
```

  - via a proxy in client (e.g. angular)

```
//proxy.conf.json
{ "/ngpersonapi": {
   "target": "http://localhost:8080",
    "secure": false
   }}

//angular.json
"projects": {
   "your-project-name": {
    "architect": {"serve": {"options": { "proxyConfig": "proxy.conf.json"}
} } } }
```

## OAuth 2.0 7.1

- authorization framework
- open standard for authorization, used by many enterprises and service providers to protect resources
- authentication done via Authorization Server (could be 3rd party, e.g. Google)
- goals
  - delegate access
  - avoid sharing passwords
  - use short-lived access tokens
  - allow secure app integration

# OAuth 2.0 (ctd.)

- main actors
  - resource owner: user owning the data
  - client: app requesting access
  - authorization server: issues token after user consent
  - resource server: api that trusts access tokens
- what OAuth is NOT
  - OAuth is not authentication
  - does not provide identity
  - access token only means "you are allowed," not "you are this person."

  -> other protocol needed for authentication!

# OpenID Connect (OIDC)



- build on top of (including) OAuth 2.0, providing authentication
- OIDC adds
  - ID Token: JSON Web token (JWT) containing user identity
  - userInfo endpoint (retrieve profile information)
  - standard scopes: openid, profile, email

# OIDC Authorization Code Flow (Typical)

- user visits the application
- app redirects to Authorization Server (e.g., Google, Keycloak).
- user logs in
- authorization Server redirects back with authorization code
- app exchanges code for
  - ID token
  - access token
  - optional refresh token
- app verifies ID token signature
- user is logged in

# Integration in Spring



- **Maven Dependency:**

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-client</artifactId>
</dependency>
```

- **application.properties (Google OIDC example)**

```
spring.security.oauth2.client.registration.google.client-id=YOUR_CLIENT_ID
spring.security.oauth2.client.registration.google.client-secret
                                                =YOUR_CLIENT_SECRET
spring.security.oauth2.client.registration.google.scope=openid,profile,email
spring.security.oauth2.client.provider.google.issuer-uri
                                                =https://accounts.google.com
```

**to get Google OAuth2 / OpenID Connect Client ID and Client Secret, create a project in Google Cloud Console and enable the OAuth consent + credentials**

# Integration in Spring (ctd.)

- **SecurityConfig.java**

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

        http
            .authorizeHttpRequests(auth -> auth
                                .requestMatchers("/public").permitAll()
                                .anyRequest().authenticated()
                    )
                    .oauth2Login()    // Enables OpenID Connect login
                    .oauth2Client();  // Enables OAuth2 client capabilities

        return http.build();

    }
}
```

# Integration in Spring (ctd.)

- **API Controller**

```
@RestController
public class ApiController {

    private  OAuth2AuthorizedClientService clientService;

    public ApiController(OAuth2AuthorizedClientService clientService) {
        this.clientService = clientService;
    }

    @GetMapping("/call-api")
    public String callApi(@AuthenticationPrincipal OidcUser user) {
        OAuth2AuthorizedClient client =
                clientService.loadAuthorizedClient("google", user.getName());

        String accessToken = client.getAccessToken().getTokenValue();
        return "Access token: " + accessToken;
    }

}
```

- **UserController**

```
@RestController
public class UserController {

    @GetMapping("/me")
        public String me(@AuthenticationPrincipal OidcUser user) {
            return "Hello " + user.getFullName();
        }

}
```

**Securing REST APIs**

1. Securing APIs
2. API Keys
3. Introduction to Spring Security
4. HTTP Basic and Digest Authentication
5. Authorizing Access
6. CORS
7. Introduction to OAuth 2.0 and OpenID Connect

# Building Web Applications with Spring MVC

**Objectives :**

- **Writing a basic controller**
- **Processing forms**

# Hello World Example - Web Version



- add the following dependencies to the pom.xml
  - Spring MVC + tomcat-starter + jackson-bind + spring-web

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

  - Thymeleaf: server-side Java template engine for web apps (instead of .jsp pages!)

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

# Hello World Example - Web Version (cont.)

- **MyController.java**

```
@Controller
public class MyController {

    @Autowired
    private HelloService helloService;

    @GetMapping("/")
    public String home(Model model) {
        model.addAttribute("message",
                "Hello " + helloService.findPerson(1).getFirstName());
        return "home";
    }

}
```

- **home.html (in src/main/resources/templates)**

```
<html xmlns:th="http://www.thymeleaf.org">
    <head>
        <title>My First Spring Web Page</title>
        <link rel="stylesheet" th:href="@{/main.css}" />
    </head>
    <body>
        <h1>Your first message from Spring:</h1>
        <h2>
            <span th:text="${message}"></span>
        </h2>
    </body>
</html>
```

# Hello World Example - Web Version (cont.)

- **main.css (in src/main/resources/static/css)**

```
h1{
    color:#0000FF;
}
h2{
    color:#FF0000;
}
```

- **application.properties**

```
server.port=8081
server.servlet.context-path=/helloworld
```

- **run the main application class**
- **open a browser and type url: http://localhost:8081/helloworld**

## The Basics 2.1

- configuration is annotation-based
- @GetMapping maps to the url requested by the client
- wire in your Service class(es)
- request-handling method can have any name
- model will be passed between controller and view
- return String value: logical name of the view that renders the result
- add features:
  - validation
  - message conversion
  - field formatting

# Accessing Request Input



- query parameters
  - use @RequestParam attribute in controller method
  - pass in URL via localhost:8080/home?myparam=x
- path variables
  - use @RequestMapping(value="/home/{myparam})
  - use via @PathVariable attribute in controller method
  - use in URL via localhost:8080/home/x
  - RESTful notation!
- form parameters
  - see 4 Processing Forms on page 420

# Example

- **MyController.java**

```
@Controller
public class MyController {

    @Autowired
    ReceptionService receptionService;

    @GetMapping("/persons/{id}")
    public String welcomePerson(@PathVariable("id") int id, Model model) {
        model.addAttribute("message", "Hello " +
                receptionService.getHelloService().findPerson(id).getFirstName());
        return "welcome";
    }

    @GetMapping("companies")
    public String showComps(@RequestParam("country") String country,
                                    Model model){
        List<Company> countryCompanies=
                receptionService.getCompaniesForCountry(country);
        model.addAttribute("comps",countryCompanies);
        return "companylist";
    }

}
```

- **Path Variable:** localhost:8080/home/persons/2
- **Request Parameter:** localhost:8080/home/companies?country=BE

# The View - Using Thymeleaf



- Java XML/XHTML/HTML5 template engine
- provides full Spring Framework integration
- Open-Source Software, licensed under the Apache License 2.0
- full (and extensible) internationalization support
- configurable, high performance parsed template cache that reduces input/output to the minimum
- uses the th-namespace:
  xmlns:th="http://www.thymeleaf.org"
- more info:
  - http://www.thymeleaf.org/
  - http://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html

# Example

- **companylist.html**

```
<html xmlns:th="http://www.thymeleaf.org">
    <head>
        <title>My First Spring Web Page</title>
    </head>
    <body>
        <h1>List of Companies</h1>
        <table border="1">
            <thead>
                <tr>
                    <th>Name</th>
                    <th>Town</th>
                </tr>
            </thead>
            <tbody>
                <tr th:each="comp : ${comps}">
                    <td th:text="${comp.name}"></td>
                    <td th:text="${comp.town}"></td>
                </tr>
            </tbody>
        </table>
    </body>
</html>
```

## Creating a Form with Thymeleaf 4.1

- standard HTML form tag
  - th:action will point to controller method
  - th:object will link with a model object (needs to be bound in the GET phase)
- basic inputs
  - <input type="text" th:field="*{myprop}"/>
    - will bind to property myprop on object
    - does conversion!
  - <input type="password" th:field="*{pass}"/>
  - <input type="hidden" th:field="*{id}" />
- buttons and links
  - <input type="submit" value="click"/>
  - <a href >

# Example

- **companyForm.html**

```
<html xmlns:th="http://www.thymeleaf.org">
    <head> <title>Create a Company</title></head>
    <body>
        <h1>Add a new Company</h1>
        <form action="#" th:action="@{/companyForm}"
                th:object="${company}" method="post">
            <input type="hidden" th:field="*{companyNumber}"/>
            <table>
                <tr>    <td>Name:</td>
                        <td><input type="text" th:field="*{name}" /></td> </tr>
                <tr>    <td>Street:</td>
                        <td><input type="text" th:field="*{street}" /></td> </tr>
                <tr>    <td>Number:</td>
                        <td> <input type="text" th:field="*{number}" /></td> </tr>
                 <tr>   <td>Zip Code: </td>
                        <td> <input type="text" th:field="*{zipCode}" /></td> </tr>
                <tr>    <td>Town:</td>
                        <td> <input type="text" th:field="*{town}" /></td> </tr>
                <tr>    <td>Country:</td>
                        <td> <input type="text" th:field="*{country}" /></td> </tr>
             </table>
            <input type="submit" value="Add Company"/>
        </form>
    </body>
</html>
```

# Processing Form Input



- new Company has to be created when calling the page
- fields in the request will be bound to the object that was passed
- Controller

```
@Controller
public class MyController {
    @Autowired
    ReceptionService receptionService;
    @GetMapping("/companyForm")
    public String showForm(Model model){
        Company c = new Company();
        c.setCompanyNumber
                (receptionService.getCompanyRepository().getKeyValue());
        model.addAttribute("company", c);
        return "companyForm";
    }
    @PostMapping("/companyForm")
    public String addForm(Model model,
                          @ModelAttribute("company") Company company){
        CompanyRepository comprepo = receptionService.getCompanyRepository();
        comprepo.insertCompany(company);
        return "redirect:/home";
    }
    // other getmapping methods
}
```

# Transporting data between pages



- add RedirectAttributes in the “PostMapping” method
- use addFlashAttribute method to add the object containing the data
- redirect to the new page
- add a @ModelAttribute object in the heading of the GetMapping method of the page you redirected to, with the name of the flash attributes

# Example

```
@Controller
public class MyController {
    @Autowired
    ReceptionService receptionService;
    @GetMapping("/companyForm")
    public String showForm(Model model){
        Company c = new Company();
        c.setCompanyNumber
                        (receptionService.getCompanyRepository().getKeyValue());
        model.addAttribute("company", c);
        return "companyForm";
    }
    @PostMapping("/companyForm")
    public String addForm(Model model,
                                @ModelAttribute("company") Company company,
                                RedirectAttributes attrs){
        CompanyRepository comprepo = receptionService.getCompanyRepository();
        comprepo.save(company);
        attrs.addFlashAttribute("addedcomp",company);
        return "redirect:/home";
    }
    @GetMapping("/showCompData")
    public String showData(Model model,
                                    @ModelAttribute("addedcomp") Company company){
        model.addAttribute("addedcomp", addedcomp);
        return "showCompData";
    }
}
```

# APPENDIX A. EXERCISES

The purpose of this integrated track is to study the possibilities of the frameworks and incorporate them into a traditional business case.

The current activities at **ABIS Training and Consulting** are supported by the **A**BIS **C**lient and **C**ourse **A**dministration (**ACCA**) system, which is used for the administration of courses, sessions, enrolments, persons, companies, ...

We will implement as a starting point the **use case to create an API for doing person administration**, i.e. create persons, update person information, retrieve person information.

Next the **enrolment process** will be considered. How can a customer enrol to a specific course session, enter all relevant data, and being informed about the enrolment status? Check which courses a person has followed?

**The exercises described in the following sections can be used to build up the application step by step, but are NOT restrictive. They are just ideas that you can build upon further.**



## *1 Spring Core*

1. Create a new Spring Boot project called PersonAPI in IntelliJ.
   Use the following configuration:

   type=maven, packaging=jar,
   group=be.abis, artifact=exercise,
   package=be.abis.exercise
   java version = 21

   Add Spring Web as dependency.
   In the pom.xml, set the version of Spring Boot to 3.5.7.

   Copy the packages/classes of the L:\Java\ExerciseSkeletons\SpringHZIV\ExerciseSkeleton directory under the correct packages. Put the persons.csv file under c:\temp\javacourses\hziv (check/change the location of the file in the FilePersonRepository class). Check the code and run the JUnit test.

2.

   a. Create a PersonService interface with the following methods:
      - List<Person> getAllPersons();
      - Person findPerson(int id);
      - Person findPerson(String emailAddress, String passWord) throws LoginException;
      - void addPerson(Person p) throws PersonAlreadyExistsException;
      - void deletePerson(int id);
      - void changePassword(Person p, String newPswd);

   b. Create a *AbisPersonService* class that implements this interface. Implement the methods by calling the equivalent PersonRepository methods.

   c. Configure the class by using @Service. Use constructor injection for the wiring of the PersonRepository.

   d. Use a Spring Boot JUnit test for the AbisPersonService to find the name of the person with id=1. Make sure that the PersonService interface is wired into your test class. You can also add tests for the other methods.

## 2 Advanced Configuration

1. Create a second class *SomePersonService* implementing PersonService.
   Make sure that *findPerson(email,password)* returns yourself. Use annotations to configure everything.
   Use profiles such that *AbisPersonService* is used in production, and *SomePersonService* while in development.
   Create a new Test class that uses the *dev* profile. All the other tests should still work by defining *prod* as the profile to be used in the *application.properties*.
2. Create a class *FallbackMemoryPersonRepository*, which contains the same data as the file, but now the persons are created in memory. This class should be used in production, but only in case the persons.csv file could not be found.
   Use *@ConditionalOnResource* to achieve this.
3. Pass the file path to be used in the FilePersonRepository class via runtime value injection, using a file called my.properties to define the value.

## 3 AOP

1. Create a LoginAspect class that logs each time after a person logs in, but also when there was a wrong login. You can just print a simple message in each case.
2. Create a timer for all find* methods in the repository package.



## 4 Spring REST

1. Create a *PersonController* class (in a package be.abis.exercise.controller).
   Add a method *findPersonByMailAndPwd(String email, String password)* that returns the full Person object in json format.
   The URL should be:
   http://localhost:8081/exercise/personapi/persons?mail=x&pwd=x
   Replace the “x” values with real data, and test in a browser.
   You can “ignore” the LoginException for now.
2. Add controller methods that correspond to (and call) all service methods. Choose the correct HTTP methods. Use the “good practices naming rules” for your URIs. Document the chosen URIs (in a simple text file). Test with Postman.
3. Change the *findPersonByMailAndPwd(email,pwd)* method so it uses POST instead of GET.
4. Change the “int age” variable to a “LocalDate birthDate” everywhere (you will also need to change the repository and the persons.csv file). Use d/M/yyyy as standard date format. Also experiment with the other Jackson annotations.
5. Add a method *List<Person> findPersonsByCompanyName(String compName).* You will also have to implement the method in the repository and the service layers. Add a test in the PersonRepositoryTest class.
   Let the contoller return XML instead of JSON when calling the method.
   Use Jackson JAXB to do the mapping.
6. Change the *addPerson()* method such that it can consume XML (but JSON should also still be possible).

## 5 _____ Creating REST Clients

1. Add Spring WebFlux to the PersonAPI. Convert the following methods from the service and controller classes to return Mono and Flux types:
   * getAllPersons()
   * findPerson(int id)
   *void changePassword(Person p, String newPswd)
   *deletePerson(int id)
   *findPersonsByCompanyName(String compName)
   -> bonus points if you keep the XML output

   You can drop the SomePersonService class as from now.
2. 
   a. Create a new Spring Boot project called “TrainingAPI”. This will be the client, which will use your PersonAPI and the provided CourseAPI.
   b. Add a PersonService interface with a *AbisPersonService* class that implements methods to call the corresponding methods from the PersonAPI. Make sure that the calls are fully reactive. Don’t handle any exceptions yet. Test with JUnit.
3. 
   a. Clone the CourseAPI which you can find at https://github.com/sschillebeeckx/CourseApiHZIV . Check out the code, and test via Postman.
   b. In the TrainingAPI, add a CourseService interface with a *AbisCourseService* class that implements methods to call the corresponding ones from the CourseAPI. In this case the calls should behave in a synchronous way. Only implement the GET methods for now. Use JUnit to test again.

## 6 _____ Exception Handling and Validation

1. Make the findPerson(email,password) and addPerson(Person p) methods in the PersonAPI reactive. Make sure to handle the exceptions correctly. Test via JUnit. What happens in the API when wrong values are passed?
   Bonus: the @AfterThrowing advice of AOP should keep working.
2. Add a CourseNotFoundException on the findCourse(int id) method in the CourseAPI (stdart from the CourseRepository level). Handle the exception with ResponseEntity and ProblemDetail. Test via Postman and also adapt the client (TrainingAPI).
3. Add exception handling to the addPerson() method in the PersonApi. Use ResponseEntity again. Test via Postman and also adapt the client (TrainingAPI).
4. Add exception handling to following methods:
   *findPerson(email,password)
   *findPerson(int id)
   *findPersonsByCompany(String compName)

   Use centralized exception handling this time.
5. Use validation in the CourseAPI to check whether the short title of a course is filled in, and has maximum 15 characters. Also make sure the number of days is a positive number. Test with Postman, and add handling in the TrainingAPI client.
6. In the PersonAPI, check whether the password of a new person is more than 5 characters. Test with Postman, and add handling in the TrainingAPI client.

## 7 _____ Database Configuration

1. Do the following in the PersonAPI, CourseAPI and TrainingAPI:
   a. Add dependencies for spring-boot-starter-jpa and an Oracle driver in the pom.xml.
   b. Add following DB settings in a application.properties file:
      Replace the “xx” by a number given to you by your instructor.

      ```
      spring.datasource.url = jdbc:oracle:thin:@//delphi.abis.be:1521/TSTA
      spring.datasource.username= tu000xx
      spring.datasource.password= tu000xx
      spring.datasource.driver-class-name=oracle.jdbc.OracleDriver
      spring.datasource.hikari.maximumPoolSize=1
      spring.jpa.hibernate.ddl-auto=none
      spring.jpa.show-sql=true
      spring.jpa.properties.hibernate.format_sql=true
      ```

   c. Add a DataSourceTest class, which checks whether the connection was set up correctly.
2. 
   a. In the database view of IntelliJ, create a new Oracle datasource, by copying the datasource url. Go to advanced->expert options, and make sure that the “introspect using JDBC metadata” option is enabled.
   b. Run the createOracleDB.sql script via the query console. Check the tables via the explorer and perform some basic queries in the console.
3. Copy the provided classes from the L:\Java\ExerciseSkeletons\SpringHZIV\trainingAPI under the correct packages in the TrainingAPI.
   Check out the code. Run the JUnit tests and also call the API methods.

## 8 ____ Entity mappings and Basic JPA repositories

1. In the CourseAPI:
   a. Add correct mappings in the Course class to mimic the DB correctly. Use GenerationType.IDENTITY for the courseId.
   b. Replace the CourseRepository interface with a CourseJpaRepository, which has to be able to call the same methods as before. Use provided methods and query methods for this. Make sure to fetch a course (by id), before you update it.
      Wire the new repository into the CourseService class, and make sure that all methods are implemented. Let the addCourse() and updateCourse() methods return a Course object as from now.
   c. Create a CourseServiceTest class to test all methods (including the DML ones). What happens in the DB? Now add @Transactional on the DML tests, and check again. Also make sure that when running all tests, they nicely run in isolation. Don’t use @Order annotations anymore.
2. Test via the Course API. Take a look at the generated SQL via the console in IntelliJ. When adding a new course, make sure not to pass the id anymore!
3. EXTRA In the TrainingAPI, Implement the methods of the AbisCourseService that were missing so far. Test via the CourseAPIController (Postman).

## 9 _____ Association Mapping and @Query

1.

   a. Add mappings in the Company and Address classes to map to the companies table.

   b. Create a CompanyJpaRepository class. Make sure to add methods to find a company by id and to find by name.

   c. Define a method to find a company by name and town. Use @Query with JPQL for this.

   d. Use a CompanyRepositoryTest class to test these methods, as well as the DML methods.

2.

   a. Add the correct mappings in the Person class to mimic the DB correctly. Replace the PersonRepository with a PersonJpaRepository, which has to be able to call the same methods as before.

   b. Wire this in the PersonService. The “add” and “update” methods should return Mono<Person>.

   c. Test via JUnit. Consider following use cases when adding a person:
      - person working for a new company
      - person working for an existing company
      - person without a company

   d. Test via the API.

   e. EXTRA Test via the PersonAPIController in the TrainingAPI

3. In the TrainingAPI, add mappings in the Session class and create a Session-JpaRepository. Wire the repository in the AbisTrainingService. Implement the *findSessionsForCourse(String courseTitle)* method. Don’t bother about cancelled sessions yet. Use JPQL. Add a JUnit test, and also test via the TrainingAPIController to find all courses for DB2BAS.



## 10 ___ More querying techniques and performance

1. In the PersonAPI:

   a. Add a PersonDTO class, such that the API will only show a person’s id, first name, last name, email, birthdate, company name and company town.

   b. Provide a PersonForm class, which passes all person data (including company and address), but in a more flat structure.

   c. Make sure that all (necessary) JSON/XML annotations are removed from the Person class, and now only appear in the PersonForm and PersonDTO classes.

   d. Foresee a PersonMapper class that transforms a Person object to and from the Form/DTO objects. Call the mapper in the PersonAPIController methods.

   e. Test via the PersonAPI.

   f. Test via the TrainingAPI Don’t forget to adapt the “model”. Adjust your package structure.

2. In the TrainingAPI

   Adjust exercise 3.3. as to return a SessionDTO object (defined as a record) from the TrainingController, which contains the session number, start date, instructor’s first name and last name, kind, cancelled indicator and long course title.
   Do the mapping from inside the AbisTrainingService this time.

3. Make sure the query now only returns the non-cancelled sessions, by using a native query.

4. In the TrainingAPI

   a. Add mappings in the Enrolment class. Use a “composite” id for the session and enrolmentInSession columns.

   b. Add a NamedQuery to find all (non-cancelled) enrolments for a person. Only fetch the e_pno, sdate and s_cid columns in the result. Create an EnrolmentJpaRepository that calls the named query.

   c. Create an EnrolmentDTO record that mentions the person’s first name and last name, his/her company name, the startdate of the session and the course title.

   d. Implement the *Flux<EnrolmentDTO> findEnrolments(int personId)* method in the AbisTrainingService and the TrainingApiController.

   e. Add a JUnit test in the TrainingServiceTest class and test the API.

## 11 _____ Exception Handling and Transactions

1. In the PersonAPI, let the PersonAlreadyExistsException (thrown when the person's emailaddress already exists in the DB) work again. Test via JUnit and Postman in the PersonAPI and via JUnit in the TrainingAPI.
2. In the TrainingAPI, create a method to delete a session by id. Throw a SessionCanNotBeDeletedException in case there are still enrolments for the session. Test via JUnit.
3. In the CourseAPI, create a method to change the price of a course. Make sure to use a modifying, native query to do this, which only changes the caprice column. Also add the necessary in the AbisCourseService class and test via JUnit.
   Extra: add a corresponding method in the API and test.
4. Add a method in the SessionJpaRepository to cancel a session. Make sure to use a modifying, native query to do this, which only changes the scancel column. Also add the necessary in the AbisTrainingService class and test via JUnit. Make sure the test also works if it is not defined as @Transactional.
   Extra: add a corresponding method in the API and test.
5. Implement the method *enrolForSession(PersonForm person, int sessionId) method* in the AbisTrainingService class. Make sure it is transactional. Throw the EnrolException if something goes wrong, and make sure everything is rolled back in that case. Add JUnit test cases in the TrainingServiceTest.
   Extra: Implement the method in the TrainingApiController and test via Postman.

## 12 _____ Advanced Topics

1. In the PersonAPI, add a List<**String**> hobbies in the Person class, linking to the *hobbies* table. Add methods to retrieve the hobbies of a person, and to add a new hobby. Test via JUnit.
2. In the TrainingAPI, create 2 subclasses of the Session class: CompanySession and PublicSession. The classes can be left "empty" for now. Make sure that the *skind* column is used as the discriminator column. Test via the return type of a *findById(int id)* method in the SessionJpaRepository class. Also try to add a companySession object in the table.



## 13 _____ Spring Security

1. Although everybody can log in (findPerson(email,pwd)) and can call changePassword, a person should only be capable of changing his OWN password (after logging in).
   Implement this by using API keys:
   - create a ApiKeyStore which holds an API key for each user id
   - send this API key to the client (TrainingAPI) when the user logs in correctly.
   - when a user wants to change a password, he has to send his API key as well
   - service checks whether the API key is correct
2. Configure Spring Security in the CourseAPI. Create an in-memory user store, with 2 users:
   - abis01 which has role AbisUser
   - abis02 which has roles AbisUser and AbisAdmin

   Call the *findCourse(int id)* method via a browser and via Postman to check what happens. Authenticate yourself to solve the issue.
3. Everybody should be able to find courses, but only the users stated above should be able to perform the other actions. Use basic authentication to achieve this. Test in Postman, and also adjust the client to provide the authentication data when needed. Test via JUnit in the client.
4. Only the user with AbisAdmin role should be able to DELETE courses. Implement authorization to do this. Test via JUnit in the client.
5. Add CORS to the CouresAPI, such that it can be reached by Angular applications. These run typically on port 4200.

1.

   a. Create a new Spring Boot project called Frontend. Add spring-web, spring-webflux and thymeleaf as dependencies.
   Create a TrainingService that calls the TrainingAPI. You only need to implement the findPerson(email,password) method. Don't bother about exceptions and api keys.

   b. Add a *WebController* class that wires in the TrainingService. Copy the provided welcome.html page under the resources\templates folder. Adjust the page with a "hello" message to the person with id 1. Show the firstName and lastName. The page should be available via the URL http://localhost:8083/exercise/welcome.

2. Create a login screen where a user needs to provide its email and password. Once logged in, the previously created page should be shown.

3. Add a logout button on the welcome page.