Learning Spring Boot 2

Bogdan Solga

Training overview

The training overview presents:

The training overview and goals

A short introduction of myself

An overview of the training setup and structure

Roadmap

Roadmap

Training overview and goals

Training recommendations

Short self-description

Training overview and goals

3m

Training overview

- The ‘Learning Spring Boot 2’ training is organized in 2 days of 4 hours per day
- Each day will consist of several sessions
- Each session will consist of:
  - A theoretical presentation part
  - An example presented by myself
  - A short hands-on example, on which we will work together
- GitHub link: https://github.com/PacktPublishing/Learning-Spring-Boot-2

Training goals

- Learn the Spring Boot 2 features and benefits
- Learn a series of best practices for Spring Boot based projects
- See and analyze multiple examples
- Work together to understand the presented topics

Short self presentation

2m

Bogdan Solga

- Java trainer and coach
- 10 years of experience in developing, architecting and testing enterprise Java projects
- 2 years of training and coaching experience

Training recommendations

2m

Training recommendations

- Ask questions, during the presented topics and at the end
- Give feedback, during the presented topics and at the end

Spring Boot intro

Lesson Objectives

In this lesson we will learn...

What is Spring Boot

The most important Spring Boot features and benefits

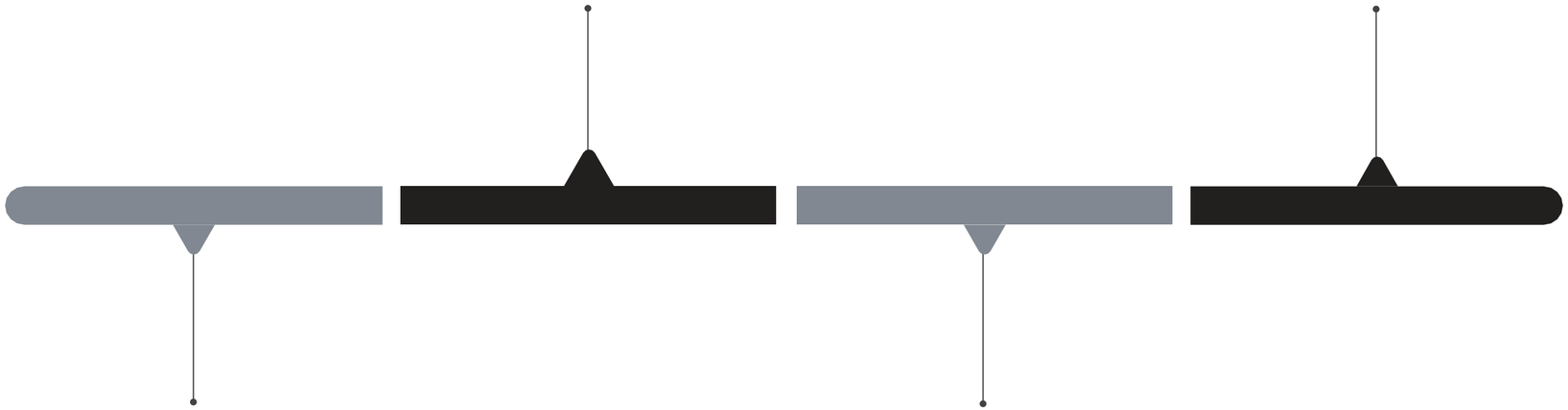
How to create a new Spring Boot project

The new features of Spring Boot 2

How to use Spring Boot's auto-configuration feature

Roadmap

Roadmap


New features in Spring Boot 2
Auto-configuration overview
Spring Boot intro
Spring Boot apps best practices

Goals

- Learn why and when we should use Spring Boot
- Learn how to generate a new Spring Boot project
- See and work together on several examples

Spring Boot intro

20m

Features and benefits

Spring Boot intro

- Set of tools & conventions for a faster start ('boot') of a Spring application
- Provides several tools to ease the usage of the Spring Framework
  - Starter modules → faster way to integrate and use application facets
  - Configuration conventions → easier usage of configuration files and profiles
- Standalone deployments → apps built using Spring Boot:
  - Run as an executable .jar file
  - Include an embedded web server
- Wide range of non-functional features - monitoring, metrics, health checks

Illustration

Spring Framework vs Spring Boot

SPRING FRAMEWORK

SPRING BOOT

Generating a new Spring Boot project

- Three main ways:
  - From the Spring Initializr site - http://start.spring.io/
  - From IntelliJ IDEA: File → New → Project → Spring Initializr
  - From the command line - using the Spring Boot CLI

All are using the Spring Initializr REST API

The Spring Initializr site

- Quick generation of a Spring Boot project
- Allows the user to choose:
  - The Spring Boot version (defaults to the latest RELEASE version)
  - The build tool (Maven / Gradle)
  - The main package
  - The used dependencies (‘ingredients’)

End result - ‘turn-key’ solution for the infrastructure of a new project

Demo

Spring Boot usage demo

Using the Spring Initializr site - https://start.spring.io/

Activity

Generating a new Spring project using the Spring Initializr site, analyzing its structure and new elements

Scenario:

Generating a new Spring Boot project from scratch

Aim:

Understanding how to create a new Spring Boot project

Steps to generate the project

1. Open the Spring Initializr site - https://start.spring.io/

2. Select the used:
   a. Build tool (Maven or Gradle) → we will work with Maven
   b. Programming language (Java, Kotlin or Groovy)
   c. Spring Boot version → defaults to the latest RELEASE version

3. Enter the project's:
   a. GroupId and artifactId
   b. Name, description and packaging
   c. Used Java version (8 or 11, the latest LTS releases)
   d. Ingredients - we'll choose just 'Web', for now

Steps to verify the generated project

1. Download and unzip the generated zip file

2. Open the project in IntelliJ IDEA

3. Verify the correct functioning of the generated project:
   a. Start the main application
   b. Verify the startup status in the console → we should see the message:

```
Started DemoApplication in 1.5 seconds
```

Analyzing the project structure

On the ‘less is more’ premise, the generated project has:

- one Maven file with:
  - two dependencies - web and testing support
  - one plugin → responsible for running and building the app
- one class → the main project class
- one test class (for the main class)
- one .properties file (→ will be further presented)

Spring Boot - new projects best practices

- Use the dependencies versions from the ‘spring-boot-starter-parent’ project, by either:
  - extending the project from the ‘spring-boot-starter-parent’ project
  - importing the Spring Boot BOM (Bill Of Materials) dependency
- Use the versions from the ‘spring-boot-dependencies’ module
- If the project uses multiple modules:
  - in the parent POM file - define the dependencies in the ‘<dependencyManagement>’ tag
    - the dependency version must be included
  - in each module - use the dependency only by its groupId and artifactId

Summary

In this lesson we learned…

- What is Spring Boot
- The most important Spring Boot features and benefits
- How to create a new Spring Boot project
- The new features of Spring Boot 2
- How to use Spring Boot's auto-configuration feature
- A few good practices for new projects

Q & A session

- Please ask your questions on the presented topics

New features in Spring Boot 2
Auto-configuration

Goals

- Spring Boot 2 - overview and new features
- Learn how to use auto-configuration in a Spring Boot project
- See and work together on several examples

Spring Boot 2

Overview and new features

10m

Spring Boot 2

- Released at the beginning of 2018
- New features and improvements:
  - **Updated Java baseline** - requires Java 8+ (Java 8 and 11 are recommended)
  - Built-in **Reactive Programming support** → its WebFlux component is built on top of 'Project Reactor', one of the main Reactive Programming libraries
    - Uses an embedded (reactive enabled) Netty server, instead of Tomcat
  - **Upgraded core components**:
    - Spring Framework: 5+
    - Tomcat: 8.5+
    - Hibernate: 5.2+

Spring Boot 2 - new features
(continued)

- Improved and simplified security configuration
  - Can be configured via the configuration files and the auto-configuration support
- Refactored and improved Actuator
- HTTP/2 support for Tomcat and Jetty
- Support for configuring Quartz via auto-configuration
- Refactored metrics support → now based on Micrometer
- Support for the Kotlin programming language

Auto-configuration overview

10m

Spring Boot auto-configuration overview

- **Auto-configuration** - automatic classpath scanning for `@Configuration` classes, for a simplified and easier project configuration
- Eases the integration of various application facets, added to the classpath via **starter modules** (further presented)
- Useful for the initial configuration of a project
  - Can/should be gradually replaced by specific `@Configuration` files

Spring Boot auto-configuration overview
(continued)

- Excluding configuration files - using:
  - The 'exclude' param of `@SpringBootApplication`/`@EnableAutoConfiguration`
  - The '`spring.autoconfigure.exclude`' configuration parameter, listed as:
    - In properties files - comma separated values
    - In YAML - list of items

- Listing all the used auto-configurations → starting the app with '`--debug`'

→ hands-on overview

@SpringBootApplication structure

- `@EnableAutoConfiguration`
  - Enables the classpath scanning for `@Configuration` annotated classes
  - Available from the classpath dependencies ('starter' modules)
    - Examples:
      - Finding the Tomcat config bean → configures the embedded servlet container
      - Finding a data-source config bean → configures the database used in the project
- `@ComponentScan`
  - Scans the classpath for `@Bean`, `@Configuration` and `@Component` classes
- `@SpringBootConfiguration`
  - Marker annotation for the main configuration of a Spring Boot application

→ hands-on overview

Starter modules (continued)

- **'Starter' module** - a set of carefully selected dependencies, used to add functionalities to a project
- Available as Maven dependencies; examples:
  - Database access, via JPA: `spring-boot-starter-data-jpa`
  - Testing support `spring-boot-starter-test`
  - Web `spring-boot-starter-web`
- Should inherit their versions from the '`spring-boot-dependencies`' module

Starter modules (continued)

- Each starter module contains a lists of curated & tested dependencies, for each functionality
- They are upgraded when new Spring Boot versions are released

- Advised naming pattern:
  - Official starter modules: `'spring-boot-starter-<name>'`
  - Unofficial starter modules: `'<name>-spring-boot-starter'`

Demo

Auto configuration support demo

Using the Spring Initializr site - https://start.spring.io/

Activity

Adding a new Spring Boot ‘starter’ module to our project

Scenario:

Adding a new Spring Boot ‘starter’ module to our project

Aim:

Understanding how to add a new facet to a Spring Boot project

Steps to add a new ‘starter’ module

1. Open the pom.xml file in IntelliJ IDEA
2. Add the following dependency in the ‘dependencies’ tag:

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

3. Start the application (from the main() method of the main class)
4. Result - the application log should contain the following message:

```
Tomcat started on port(s): 8080 (http) with context path ''
```

5. That confirms that the application now starts an embedded Tomcat server

Summary

In this lesson we learned…

- An overview of the new features from Spring Boot 2
- An overview of the auto-configuration support from Spring Boot
- The anatomy of the `@SpringBootApplication` annotation
- An overview of the Spring Boot starter modules
- How to add a new facet to a Spring Boot project, by adding a starter module

Q & A session

- Please ask your questions on the presented topics

Configuration files, Profiles and Spring web support

Goals

- Learn how to use configuration files in a Spring Boot project
- Learn how to use Profiles in a Spring Boot project
- Learn how to use the Spring Web component, expose a few simple REST endpoints

Configuration files

10m

Configuration files in Spring Boot

- Default configuration file - application.properties / application.yml
- Can be used interchangeably
  - Preference - by readability and needs (further discussed)
- Properties format:
  - The default ‘key=value’ format
  - Flat structure, repetitive items
- YAML format (.yml extension):
  - Hierarchical data definition format → superset of the JSON format
  - Tree structure for the config options
  - More complex structures (lists, maps, classes)

Configuration files - entries types

- Two types of configuration entries:
  - Predefined
    - Automatically used by Spring Boot
    - Can be wired using `@Value` (further presented)
  - User defined
    - Wired using the `@Value` annotation
- Valid for both formats (.properties and .yml)

Configuration files - using .properties files

```
# a comment example
spring.config.name=application
spring.application.name=
```

Comment

Default value

Config properties without a default value

Category Sub-category

Referring to the Spring Boot appendix values

Hands-on: using properties files

Configuration files - using .yml files

```
# a comment example
spring:
    application:
        name: 'the app name'
```

Comment

Category

Sub-category

Key value

Hands-on: using YAML files

Open discussion - using properties or YAML files

- What format do you consider more:
  - Easy to use
  - Error prone
- Why?

Loading non-default properties files

- @PropertySource - loading non-profile .properties files (*not* .yml files)
- Usage - on one/more `@Configuration` annotated classes
- The config files are loaded from:
  - The 'resources' folder (by default)
  - The classpath → 'classpath:' prefix
  - The filesystem → 'file:' prefix

Demo

Configuration files usage demo

Loading non-default configuration files

Loading a products.properties file from the user's home folder

Activity

Adding a configuration file to our project

Scenario:

Adding a configuration file to our project

Aim:

Hands-on exercise -- integrating a configuration file in our project

Steps to add a configuration file

1. Open the project in IntelliJ IDEA (if it's not already open)
2. Create either a properties file or a YAML file in the `src/main/resources` folder
3. Add the following entries in it:
   a. if you're using a properties file:

```
server.port=8888
custom.property=a custom value
```

   b. if you're using a YAML file:

```
server:
    port: 8888
custom:
    property: 'a custom value'
```

Steps to add a configuration file (continued)

1. Create a package named '`com.packtpub.learning.spring.boot.service`'
2. Create a class named `PropertiesUsageService` in it
3. Add the `@Component` annotation on it
4. Add the following field in it:

```
@Value("${custom.property}")
private String customProperty;
```

5. Add a method with the following body in the class:

```
@PostConstruct
public void initialize() {
   System.out.println("The custom property is '" + customProperty + "'");
}
```

6. Start the main class →
   1. The web server should start on the port 8888 (the port is displayed in the console)
   2. The custom property should be displayed

Summary

In this lesson we learned…

- An overview of the supported configuration files in Spring Boot 2
  - Properties files
  - YAML files
- The way to use both configuration formats in a Spring application
- How to add a new configuration file and configuration entry to a project

Q & A session

- Please ask your questions on the presented topics

Configuration files, Profiles and Spring Web support

Goals

- Learn how to use configuration files in a Spring Boot project
- Learn how to use Profiles in a Spring Boot project
- Learn how to use the Spring Web component, expose a few simple REST endpoints

Profiles, configuration files per profile

10m

Spring Profiles

- **Spring bean profile** mechanism - registering beans *only* in some scenarios
- Common use-cases - development environments (dev, QA, prod)
  - Dev environment - in-memory database
  - QA environment - some profiling tool
  - Prod environment - a real database server (Oracle, PostgreSQL etc)
- **Usage**:
  - **Definition** - using the `@Profile` annotation, added on classes / methods
  - **Specifying** the used profile(s):
    - From the command line - using '`spring.profiles.active`' as a system property
    - As an OS environment variable - using the SPRING_PROFILES_ACTIVE property

Spring Profiles [continued]

- Interfaces can have multiple (bean) classes → using a class per profile
  - Each class will be used only when the corresponding profile is active
- Profiles are orthogonal - multiple profiles can be used at the same time
  - Their definition and management → programmer's responsibility
  - Examples: using different web server profiles and development environments
- 'default' profile - active if no profiles are specified
  - Specifying the default profiles → 'spring.profiles.default=<profile-name>'

Hands-on: using the Spring `@Profile` annotation

Spring Boot - extending the Profiles usage

- Spring Boot extends the @Profile support from Spring
- Per-profile configuration files: application-<profile-name>.(yml | properties)
  - Automatically loaded by Spring Boot, when specifying a profile
- Application properties hierarchy:
  - The profile specific files are *extending* the main config file (application.props|yml)
  - The config entries are first loaded from the main config file
    - If a property is defined for a profile -- that value has prevalence
    - If a config property is found in both files - the one from the profile config is used

Demo

Demoing the usage of Spring & Spring Boot Profiles

Using configuration files per profile

We will see the automatic usage of a ‘dev’ profile

Activity

Defining a profile and adding its configuration file in the project

Scenario:

Adding a new Spring profile to our project

Aim:

Understanding the steps needed to add a new profile to a Spring Boot project

Steps to add a new Spring profile

1. Choose a short name for the new profile - example: int (from integration)
2. Create a file for the new profile - application-int.properties or application-int.yml
3. Add a custom property in the new created file; example: 'custom.int.property'
   a. Make sure to format the custom property according to the used file type
4. Create a class named `RunProfiles.java` class in the main package
   a. Keeps the defined profiles in a structured mode, avoids hard-coding them
5. Add a public constant for the new profile:

```
public static final String INT = "int";
```

Steps to add a new Spring profile

1. Set the new profile in the application, by using a SpringApplication object:

```
final SpringApplication application = new
SpringApplication(NewSpringProfileDemo.class);
application.setAdditionalProfiles(RunProfile.INT);
application.run(args);
```

2. Add a @Value annotated field in the app, to tell Spring to wire the value there

```
@Value("${custom.property}")
public String customProperty;
```

3. Add a `@PostConstruct` annotated method to display the configured property:

```
@PostConstruct
public void displayProperty() {
  System.out.println("The custom property is " + customProperty);
}
```

Steps to add a new Spring profile

1. Start the application (from the main() method of the main class)

2. Observe the custom property being displayed

Summary

In this lesson we learned…

- An overview of Spring profiles:
  - What they are
  - How they can be used in a Spring Boot based application
  - How they can help structuring and using a Spring application
- An overview of how we can use Spring profiles
- An overview of the way Spring Boot extends the Spring profiles support
- How to add a new Spring profile to an existing application

Q & A session

- Please ask your questions on the presented topics

Configuration files, Profiles and Spring Web support

Goals

- Learn how to use configuration files in a Spring Boot project
- Learn how to use Profiles in a Spring Boot project
- Learn how to use the Spring Web component, expose a few simple REST endpoints

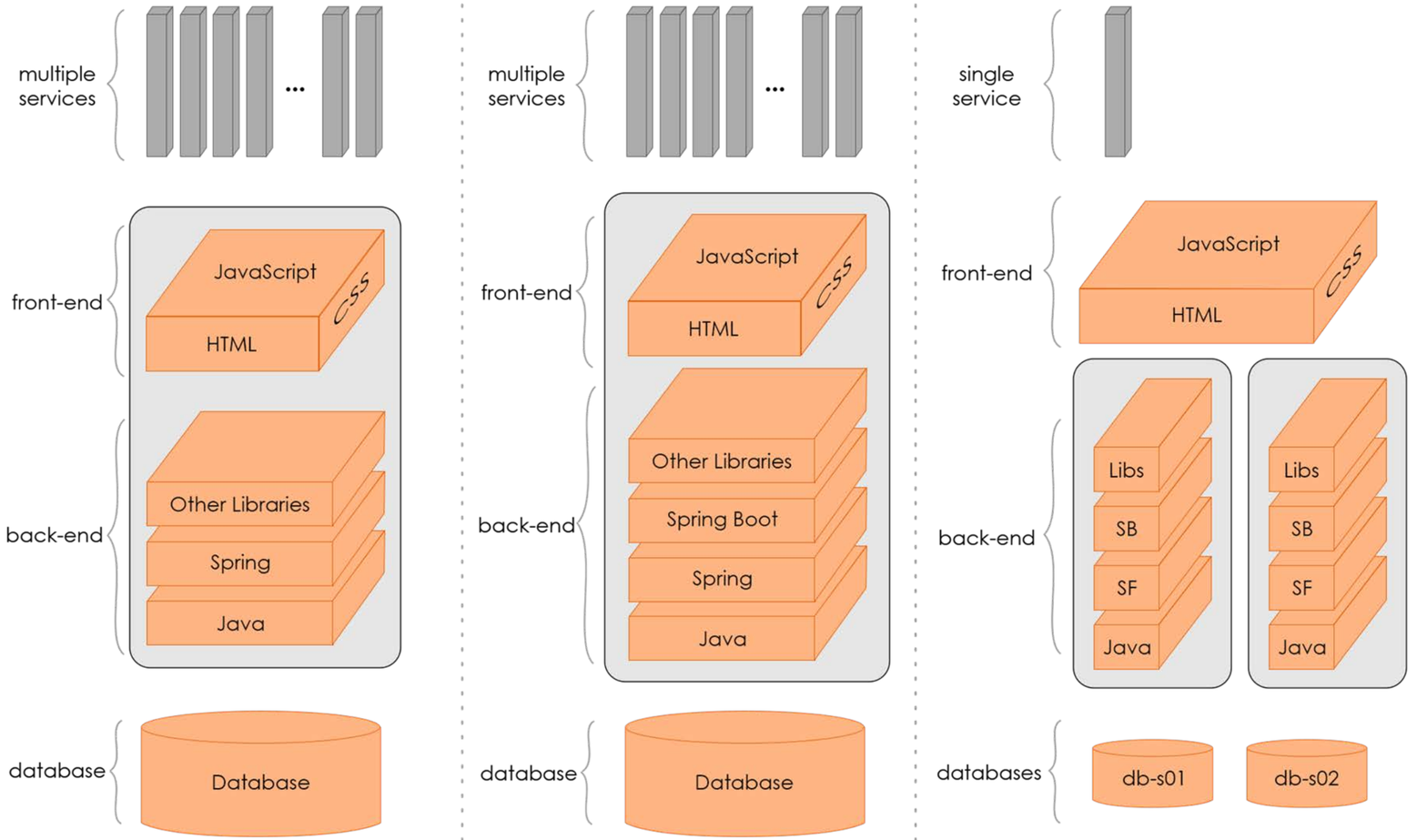
Spring Web overview, short REST demo

10m

Web Applications - overview

- Webapp - a Java application which is:
  - Deployed and ran from a web server
  - (Usually / mostly) accessed from a web browser
- Exposes web accessible endpoints, for CRUD requests - types:
  - MVC
  - REST
- Requests: served by servlets - Java code ran by a web server
  - Tomcat, in our case
- Packaging - done by the build tool (Maven / Gradle)

Enterprise Java architectures - application structuring phases



Spring / Java Based monolith

Deployed in web / application server

1

Spring and Spring Boot monolith

ran as stand-alone application,
starts an embedded web server

2

Spring and Spring Boot and Spring Cloud micro-service

ran as stand-alone services,
each service starts an embedded web server

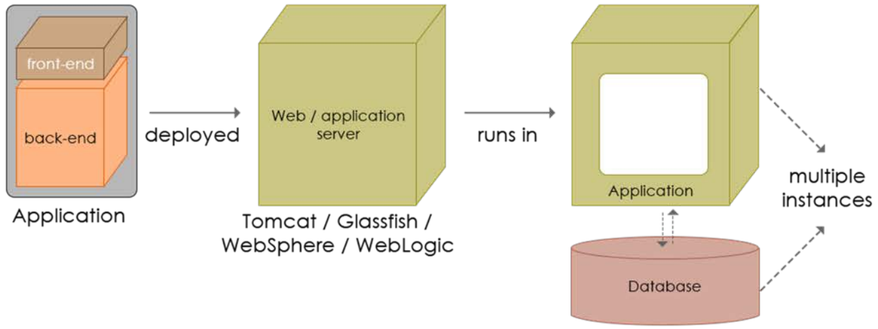
3

Enterprise Java architectures - deployment modes

1

Java / Spring monolith

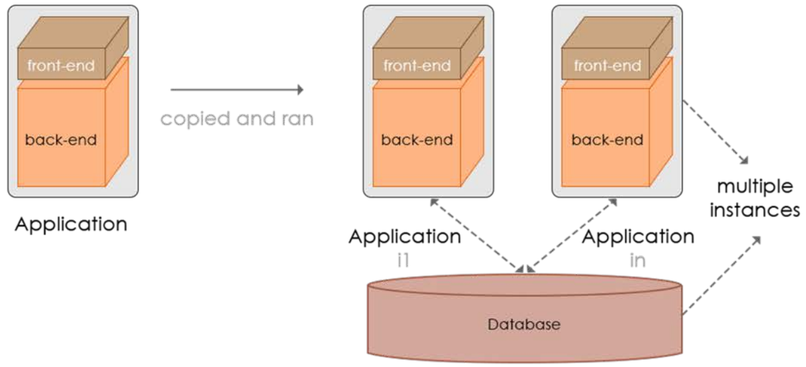
deployed in a web / application server



2

Spring and Spring Boot monolith

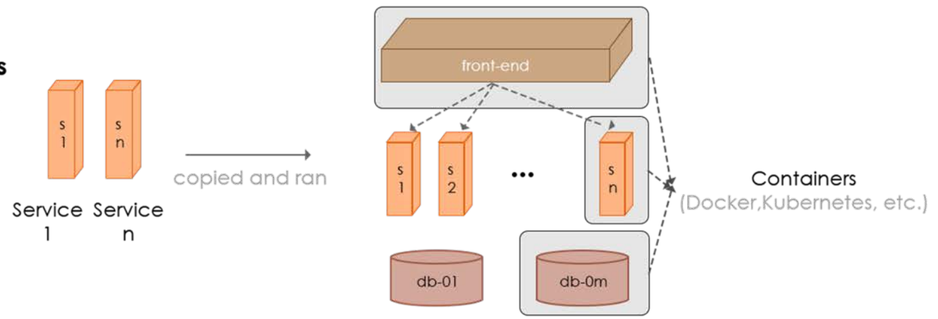
- ran as a stand-alone application
- starts an embedded web server



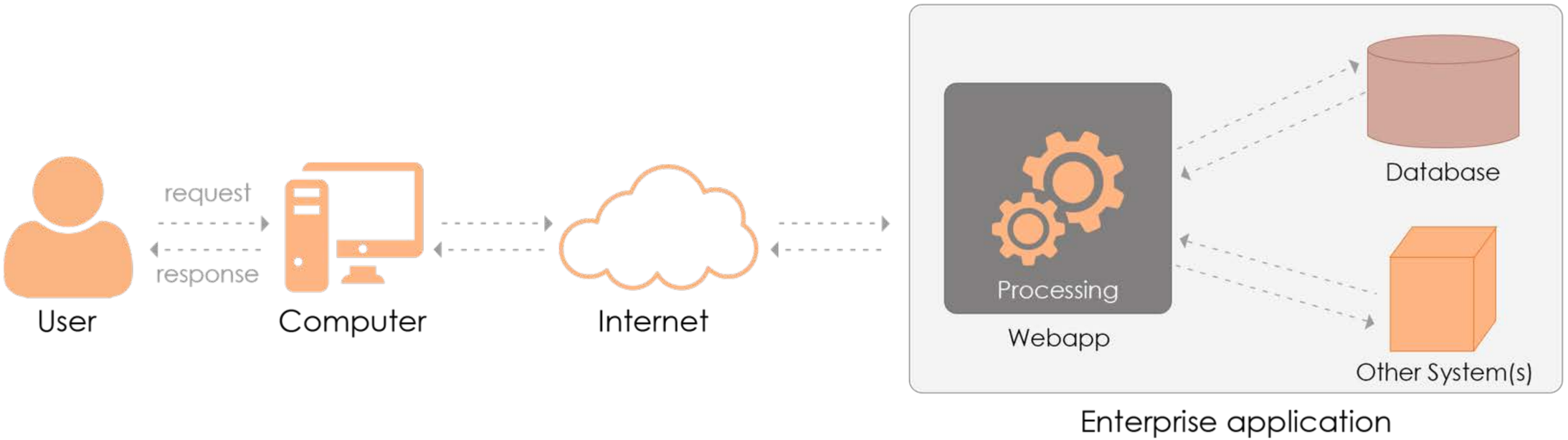
3

Spring, Spring Boot & Spring Cloud micro-services

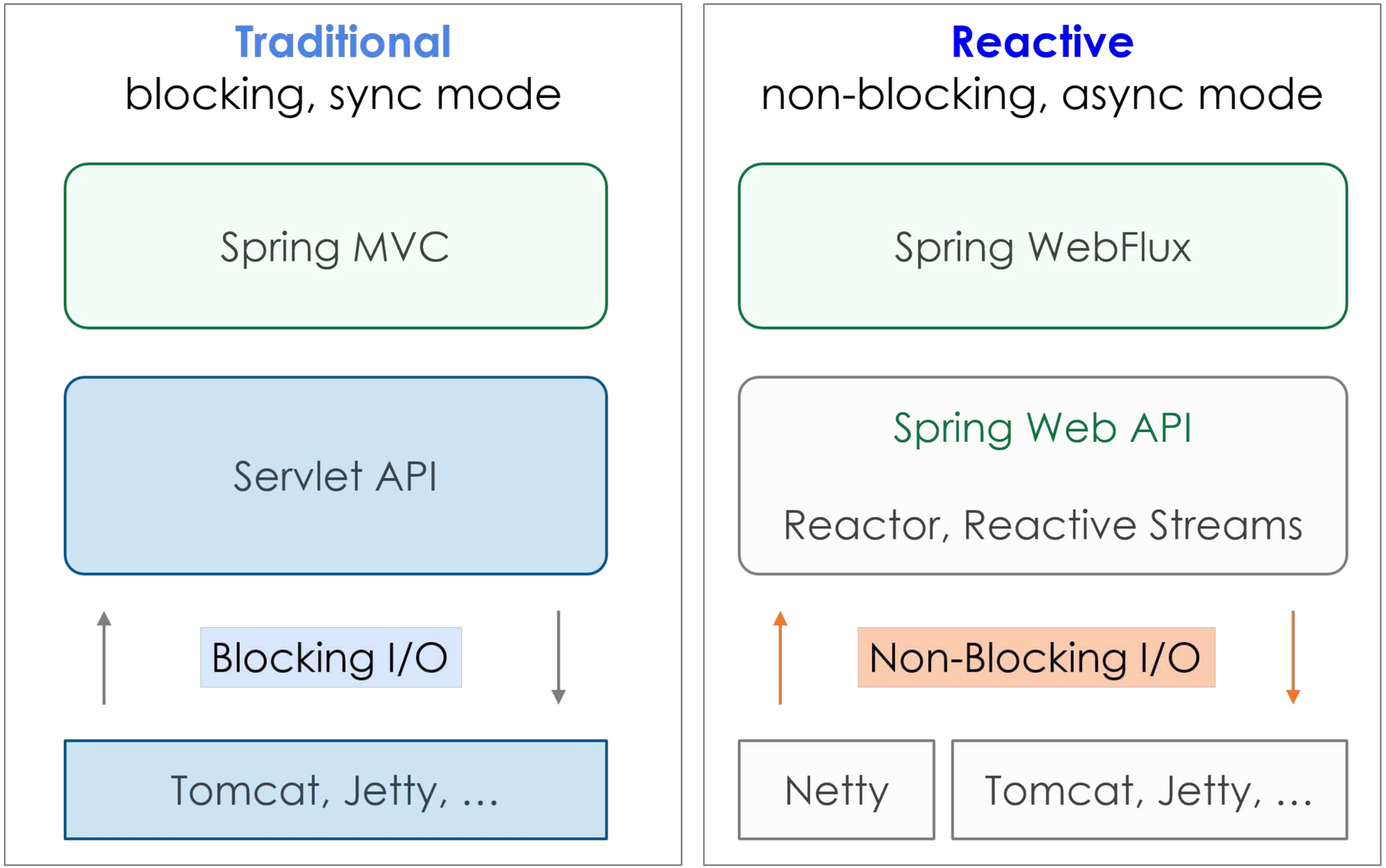
- ran as stand-alone services
- each services starts an embedded web server



Using web applications - high level overview



Spring Web - two usage modes



REST & MVC applications - request & response lifecycle

- Every access to a webapp is called an HTTP request
- It is always followed by an HTTP response

- Wrapping interfaces: HttpServletRequest & HttpServletResponse

- The **core servlet method** which processes them - `service`(req, res)

```
void service(HttpServletRequest req, HttpServletResponse res)
```

- The **main Spring servlet** - `DispatcherServlet` → dispatches web requests to the `@Controller` and `@RestController` annotated classes

Tomcat

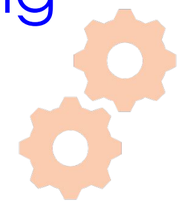
- The most used open source Java web server
- Runs web applications in two modes:
  - Traditional deployment:
    - Webapps are deployed as .war files (web archive) → tomcat/webapps
    - Tomcat unpacks, validates and runs the deployed archive
    - Can run multiple webapps in parallel - they must have different context paths
  - Embedded deployment - ran from a main() method → a single webapp
- Context path - the path on which a webapp can be accessed ('/', by default)

Spring Web - requests handling

- Designed around:
  - The `'DispatcherServlet'` servlet - the core Spring requests dispatching servlet
  - The @Controller, @RestController and @RequestMapping annotations

- **Handlers** - classes annotated with:
  - MVC architecture - `@Controller`
  - REST architecture - `@RestController`

- **Handler mappings** - mapped requests paths, using @RequestMapping



‘Hello, Spring web!’ anatomy

```
@RequestMapping(
    method = RequestMethod.GET,
    path = "/hello"
)
public @ResponseBody String helloSpring() {
    return "Hello, Spring web!";
}
```

- `@RequestMapping(` — Defines a handler mapping
- `method = RequestMethod.GET,` — The used HTTP methods
- `path = "/hello"` — The defined request path(s)
- `public @ResponseBody String helloSpring() {` — Return the response in the response body
- `return "Hello, Spring web!";` — The returned object

Main HTTP methods and response codes

- **HTTP methods**:
  - POST - write content (C)
  - GET - read content / pages (R)
  - PUT - update content (U)
  - DELETE - delete content (D)
  - OPTIONS and PATCH - retrieve endpoint options / characteristics
- Response codes:
  - 200 - OK
  - 400 - Bad request
  - 401 - Not authorized
  - 404 - Not found
  - 500 - Internal server error

Main web related annotations

- `@RequestMapping` - maps web requests on specific handler classes / methods
  - Supports many configuration parameters (further presented)
- `@RequestParam` - a method parameter bound to a web request parameter
- `@PathVariable` - a method param bound to a URI template variable
- `@RequestBody` - a method param bound to the method's request body
- `@ResponseBody` - a method return value bound to the request's response body
- `@RequestHeader` and `@CookieValue` - binding request headers and HTTP cookies

Demo

Spring web demo

A simple REST endpoint

We will see a simple REST endpoint exposed through a REST controller

@RequestMapping

- Used to define request handlers - classes and methods
  - Can be used on both, for hierarchical mappings
- The most configurable annotation - it can specify:
  - The HTTP methods
  - The request paths (multiple can be defined)
  - The request and response types (further presented)
  - Header options

@RequestParam and @PathVariable

- **@RequestParam** - used to specify *request* parameters:
  - On the request path, the parameters:
    i. begin with '?'
    ii. are chained via '&'
  - Example: ?color=blue&weight=light

- **@PathVariable** - used to send *path* parameters:
  - Specified by the '{<parameter-name>}' syntax, separated by '/'
  - Example: '{first}/{second}

@RequestParam and @PathVariable

- **@RequestParam** - used to specify *request* parameters:
  - On the request path, the parameters:
    i. begin with '?'
    ii. are chained via '&'
  - Example: ?color=blue&weight=light

- **@PathVariable** - used to send *path* parameters:
  - Specified by the '{<parameter-name>}' syntax, separated by '/'
  - Example: '{first}/{second}

Demo

Spring web demo

A few @RequestParam and @PathVariable examples

@RequestBody and @ResponseBody

- @RequestBody - map an HTTP request payload (body) to a method param
  - Used especially for POST, PUT and PATCH requests (GET is not sending payloads)
- @ResponseBody - return an object in the response body
  - Automatically included for `@RestController` mapped controllers

Demo

Spring web demo

A few @RequestBody and @ResponseBody examples

*Mapping annotations

Spring Framework 4.3+ - shorter mapping annotations:

- `@PostMapping → @RequestMapping & method = HttpMethod.GET`
- `@GetMapping`
- `@PutMapping`
- `@DeleteMapping`

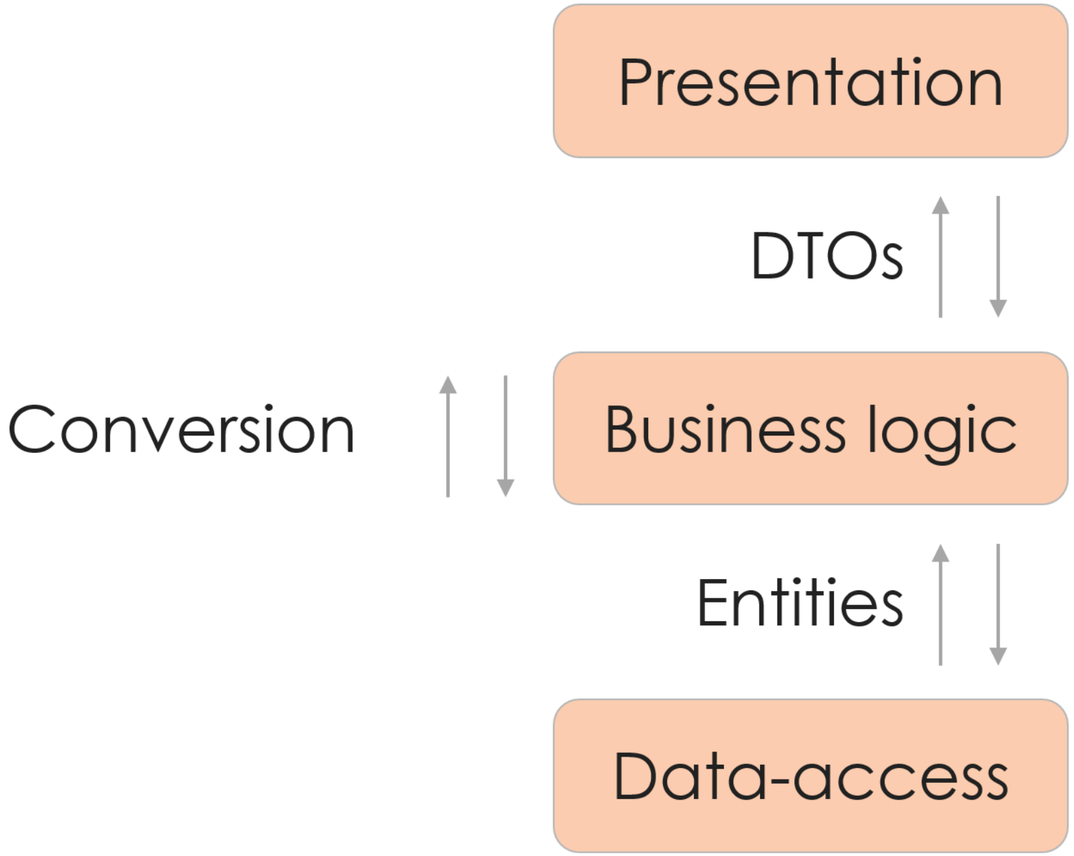
Spring OXM support

- **OXM** - Object to XML mapping support
  - Also applies to the JSON format
- Automatic **serialization** / **deserialization** of / into Java objects
  - Also called **marshalling** / **unmarshalling**
- Out-of-the box support for multiple content types:
  - `text/plain`
  - `application/xml` (JAXB -> @XmlRootElement annotation)
  - `application/json` (using the Jackson library → default in 'spring-web')

Using DTOs - benefits and drawbacks

- **DTO** (Data Transfer Object) → design pattern used to decouple the data transfer between the presentation layer and the data-access layer
- Benefits:
  - Decouples the two layers in terms of data representation
  - Keeps the JPA entities clean from pres. layer related annotations
- Drawbacks:
  - Requires an extra effort to be written and converted (to/from)
  - Duplicates the properties of some JPA entities, if they are serialized in their entirety

Using DTOs - visual representation



Activity

Adding REST support to the project, exposing a few REST endpoints

Scenario:

Adding a few REST endpoints to our project, to learn the usage of:

- `RequestMapping`
- `RequestParam`
- `PathVariable`

Aim:

Understanding the usage of the main Spring Web annotations

Steps to add a few REST endpoints

1. Create a class named SimpleRESTEndpoints
2. Annotate it with `@RestController` (presented more in-depth soon)
3. We will create a few methods and expose them via REST endpoints together

Summary

In this lesson we learned…

- An overview of web applications (in general)
- An overview of the Java web applications and their:
  - Application structuring phases / stages
  - Deployment phases / stages
- An overview of the web applications:
  - High level overview usage
  - Request and response lifecycle
- An overview of the two main Spring Web usage modes
  - MVC and REST
  - WebFlux
- An overview of the Spring REST main annotations

Q & A session

- Please ask your questions on the presented topics

Spring web support continued, exception and error handling

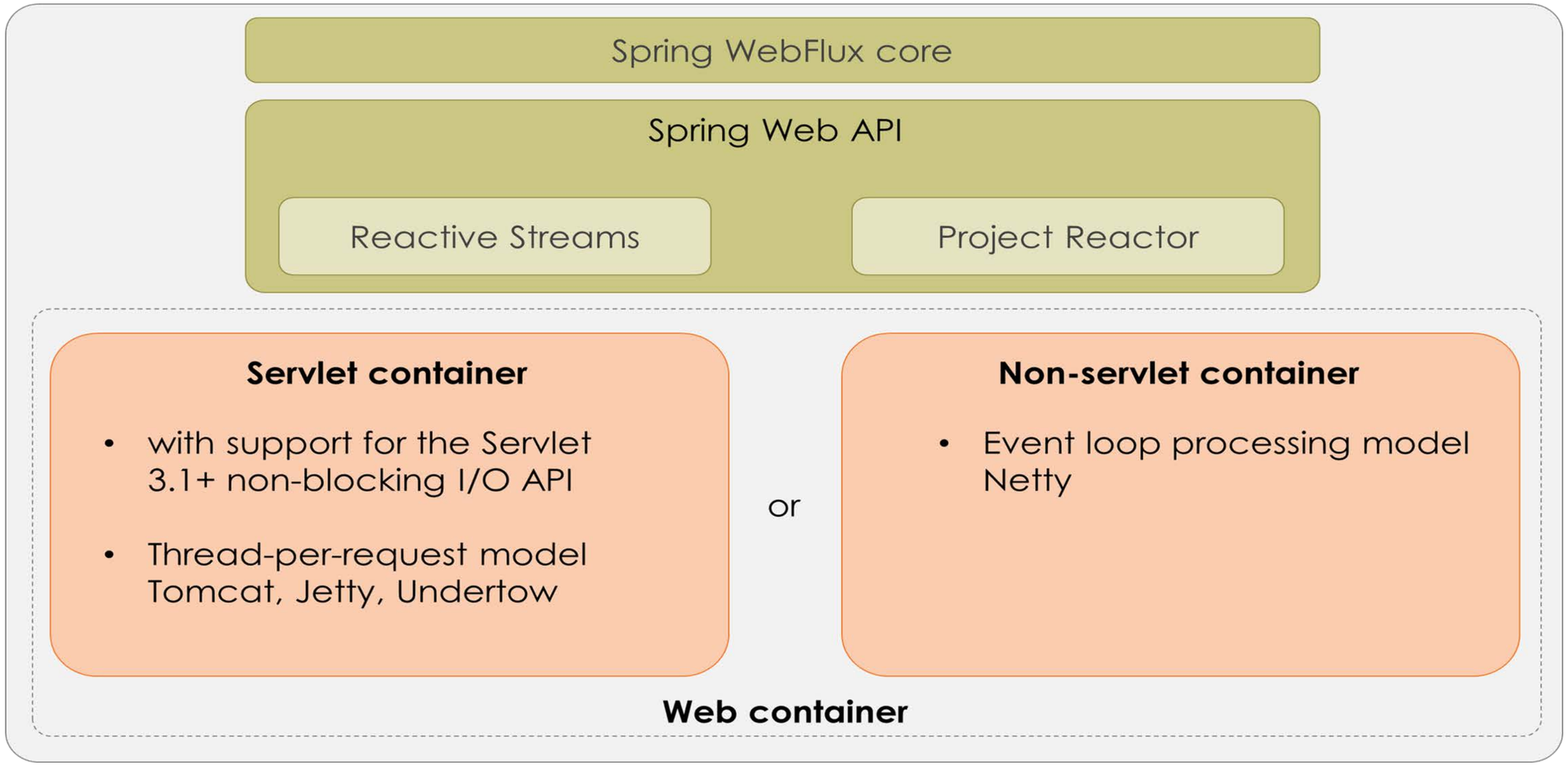
Goals

- Learn how to use configuration files in a Spring Boot project
- Learn how to use Profiles in a Spring Boot project
- Learn how to use the Spring Web component, expose a few simple REST endpoints

Spring WebFlux overview, Reactive Programming overview

10m

Spring WebFlux overview



Reactive Programming - short overview

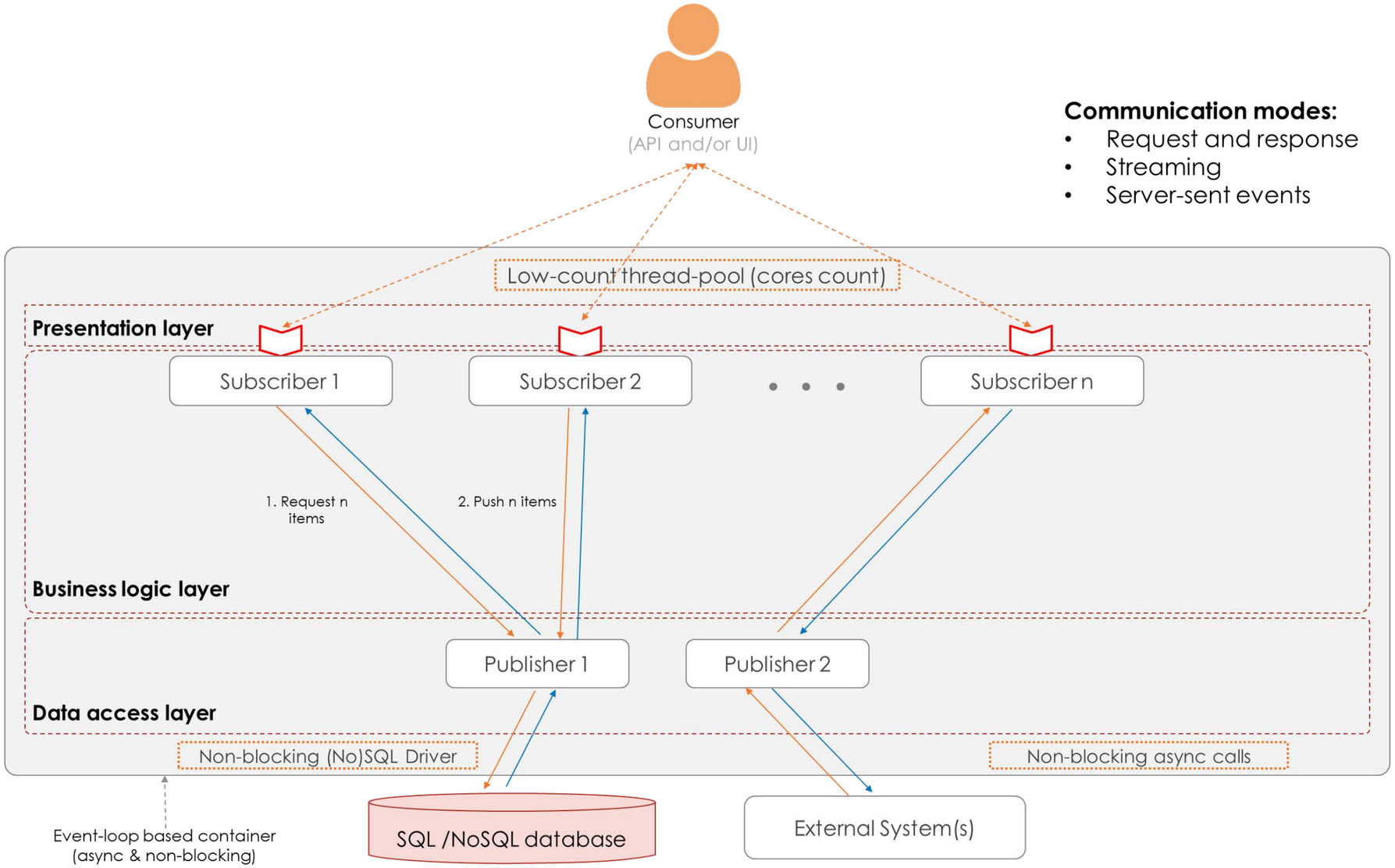
- **Core** concept - programming model that is built around:
  - Reactivity to changes (hence the name 'reactive')
  - Availability and processability of events/messages
- **Benefits**:
  - Completely non-blocking and asynchronous processing
  - Improved (/optimal) hardware utilization → better performance from the same hardware resources
- **Challenges:**
  - Cannot coexist with blocking codebases -- JDBC and Servlet-API, mainly
  - Non-blocking SQL access is still in development
  - Still in the early stages of adoption

Reactive Programming, Systems and Streams

Four main Reactive notions used in the Java programming ecosystem:

- **Programming** - the programming model used for creating Reactive Systems
- **Systems** - architectures/systems designed to be reactive to change
- **Streams** - an initiative to provide a standard for asynchronous stream processing, with non-blocking back-pressure
  - A collaboration between engineers from Netflix, Pivotal, Red Hat, Twitter, Typesafe
- **Manifesto** - an initiative to standardize and unify the characteristics of Reactive Systems

Reactive Programming model



Communication modes:

- Request and response
- Streaming
- Server-sent events

Project Reactor

- A 4th generation reactive library, one of the most used
- The core of Spring WebFlux, the Reactive Programming component from Spring 5
- Implements:
  - The Reactive Streams API specifications
  - The Reactive Extensions specifications → compatible with ReactiveX & RxJava
- Two main publishers:
  - Mono - a publisher of 0 or 1 items
  - Flux - a publisher of 0 to n items, potentially unbounded (infinite)

Project Reactor characteristics

- Efficient data demand management → built-in back-pressure handling
- Integrates directly with the Java 8 API → can return:
  - Streams API
  - `CompletableFuture`
  - `Duration`
- A rich set of operators that align with the ReactiveX operators:
  - `buffer`
  - `compose / concat`
  - `window`
  - `zip`

The Mono publisher

- **Mono** = a Reactive Streams Publisher with basic flow operators
- Emits 0 → 1 elements
- Many static factory methods for generating a publisher (further pres.)
- Completes eiter:
  - Successfully - after emitting 0/1 element → the .onComplete() method
  - With an error → the .onError() method
- Can be used in implementations and as return types (Spring WebFlux)

Demo

A few simple `Mono` publisher examples

The `Flux` publisher

- **Flux** = a Reactive Streams Publisher with basic flow operators
- Emits 0 → n elements → *potentially unbounded (infinite)*
- Many static factory methods for generating a publisher (further pres.)
- Completes eiter:
  - Successfully - after emitting all elements → the `.onComplete()` method
  - With an error → the `.onError()` method
- Can be used in implementations and as return types (Spring WebFlux)

Demo

A few simple `Flux` publisher examples

Reactive REST endpoints

- When using Spring WebFlux, endpoints can be defined in two ways:
  - Declarative: Spring MVC style `@Controller` & `@RestController` endpoints
  - Functional: returning functional reactive endpoints (further presented)
- For both definition modes, the endpoints can return the presented Reactor publishers:
  - `Flux`
  - `Mono`
- They will be handled by the underlying stack:
  - Servlet 3.1+ (with non-blocking I/O): Tomcat, Jetty, Undertow
  - Spring WebFlux: Netty

Declarative REST endpoints

```
@GetMapping("/mono")
public Mono<Product> mono() {
  return Mono.just(new Product(1, "iSomething", 2000));
}

@GetMapping("/flux")
public Flux<Product> flux() {
  return Flux.just(
    new Product(1, "First", 100), new Product(2, "Second", 200)
  );
}
```

Demo

Exposing Mono and Flux objects as declarative REST endpoints

Functional REST endpoints

- Declaration - returning a RouterFunction<ServerResponse> @Bean
- Configuration - chaining CRUD requests/responses in a functional way

```
rf = route(GET("/mono"), request -> ok().body(mono, Product.class))
  .andRoute(GET("/flux"), request -> ok().body(flux, Product.class));

// dynamic linking / registration
if (condition) {
   rf.andRoute(GET("/l"), request -> ok().body(just(1L), Long.class));
}
```

Demo

Exposing Mono and Flux objects as functional REST endpoints

Activity

Adding Spring WebFlux to our project, exposing a few reactive endpoints

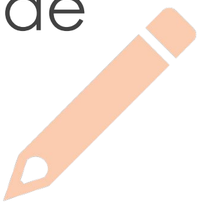
Scenario:

- Adding the Spring WebFlux component to our project
- Exposing a few reactive REST endpoints in our project in a:
  - Declarative mode
  - Functional mode

Aim:

Understanding:

- how to integrate Spring WebFlux in a project
- how to expose REST endpoints in a declarative and functional mode



Steps to add the Spring WebFlux component

1. Open the pom.xml file
2. Replace the dependency with the artifactId:

   ```
   <artifactId>spring-boot-starter-web</artifactId>
   ```

   with the artifactId:

   ```
   <artifactId>spring-boot-starter-webflux</artifactId>
   ```

3. Clean the project → run a Maven ‘clean’ task from the IDE (we will do it together)

Steps to add a few declarative reactive REST endpoints

1. Create a package named

   'com.packt.learning.spring.boot.controller.reactive' in our project

2. Create a class named `DeclarativeRESTController` in the new package

3. Annotate it with the `@RestController` annotation

4. Expose:

   a. A REST endpoint which returns a Mono<Product>

   b. A REST endpoint which returns a Flux<Product>

      For both endpoints - use the static factory methods from Mono and Flux

5. Start the application and invoke the exposed endpoints from a browser

Steps to add a few functional reactive REST endpoints

1. Create a package named 'com.packt.learning.spring.boot.config.reactive' in our project
2. Create a class named FunctionalRESTConfig in the new created package
3. Annotate it with the @Configuration annotation
4. Create a method which returns a @Bean of RouterFunction<ServerResponse>
   a. We will write the code for them together
5. Start the application and invoke the functionally exposed endpoints from a browser

Summary

In this lesson we learned…

- An overview of:
  - the Reactive Programming paradigm
  - Spring WebFlux
  - the main Publisher types from the Project Reactor library - Mono and Flux
- How to expose reactive endpoints in a:
  - declarative mode
  - functional mode

Q & A session

- Please ask your questions on the presented topics

Configuration files, Profiles and Spring Web support

Goals

- Learn how to use configuration files in a Spring Boot project
- Learn how to use Profiles in a Spring Boot project
- Learn how to use the Spring Web component, expose a few simple REST endpoints

Serving static content

10m

Serving static content

- Static content - HTML, JavaScript, CSS, images, fonts
- The following folders are configured (by default) to serve static content:
  - /static
  - /public
  - /resources
- Customized by the '`spring.resources.static-locations`' property
- Spring MVC - supports the usage of static contents via:
  - JSP pages (with some limitations)
  - Templating engines:
    - Thymeleaf
    - FreeMarker

Serving static content

- In case the project uses a templating engine (JSP, Thymeleaf) - the application will be used in an MVC mode:
  - It will use `@Controller` annotated classes, not `@RestControllers`
  - Each controller will return:
    - the name of the page which will render the response
    - the model and view details that will be populated in the rendered page

Demo

A simple static page example

Activity

Adding a short example of a static content to the project

Scenario:

- Adding a new HTML page to our project
- Accessing the new page from a browser

Aim:

Understanding:

- How to use and configure the static content in a Spring Boot project
- How to add new static content to a Spring Boot project

Steps to add a static folder to a Spring Boot project

1. Navigate to the src/main/resources folder of our project
2. Create a folder named ‘static’ in the ‘resources’ folder
3. Create a file named ‘index.html’ in the ‘static’ folder
4. Add some HTML content in the new created file → we will write it together
5. Start the project
6. Open a browser and enter the address http://localhost:8080
7. Expected result - we should see the rendering of the created HTML page

Summary

In this lesson we learned…

- An overview of:
  - the Reactive Programming paradigm
  - Spring WebFlux
  - the main Publisher types from the Project Reactor library - Mono and Flux
- How to expose reactive endpoints in a:
  - declarative mode
  - functional mode

Q & A session

- Please ask your questions on the presented topics

Configuration files, Profiles and Spring Web support

Goals

- Learn how to use configuration files in a Spring Boot project
- Learn how to use Profiles in a Spring Boot project
- Learn how to use the Spring Web component, expose a few simple REST endpoints

Other web access methods - overview

10m

Main web access methods

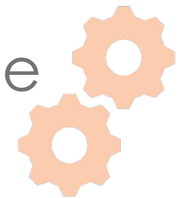
The most used web access methods (in recent projects):

- **REST**:
  - CRUD operations exposed via `@RestController` annotated classes
  - The endpoints access types is defined by the used HTTP methods
  - Using JSON/XML to send/receive data
  - Better decoupling between the frontend and backend components
- **MVC**:
  - CRUD operations exposed via `@Controller` annotated classes
  - The endpoints are returning page names
  - The data is filled using the returned model and view details
  - Tighter coupling between the frontend and backend components

Other web access methods

The main alternatives to the REST and MVC web access methods are:

- **Web Services (SOAP, RPC via XML)**:
  - Web endpoints exposed (usually) via POST or GET endpoints
  - The request and response payload is in XML format
  - The main inter-system communication method used until REST appeared
- **WebSockets, Server Sent Events (SSE)**:
  - Bi-directional communication between the server and the client
  - The client initiates the communication and then requests an upgrade towards bi-directional communication
  - Used especially in contexts when the server needs to send events to the client(s), without their prior request(s)



Summary

In this lesson we learned…

- A high-level overview of:
  - The differences between the REST and MVC web access methods
  - The main alternative web access methods:
    - Web Services (SOAP, RPC using XML)
    - WebSockets and Server-Sent Events (SSE)

Q & A session

- Please ask your questions on the presented topics

Configuration files, Profiles and Spring Web support

Goals

- Learn how to use configuration files in a Spring Boot project
- Learn how to use Profiles in a Spring Boot project
- Learn how to use the Spring Web component, expose a few simple REST endpoints

Exception and error handling

10m

Request and response content types

- **Request types** - sent by the HTTP client → specify the sent content type
- Usage - the 'Content-Type' request header
- Most used content types (in our context) - 'application/json'
- Other types:
  - `application/xml`
  - `application/text`
- **Response types** - specifies the returned response type → instructs the OXM serializer which response format to use (ex: XML / JSON)

Setting the HTTP request and response types

```
@PostMapping(
    path = "/product",
    consumes = MediaType.APPLICATION_JSON_VALUE
)
public ResponseEntity createProduct(@RequestBody ProductDTO product) {...}

@GetMapping(
    path = "/product",
    produces = MediaType.APPLICATION_JSON_VALUE
)
public List<ProductDTO> getProduct(@RequestParam String productName) {...}
```

Setting the HTTP response status

```
@PostMapping(
    path = "/product",
    consumes = MediaType.APPLICATION_JSON_VALUE
)
public ResponseEntity createProduct(@RequestBody ProductDTO product) {...}

@GetMapping(
    path = "/product",
    produces = MediaType.APPLICATION_JSON_VALUE
)
public List<ProductDTO> getProduct(@RequestParam String productName) {...}
```

Centralized exception handling

- **@ControllerAdvice** - centralized exception handling annotated class
- @ExceptionHandler annotated methods - exception type handling

```
@ExceptionHandler(IllegalArgumentException.class)
@ResponseStatus(value = HttpStatus.BAD_REQUEST)
public @ResponseBody String iae(IllegalArgumentException e) {
  return e.getMessage();
}
```

Defined handler

Returned HTTP status

Handled exception

Demo

A simple centralized exception handling example

Using the `ResponseStatusException` class

- Spring 5 introduced the `ResponseStatusException` class
- The exception class can be used to combine the:
  - returned exception message(s)
  - returned HTTP code(s)
- Can be seen as an alternative to the usage of @ControllerAdvice class and @ExceptionHandler methods
  - It provides a decentralized way to use exceptions from the code
  - Easier for initial prototyping, may lead to duplicated usage in the long term

Demo

Using the ResponseStatusException class

Activity

Adding exception handling support to our project

Scenario:

- Adding a centralized exception handling mechanism to a project
- Defining and implementing the main handled exceptions

Aim:

Understanding:

- How to add a centralized exception handling method to a project
- How to define and integrate custom exception classes

Steps to configure the centralized exception handling in a project

1. Navigate to the main project package (‘com.packt.learning.springboot’)
2. Create a package named ‘errorhandling’ in the main package
3. Create a class named `ExceptionHandlers` in the ‘errorhandling’ package
4. Add a few @ExceptionHandler annotated methods → we’ll add them together
5. Throw one / several of the defined exceptions from your business logic code
6. Start the application and invoke the exception throwing code from it
7. Expected result → the exceptions should be automatically caught by our defined exception handler

Summary

In this lesson we learned…

- An overview of the Spring Boot exception handling mechanism
- How to integrate a @ControllerAdvice annotated method to a project
- How to define several @ExceptionHandler annotated methods to the @ControllerAdvice class
- How to test the exceptions from a browser

Q & A session

- Please ask your questions on the presented topics

Database configuration, using an embedded database

Goals

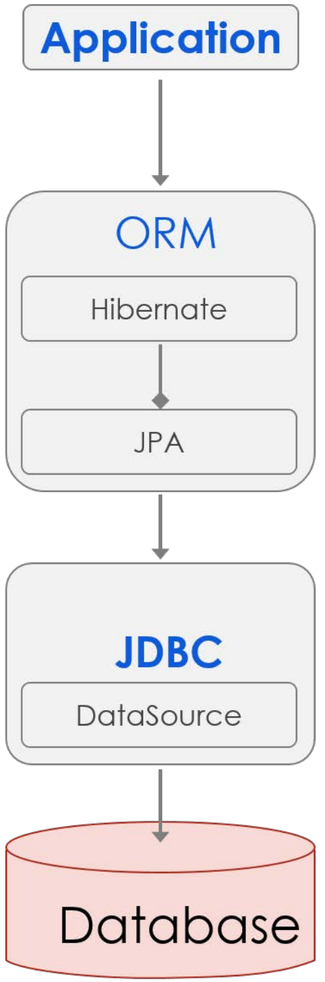
- Learn the main database actors used in a Spring Boot project
- Learn how to configure an embedded database in a Spring Boot project
- Learn how to persist a simple JPA entity in the database

Database access - actors, JDBC vs JPA

10m

Database access actors

- **JDBC** [Java DataBase Connection] - database access layer
- **`DataSource`** - interface used to provide db connections
  - Implemented by all the connection pools
- **ORM** [Object-Relational Mapping] - map classes to DB tables
- **JPA** [Java Persistence Architecture] - Java ORM API
- **Hibernate** - open source JPA implementation
  - The most used JPA implementation
  - Default when the 'spring-boot-starter-data-jpa' starter is used



JDBC or JPA - which one to choose?

- **JDBC** - if the database access time is critical
  - JdbcTemplate → methods for the most common DB operations
  - + the fastest and the most reliable access
  - - procedural access, more boilerplate code
- **JPA** - if the database access time is not critical and/or the OOP design is more important
  - Several ways to access a database via JPA:
    - Pure JPA access: using an `EntityManager` bean, using Spring Data JPA
    - Hibernate based access: using a `SessionFactory` bean

Database usage types

- Two types of database usage:
  - Embedded (on disk / in memory): H2, HSQL, Derby
  - Real databases: MySQL, PostgreSQL, Oracle etc
- Recommended usage:
  - Development - embedded database
    - Doesn't need anything pre-installed
    - Faster startup / teardown
  - Production - real database
    - Real-world usage - improved connection and transaction management, replication, etc
- Toggled via: profiles

Database design and bootstrapping modes

Two main modes:

- **Top-down**:
  - Creating the JPA entities
  - Setting the value of the 'spring.jpa.hibernate.ddl-auto' property to 'create' → the database will be created from the JPA entities
- **Bottom-up**:
  - Creating the SQL tables, usually using SQL scripts → setting the 'schema' and 'data' SQL script names
  - Exporting the JPA entities from the database
  - Fine-tuning the entities and tables relationships - FKs, indexes, cascades

Database initialization modes via config files

- Using JPA only: 'spring.jpa.generate-ddl' (true / false)
  → JPA implementation agnostic → keeps the config decoupled from it
- Using Hibernate: 'spring.jpa.hibernate.ddl-auto' → enum values:
  none, validate, update, create and create-drop (! for prod)
  → default values: 'create-drop' for embedded, 'none' otherwise
- Using DDL scripts → the 'spring.datasource.schema' and 'spring.datasource.data' properties → using SQL scripts for them

Using an embedded database

1. Add the 'spring-boot-starter-data-jpa' starter dependency
2. Add the Maven database dependency (H2 / HSQL / Derby)
3. Configure the database connection properties:

```
spring.datasource.platform=h2
spring.datasource.url=jdbc:h2:mem:database-name
spring.datasource.username=sa
spring.datasource.password=<choose-a-password>
```

4. Start the project

Demo

Studying the JPA integration in the project

- pom.xml dependencies
- the Product and Section JPA entities

Activity

Adding the embedded database support to the project, persisting an entity in it

Scenario:

- Adding the embedded database support to our project
- Persisting a JPA entity in it

Aim:

Understanding:

- How to add support for using an embedded database in a project
- How to persist a simple JPA entity in it

Steps to add the embedded database support in a project

1. Open the project's pom.xml file
2. Add the following two dependencies to it:

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

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

3. Add the datasource config in the project's application.(properties | .yaml) file

```
spring.datasource.platform=h2
spring.datasource.url=jdbc:h2:mem:learning-spring-boot
spring.datasource.username=sa
spring.datasource.password=learning-database-access
```

Steps to add the embedded database support in a project (continued)

1. Create a package named ‘com.packt.learning.springboot.domain’
2. Create two sub-packages named ‘entity’ and ‘repository’ in the new package
3. Create a `Product` class in the ‘entity’ package - we’ll add the properties together
4. Create a `ProductRepository` interface in the ‘repository’ package - we’ll configure it together

Steps to add the embedded database support in a project (continued)

1. Create a package named 'com.packt.learning.springboot.service'
2. Create a service class named `ProductService` in the new create package, annotate it with `@Service`
3. Autowire the `ProductRepository` in the `ProductService` class
4. Add a method named 'testProductSaving()' in the new class, so that we can test the product saving and retrieving
5. Annotate the method with `@PostConstruct`, so that it's automatically ran

Steps to add the embedded database support in a project (continued)

1. Add the following code in the 'testProductSaving()' method:

```
Product product = new Product(1, “Phone”);
productRepository.save(product);
System.out.println(“The product was successfully saved”);
Assert.isTrue(productRepository.count() == 1, “No products”);
```

2. Run the application and observe the displayed messages

Summary

In this lesson we learned...

- An overview of the the Java database actors and their correlations
- An overview of the differences between using JDBC and JPA
- How to integrate an embedded database in a Maven project
- How to create a simple JPA entity and a repository for it
- How to verify that an entity is persisted in the database

Q & A session

- Please ask your questions on the presented topics

Using a real database (PostgreSQL)

Goals

- Learn the main database actors used in a Spring Boot project
- Learn how to configure a real (PostgreSQL) database in a Spring Boot project
- Learn how to persist a simple JPA entity in the real database

Using a real database server - PostgreSQL

10m

Database design and bootstrapping modes

Two main modes:

- **Top-down**:
  - Creating the JPA entities
  - Setting the value of the 'spring.jpa.hibernate.ddl-auto' property to 'create' → the database will be created from the JPA entities
- **Bottom-up**:
  - Creating the SQL tables, usually using SQL scripts → setting the 'schema' and 'data' SQL script names
  - Exporting the JPA entities from the database
  - Fine-tuning the entities and tables relationships - FKs, indexes, cascades

Database initialization modes via config files

- Using JPA only: 'spring.jpa.generate-ddl' (true / false)
  → JPA implementation agnostic → keeps the config decoupled from it
- Using Hibernate: 'spring.jpa.hibernate.ddl-auto' → enum values:
  none, validate, update, create and create-drop (! for prod)
  → default values: 'create-drop' for embedded, 'none' otherwise
- Using DDL scripts → the 'spring.datasource.schema' and 'spring.datasource.data' properties → using SQL scripts for them

More info can be found here.

Connecting to a real database

The following steps are needed to connect to a real database:

1. Add the Maven dependency of the database driver
2. Set the datasource connection properties
   - driver-class-name data-source connection class (database dependant)
   - platform database platform / type (oracle, postgresql, mysql, etc)
   - name database name
   - username database connection user
   - password database connection password
3. Run the project

Demo

Studying the connection to a real database:

- pom.xml dependencies
- YAML configuration - Hibernate and JPA properties

Activity

Adding the support for the PostgreSQL database to the project, persisting an entity in it

Scenario:

- Connecting the project to a real database server
- Persisting a simple JPA entity in it

Aim:

Understanding:

- How to integrate a real database server in a Spring Boot project
- How to persist an entity in it
- How to verify the entity is successfully persisted

Steps to add the real database support in a project

1. Open the project's pom.xml file
2. Replace the H2 dependency with the following dependency:

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

3. Replace the datasource configuration properties with the following:

```
spring.datasource.driver-class-name=org.postgresql.Driver
spring.datasource.name=spring_boot
spring.datasource.password=postgres
spring.datasource.platform=postgresql
spring.datasource.username=postgres
spring.datasource.url=jdbc:postgresql://localhost:5432/spring_boot
```

Steps to add the real database support in a project (continued)

1. Verify the following database exists in your local PostgreSQL instance
   - Database name: spring_boot
   - Database user: postgresql
   - Password: postgres
2. If the database does not exist - we can create it with the following commands:

```
CREATE USER postgres WITH PASSWORD 'postgres';
CREATE DATABASE spring_boot;
GRANT ALL PRIVILEGES ON DATABASE spring_boot TO postgres;
```

3. Run the application and observe the displayed messages

Summary

In this lesson we learned…

- An overview of the the database design modes
- An overview of the integration of a real database
- How to reuse the previously created:
  - service
  - repository
  - entity
- How to verify that an entity is persisted in the database

Q & A session

- Please ask your questions on the presented topics

Using a custom data-source
Connection pooling
Spring Data JPA overview

Goals

- Learn how to use a custom data-source in a Spring Boot project
- Learn a high-level overview of connection pooling
- Learn a high-level overview of Spring Data

Using a custom defined datasource

1. Auto-configured databases are good (mostly) for initial prototyping
2. Spring Boot recommends their users to gradually change to own configs
3. Defining our own datasource:
   - Create a `@Configuration` annotated class
   - Disable the automatic database config:
     ```
     @EnableAutoConfiguration(exclude = DataSourceAutoConfiguration.class)
     ```
   - Create a `@Bean` annotated method which returns a configured `DataSource`
   - Run the project

Demo

Studying the connection to a custom datasource:

- @Configuration class
- application.yaml configuration

Spring Data JPA overview

- Spring Data - project used to simplify the usage of JPA repositories
- Creating a new repository → extending the `CrudRepository` interface
  - Generic interface → specifies:
    - Entity class type
    - Primary key class type
- The repository will automatically inherit the usual CRUD methods
- New functionalities can be added by:
  - Adding methods with an SQL-like naming → 'findByName' (for a `Product`)
  - Adding methods annotated with `@Query` → using JPQL or native queries

Spring Data JPA integration

Two main annotations:

1. `@EnableJpaRepositories`
   - Specifies the base package of the used JPA repositories
   - Can also specify other details - entity manager, transaction manager, etc
2. `@EntityScan`
   - Specify the base package of the JPA entities
   - Useful only if they are not under the main package of the Spring Boot app

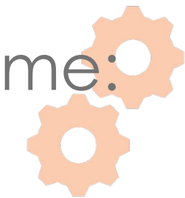
Demo

Using the two annotations - @EnableJpaRepositories and @EntityScan

Paging and sorting using Spring Data JPA

Spring Data projects support paging and sorting in several ways:

- Extending PagingAndSortingRepository (it extends CrudRepository):
  - Contains two 'findAll' methods for supporting paging and sorting:
    - `Iterable<T> findAll(Sort sortMode)`
    - `Page<T> findAll(Pageable pageable)`
  - The Page object also returns paging info
- Sending Pageable parameters in a method added in a repository:
  - `findByPrice(double price, Pageable pageable)`
- Creating methods which contain 'asc' & 'desc' keywords in their name:
  - `findByPriceOrderByNameAsc(double price, Pageable pageable)`



Transaction management overview

- **How**:
  - Using an (auto)configured `TransactionManager`
  - Using the `@Transactional` annotation on `@Service` CRUD methods
- `@Transactional`:
  - Performs transaction management on annotated methods
  - Applied as an AOP advice
  - Acts as a 'transactional umbrella' on the advised methods
  - Used on the service layer → wraps calls to multiple repositories

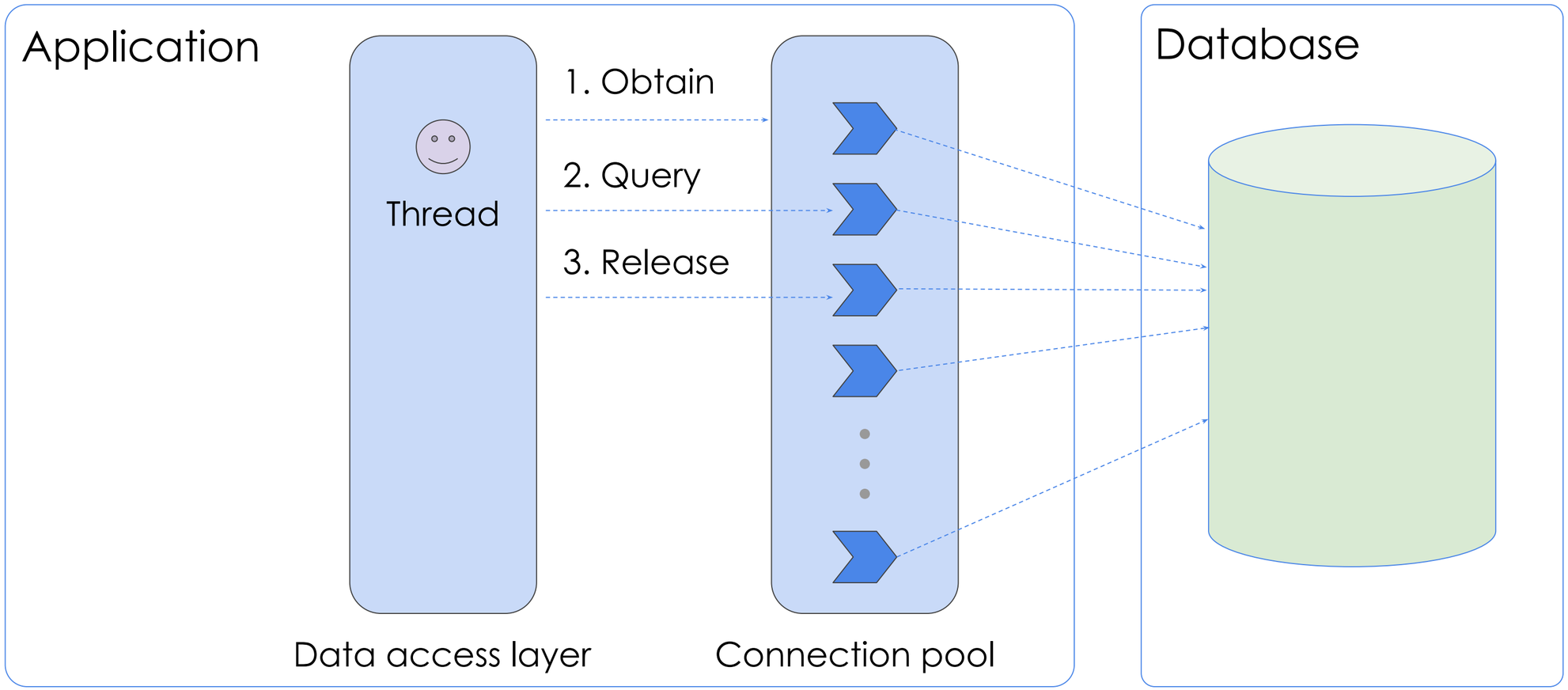
Demo

Using the @Transactional annotation on @Service methods

Connection pooling

- Connection pooling - creating and managing a set of reusable database connections
- Connections are established when the container is initialized, instead of when they are needed
- The connection pool implementation is not tied to the actual usage
  - It can be changed any time → changing the `DataSource` implementation
- Wiring - the connection pool's main class implements the DataSource interface

Connection pooling
Visual representation



Connection pools

- The most common / used are:
  - HikariCP (default in Spring Boot 2.0)
  - Tomcat JDBC CP
  - Commons DBCP
  - C3P0

Demo

Detailing the integration of HikariCP in our project

Activity

Adding a Spring Data JPA repository, persisting an entity with it

Scenario:

- Adding a new JPA entity and the Spring Data repository for it
- Persisting the new entity using the created repository

Aim:

Understanding:

- How to add a new JPA entity and repository to a project
- How to persist the newly added entity

Steps to add a new JPA entity and repository

1. Navigate to the package ‘com.packt.learning.springboot.domain’
2. Create a new JPA entity named Section in the ‘entity’ package
3. Configure the bidirectional relation of the `Product` and `Section` entities → we’ll code them together
4. Create an interface named `SectionRepository` in the ‘repository’ package
5. We’ll configure the repository integration together

Steps to add a new JPA entity and repository (continued)

1. Create a service class named `SectionService` in the 'service' package, annotate it with `@Service`
2. Autowire the `SectionRepository` in the `SectionService` class
3. Add a method named 'testSectionSaving()' in the new class, so that we can test the section saving and retrieving
4. Annotate the method with `@PostConstruct`, so that it's automatically ran

Steps to add a new JPA entity and repository (continued)

1. Add the following code in the 'testRepositorySaving()' method:

```
Section section = new Section(1, "Electronics");
sectionRepository.save(section);
System.out.println("The section was successfully saved");
Assert.isTrue(sectionRepository.count() == 1, "No sections");
```

2. Run the application and observe the displayed messages

Summary

In this lesson we learned…

- How to define and use a custom data-source in a Spring Boot project
- An overview of Spring Data JPA
- An overview of transaction management in Spring
- An overview of connection pooling and how to use it in a Spring project

Q & A session

- Please ask your questions on the presented topics

End of Day 1