Packt®

Learning Spring Boot 2

Bogdan Solga

Unit and integration testing, Spring Security

Goals

- ✓ Learn an overview of automated testing
- ✓ Learn how to use unit tests in a Spring Boot project
- ✓ Learn the unit testing principles and best practices

Automated testing overview

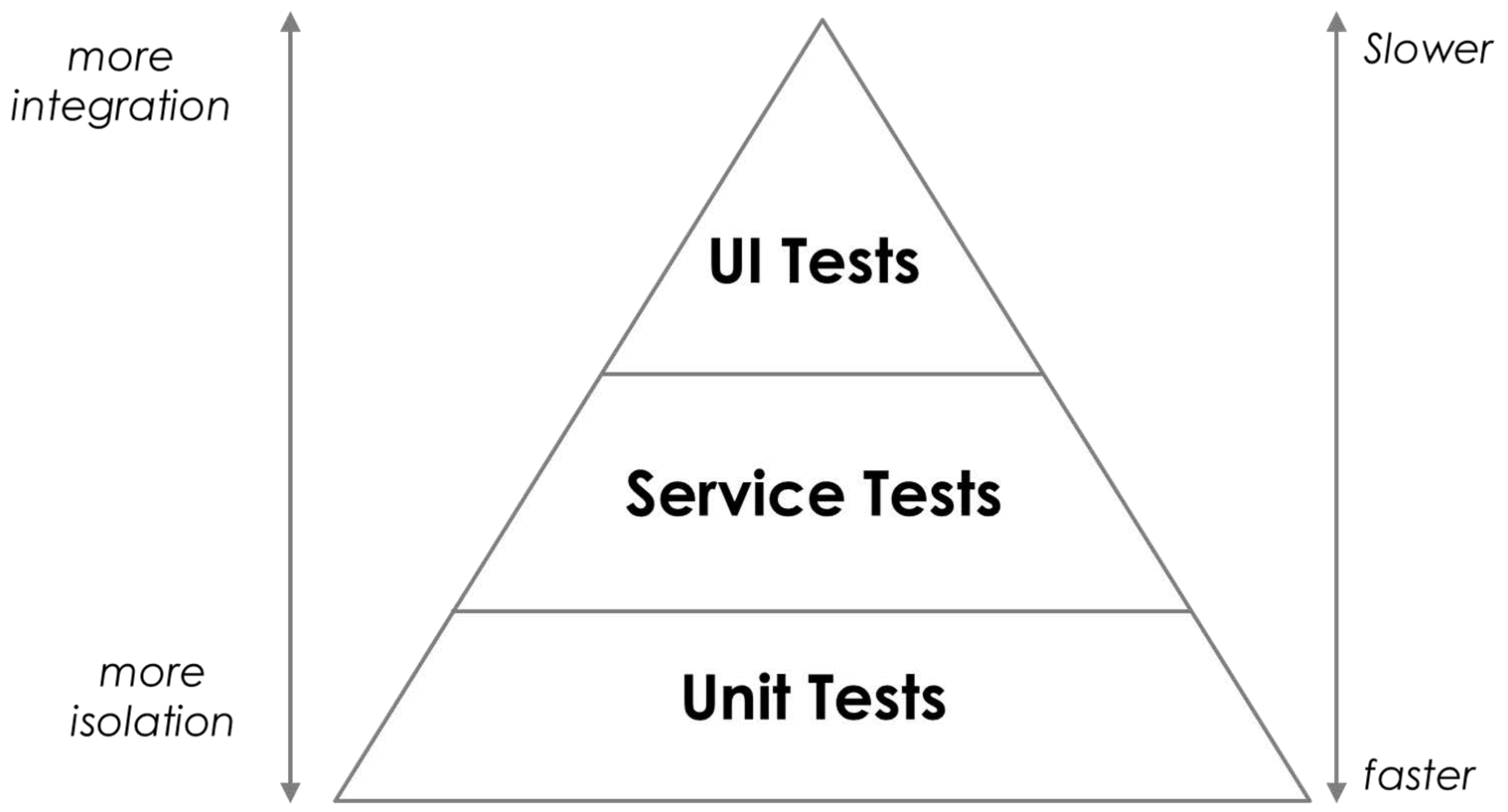
Unit testing overview

10m

Enterprise application testing

- **Testing** - verifying that an application behaves as expected, in terms of:
  - The correct functioning of the developed features - unit and integration tests
  - Reliable response times under heavy load - load testing
- **Automated** testing - tests are executed automatically by a CI tool
  - The most known and used: Jenkins, Bamboo, TeamCity
  - They can run the tests:
    - Periodically (example: every hour, at the end of every day)
    - When each developer commits something
    - When invoked manually
- **Code coverage** - the percentage of code covered by unit tests

The testing pyramid



Takeaways:

1. Write tests with different granularity
2. The more high-level you get the fewer tests you should have

Unit testing

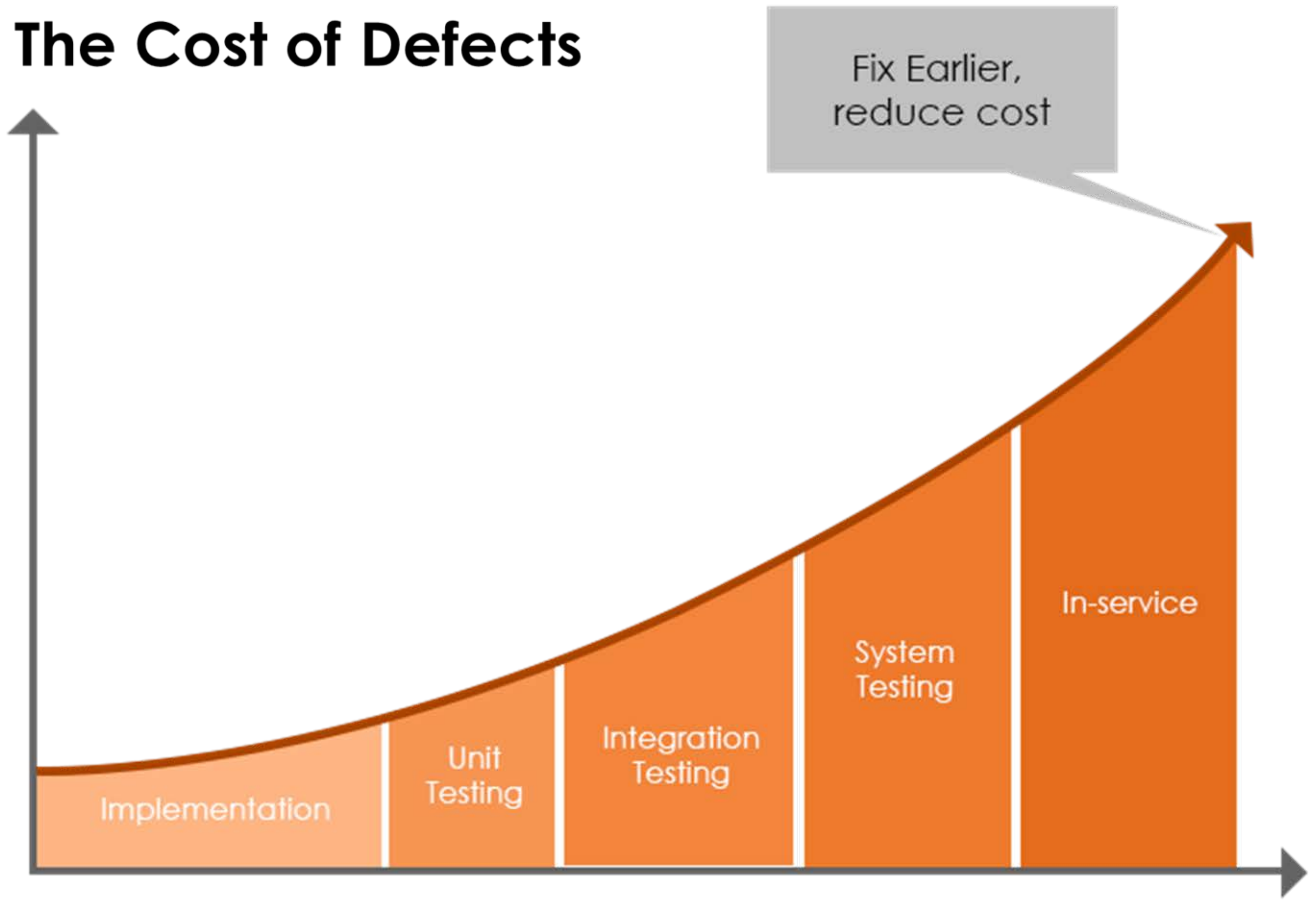
- Automated way to test the implementation behaves as expected
- A way for:
  - Automatically testing the correct functioning
  - Regression test the changes → test functionalities after changes / bug-fixes
- Tested application parts - service methods (especially)
- Simulating collaborators behavior - mocking / stubbing
  - Mocking - pre-programming objects to behave in a certain way
  - Stubbing - objects which respond with hard-coded (non-programmable) responses

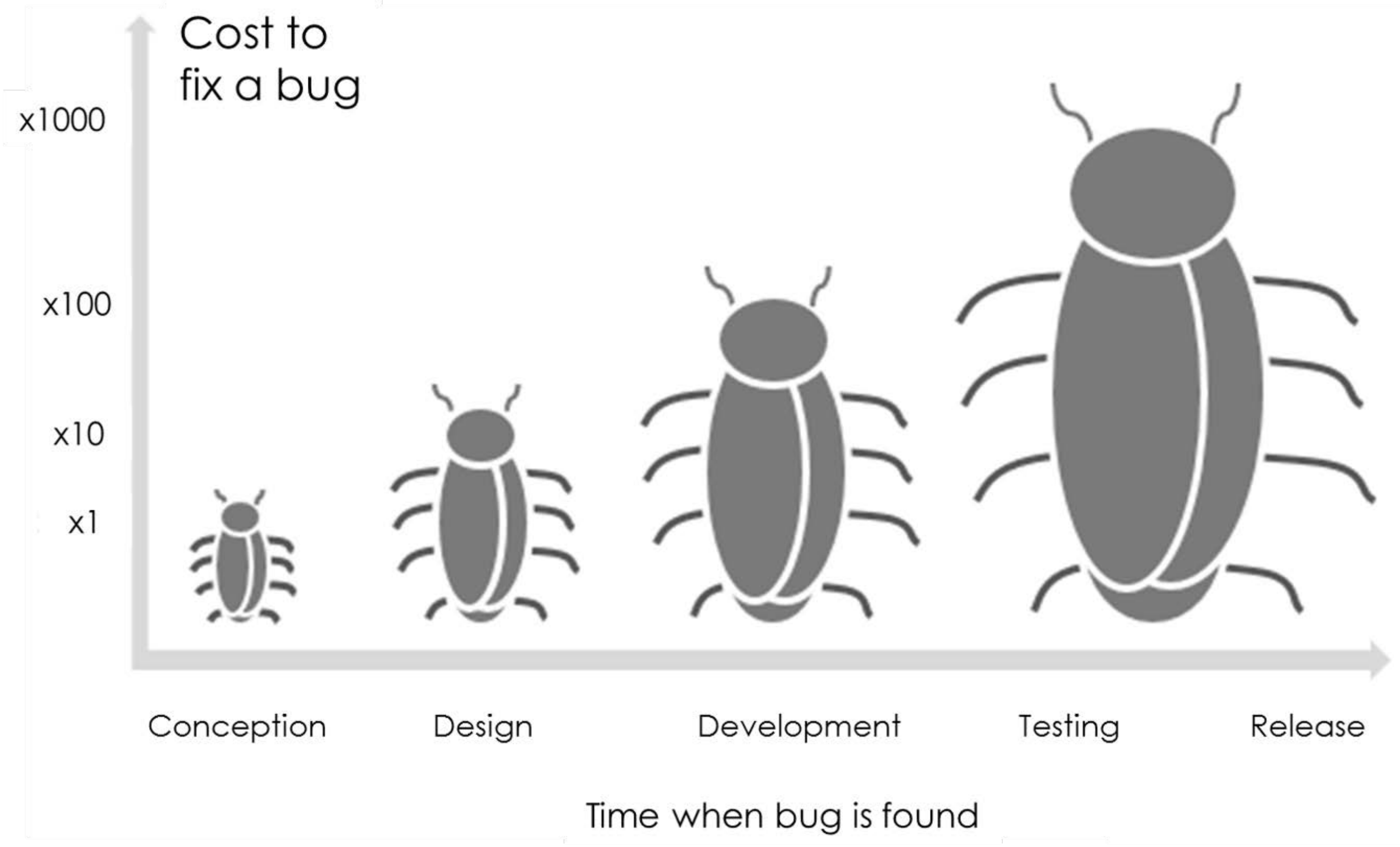
Writing unit tests - pros and cons

- **Pros:**
  - Automated and fast testing of the functionalities
  - Quick way to detect problems - fail fast, fail quickly
  - Regression tests - verify the proper functioning after (quick) fixes
- **Cons:**
  - Writing unit tests takes may take *a lot* of time (and money)
  - Writing them involve knowing the testing / mocking libraries → additional study
  - Tight deadlines

Cost of writing unit tests < sum (bug fixing time)

Bug fixing costs (estimations)





Unit testing libraries overview

- **JUnit** - the 'de facto' unit testing framework
    - Provides a rich set of classes and annotations for running tests
    - Integrates with other tools for more complete functionalities
- **TestNG** - an alternative to JUnit (especially before JUnit 5)
    - Can use automated sets of data for testing (among other benefits)
- **Mockito** - the most used mocking library
    - Powerful mocking capabilities:
        - Defining the methods behavior: `when(mtd.call()).thenReturn(resp)`
        - Verifying method invocations: `verify(obj, times(1)).method(param)`
        - Using wildcard matchers: `is(value), not(value)`

JUnit versions

- 4: The ‘old’ version, exists since 2006
  - The most used testing framework → >30k projects and libraries use it
  - Uses Java 5 as the baseline
- 5: The current version, released in September 2017
  - Uses Java 8 as the baseline → built-in lambda expressions support
  - Several new features:
    - Parameterized tests → specify one or more sources that supply parameter values for a unit test method
    - Nested unit tests → test classes can contain inner classes (which can contain classes)

JUnit tests overview

- Unit tests - methods annotated with @Test
- Should test *a single functionality* from the app → *unit* testing
- Can run methods:
  - Before and after all the tests have been executed
  - Before and after each test has been executed
- Can interact with other libraries for more powerful tests:
  - Mockito - mocking library
  - Hamcrest - powerful matchers

Running the tests

- Manually:
    - From the IDE - all IDEs have unit tests support
    - Via Maven - using the ‘maven-surefire-plugin’ plugin
        - Can be skipped using the ‘-DskipTests’ property
    - Via Gradle - built-in support for running the tests
- Automatically - using a CI tool
    - Frequency:
        - Periodically → daily / nightly builds / on each commit
        - On request
    - Runner: the used build tool (Maven/Gradle)

Unit tests principles

FIRST principles for unit testing:

- **F**ast → tests execution time should be short → they can be ran frequently
- **I**solated / independent → there should not be any dependency between the tests running order
- **R**epeatable and deterministic → tests should not depend on any environment data, their execution should be similar each time they run
- **S**elf-validating → they shouldn't require any additional validation, after running
- **T**horough → tests should cover *all* the inputs, corner-cases, exceptions, boundaries and scenarios

Demo

- Studying the project updates
- Studying the tests for the ProductService class

Elements to study

- Introducing the project updates
  - Maven dependencies
  - Code changes
- Studying the `ProductService` unit tests
  - Class layout
  - Used annotations
  - Tests
    - Mocking
    - Assertions, matchers
    - Verifying invocations
    - Running the tests from the IDE and with Maven

Unit tests best practices

- **Test naming**:
  - Advised mode: ‘given-when-then’ → gives more context to each test
    - Since JUnit 5 - @DisplayName can be used for describing tests
  - Should not contain the word ‘test’ → it’s redundant
- **Test methods structuring** - should contain three (/ four) stages:
  - Arrange, Act, Assert
  - Setup, Act, Verify, Teardown (Gerard Meszaros, Four Phase tests)
  - Given, When, Then
- ***Must* test the *functionality***, *not* the class → the class needs to adapt to the test

Activity

Adding unit and integration tests support to our project

Writing a simple test for the ProductService

Scenario:

- Adding unit tests support to our project
- Writing a few simple tests for our `ProductService`

Aim:

Understanding:

- How to add unit and integration tests support to an application
- How to write a few simple unit tests for an existing class

Steps to add unit and integration tests support

1. Open the project's pom.xml file
2. Add the following dependency:

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

3. Create a class named ProductServiceTest in the 'com.packt.learning.springboot' package of the 'src/test/java' folder
4. We will write the tests in the class together

Summary

In this lesson we learned…

- An overview of enterprise automated testing
- An overview of the main types of tests - unit, integration and load tests
- An overview of the main Java unit testing libraries:
  - JUnit
  - Mockito
  - TestNG

Q & A session

- Please ask your questions on the presented topics

Integration testing in Spring Boot

Goals

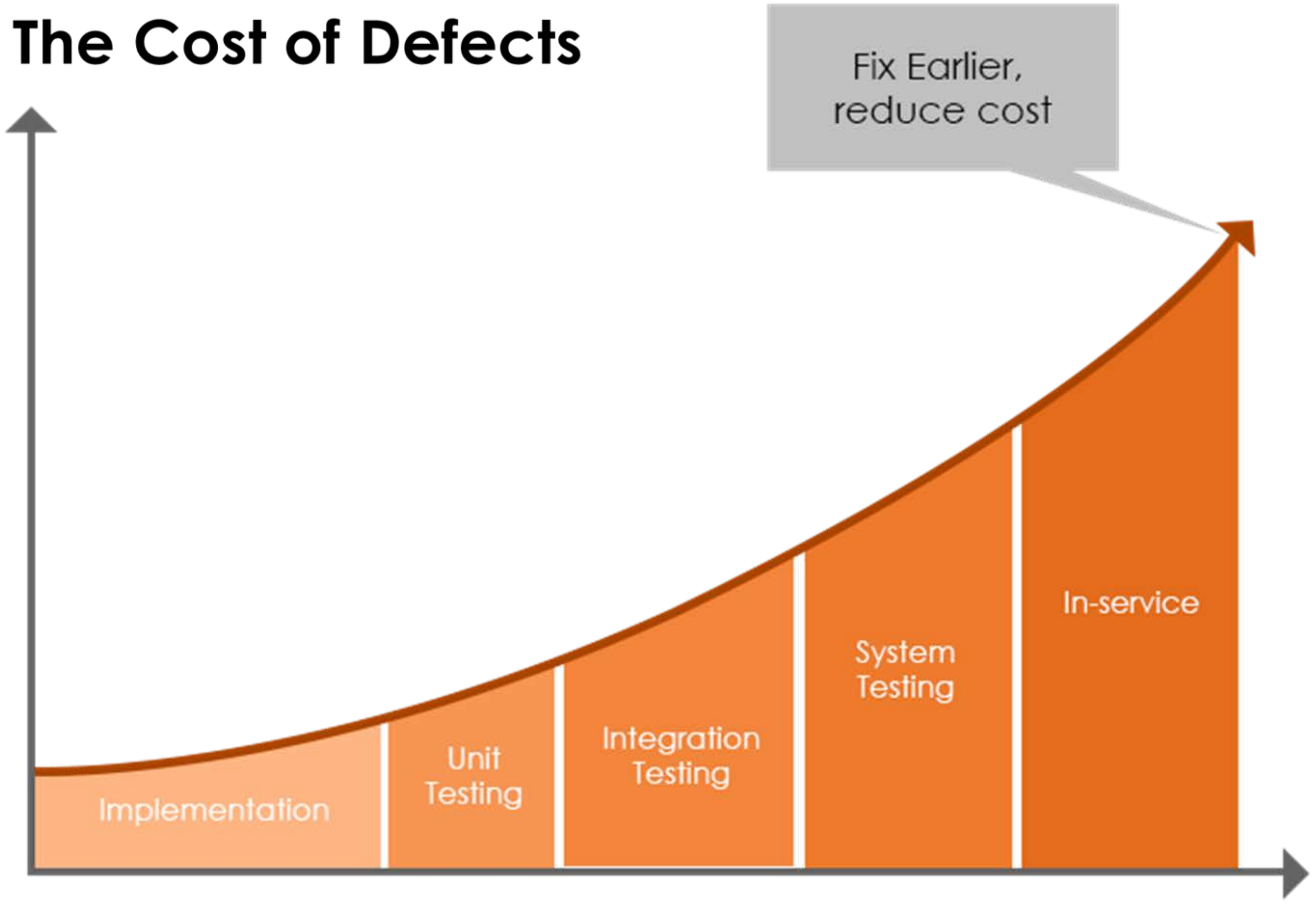
- ✓ Learn an overview of automated testing
- ✓ Learn how to use unit and integration tests in a Spring Boot project
- ✓ Learn how integrate Spring Security in a project

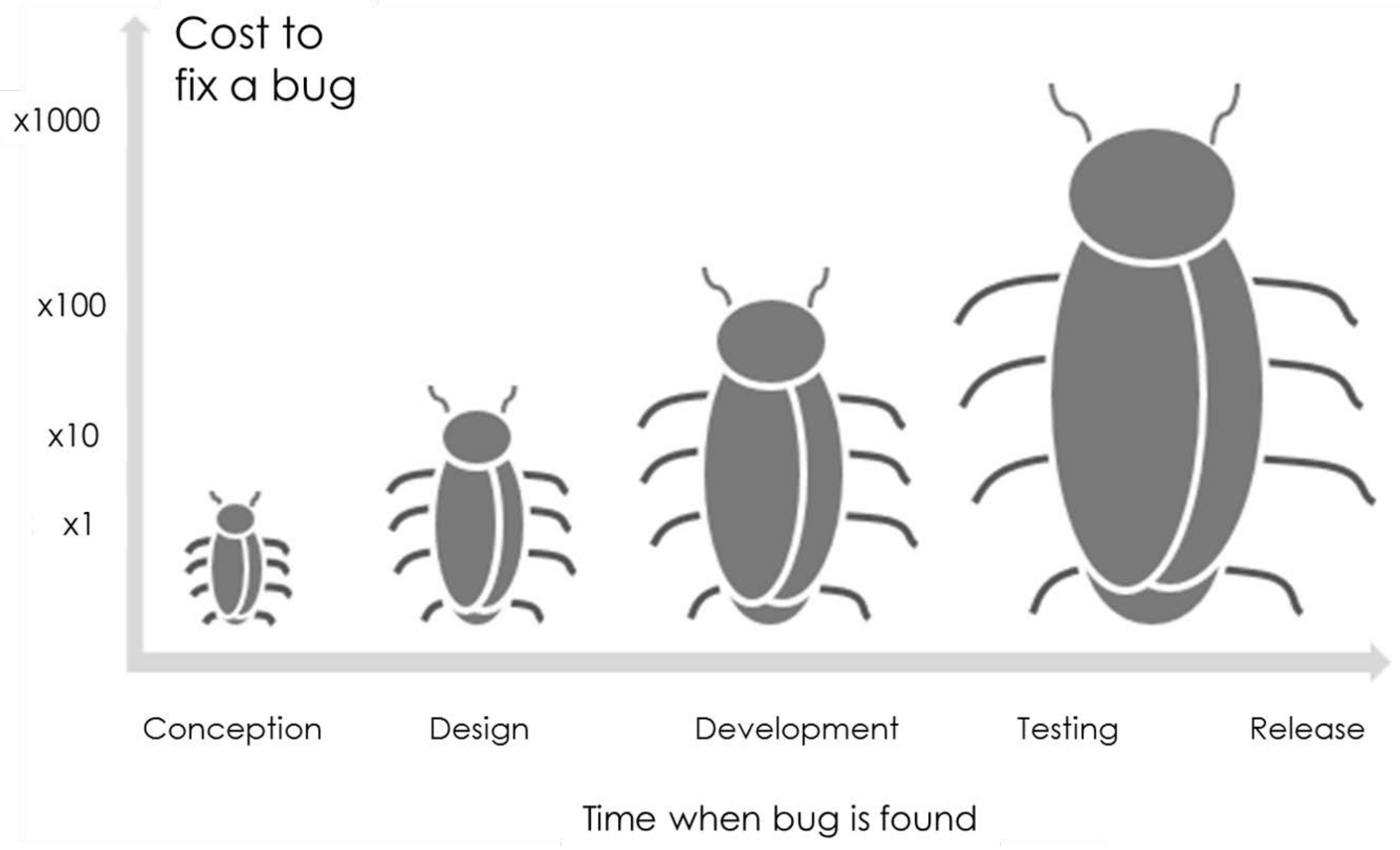
Automated testing overview

Unit and integration tests

10m

Bug fixing costs (estimations)





Integration testing

- **Integration test** - testing the end-to-end (E2E) functionality of a project
- Mostly useful for web applications
  - → Testing the application from the presentation layer to the database
- Maven integration
  - Ran together with the unit tests (by the maven-surefire plugin)
  - Can be separated to run at different build stages
- Can / should do cleanups, if needed
  - Database data
  - Other files and resources

Integration testing in Spring / Boot

- Spring & Spring Boot offer extensive support for integration testing
  - No need to deploy the app in a (web | application) server
  - Support for random port assignment on app startup
  - Init and teardown hooks

- Main annotation - @SpringBootTest

- Integration with other powerful testing libraries:
  - RESTAssured → cleaner & simpler REST tests, using given-when-then syntax
  - TestNG → used especially with JUnit 4, for using parameterized tests

  Reference - https://docs.spring.io/spring/docs/current/spring-framework-reference/testing.html#integration-testing

Spring Boot integration tests
Usage modes

- Using the @SpringBootTest annotation is not starting an embedded server → it is intended for testing the web layer
  - The web server starting - activated using the 'webEnvironment' param
- Several other annotations can be used, to help the testing:
  - `@AutoConfigureMockMvc` → auto-configures a MockMvc object, used to perform HTTP calls
  - `@AutoConfigureWebTestClient` → auto-configures a WebTestClient
  - `@DataJpaTest / @JdbcTest` → testing the persistence part of an app

Demo

- Studying the project updates
- Studying the integration tests for the `ProductController` class

Elements to study

- Introducing the project updates
  - Maven dependencies
  - Code changes
- Studying the `ProductController` integration tests
  - Class layout
  - Used annotations
  - Tests
    - The used annotation
    - The usage of the given-when-then tests structuring

Demo

- Studying the two types of integration tests from our project

Activity

Adding integration tests support to our project

Writing a simple integration test for the ProductController

Scenario:

- Adding integration tests support to our project
- Writing a simple integration test for our `ProductController`

Aim:

Understanding:

- How to add integration tests support to an application
- How to write a few simple integration tests for an existing class

Steps to add unit and integration tests support

1. Open the project's pom.xml file
2. Add the following dependency:

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

3. Create a class named ProductControllerTest in the 'com.packt.learning.springboot.integration' package of the 'src/test/java' folder
4. We will write the tests in the class together

Summary

In this lesson we learned...

- An overview of integration testing in a Spring Boot project
- An overview of the main integration testing libraries used together with Spring Boot:
  - TestNG
  - RESTAssured

Q & A session

- Please ask your questions on the presented topics

Other Spring Boot tests

Goals

- ✓ Learn an overview of the other Spring Boot possible tests
- ✓ Learn how they can be useful in a Spring Boot project
- ✓ Learn how and when to use them, if needed

Other Spring Boot tests - overview

- Spring Boot supports the concept of ‘tests slicing’ → testing just the useful / needed application layer
- When using a certain test slice - Spring will create a more lightweight `ApplicationContext` for that slice → faster test execution
- The slices are defined via annotations; the most common are:

○ *@JsonTest*:	registers JSON relevant components
○ *@DataJpaTest*:	registers JPA beans, including the ORM available
○ *@JdbcTest*:	raw JDBC tests, takes care of the datasource & in memory DB
○ *@DataMongoTest*:	in-memory MongoDB testing setup
○ *@WebMvcTest*:	a mock MVC testing slice, without the rest of the app

Other Spring Boot tests - helper annotations

- There are several other annotations which can be used to setup the Spring Boot integration tests:
    - `@ActiveProfiles("test")` -- used to specify the active profile(s) during the test execution
    - @AutoConfigureWireMock -- auto-configures a WireMock HTTP server, which can be configured to return predefined responses
    - `@AutoConfigureMockMvc` -- auto-configures a MockMvc object, which can be used to perform mocked MVC calls

Summary

In this lesson we learned…

- An overview of the Spring Boot test slices
- An overview of the other annotations that we can use in Spring Boot tests, for:
  - simpler and more granular tests
  - writing less code

Q & A session

- Please ask your questions on the presented topics

Spring Security - overview and integration

Goals

- ✓ Learn an overview of the security concepts
- ✓ Learn how to integrate Spring Security in a Spring Boot project
- ✓ Learn how to integrate session persistence in a Spring Boot project

Integrating Spring Security

10m

Spring Security overview

- The de facto Spring A&A [authentication and authorization] framework
- Supports the most common A&A methods and protocols
    - OAuth [1 and 2]
    - SAML
    - Kerberos
    - X509, ...
- Integration with multiple authentication providers - database, LDAP, etc
- Protection against web-related hacks (CORS, CSRF, session stealing, etc)
- Built-in ‘remember me’ functionality
- Easy session persistence integration, with several providers

Security core concepts

- **Authentication** - the process used to authenticate an user (verify user & pass)
- **Authorization** - verify the roles [set of privileges] that an user has in the application
- **Role[s]** - the set of privileges that an user was granted in an application
  - Also called authorities
- **Principal** - the user who is currently logged in

Steps to integrate Spring Security in a Spring Boot project

- Add the ‘spring-boot-starter-security’ module to the project’s pom.xml
- Implement a configuration class and:
  - Add:

    ```
    @EnableWebSecurity

    @EnableGlobalMethodSecurity
    ```

  - Extend:

    ```
    WebSecurityConfigurerAdapter
    ```

- Configure the needed components →

Configuring the security details

- `AuthenticationManagerBuilder`
  - Defines an AuthenticationManager → the user authentication repository
  - Can also use an in-memory AuthenticationManager → easier testing
- `HttpSecurity`
  - Configures the:
    - Role-based access per HTTP endpoints and resources
    - Post-login and logout handlers
- `WebSecurity`
  - Configures the unrestricted access endpoints

Form authentication - login / logout flow

- **Login**:
  - Method & URI: POST /login
  - Params: username, password
    - On success:
      - A JSESSIONID Cookie is generated
      - Automatically sent in the next requests
    - On failure: A 401 HTTP response is returned
- **Logout**:
  - Method & URI: POST /logout
  - Params: none

Demo

- Studying the project updates
- Form based login → auto-generated page and form
- Simple in-memory authentication

Elements to study

- Maven dependencies
- Security configuration class
    - Overriden methods
    - Configuration items

Authentication providers

- Pluggable modules referencing an UserDetailsService → auth services
  - Implementations: in-memory, JDBC, LDAP, caching
- An `AuthenticationManager` can reference multiple auth providers
  - The authentication is tried sequentially on them, until one succeeds
- The authentication providers can use a `PasswordEncoder`, for encoding / matching the password
  - Encoding: when the user is saved
  - Matching: when the user is authenticated

Authorization by annotations

- Enabling pre / post authorization annotations:

```
@EnableGlobalMethodSecurity(
    prePostEnabled = true, securedEnabled = true)
```

- 'prePostEnabled' -- activates the usage of:
  - `@PreAuthorize  and    @PostAuthorize`
  - `@PreFilter     and    @PostFilter`
- 'securedEnabled' -- activates the usage of:
  - `@Secured`

Authorization via annotations

- **@PreAuthorize**

```
@PreAuthorize(“isAuthenticated() AND hasRole(“ROLE_ADMIN”)”)
public List<ProductDTO> get(int start, int pageSize) {...}
```

- **@PreFilter**

```
@PreFilter("products.userId == authentication.userId")
public void addProducts(List<ProductDTO> products) {...}
```

- **@Secured**

```
@Secured({"ROLE_ADMIN", "ROLE_MANAGER"})
public void deleteProduct(int productId) {...}
```

@AuthenticationPrincipal

- Usefulness - retrieve the details of the currently authenticated user
- Used on - presentation layer endpoints
    - Pass it onwards to the service methods

```
public void getAuthUser(@AuthenticationPrincipal UserDetails
userDetails) {
  String username = userDetails.getUsername();
  // further use the username
}
```

Demo

- Pre- and post-authorization annotations
- Using the @AuthenticationPrincipal annotation

Password encoding

- Usefulness - securely storing and processing sensitive data:
  - Passwords
  - Other sensitive data - ex: credit card numbers
- Used in conjunction with hashing and salting → improved security
- Several predefined encoders, others can be defined → implement the `PasswordEncoder` interface
- Example:

```
@Autowired PasswordEncoder passwordEncoder;
// save → String encodedPassword = passwordEncoder.encode(password);
// login → boolean matches = passwordEncoder.matches(rawPass, encPass);
```

Post successful / failed form auth handlers

- Perform post successful / failed authentication actions:
  - Expiration verifications
  - Failed passwords number validations
  - Other validations
- Linked from the `HttpSecurity.FormLoginConfigurer` object
- Main configured actions:
  - `successHandler`
  - `failureHandler`

Integrating session persistence

- Session persistence - persisting the authentication sessions in a clustered environment, for high availability
- Library used for session persistence - Spring Session
- Can use several backing stores for persisting the session
- Usage:
  - Maven:

```
<groupId>org.springframework.session</groupId>
<artifactId>spring-session</artifactId>
```

  - Config: spring.session.store-type=(Mongo | Redis | Hazelcast | JDBC)

Activity

Securing a REST endpoint, testing the security on it

Scenario:

- Configuring Spring Security for our project
- Securing a REST endpoint

Aim:

Understanding:

- How to integrate Spring Security in a new project
- How to add security to an existing REST controller

Steps to integrate Spring Security in our project

1. Open the project’s pom.xml file
2. Add the following dependency:

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

3. Create a class named SecurityConfiguration in the ‘com.packt.learning.spring.boot.d02s01’ package of the ‘src/test/java’ folder
4. We will write the class together, copying the code from the existing project

Steps to integrate secure an existing REST controller

1. Open the ProductController class (from the 'com.packt.learning.spring.boot.controller' package)
2. Add the following annotation on it:

```
@PreAuthorize("isAuthenticated()")
```

3. Run the project, by executing the main() method from the main class
4. Access the endpoint http://localhost:8080/product from a browser
5. Observe the returned 401 (Unauthorized) response

Summary

In this lesson we learned...

- An overview of the main web security concepts
- An overview of Spring Security
- How to integrate Spring Security in an existing project
- How to secure a REST controller using the `@PreAuthorize` annotation
- An overview of how to integrate session persistence in a project

Q & A session

- Please ask your questions on the presented topics

Conditional annotations, ConfigurationProperties and Spring Boot events

Goals

- ✓ Learn an overview of what conditional annotations are
- ✓ Learn the main conditional annotations from Spring Boot
- ✓ Learn the benefits of using the annotations in a project

Conditional annotations overview

10m

Conditional annotations

- @Conditional annotations - conditionally load one / several beans
- Can be used for classes and individual beans
- Used for:
  - Beans auto-configuration
  - Loading certain beans conditionally
- Main annotations:
  - `@Conditional` - one or several Conditions must be met
  - `@ConditionalOnClass` - if a class is present in the classpath
  - `@ConditionalOnBean` - if a bean is present in the classpath
  - `@ConditionalOnProperty` - if a property is found
  - `@ConditionalOnJava` - JVM version condition

Conditional annotations - use-cases

The most common use-cases:

- Feature toggles - using some @Beans only when a condition is true
- Activating some functionalities based on:
  - The current Java version
  - The existence of a class
  - The existence of a Spring bean
- The main functionality used by Spring Boot's auto-configuration support

Demo

- Seeing a few Conditional annotations examples

Creating our own configuration properties

- When using multiple external configuration options → create a @ConfigurationProperties class to ease their usage
- Takes the config options prefix as parameter
  - Example:

```
@ConfigurationProperties(prefix = “spring-boot”)
```

- Usage - config options values can be used:
  - Through the @Value(“{}”) annotation

```
@Value(“${spring-boot.version}”)
private String springVersion;
```

  - Wiring the configuration class and using the values as simple method calls

Wiring the created config class

- The created class → wired as a regular bean
- The config options are retrieved as simple immutable class properties
- Example:
  - Wiring it:

```
@Autowired
private DomainConfigProperties domainConfig;
```

  - Using it:

```
domainConfig.getEnvironmentName();
```

Demo

- Using a `@ConfigurationProperties` annotated class

Activity

- Adding a conditional annotation to the project
- Testing its correct functioning

Scenario:

- Creating a ‘feature toggle’ in our project
- Adding a conditional annotation to use that feature

Aim:

Understanding:

- How to create and use ‘feature toggles’ in a project
- How to use conditional annotations to define the conditionally loaded classes / components

Steps to create a feature toggle in a project

1. Define the feature that should be toggled via some @Conditional annotations

2. For our scenario, we'll consider the Section service to be the toggled feature

3. We will the `@ConditionalOnProperty` annotation on the following classes
   a. `SectionController`
   b. `SectionService`
   c. `SectionRepository`

4. On each class, we will add the annotation in the following form:

```
@ConditionalOnProperty("enable.section.service")
```

Steps to create a feature toggle in a project

1. Run the project first without setting a value for the property → the feature won't be enabled
2. Verify the availability of the '/section' endpoint by accessing it from the browser → http://localhost:8080/section

```
System.setProperty("enable.section.service", "true");
```

3. Add the following line in the main() method of the main class:
4. Start the project again and access the '/section' endpoint again → http://localhost:8080/section
5. Result - the section service should be now accessible

Summary

In this lesson we learned...

- What 'conditional annotations' are
- How to define and use conditional annotations
- An overview of the 'feature toggle' concept and it's usage in a Spring Boot project, by using the `@Conditional` derived annotations

Q & A session

- Please ask your questions on the presented topics

Using ConfigurationProperties classes

Goals

- ✓ Learn an overview of when to use @ConfigurationProperties classes
- ✓ Learning how to define @ConfigurationProperties classes
- ✓ Learn the difference between simple and grouped properties

Using ConfigurationProperties

10m

Creating our own configuration properties

- When using multiple external configuration options → create a `@ConfigurationProperties` class to ease their usage
- Takes the config options prefix (namespace) as parameter
  - Example:

```
@ConfigurationProperties(prefix = “spring-boot”)
```

- Usage - config options values can be used:
  - Through the @Value(“{}”) annotation

```
@Value(“${spring-boot.version}”)
private String springVersion;
```

  - Wiring the configuration class and using the values as simple method calls

Wiring the created config class

- The created class → wired as a regular bean
- The config options are retrieved as simple immutable class properties
- Example:
    - Wiring it:

```
@Autowired
private DomainConfigProperties domainConfig;
```

    - Using it:

```
domainConfig.getEnvironmentName();
```

Demo

- Using a `@ConfigurationProperties` annotated class

Activity

- Creating a new ConfigurationProperties class
- Testing its correct functioning

Scenario:

- Integrate a @ConfigurationProperty annotated class in our project
- Use the properties loaded through that class

Aim:

Understanding:

- How to create & integrate @ConfigurationProperties classes
- How to use the properties from them in the project classes

Steps to create and integrate a `@ConfigurationProperties` class

1. Establish the namespace you will use in the `@ConfigurationProperties` class
    a. As a simple alternative - you can use the ‘learning.spring-boot’ namespace
2. Create a class named ConfigPropertiesExample, in the ‘com.packt.learning.spring.boot.config’ package
3. Annotate the created class with the following annotations:
    ```
    @Configuration
    @ConfigurationProperties(“learning.spring-boot”)
    ```
4. Create a few properties in the new class
5. Define the properties with the same name in the project’s configuration file

Steps to create and integrate a `@ConfigurationProperties` class

1. Autowire the created class in one of the project's service classes (ex: in the `ProductService` class)
2. Create an init() method with the following content in the class:

```
@PostConstruct
public void init() {
  System.out.println(configProperties.getCustomPropertyName());
}
```

3. Start the application and verify the configured property is displayed

Summary

In this lesson we learned...

- The benefits and usage of the `@ConfigurationProperties` annotation
- How to create and integrate a `@ConfigurationProperties` class in a project

Q & A session

- Please ask your questions on the presented topics

Spring and Spring Boot events

Goals

- ✓ Learn an overview of the internal Spring and Spring Boot events
- ✓ Learn how and when to use the events in an application
- ✓ Learn how the events can help in several scenarios

Spring and Spring Boot events - overview

10m

Events overview

- **Event** - application published action, may have (or not) a return type
- **Why** - a mean for loosely coupled components to exchange information

- Publish / subscribe model (‘pub / sub’):
    - A way to mix multiple publishers and their subscribers
    - **Publisher** - the class that publishes events
    - **Subscriber(s)** - the classes and methods subscribed to the published events

- Spring events:
    - Container published
    - Application published

IoC container generated events

- Spring generates (emits) several events during it's IoC container init:
  - `ContextRefreshedEvent` - triggered on context start and refresh events
    - Most commonly used for initializing data at startup
  - `ContextStartedEvent` - triggered on context start
    - Difference between ContextRefreshedEvent - invoked only on context start, not on context refresh
  - `ContextStopedEvent` - triggered when the context is stopped
  - `ContextClosedEvent` - triggered when the context is closed
- Can be used to perform actions when they are emitted (further presented)

Spring Boot emitted events

Additional to the Spring events:

- **`ApplicationStartingEvent`** - at the start of a run, before any processing
- **`ApplicationEnvironmentPreparedEvent`** - when the Environment is known, before the context is created
- **`ApplicationPreparedEvent`** - before the refresh is started, after bean definitions have been loaded
- **`ApplicationReadyEvent`** - after the refresh and any related callbacks have been processed, indicating the application is ready to service requests
- **`ApplicationFailedEvent`** - if there is an exception on startup

Application emitted events

- **Publishing** events - `ApplicationEventPublisher` class

```
appEventPublisher.publishEvent(new ProductUpdatedEvent())
```

- **Listening** / subscribing to events:

```
@EventListener(ProductUpdatedEvent.class)
public void processUpdate(final ProductUpdatedEvent event) {
  // handle the event
}
```

Application emitted events

- **Publishing** events - `ApplicationEventPublisher` class

```
appEventPublisher.publishEvent(new ProductUpdatedEvent())
```

- **Listening** / subscribing to events:

```
@EventListener(ProductUpdatedEvent.class)
public void processUpdate(final ProductUpdatedEvent event) {
  // handle the event
}
```

Demo

- Listening to Spring generated events
- Listening to Spring Boot generated events
- Creating our own events, publishing and listening to them

Sync and async events

- Default event publishing - synchronous (blocking)
  - The publishing thread will block until all the listeners will get the event
  - Advantage: for transactional contexts - the publisher and listeners will run in the same transaction context

- Event listeners can be made async using the `@Async` annotation

```
@Async
@EventListener(ProductUpdatedEvent.class)
public void productUpdated(ProductUpdatedEvent event) { ... }
```

Event filtering

- Events can be filtered based on their internal properties
- The properties are accessed using the Spring Expression Language (SpEL)
- Usage - with the ‘condition’ property of the `@EventListener`
    - Since Spring 4.3 - can reference a bean name: “@beanName.method”
- Example:

```
@EventListener(condition = “#product.name.length > 0”)
public void processProduct(Product product) {...}
```

Demo

- Event filtering
- Event processing

Transaction bound events

- For sync events, the listener(s) can be bound to a transaction's life cycle
- Events can be processed based on the transaction's phase
    - On commit
    - On rollback

- Usage: `@TransactionalEventListener` & set the transaction phase:
    - After commit
    - After rollback
    - After completion

```
@TransactionalEventListener(phase = AFTER_COMMIT)
public void afterCommit(ProductSavedEvent event) {...}
```

Demo

- Transaction bound events

Activity

- Adding a Spring Boot event on a class
- Testing its correct functioning

Scenario:

- Using Spring and Spring Boot events in a project
- Using transaction bound events (ifEnoughTime)

Aim:

Understanding:

- How to use Spring and Spring Boot events in a project
- How to use transaction bound events in a project

Steps to use Spring events in our project

1. Open the ProductService class
2. Create a method called `springEventListener()` with the content:

```
@EventListener(ContextStartedEvent.class)
public void springEventListener() {
  System.out.println("Received a ContextStartedEvent event");
}
```

→ continued on the next slide

Steps to use Spring Boot events in our project

1. Create a method called `springBootEventListener()` with the content:

```
@EventListener(ApplicationReadyEvent.class)
public void springBootEventListener() {
  System.out.println(“Received an ApplicationReadyEvent”);
}
```

2. Start the main application
3. Observe the console - the messages from the two methods should be displayed

Steps to add a transaction bound event listener

1. Create a package named ‘com.packt.learning.spring.boot.events’
2. Create a class named `ProductRetrieved` in it
3. Create a method called `transactionalEventListener()`:

```
@Transactional(propagation = Propagation.SUPPORTS)
public Product transactionalEventListener(int id) {
  appEventPublisher.publishEvent(new ProductRetrieved("Tablet"));
  return new Product(id, "iSomething");
}
```

Steps to add a transaction bound event listener

1. Create a class named ProductServiceEventsListener in the ‘com.packt.learning.spring.boot.events’ package
2. Add the following event listener in it:

```
@TransactionalEventListener(phase = AFTER_COMPLETION)
  public void processSavedProduct(ProductRetrieved event) {
  System.out.println(“The product was saved”);
}
```

3. Start the main class and observe the displayed messages

Summary

In this lesson we learned...

- An overview of the Spring and Spring Boot built-in events
- How to listen for those events in a project
- How to define custom events, how to publish and subscribe to them
- How to use transaction bound events in a Spring & Spring Boot project

Q & A session

- Please ask your questions on the presented topics

Spring Boot messaging support overview, Spring Boot Actuator and developer tools

Goals

- ✓ Learn an overview of the messaging support built in Spring Boot
- ✓ Learn how Spring and Spring Boot abstracts the used messaging framework
- ✓ Learn an overview of the messaging usage scenarios

Packt

Spring Boot messaging support overview

10m

Messaging systems overview

- **Messaging systems -** systems that communicate (or facilitate the communication) via async messages
- **Main technologies**:
  - JMS: Java Messaging System (Java specific messaging)
  - AMQP: Advanced Message Queuing Protocol (platform-independent)
  - WebSocket: bi-directional messages, usually exchanged between an UI and the backend
- **Main Java messaging systems / brokers:**
  - Apache ActiveMQ and ActiveMQ-Artemis
  - RabbitMQ
  - Apache Kafka (a high-throughput distributed messaging system)

Spring Framework and Spring Boot support for messaging systems

The Spring Framework and Spring Boot provide extensive support for integrating messaging systems:

- Simplified usage of the:
  - JMS API - through the `JmsTemplate` class
  - AMQP - through the Spring AMQP component
- An end-to-end infrastructure to send and receive async messages
- Spring Boot provides auto-configuration support for:
  - JMS: autowiring a `JmsTemplate` object
  - RabbitMQ: autowiring a `RabbitMessagingTemplate` object
  - Kafka: autowiring a `KafkaTemplate` object

Using JMS

- The Spring Framework provides a higher-level messaging abstraction on top of the `ConnectionFactory` class → core connection handling class
- Spring Boot auto-configures the necessary beans to allow the message sending and receiving
- Code usage:
  - Sending messages: through an autowired `JmsTemplate` object
  - Listening to messages: annotating a bean method with `JmsListener`:

```
@JmsListener(destination = "products")
public void processMessage(String product) {
  // ...
}
```

Using AMQP

- **AMQP** - platform-neutral protocol for message-oriented architectures
- Main messaging brokers:
  - RabbitMQ - lightweight AMQP message broker; scalable, reliable & portable
  - Apache ActiveMQ and Artemis - open-source AMQP brokers
    - Auto-configured by Spring Boot, if they are found on the class path
- Code usage:
  - Sending messages: an `AmqpTemplate` / `RabbitMessagingTemplate`
  - Listening to messages: annotating a bean method with `RabbitListener`:

```
@RabbitListener(destination = "products")
public void processMessage(String product) {
  // ...
}
```

Using Apache Kafka

- **Apache Kafka** - streaming platform, used for building real-time streaming data pipelines / applications
- Spring Boot supports Kafka, provides auto-config support for the `spring-kafka` project
- Code usage:
  - Sending messages: autowiring a `KafkaTemplate` object
  - Listening to messages: annotating a bean method with `KafkaListener`:

```
@KafkaListener(destination = "products")
public void processMessage(String product) {
  // ...
}
```

Spring Cloud Stream

- Spring Cloud Stream - ‘framework for building highly scalable event-driven microservices, connected via shared messaging systems’
- Building blocks:
  - Destination Binders: integrations with the external messaging systems:
    - Kafka, RabbitMQ, cloud provider solutions (GCP, Amazon, Azure, ...)
  - Destination Bindings: bridges between the:
    - External messaging systems
    - Application provided Publishers and Subscribers (created by the binders)
  - Messages: data structure used by pubs & subs to communicate with Binders

Integration scenario

- Scenario: async & reactive orders processing
- Implementation: a publisher and a subscriber of Order messages (n products)
- Messaging broker: Kafka
- Publisher:
  - `@EnableBinding(Source.class)` → messages publisher
  - Uses the Source interface to publish messages
- Subscriber:
  - `@EnableBinding(Sink.class)` → messages receiver
  - `@StreamListener + @Input(Sink.INPUT) + Flux<Order>`

Demo

- A messaging example using Spring Cloud Stream and Kafka

Elements to study

- The used Maven dependencies
- The message binders and bindings
- The loose coupling for the messaging components

Summary

In this lesson we learned...

- An overview of the Spring and Spring Boot support for async messaging systems
- An overview of the main messaging systems:
  - JMS
  - AMQP
  - Apache Kafka
- An overview of Spring Cloud Stream, an abstraction over several messaging systems

Q & A session

- Please ask your questions on the presented topics

Spring Boot Actuator support

Goals

- ✓ Learn an overview of the Spring Boot Actuator set of tools
- ✓ Learn how the tools can be used to monitor and audit a system
- ✓ Learn the information types available through the Actuator

Spring Boot Actuator overview

10m

Spring Boot Actuator overview

- **Actuator** - set of management and monitoring tools
- Manage / monitor an app using:
  - HTTP
  - JMX
- Provides endpoints for:
  - Health checks
  - Auditing
  - Metrics

Adding Spring Boot Actuator to a project

- Add the '`spring-boot-starter-actuator`' to your Maven file:

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

- Configure the enabled endpoints - list of predefined endpoints
- Run the app

Spring Boot Actuator - predefined endpoints (selection)

auditevents	- Exposes audit events information
beans	- Displays a complete list of all the Spring beans in the app
env	- Exposes properties from Spring's ConfigurableEnvironment
health	- Shows application health information
info	- Displays arbitrary application info
loggers	- Shows and modifies the loggers configuration
metrics	- Shows 'metrics' information for the current application
mappings	- Displays a collated list of all @RequestMapping paths
shutdown	- Lets the application be gracefully shutdown
threaddump	- Performs a thread dump

Configuring the endpoints

- Config namespace - ‘management.endpoints’
- Options:
  - ID
  - Enabled / disabled
  - Sensitive (secured)
- Example:

```
management:
  endpoint:
    env:
      enabled: true
```

Demo

- Study the actuator endpoints config
- Run the Postman collection
- Analyze the endpoints responses

Adding our own endpoints

1. Create a `@Component` annotated class
2. Add the `@Endpoint` annotation
3. Add one or more operation methods to it → annotate them with:
   a. `@ReadOperation` → can be invoked via GET HTTP requests
   b. `@WriteOperation` → can be invoked via POST requests
      Accepts 'application/vnd.spring-boot.actuator.v2+json' & 'application/json'
   c. `@DeleteOperation` → can be invoked via DELETE requests
4. Return / write the wanted info from / in them
5. Test it

Demo

- A custom endpoint

Activity

Adding Actuator support to the project

Testing its correct functioning

Scenario:

- Adding the Spring Boot Actuator features to a project
- Testing the added functionality / endpoints

Aim:

Understanding:

- How to add and configure the Spring Boot Actuator to a project
- How to configure the exposed endpoints

Steps to integrate Spring Boot Actuator in our project

1. Open the project’s pom.xml file
2. Add the following dependency:

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

3. Open the project’s configuration file (application.properties | .yml)

→ continued on the next slide

Steps to integrate Spring Boot Actuator in our project

1. Add the following entries in the configuration file (YAML format):

```
management:
 endpoints:
   enabled-by-default: true
   web:
     exposure:
       include: '*'
 endpoint:
   health:
     show-details: always
```

2. Start the project
3. Test the endpoints, by accessing the 'health' endpoint from a browser:

   http://localhost:8080/actuator/health

Summary

In this lesson we learned…

- An overview of the Spring Boot Actuator
- An overview of the benefits brought by it:
    - Monitoring endpoints
    - Metrics and audit support
- A simple example of how to add a custom endpoint

Further information

- The official Spring Boot Actuator documentation

Q & A session

- Please ask your questions on the presented topics

Spring Boot developer tools

Goals

- ✓ Learn an overview of the Spring Boot developer tools
- ✓ Learn how the tools can be used to improve the development speed
- ✓ Learn their integration in an existing Spring Boot project

Spring Boot developer tools

10m

Spring Boot developer tools

- **What** - a set of tools meant to improve & speed-up the development experience
- Integration: adding the ‘`spring-boot-devtools`’ dependency
  - Maven:

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-devtools</artifactId>
   <optional>true</optional>
</dependency>
```

  - Gradle:

```
dependencies {
   developmentOnly("org.springframework.boot:spring-boot-devtools")
}
```

Development usage only

- The developer tools are automatically disabled when running a packaged application
  - the tools are meant for development *only*, not for production usage
  - disabled automatically when the app is launched via the 'java -jar' command
- The developer tools are not packaged in repackaged archives
  - The repackaging operation is done by the ‘spring-boot’ Maven/Gradle plugin (further presented)
-

Development features and improvements

- Automatically applied **development improvements**:
  - disabling the MVC and Thymeleaf caching
  - setting the logging level to DEBUG for 'spring-web' and 'spring-webflux'
- **Automatic restart** when a file on the classpath is changed
  - The class path updating is IDE dependant
  - Some resources can be excluded for some files (ex: static resources)
- **LiveReload** - live reloading of the UI changes
  - triggers a browser refresh when a resource is changed
  - works by pairing with a browser extension

Development features and improvements (continued)

Remote updates and restarts → triggering remote application restarts

- Supported via two parts:
  - A server-side endpoint that accepts remote connections
  - A client (Java) app that runs in an IDE
- Requirements:
  - Configuring the Maven/Gradle plugin to include the devTools library → setting the 'excludeDevtools' property to false
  - Setting a 'spring.devtools.remote.secret' property on the server app
  - Running 'o.s.b.d.RemoteSpringApplication' with the URL of the managed app
- Main benefit - the remote app can be updated from the local client, avoiding the need to perform redeploys

Demo

- Seeing the Spring Boot developer tools in action

Activity

Adding the Spring Boot developer tools to the project

Testing their correct functioning

Scenario:

- Adding the Spring Boot developer tools to a project
- Testing their proper integration in the project

Aim:

Understanding:

- How to add the Spring Boot developer tools in a project
- How to use their functionalities

Steps to integrate the Spring Boot developer tools in our project

1. Open the project's pom.xml file
2. Add the following dependency:

```
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-devtools</artifactId>
<optional>true</optional>
```

→ continued on the next slide

Steps to integrate the Spring Boot developer tools in our project (continued)

1. Start the project
2. Open the ProductService class
3. Make a change in it → add a new method, for example
4. Desired outcome - the application should be automatically restarted
   a. It should start much faster than a cold restart (when all the classes are reloaded)

Summary

In this lesson we learned…

- The integration of the Spring Boot developer tools in a project
- An overview of their benefits
    - Automatic restart
    - LiveReload
    - Remote restarts

Further information

- Spring Boot developer tools

Q & A session

- Please ask your questions on the presented topics

Packaging and running the app, Spring Boot Admin

Goals

- ✓ Learn an overview of the Spring Boot Maven plugin
- ✓ Learn how to configure and use the plugin in a project
- ✓ Learn the main configuration options and when to use them

Spring Boot Maven plugin

10m

Spring Boot Maven plugin - overview

- **Spring Boot Maven plugin** - a plugin used to:
  - Build a Spring Boot application
  - Repackage a Spring Boot application → the spring-boot:repackage goal
    - Repackage = rebuild the archive to contain all the needed libraries in it → allow it to run in a standalone mode
    - Built formats:
      - jar
      - war
  - Run a Spring Boot application → the spring-boot:run goal

Spring Boot Maven plugin - default structure

```
<plugin>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-maven-plugin</artifactId>
   <version>2.1.3.RELEASE</version>
   <executions>
      <execution>
         <goals>
            <goal>repackage</goal>  → repackage the .jar/.war file during Maven's 'package' goal
         </goals>
      </execution>
   </executions>
</plugin>
```

Configuration options

The plugin allows multiple configuration options → ‘configuration’ tag:

- Setting system properties:

```
<systemPropertyVariables>
   <propertyExample>value</propertyExample>
</systemPropertyVariables>
```

- Setting environment variables:

```
<environmentVariables>
   <ENV1>1000</ENV1>
</environmentVariables>
```

- Running the app in debug mode:

```
<jvmArguments>
   -Xdebug -Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=5005
</jvmArguments>
```

- Specify active profile(s):

```
<profiles>
   <profile>dev</profile>
</profile>
```

Demo

- Seeing the Spring Boot Maven plugin in action

Activity

Adding the Spring Boot plugin to the project

Testing their correct functioning

Scenario:

- Adding the Spring Boot Maven plugin to our project
- Testing the plugin integration in the project

Aim:

Understanding:

- How to add the Spring Boot Maven plugin in a project
- How to configure it

Steps to integrate the Spring Boot Maven plugin in our project

1. Open the project's pom.xml file
2. Open the Spring Boot Maven plugin official page - `https://docs.spring.io/spring-boot/docs/current/maven-plugin/`
3. Open the 'Usage' page
4. Copy the plugin body in the <build><plugins> section of the pom.xml file

→ continued on the next slide

Steps to configure the Spring Boot Maven plugin

1. Add the ‘executions’ section to the plugin, to configure the plugin to repackage the app

```
<executions>
   <execution>
      <goals>
         <goal>repackage</goal>
      </goals>
   </execution>
</executions>
```

2. We will perform the app repackaging and running in the next session

Summary

In this lesson we learned…

- The integration of the Spring Boot Maven plugin in a project
- Its configuration options
- The configuration of the ‘executions’ tag in it

Further information

- Spring Boot Maven plugin - official documentation

Q & A session

- Please ask your questions on the presented topics

Packaging and running the app

Goals

- ✓ Learn an overview of to package and run a Spring Boot app
- ✓ Learn the usage of the Spring Boot plugin for these tasks
- ✓ Learn the specifics of ‘jar’ and ‘war’ packaging

Packaging and running the app

10m

Creating JAR and WAR archives

- Spring Boot apps are packaged as JAR files, by default
  - The default configuration of the 'spring-boot' Maven plugin
- WAR files can be also created - the following changes are needed:
  - Maven changes:
    - The 'maven-war-plugin' must be added and configured
    - The value of the 'packaging' property must be set to 'war'
  - Code changes - the main class must:
    - Extend the `SpringBootServletInitializer` class → binds the Servlet, Filter and `ServletContextInitializer` beans to the running web server
    - Override the 'configure' method, to specify the main Spring Boot class

Demo

- Using the Spring Boot Maven plugin to:
    - Run a project
    - Repackage the application

Activity

Packaging and running the application

Scenario:

- Using the Spring Boot Maven to:
  - Run our project
  - Repackage the built JAR file

Aim:

Understanding:

- How to use the Spring Boot Maven plugin to run and repackage a project

Steps to use the Spring Boot Maven plugin in our project

1. Open the project’s pom.xml file
2. Open the Spring Boot Maven plugin official page -
   https://docs.spring.io/spring-boot/docs/current/maven-plugin/
3. Open the ‘Usage’ page
4. Copy the plugin body in the <build><plugins> section of the pom.xml file

→ continued on the next slide

Steps to use the Spring Boot Maven plugin to run our project

1. Open the Maven window from the IDE → upper-right side of the screen
2. Expand the ‘Lifecycle’ section
3. Run the ‘clean’ tasks
4. Expand the plugins → ‘spring-boot’ section
5. Double click on the ‘spring-boot:run’ goal
6. Desired outcome → the application should be compiled and started by the plugin

Steps to use the Spring Boot Maven plugin to repackage our project

1. Expand the ‘Lifecycle’ and ‘Plugins’ → ‘spring-boot’ sections of the Maven window
2. Click on the ‘clean’, ‘package’ and ‘spring-boot:repackage’ tasks, by holding down the Cmd / Ctrl key
3. Right click + ‘Create [clean,package,spring-boot:run]’ run configuration
4. Execute the created run configuration
5. Desired outcome → the folder ‘target’ should contain two .jar files: the original jar file and the repackaged one

Summary

In this lesson we learned…

- The integration of the Spring Boot Maven plugin in a project
- Its configuration options
- An overview of its usage to:
    - Run a project
    - Repackage an already packaged project

Further information

- Spring Boot Maven plugin - official documentation

Q & A session

- Please ask your questions on the presented topics

Using an application as a service

Spring Boot Admin overview

Goals

- ✓ Learn an overview of how to use a Spring Boot app as an OS service
- ✓ Learn an overview of the Spring Boot Admin tool
- ✓ Learn how to use Spring Boot Admin to manage a Spring Boot app

Using an application as a service

5m

Installing an app as an OS service

A Spring Boot application (JAR file *) can be installed as an:

- On Linux systems:
  - init.d service
  - systemd service
- Windows service

* - WAR files need to be deployed in a web or application server

Creating an executable JAR file

The following configuration option must be configured in the Spring Boot Maven project:

```
<configuration>
  <executable>true</executable>
</configuration>
```

Installing an app as an OS service
Linux init.d systems

The following steps are needed to install the app as an init.d process:

- Create a symlink for the jar file to the init.d folder → support for the start, stop, restart, and status commands.
- The script supports the following features:
  - Start the services (as the user that owns the jar file)
  - Tracks the application's PID - /var/run/<appname>/<appname>.pid
  - Writes console logs to /var/log/<appname>.log

Installing an app as an OS service Linux system.d systems

- For a Spring Boot application installed in /var/great-app:

```
[Unit]
Description=great-app
After=syslog.target

[Service] User=great-app
ExecStart=/var/great-app/great-app.jar
SuccessExitStatus=143

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

- Automatic startup: `systemctl great-app myapp.service`

Installing an app as an OS service
Windows systems

- Using a Spring Boot application as a Windows service

Spring Boot Admin

5m

Spring Boot Admin
overview

Spring Boot Admin:

- An admin interface for managing Spring Boot applications
- Applications are managed through the Spring Boot Actuator endpoints
    - Applications are self-registering to the Spring Boot Admin instance
- Available operations:
    - Show health status, build number, JVM, memory and DB metrics
    - Change the logging levels
    - View / download logs, thread and heap dumps
    - View scheduled tasks, audit events, ...

Spring Boot Admin
overview (continued)

- Can be configured to send notifications via:
  - Email
  - Slack channels
  - HTTP endpoints

Spring Boot Admin
components

- A Spring Boot app to which all the monitored apps need to register
- The monitored apps → they will register to the Spring Boot Admin Server
  - Multiple apps and/or instances can register to the same SBA instance

Spring Boot Admin
integration

- Including the Maven dependency:

```
<dependency>
   <groupId>de.codecentric</groupId>
   <artifactId>spring-boot-admin-starter-server</artifactId>
</dependency>
```

- Creating the Spring Boot Admin server class

```
@Configuration
@EnableAutoConfiguration
@EnableAdminServer
public class SpringBootAdminApplication {
   public static void main(String[] args) {
          SpringApplication.run(SpringBootAdminApplication.class, args);
     }
}
```

Spring Boot Admin - security integration

From the security perspective, the Spring Boot Admin tool can be used in two ways:

- Without security:
  - Easier to setup / integrate → no security configuration is needed
  - Advised for internal usage only, where no security is needed
- With security:
  - The Spring Boot Admin requires integration with Spring Security
  - Advised for enterprise usage modes, where either:
    - The exposed information is sensitive
    - The access to the Spring Boot Admin server must require authentication

Demo

- Integrating and using Spring Boot Admin

Summary

In this lesson we learned…

- An overview of the Spring Boot Admin project
- Its integration in a Spring Boot project
- The integration between the Spring Boot Admin and Spring Security, if/when needed

Further information

- Spring Boot Admin Reference
- Securing a Spring Boot Admin app

Q & A session

- Please ask your questions on the presented topics

Training wrap-up

Course overview

We have learned an overview of:

- What is Spring Boot, how it can help us in building reliable applications
- The new features brought by Spring Boot 2
- The core Spring & Spring Boot features:
  - Spring Boot starter modules
  - Configuration files and profiles usage
  - Conditional annotations
- The web and database access characteristics
- The additional tools and plugins which can be used in a Spring Boot project

Small advice

- Start a small project, grow it using several technologies
  - Not only Spring related; whatever you want / wish to learn
- Choose an useful topic, for you or for the community
  - **Must** be on a topic which interests you:
    - Social
    - Media
    - Hobbies
    - Financing
- Grow, improve, refactor, test, release it

→ The best source / means of learning, by far

Deliberate practice

if (enoughTime)

```
Optional.ofNullable(whatWentGoodInTheTraining)
      .and(whatCanBeImprovedForTheFuture)
      .forEach(participant → sayFeedback(participant));

ThankYou sayFeedback(Participant participant) {
  sayFeedbackFor(“Continue doing“);
  sayFeedbackFor(“Start doing”);
  sayFeedbackFor(“Stop doing”);
}
```

Thank you!