Mastering the Spring Framework

Chapter 3:
Introduction to SpringBoot

Chapter Objectives

In this chapter, we will discuss:

- Understanding what a Spring Boot & its features
- JPA & CrudRepository

What is SpringBoot?

- It is a Spring module which provides RAD (Rapid Application Development) feature to Spring framework.
- no requirement for XML configuration.
- uses convention over configuration software design paradigm
- It provides opinionated 'starter' POMs to simplify your Maven configuration.
- It automatically configure Spring whenever possible.
- It provides production-ready features such as metrics, health checks and externalized configuration.

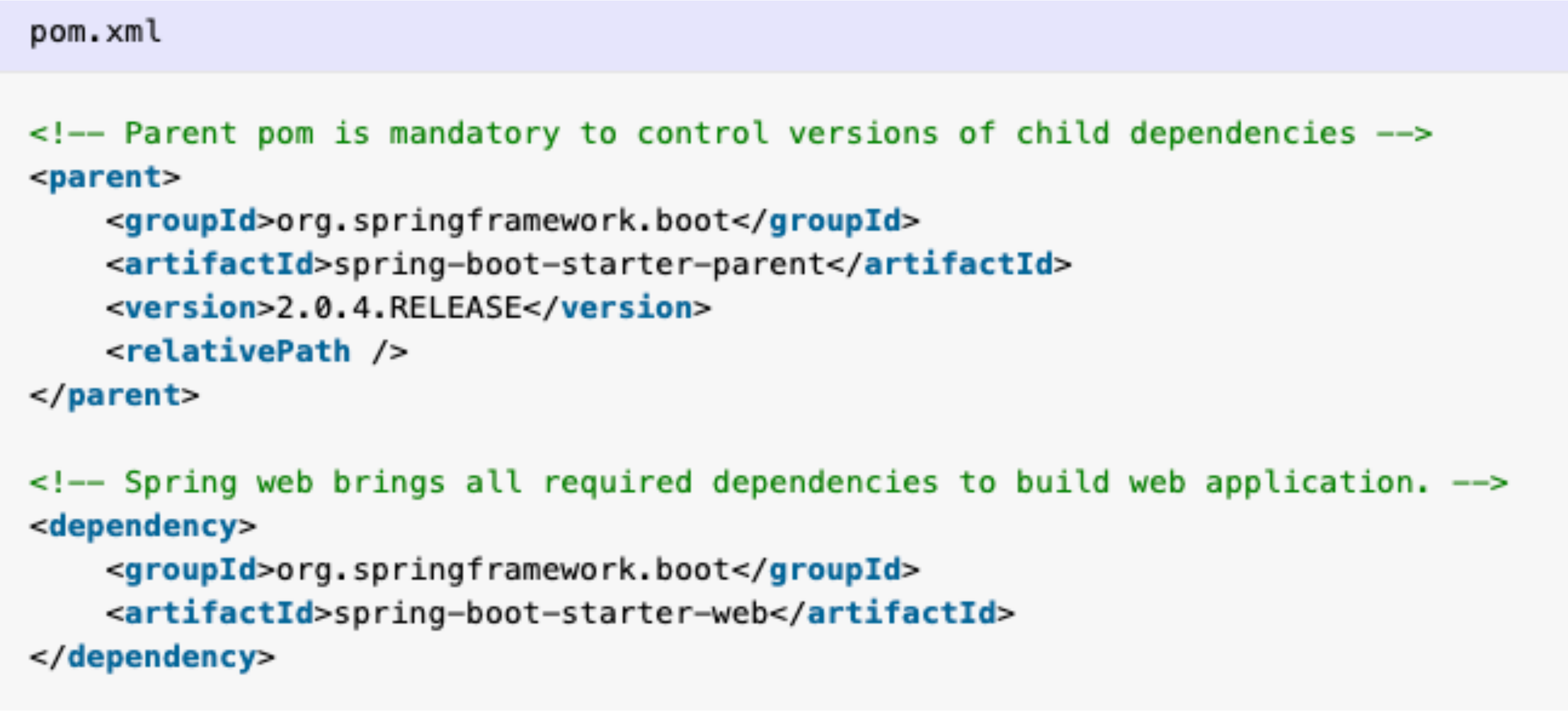
Springboot features

- Web Development
- SpringApplication
- Application events and listeners
- Admin features
- Externalized Configuration
- Properties Files
- YAML Support
- Type-safe Configuration
- Logging
- Security

Starter Template

- Spring Boot starters are templates that contain a **collection of all the relevant transitive dependencies** that are needed to start a particular functionality.
- For example, If you want to create a Spring WebMVC application then in a traditional setup, you would have included all required dependencies yourself. It leaves the chances of **version conflict** which ultimately result in more **runtime exceptions**.
- With String boot, to create MVC application all you need to import is spring-boot-starter-web dependency.

Starter Template



pom.xml

```
<!-- Parent pom is mandatory to control versions of child dependencies -->
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.0.4.RELEASE</version>
    <relativePath />
</parent>

<!-- Spring web brings all required dependencies to build web application. -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Spring boot autoconfiguration

- Autoconfiguration is enabled with @EnableAutoConfiguration annotation.
- Spring boot auto configuration scans the classpath, finds the libraries in the classpath and then attempt to guess the best configuration for them, and finally configure all such beans.
- Auto-configuration tries to be as intelligent as possible and will back-away as you define more of your own configuration.
- Auto-configuration is always applied after user-defined beans have been registered.
- Spring boot auto-configuration logic is implemented in **spring-boot-autoconfigure.jar**.

Embedded Server

- Spring boot applications always include **tomcat** as **embedded server** dependency.
- It means you can run the Spring boot applications from the command prompt without needling complex server infrastructure.
- You can exclude tomcat and include any other embedded server if you want. Or you can make exclude server environment altogether. It’s all configuration based.

Bootstrap the application

- To **run the application**, we need to use @SpringBootApplication annotation.
- Behind the scenes, that’s equivalent to @Configuration, @EnableAutoConfiguration, and @ComponentScan together.
- It enables the scanning of config classes, files and load them into **spring context**.
- execution start with main() method. It start loading all the config files, configure them and bootstarp the application based on application properties in **application.properties** file in /resources folder.

Advantages of Springboot

- Spring boot helps in **resolving dependency conflict**. It identifies required dependencies and import them for you.
- It has information of **compitable version** for all dependencies. It minimizes the runtime **classloader** issues.
- It's "opinionated defaults configuration" approach helps you in configuring most important pieces behind the scene. Override them only when you need. Otherwise everything just works, perfectly. It helps in avoiding **boilerplate code**, annotations and XML configurations.
- It provides embedded HTTP server Tomcat so that you can develop and test quickly.

Spring JPA

- **Spring Boot configures *Hibernate* as the default JPA provider**
- **Spring Boot can also auto-configure the *dataSource* bean, depending on the database used**. In the case of an in-memory database of type *H2*, *HSQLDB* and *Apache Derby*, Boot automatically configures the *DataSource* if the corresponding database dependency is present on the classpath.
- If we want to use JPA with *MySQL* database, then we need the *mysql-connector-java* dependency, as well as to define the *DataSource* configuration.

How does JPA work?

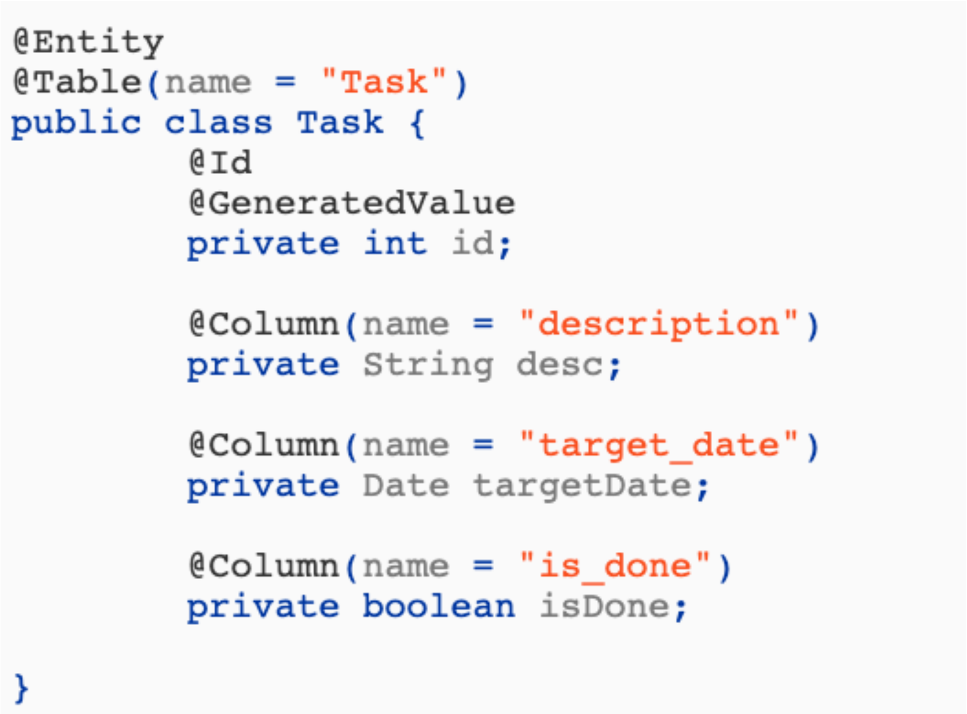
- JPA evolved as a result of a different thought process. How about mapping the objects directly to tables?
  - Entities
  - Attributes
  - Relationships
- This Mapping is also called ORM - Object Relational Mapping. Before JPA, ORM was the term more commonly used to refer to these frameworks.
- Thats one of the reasons, Hibernate is called a ORM framework.

JPA vs Hibernate

- Hibernate is one of the most popular ORM frameworks.
- JPA defines the specification. It is an API.
  - How do you define entities?
  - How do you map attributes?
  - How do you map relationships between entities?
  - Who manages the entities?
- Hibernate is one of the popular implementations of JPA.
  - Hibernate understands the mappings that we add between objects and tables.
  - It ensures that data is stored/retrieved from the database based on the mappings.
  - Hibernate also provides additional features on top of JPA.

JPA annotations

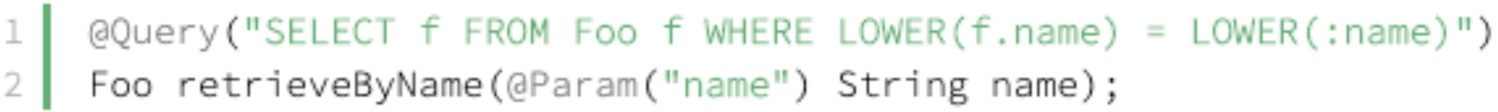
- Some JPA annotations
  - @Table(name = “Task”)
  - @Id
  - @GeneratedValue
  - @Column(name = “description”)



```
@Entity
@Table(name = "Task")
public class Task {
        @Id
        @GeneratedValue
        private int id;

        @Column(name = "description")
        private String desc;

        @Column(name = "target_date")
        private Date targetDate;

        @Column(name = "is_done")
        private boolean isDone;

}
```

Manual queries in JPA

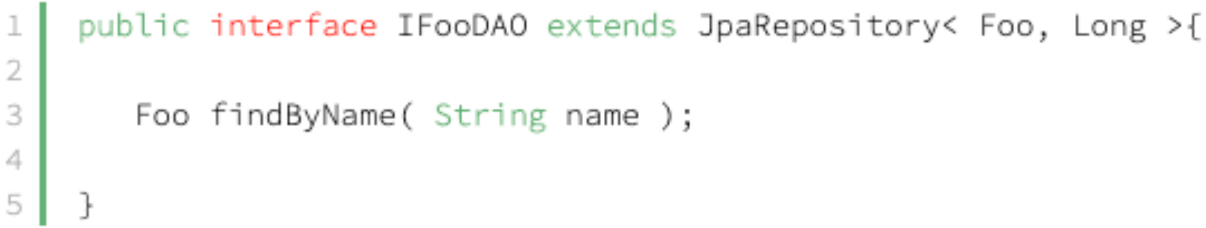
- a custom query that we will define via the *@Query* annotation:



```
@Query("SELECT f FROM Foo f WHERE LOWER(f.name) = LOWER(:name)")
Foo retrieveByName(@Param("name") String name);
```

Automatic custom queries

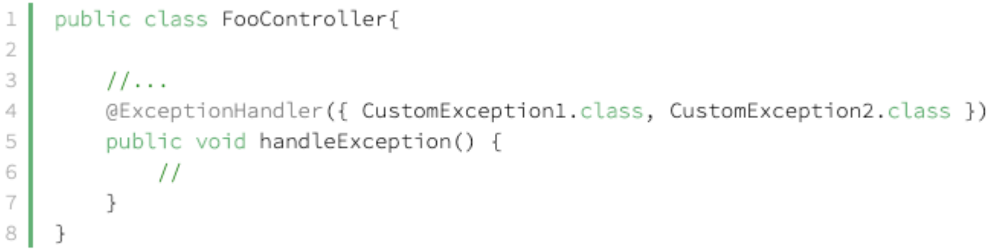
- When Spring Data creates a new *Repository* implementation, it analyses all the methods defined by the interfaces and tries to **automatically generate queries from the method names**.



```
public interface IFooDAO extends JpaRepository< Foo, Long >{

   Foo findByName( String name );

}
```

The Controller level *@ExceptionHandler*

- The first solution works at the *@Controller* level – we will define a method to handle exceptions, and annotate that with *@ExceptionHandler*:

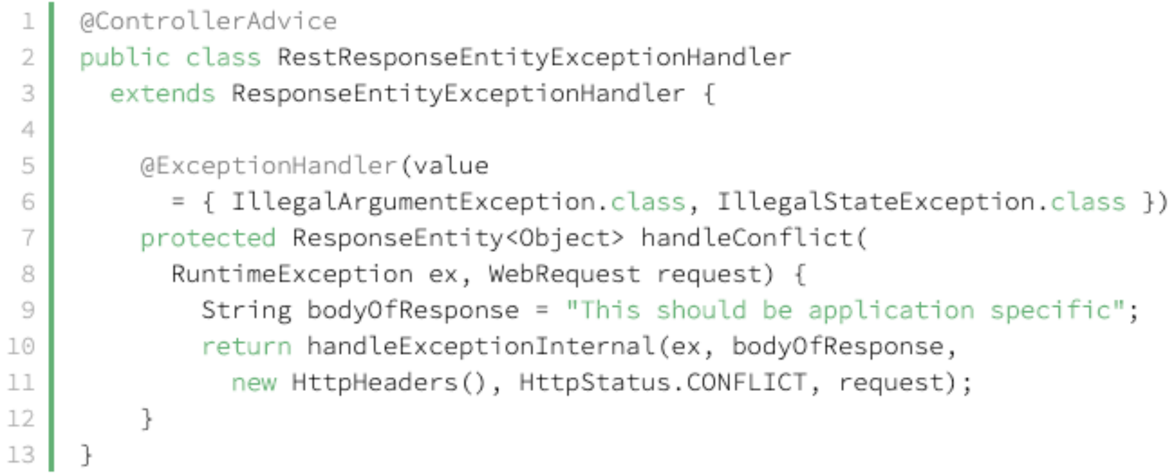


```
public class FooController{

    //...
    @ExceptionHandler({ CustomException1.class, CustomException2.class })
    public void handleException() {
        //
    }
}
```

- **the *@ExceptionHandler* annotated method is only active for that particular Controller**, not globally for the entire application.

@ControllerAdvice

- Spring 3.2 brings support for **a global *@ExceptionHandler* with the *@ControllerAdvice* annotation**. This enables a mechanism that breaks away from the older MVC model and makes use of *ResponseEntity* along with the type safety and flexibility of *@ExceptionHandler*:

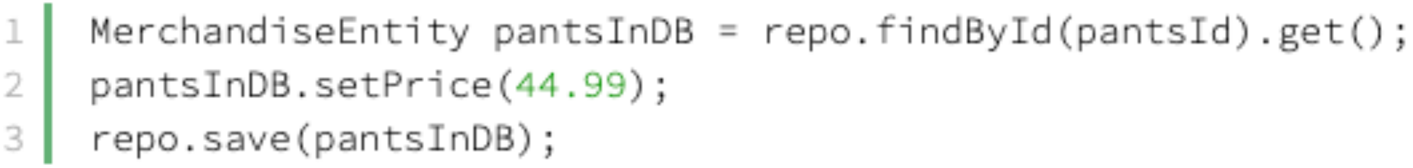


```
@ControllerAdvice
public class RestResponseEntityExceptionHandler
  extends ResponseEntityExceptionHandler {

    @ExceptionHandler(value
      = { IllegalArgumentException.class, IllegalStateException.class })
    protected ResponseEntity<Object> handleConflict(
      RuntimeException ex, WebRequest request) {
        String bodyOfResponse = "This should be application specific";
        return handleExceptionInternal(ex, bodyOfResponse,
          new HttpHeaders(), HttpStatus.CONFLICT, request);
    }
}
```

@ControllerAdvice

- The*@ControllerAdvice* annotation allows us to **consolidate our multiple, scattered *@ExceptionHandler*s from before into a single, global error handling component**.
- The actual mechanism is extremely simple but also very flexible. It gives us:
- Full control over the body of the response as well as the status code
- Mapping of several exceptions to the same method, to be handled together, and
- It makes good use of the newer RESTful *ResposeEntity* response

CrudRepository



```java
MerchandiseEntity pantsInDB = repo.findById(pantsId).get();
pantsInDB.setPrice(44.99);
repo.save(pantsInDB);
```