



Creating RESTful web  
services with Spring  
Boot

# The Spring framework

- Free and open source
- Inversion of Control Container (IoC)
- Modules
  - DI / AOP
  - Data /Security
  - Web MVC/ REST
  - So much more +++



# What is Spring Boot ?



- Getting started with Spring Boot is easy
- Convention over configuration
- SPRING CLI
- Entry Point
- Starter POMs
- Production Ready
- Run anywhere!



# Starter POMs

- spring-boot-starter-\*
- Set of convenient dependency descriptors
- So what starter POMs are available ?
  - docs.spring.io
  - The most common ones :
    - **spring-boot-starter**
    - **spring-boot-starter-actuator**
    - **spring-boot-starter-jdbc**
    - **spring-boot-starter-data-\***
    - **spring-boot-starter-test**
    - **spring-boot-starter-security**
    - **spring-boot-starter-web**



# Create a BOOtiful app on the fly



- ... With Spring Initializr Project -> <http://start.spring.io>

The screenshot shows the Spring Initializr web application interface. At the top, it says "SPRING INITIALIZR bootstrap your application now". Below this, there are three dropdown menus: "Generate a" with "Maven Project" selected, "with" with "Java" selected, and "and Spring Boot" with "1.5.9" selected. The interface is divided into two main sections: "Project Metadata" and "Dependencies".  
**Project Metadata**  
Artifact coordinates  
Group:   
Artifact:   
**Dependencies**  
Add Spring Boot Starters and dependencies to your application  
Search for dependencies:   
Selected Dependencies: (empty)  
At the bottom center, there is a green button labeled "Generate Project" with a keyboard shortcut "alt + ⌘". Below the button, there is a link: "Don't know what to look for? Want more options? [Switch to the full version.](#)"

# Executable JARs

- Spring Boot includes a Maven plugin that can package the project as an executable jar.
- Add the plugin to your <plugins> section if you want to use it

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



# Spring Boot: Auto Configuration



- Spring Boot Auto Configuration attempts to automatically configure your Spring application based on the jar dependencies that you have added
- Autoconfiguration is not invasive



## Auto Configuration

Spring Boot: Auto Configuration

# REST



- architectural principle for managing state information

## **A few RESTful principles**

- In REST principles, all resources are identified by the Uniform Resource Identifier (URI).
- HTTP REST resources are represented in some media types, such as XML, JSON, and RDF.
- RESTful resources are self-descriptive
- In REST principles, the server will not keep any state about the client session on the server side; hence, it's stateless
- With RESTful web services, a client can cache any response coming from the server. The server can mention how, and for how long, it can cache the responses.



# HTTP Verbs



HTTP METHOD	PATH	DESCRIPTION
GET	/users	Gets all users in the repository
GET	/users/{id}	Gets the user with an id of 1
POST	/users	Creates a new user
PUT	/users/1	Creates or updates an existing user
DELETE	/users/1	Deletes user with an id of 1

# HTTP Status Codes

Status Code	Description
1xx	Informational
2xx	Success
3xx	Redirection
4xx	Client Error

# Spring MVC



- `FrontController = DispatcherServlet`
- `@EnableWebMvc` : Spring Boot adds it automatically when it sees Spring MVC on the classpath
- This flags the application as a webapp & activates features like : setting up the `DispatcherServlet` for us
- Web service components have special annotations like `@Service`, `@RestController` – provide hints of the role that the class has

# CRUD operations with Spring Boot



```
@RestController
@RequestMapping("/user")
public class UserController {

    @Autowired
    UserService userService;

    @RequestMapping("")
    public List<User> getAllUsers() { return userService.getAllUsers(); }

    @GetMapping("/{id}")
    public User getUser(@PathVariable("id") Integer id) { return userService.getUser(id); }

    @PostMapping(value = "")
    public Map<String, Object> createUser(@RequestParam(value = "userid") Integer userid,
                                         @RequestParam(value = "username") String username) {
        Map<String, Object> map = new LinkedHashMap<>();
        userService.createUser(userid, username);
        map.put("result", "added");
        return map;
    }
}
```

# CRUD operations with Spring Boot



```
💡 @PutMapping(value =("/{userid}")
public Map<String, Object> updateUser(@PathVariable Integer userid,
                                     @RequestParam(value = "username") String username) {
    Map<String, Object> map = new LinkedHashMap<>();
    userService.updateUser(userid, username);
    map.put("result", "updated");
    return map;
}

@DeleteMapping(value =("/{id}")
public Map<String, Object> deleteUser(
    @PathVariable("id") Integer userid) {
    Map<String, Object> map = new LinkedHashMap<>();
    userService.deleteUser(userid);
    map.put("result", "deleted");
    return map;
}
}
```

# Data Access with Spring Boot



- Creating Entities & Repositories

- Spring-boot-starter-jpa
- we're using JPA , a way allows us to map objects to r
- it has Hibernate, one of the most popular JPA impleme
- it gives us spring-data-jpa which makes it easy to in
- it gives us spring-orm which gives us core ORM support
- flag a POJO with @Entity if you want it to become a J

A screenshot of a code editor window showing the implementation of a JPA entity named Post. The code includes imports for javax.persistence.Entity, GeneratedValue, Id, and ManyToOne. The Post class is annotated with @Entity and contains fields for id (with @Id and @GeneratedValue), title, body, postedOn, and a ManyToOne relationship with an Author. A private no-arg constructor is also shown.

```
4
5 import javax.persistence.Entity;
6 import javax.persistence.GeneratedValue;
7 import javax.persistence.Id;
8 import javax.persistence.ManyToOne;
9
10 @Entity
11 public class Post {
12
13     @Id
14     @GeneratedValue
15     private Long id;
16
17     private String title;
18     private String body;
19     private Date postedOn;
20
21     // Author
22     @ManyToOne
23     private Author author;
24
25     // private no arg constructor is needed by JPA
26     private Post(){
27
```

# Data Access with Spring Boot



- We get a lot of methods out of the box from CrudRepository : save(entity), findOne(id), exists(id), delete(id), delete(entity)
- construct queries with very little code. queries get constructed by Spring from the me

A screenshot of an IDE window showing a Java source file named PostRepository.java. The code defines a Spring Data JPA repository interface. It starts with a package declaration 'com.therealdanvega.repository', followed by an import for 'java.util.List'. The interface is annotated with '@Repository' and extends 'CrudRepository<Post, Long>'. It contains several method signatures, with the last one highlighted: 'List<Post> findAllByAuthorFirstNameIgnoreCaseAndAuthorLastNameIgnoreCase(String first, String last);'. Other methods include 'findAllByAuthorFirstName', 'findAllByAuthorFirstNameIgnoreCase', and 'findAllByAuthorFirstNameIgnoreCaseOrderByPostedOnDesc'. There are also comments for 'AUTHOR', 'KEYWORDS', 'ACTIVE', 'AUTHOR & KEYWORDS', and 'QUERY' sections.

```
1 package com.therealdanvega.repository;
2
3 import java.util.List;
4
11
12 @Repository
13 public interface PostRepository extends CrudRepository<Post, Long> {
14
15     // AUTHOR -----
16
17     List<Post> findAllByAuthorFirstName(String first);
18
19     List<Post> findAllByAuthorFirstNameIgnoreCase(String first);
20
21     List<Post> findAllByAuthorFirstNameIgnoreCaseOrderByPostedOnDesc(String first);
22
23     List<Post> findAllByAuthorFirstNameIgnoreCaseAndAuthorLastNameIgnoreCase(String first, String last);
24
25     // KEYWORDS -----
26
27
28     // ACTIVE -----
29
30
31     // AUTHOR & KEYWORDS -----
32
33
34     // QUERY -----
```

# Testing RESTful web services



- Junit
- MockMvc
- @MockBean
- JPA components Tests
- WebMVC component Test



# Security & JWT



- JWT tokens are URL-safe and web browser-compatible especially for Single Sign-On (SSO) contexts.
- JWT has three parts:
  - Header
  - Payload/body
  - Signature
- OAuth2.0

# Building a REST Client



- If we want to build a REST Client to consume another REST API, SpringTemplate is a Spring class used to consume REST API from the client side .
- RestTemplate can be used use to call GET, POST, PUT, DELETE, and other advanced HTTP methods (OPTIONS, HEAD)

```
@RestController
@RequestMapping("/client")
public class ClientController {

    private final Logger _log = LoggerFactory.getLogger(this.getClass());

    @Autowired
    private RestTemplate template;

    @ResponseBody
    @RequestMapping("/test")
    public Map<String, Object> test(){
        Map<String, Object> map = new LinkedHashMap<>();

        String content = template.getForObject("http://localhost:8080/", String.class);
        map.put("result", content);

        return map;
    }
}
```

# Error Handling



- usually when we deal with an unexpected situation in REST APIs, it will automatically throw an HTTP error such as 404
- we might need a JSON format result regardless of whether things go right or wrong
- we can use Spring Boot's `@ExceptionHandler` to manage errors & display them in JSON format

```
45 public Post read(@PathVariable(value="id") long id) throws PostNotFoundException {
46     Post post = postService.read(id);
47     if( post == null ){
48         throw new PostNotFoundException("Post with id: " + id + " not found.");
49     }
50     return post;
51 }
52
53 @RequestMapping( value =("/{id}", method = RequestMethod.PUT )
54 public Post update(@PathVariable(value="id") long id, @RequestBody Post post){
55     return postService.update(id,post);
56 }
57
58 @RequestMapping( value =("/{id}", method = RequestMethod.DELETE )
59 public void delete(@PathVariable(value="id") int id){
60     postService.delete(id);
61 }
62
63 @ExceptionHandler(PostNotFoundException.class)
64 public void handlePostNotFound(PostNotFoundException exception, HttpServletResponse response) throws IOException{
65     response.sendError( HttpStatus.NOT_FOUND.value(), exception.getMessage() );
66 }
67 }
68
```

QUESTIONS

&

ANSWERS

