Introduction to STS Programming

A practical example using a `RESTful` services

M.I. Capel

ETS Ingenierías Informática y
Telecomunicación
Departamento de Lenguajes y Sistemas Informáticos
Universidad de Granada
Email: manuelcapel@ugr.es

DSBCS

Máster en Ingeniería Informática

October 2, 2024

Índice

What is STS (Spring Tool Suite)?

Fundamentals

- STS is an Eclipse-based development environment tailored for building Spring applications.
- It provides various tools and features to simplify development in Spring Framework.

Overview of the programming example

Application11 MV and Java STS project

- Application11 is a RESTful Spring Boot project demonstrating CRUD operations.
- Uses technologies: Spring Boot, Spring Data JPA, and an H2 in-memory database.

Project structure

It describes the typical structure of a Spring Boot project

- `src/main/java`: Contains source code.
- `src/main/resources`: For properties and templates.
- `pom.xml`: The Maven configuration file.

Creating the Application Class

What is it for?

- `Application11Aplication.java` contains the main code of your application
- It uses `@SpringBootApplication` to start the application.
- Contains a `CommandLineRunner` to execute some code at startup.

```
1  @SpringBootApplication
2  public class Application11Application {
3          private static final Logger log = LoggerFactory.
               getLogger(Application11Application.class);
4          public static void main(String[] args) {
5                  SpringApplication.run(Application11Application.
                       class, args);
6          }
7          @Bean
8      CommandLineRunner jpaSample(TodoRepository todoRepo) {
9        return (args) -> {
10       ....// RESTful code
11     }
12  }
```

Creating an Entity Class

Class - “Todo”

- The `Todo` class represents a task entity.
- Uses `@Entity, @Id`, and other JPA annotations for database mapping.
- Demonstrates dependency injection and properties like `summary, description, dueDate` class attributes.

Creating an Entity Class–2

```
1  @Entity
2  @Data
3  public class Todo implements ITodo {
4          @GeneratedValue(strategy = GenerationType.AUTO)
5          @Id
6          private long id;
7          private String summary;
8          private String description;
9          private Boolean done;
10         private Date dueDate;
11         public Todo() { }
12         @Autowired
13         public Todo(@Qualifier("summary") String summary) {
14               this.summary = summary;      }
15         @Override
16         public long getId() {
17                 return id;       }
18         @Override
19         public String getSummary() {
20                 return summary;
21         }
22         .....
23 }
```

Dependency injection and `@Autowired`

Inyección de dependencias

- `@Autowired` is used to inject dependencies into the `Todo` class.
- Qualifiers like `summary` and `description` are used to define default values for the Todo entity.

Dependency injection and `@Autowired`

```
1
2  @Autowired
3          public Todo(@Qualifier("summary") String summary) {
4                  this.summary = summary;
5           }
6
7           ...
8  @Autowired
9      @Qualifier("description")
10         @Override
11         public String getDescription() {
12                 return description;
13         }
```

Creating a repository interface

Extending Jpa repository

- The `TodoRepository` interface extends JpaRepository.
- Uses `@RepositoryRestResource` to expose repository as a RESTful resource.

The TodoRepository

```
1 import java.util.List;
2 import org.springframework.data.jpa.repository.JpaRepository;
3 import org.springframework.data.rest.core.annotation.
      RepositoryRestResource;
4
5 @RepositoryRestResource(collectionResourceRel="tasks",path="
      tasks")
6 public interface TodoRepository extends
7         JpaRepository<Todo, Long> {
8
9 }
```

Configuring a REST Controller

The nuts and bolts of REST Controllers

- The `RootUriController` maps the root URL `/index` to a template.
- Uses `@Controller` and `@RequestMapping`.

Configuring a REST Controller

```
1  import org.springframework.stereotype.Controller;
2  import org.springframework.web.bind.annotation.RequestMapping;
3  @Controller
4  public class RootUriController {
5          @RequestMapping(value = "/index")
6      public String index() {
7        return "index";
8      }
9  }
10 ////////////////////
11 <!DOCTYPE html>
12 <html xmlns:th="http://www.thymeleaf.org">
13 <head>
14 <!--     <meta charset="UTF-8"/> -->
15     <title>Spring Training</title>
16     <meta http-equiv="Content-Type" content="text/html; charset=
           UTF-8" />
17 </head>
18 <body>
19     <div id="content">Hola Spring!Comenzamos las practicas de
           DSS....</div>
20 </body>
21 </html>
```

Exposing RESTful Endpoints

Contents accessed only through Endpoints

- The `@RepositoryRestResource` on `TodoRepository` automatically exposes CRUD endpoints.
- Examples of REST endpoints:
  - `GET /tasks` - Fetch all `Todo` items.
  - `POST /tasks` - Add a new `Todo`.
- RESTful endpoints can be tested using the free application Postman

Getting and Posting through the endpoints

```
1  RestTemplate restTemplate = new RestTemplate();
2     //Ahora los vamos a obtener del servidor REST
3     Todo firstTodo = restTemplate.getForObject("http://localhost
          :8080/rest/tasks/1", Todo.class);
4   System.out.println(firstTodo);
5  ////////////
6  Todo newTodo = new Todo("New Todo entity");
7     newTodo.setDescription("Todo added by the API REST");
8     newTodo.setDone(true);
9     ResponseEntity<Todo> postForEnt = restTemplate.postForEntity
          ("http://localhost:8080/rest/tasks", newTodo, Todo.
          class);
10    System.out.println("Posted entity in repo"+postForEnt);
```

Configuration and Beans

Where are my Beans?

- `Config` class demonstrates defining beans for the `Todo` entity properties.
- `@Bean` and `@Qualifier` annotations are used for defining and identifying different beans.

Configuration and Beans–2

```
1  @Configuration
2  public class Config {
3          @Bean
4      public Long getId() {
5          return Long.valueOf(0);
6      }
7      @Bean
8      @Qualifier("summary")
9      public String getSummary() {
10         return "Spring: prueba de Inyeccion de Dependencias";
11     }
12     @Bean
13     @Qualifier("description")
14     public String getDescription() {
15         return "Spring: prueba de Inyeccion de Dependencias y
               todo lo demas";
16     }
17     @Bean
18     public Boolean isDone() {
19         return Boolean.FALSE;
20     }
21     @Bean
22     public Date getDueDate() {
23         return new Date();
24     }
25 }
```

Configuring `application.properties`

Important idea

- The `application.properties` file contains the configuration for your Spring Boot application.
- Sets up properties for REST endpoints, database connections, and other application settings.

The `Pom.xml` file

- The `pom.xml` (Project Object Model) file is a core configuration file for any Maven project, including Spring Boot applications.
- It manages dependencies, plugins, build configurations, and project metadata.

Starting and Testing the Application

First steps with STS IDE

- Run the application using STS.
- Check the REST endpoints and H2 database console (via /h2-console).