

Boot_Rest_MVC_Demo

Created by :
Sangeeta Joshi

Spring Boot_REST demo

Example: a Restful CRUD API for a simple note-taking application

- A Note can have a title and some content.
- We'll first build the apis to create, retrieve, update and delete a Note
- then test them using postman

Spring Boot_REST demo

Spring Boot provides a web tool called [Spring Initializer](#) to bootstrap an application quickly.

- step 0: go to <http://start.spring.io>
- Step 1 : Click Switch to full version

Spring Boot_REST demo

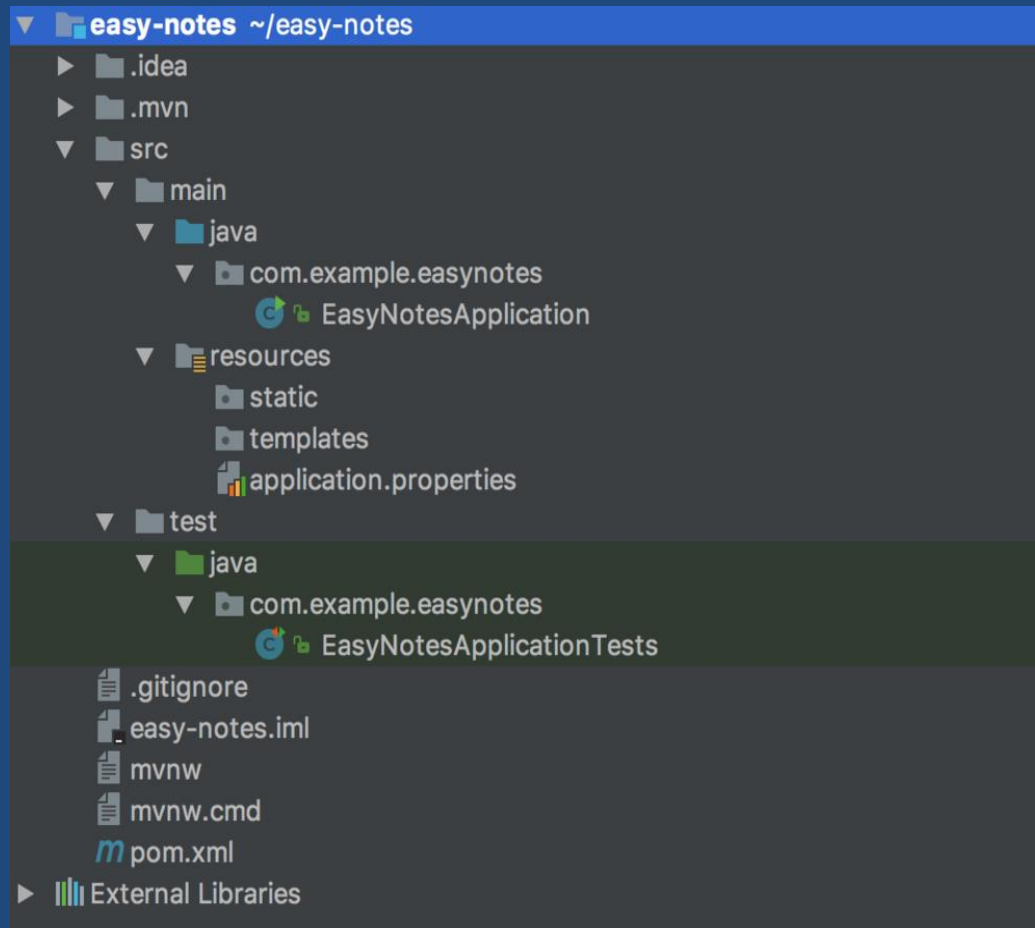
- Step 2 : Enter the details as follows -
 - Group : com.example
 - Artifact : easy-notes
 - Name : easy-notes
 - Description : Rest API for a Simple Note Taking Application
 - Package Name : com.example.easynotes
 - Packaging : jar (This is the default value)
 - Java Version : 1.8 (Default)
 - Dependencies : **Web, JPA, MySQL, DevTools**

Spring Boot_REST demo

- click Generate Project to generate and download your project.
- **Spring Initializer** will generate the project with the details you have entered
- download a zip file with all the project folders.
- Next, Unzip the downloaded zip file and import it into your favorite IDE

Spring Boot_REST demo

- Exploring the Directory Structure



Spring Boot_REST demo

1. EasyNotesApplication

- This is the main entry point of our Spring Boot application.
package com.example.easynotes;

```
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;
```

```
@SpringBootApplication
```

```
public class EasyNotesApplication {
```

```
    public static void main(String[] args) {
```

```
        SpringApplication.run(EasyNotesApplication.class, args);
```

```
    }
```

```
}
```

Spring Boot_REST demo

@SpringBootApplication : which is a combination of following

- **@Configuration** : Any class annotated with @Configuration annotation is **bootstrapped** by Spring and is also **considered as a source of other bean definitions**.
- **@EnableAutoConfiguration** : This annotation tells Spring to **automatically configure** your application based on the dependencies that you have added in **the pom.xml** file.
- For example, If spring-data-jpa is in the classpath, then it automatically tries to configure a DataSource by reading the database properties from application.properties file.
- **@ComponentScan** : It tells Spring to scan and bootstrap other components defined in the current package (com.example.easynotes) and all the sub-packages.
- The main() method calls Spring Boot's SpringApplication.run() method to launch the application.

Spring Boot_REST demo

2. *resources/*

- This directory, is dedicated to all the static resources, templates and property files.
- `resources/static` - contains static resources such as css, js and images.
- `resources/templates` - contains server-side templates which are rendered by Spring.

Spring Boot_REST demo

resources/application.properties -

- It contains application-wide properties.
- Spring reads the properties defined in this file to configure your application.
- You can define server's default port, server's context path, database URLs etc, in it

3. **EasyNotesApplicationTests** - Define unit and integration tests here.

4. **pom.xml** - contains all the project dependencies

Spring Boot_REST demo

- Configuring MySQL Database
- Spring Boot tries to auto-configure a DataSource if *spring-data-jpa is in the classpath* by reading the database configuration from application.properties file.
- So, we just have to add the configuration and Spring Boot will take care of the rest.
- Open application.properties file and add the following properties to it.

Spring Boot_REST demo

Spring DATASOURCE (DataSourceAutoConfiguration & DataSourceProperties)

```
spring.datasource.url = jdbc:mysql://localhost:3306/notes_app?useSSL=false  
spring.datasource.username = root  
spring.datasource.password = root
```

Hibernate Properties

The SQL dialect makes Hibernate generate better SQL for the chosen database

```
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5Dialect
```

Hibernate ddl auto (create, create-drop, validate, update)

```
spring.jpa.hibernate.ddl-auto = update
```

Spring Boot_REST demo

- create a database named notes_app in MySQL
- Spring Boot uses *Hibernate as the default JPA implementation*.
- The property *spring.jpa.hibernate.ddl-auto* is used for database initialization.
- value “update” for this property does two things -
- When you define a domain model, a table will automatically be created in the database and the fields of the domain model will be mapped to the corresponding columns in the table.
- Any change to the domain model will also trigger an update to the table. For example, If you change the name or type of a field, or add another field to the model, then all these changes will be reflected in the mapped table as well.

Spring Boot_REST demo

- Creating the Note model
- Note model has following fields :
 - id: Primary Key with Auto Increment.
 - title: The title of the Note. (NOT NULL field)
 - content: Note's content. (NOT NULL field)
 - createdAt: Time at which the Note was created.
 - updatedAt: Time at which the Note was updated.

Spring Boot_REST demo

Create Note model class:

```
@Entity
@Table(name = "notes")
@EntityListeners(AuditingEntityListener.class)
@JsonIgnoreProperties(value = {"createdAt", "updatedAt"},
    allowGetters = true)
public class Note implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @NotBlank
    private String title;

    @NotBlank
    private String content;
```

Spring Boot_REST demo

```
@Column(nullable = false, updatable = false)
@Temporal(TemporalType.TIMESTAMP)
@CreatedDate
private Date createdAt;
```

```
@Column(nullable = false)
@Temporal(TemporalType.TIMESTAMP)
@LastModifiedDate
private Date updatedAt;
```

```
// Getters and Setters ... (Omitted here)
```

```
}
```

Spring Boot_REST demo

Enable JPA Auditing

- In our Note model we have annotated createdAt and updatedAt fields with @CreatedDate and @LastModifiedDate annotations respectively.
- these fields should automatically get populated whenever we create or update an entity.

To achieve this, we need to do two things -

- **1. Add Spring Data JPA's AuditingEntityListener to the domain model.**

We have already done this in our Note model with the annotation @EntityListeners(AuditingEntityListener.class).

- **2. Enable JPA Auditing in the main application.**

Open EasyNotesApplication.java and add @EnableJpaAuditing annotation.

Spring Boot_REST demo

Creating Note Repository:

- Spring Data JPA comes with a JpaRepository interface which defines methods for all the CRUD operations on the entity,
 - a default implementation of JpaRepository called **SimpleJpaRepository**.
-
- First, Create a new package called repository
 - create an interface called NoteRepository and extend it from JpaRepository

Spring Boot_REST demo

```
package com.example.easynotes.repository;

import com.example.easynotes.model.Note;
import org.springframework.data.jpa.repository.JpaRepository;
@Repository
public interface NoteRepository extends
JpaRepository<Note, Long> {

}
```

Spring Boot_REST demo

- `@Repository` annotation: This tells Spring to bootstrap the repository during component scan.
- We are now be able to use `JpaRepository`'s methods like `save()`, `findOne()`, `findAll()`, `count()`, `delete()` etc.
- We don't need to implement these methods. They are already implemented by Spring Data JPA's `SimpleJpaRepository`.
- This implementation is plugged in by Spring automatically at runtime.

Spring Boot_REST demo

- Creating Custom Business Exception

@ResponseStatus(value = HttpStatus.NOT_FOUND)

```
public class ResourceNotFoundException extends RuntimeException {
```

```
    private String resourceName;
```

```
    private String fieldName;
```

```
    private Object fieldValue;
```

```
    public ResourceNotFoundException( String resourceName, String  
        fieldName, Object fieldValue) {
```

```
        super(String.format("%s not found with %s : '%s'", resourceName, fieldName,  
            fieldValue));
```

```
        this.resourceName = resourceName;
```

```
        this.fieldName = fieldName;
```

```
        this.fieldValue = fieldValue;
```

```
    }
```

Spring Boot_REST demo

```
public String getResourceName() {  
    return resourceName;  
}
```

```
public String getFieldName() {  
    return fieldName;  
}
```

```
public Object getFieldValue() {  
    return fieldValue;  
}  
}
```

Spring Boot_REST demo

Creating note controller:

The Final Step - now create the REST APIs for

creating, retrieving, updating and deleting a Note.

```
@RestController
```

```
@RequestMapping("/api")
```

```
public class NoteController {
```

```
    @Autowired
```

```
    NoteRepository noteRepository;
```

```
    // Get All Notes    // Create a new Note    // Get a Single Note
```

```
    // Update a Note    // Delete a Note
```

```
}
```

Spring Boot_REST demo

- ***@RestController*** : a combination of Spring's ***@Controller*** and ***@ResponseBody*** annotations.

@Controller annotation is used to define a controller

@ResponseBody annotation is used to indicate that the return value of a method should be used as the response body of the request

@RequestMapping("/api") declares that the url for all the apis in this controller will start with /api.

Spring Boot_REST demo

- Controller Methods:

```
// Get All Notes
```

```
@GetMapping("/notes")
```

```
public List<Note> getAllNotes() {  
    return noteRepository.findAll();  
}
```

Spring Boot_REST demo

- Create a new Note (POST /api/notes)

// Create a new Note

@PostMapping("/notes")

```
public Note createNote(@Valid @RequestBody Note note) {  
    return noteRepository.save(note);  
}
```

Spring Boot_REST demo

- Get a Single Note (Get /api/notes/{noteId})

// Get a Single Note

```
@GetMapping("/notes/{id}")
```

```
public Note getNoteById(@PathVariable(value = "id")  
Long noteId) {
```

```
    return noteRepository.findById(noteId)
```

```
        .orElseThrow(() -> new
```

```
ResourceNotFoundException("Note", "id", noteId));
```

```
}
```

Spring Boot_REST demo

- **Update a Note (PUT /api/notes/{noteId})**

```
// Update a Note
@PutMapping("/notes/{id}")
public Note updateNote(@PathVariable(value = "id") Long noteId,
                      @Valid @RequestBody Note noteDetails) {

    Note note = noteRepository.findById(noteId)
        .orElseThrow(() -> new ResourceNotFoundException("Note", "id", noteId));

    note.setTitle(noteDetails.getTitle());
    note.setContent(noteDetails.getContent());

    Note updatedNote = noteRepository.save(note);
    return updatedNote;
}
```

Spring Boot_REST demo

- **Delete a Note (DELETE /api/notes/{noteId})**

```
// Delete a Note
@DeleteMapping("/notes/{id}")
public ResponseEntity<?> deleteNote(@PathVariable(value = "id")
Long noteId) {
    Note note = noteRepository.findById(noteId)
        .orElseThrow(() -> new ResourceNotFoundException("Note",
"id", noteId));

    noteRepository.delete(note);

    return ResponseEntity.ok().build();
}
```

Spring Boot_REST demo

Running the Application

Let's now run the app and test the apis.

Just go to the root directory of the application and type the following command to run it -

```
$ mvn spring-boot:run
```

The application will start at Spring Boot's default tomcat port 8080.