Spring Boot CRUD Application with Thymeleaf

Last updated: May 11, 2024

Written by: Alejandro Ugarte
Reviewed by: Emily Cheyne

Spring Boot Spring MVC Thymeleaf

1. Overview

The implementation of DAO layers that provide CRUD functionality on JPA entities can be a repetitive, time-consuming task that we want to avoid in most cases.

Luckily, Spring Boot makes it easy to create CRUD applications through a layer of standard JPA-based CRUD repositories.

In this tutorial, we'll learn how to develop a CRUD web application with Spring Boot and Thymeleaf.

Further reading:

Spring Request Parameters with Thymeleaf

Learn how to use request parameters with Spring and Thymeleaf.

Read more →

Changing the Thymeleaf Template Directory in Spring Boot

Learn about Thymeleaf template locations.

Read more →

2. The Maven Dependencies

In this case, we'll rely on spring-boot-starter-parent for simple dependency management, versioning and plugin configuration.

As a result, we won't need to specify the versions of the project dependencies in our *pom.xml* file, except for overriding the Java version

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
</parent>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-thymeleaf</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
    </dependency>
</dependencies>
```

3. The Domain Layer

With all the project dependencies already in place, let's now implement a naive *domain* layer.

For simplicity's sake, this layer will include one single class that will be responsible for modeling *User* entities:

```
@Entity
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long id;

    @NotBlank(message = "Name is mandatory")
    private String name;

    @NotBlank(message = "Email is mandatory")
    private String email;

    // standard constructors / setters / getters / toString
}
```

Let's keep in mind that we've annotated the class with the *@Entity* annotation. **Therefore, the JPA implementation, which is Hibernate, in this case, will be able to perform CRUD operations on the domain entities.** For an introductory guide to Hibernate, visit our tutorial on Hibernate 5 with Spring.

In addition, we've constrained the *name* and *email* fields with the *@NotBlank* constraint. This implies that we can use Hibernate Validator for validating the constrained fields before persisting or updating an entity in the database.

For the basics on this, check out our associated tutorial on Bean Validation.

4. The Repository Layer

At this point, our sample web application does nothing. But that's about to change.

Spring Data JPA allows us to implement JPA-based repositories (a fancy name for the DAO pattern implementation) with minimal fuss.

Spring Data JPA is a key component of Spring Boot's *spring-boot-starter-data-jpa* that makes it easy to add CRUD implementation through a powerful *layer of abstraction* placed on top of a JPA *implementation*. This abstraction layer allows us to access the persistence layer without having to provide our own DAO implementations from scratch.

To provide our application with basic CRUD functionality on *User* objects, we just need to extend the *CrudRepository* interface:

```
@Repository
public interface UserRepository extends CrudRepository<User, Long> {}
```

And that's it! By extending the *CrudRepository* interface, Spring Data JPA will provide implementations for the repository's CRUD methods for us.

5. The Controller Layer

Thanks to the layer of abstraction that *spring-boot-starter-data-jpa* places on top of the underlying JPA implementation, we can easily add some CRUD functionality to our web application through a basic web tier.

In our case, a single controller class will suffice for handling GET and POST HTTP requests and then mapping them to calls to our *UserRepository* implementation.

The controller class relies on some of Spring MVC's key features. For a detailed guide on Spring MVC, check out our Spring MVC tutorial.

Let's start with the controller's *showSignUpForm()* and *addUser()* methods.

The former will display the user *signup* form, while the latter will persist a new entity in the database after validating the constrained fields.

If the entity doesn't pass the validation, the signup form will be redisplayed.

Otherwise, once the entity has been saved, the list of persisted entities will be updated in the corresponding view:

```
@Controller
public class UserController {

    @GetMapping("/signup")
    public String showSignUpForm(User user) {
        return "add-user";
    }

    @PostMapping("/adduser")
    public String addUser(@Valid User user, BindingResult result, Model model) {
        if (result.hasErrors()) {
            return "add-user";
        }

        userRepository.save(user);
        return "redirect:/index";
    }

    // additional CRUD methods
}
```

We'll also need a mapping for the */index* URL:

```
@GetMapping("/index")
public String showUserList(Model model) {
    model.addAttribute("users", userRepository.findAll());
    return "index";
}
```

Within the *UserController*, we will also have the *showUpdateForm()* method, which is responsible for fetching the *User* entity that matches the supplied *id* from the database.

If the entity exists, it will be passed on as a model attribute to the update form view.

So, the form can be populated with the values of the *name* and *email* fields:

```
@GetMapping("/edit/{id}")
public String showUpdateForm(@PathVariable("id") long id, Model model) {
    User user = userRepository.findById(id)
      .orElseThrow(() -> new IllegalArgumentException("Invalid user Id:" + id));

    model.addAttribute("user", user);
    return "update-user";
}
```

Finally, we have the *updateUser()* and *deleteUser()* methods within the *UserController* class.

The first one will persist the updated entity in the database, while the last one will remove the given entity.

In either case, the list of persisted entities will be updated accordingly:

```
@PostMapping("/update/{id}")
public String updateUser(@PathVariable("id") long id, @Valid User user,
  BindingResult result, Model model) {
    if (result.hasErrors()) {
        user.setId(id);
        return "update-user";
    }

    userRepository.save(user);
    return "redirect:/index";
}

@GetMapping("/delete/{id}")
public String deleteUser(@PathVariable("id") long id, Model model) {
    User user = userRepository.findById(id)
      .orElseThrow(() -> new IllegalArgumentException("Invalid user Id:" + id));
    userRepository.delete(user);
    return "redirect:/index";
}
```

6. The View Layer

At this point, we've implemented a functional controller class that performs CRUD operations on *User* entities. **Even so, there's still a missing component in this schema: the *view* layer.**

Under the *src/main/resources/templates* folder, we need to create the HTML templates required for displaying the signup form and the update form as well as rendering the list of persisted *User* entities.

As stated in the introduction, we'll use *Thymeleaf* as the underlying template engine for parsing the template files.

Here's the relevant section of the *add-user.html* file:

```
<form action="#" th:action="@{/adduser}" th:object="${user}" method="post">
    <label for="name">Name:</label>
    <input type="text" th:field="*{name}" id="name" placeholder="Name">
    <span th:if="${#fields.hasErrors('name')}" th:errors="*{name}"></span>
    <label for="email">Email:</label>
    <input type="text" th:field="*{email}" id="email" placeholder="Email">
    <span th:if="${#fields.hasErrors('email')}" th:errors="*{email}"></span>
    <input type="submit" value="Add User">
</form>
```

Notice how we've used the *@{/adduser}* URL expression to specify the form's *action* attribute and the *${}* variable expressions for embedding *dynamic* content in the template, such as the values of the *name* and *email* fields and the *post*-validation errors.

Similar to *add-user.html*, here's how the *update-user.html* template looks:

```
<form action="#"
  th:action="@{/update/{id}(id=${user.id})}"
  th:object="${user}"
  method="post">
    <label for="name">Name:</label>
    <input type="text" th:field="*{name}" id="name" placeholder="Name">
    <span th:if="${#fields.hasErrors('name')}" th:errors="*{name}"></span>
    <label for="email">Email:</label>
    <input type="text" th:field="*{email}" id="email" placeholder="Email">
    <span th:if="${#fields.hasErrors('email')}" th:errors="*{email}"></span>
    <input type="submit" value="Update User">
</form>
```

Finally, we have the *index.html* file that displays the list of persisted entities along with the links for editing and removing existing ones:

```
<div th:switch="${users}">
    <h2 th:case="null">No users yet!</h2>
    <div th:case="*">
        <h2>Users</h2>
        <table>
            <thead>
                <tr>
                    <th>Name</th>
                    <th>Email</th>
                    <th colspan="2">Action</th>
                </tr>
            </thead>
            <tbody>
                <tr th:each="user : ${users}">
                    <td th:text="${user.name}"></td>
                    <td th:text="${user.email}"></td>
                    <td><a th:href="@{/edit/{id}(id=${user.id})}">Edit</a></td>
                    <td><a th:href="@{/delete/{id}(id=${user.id})}">Delete</a></td>
                </tr>
            </tbody>
        </table>
    </div>
    <p><a href="/signup">Add a new user</a></p>
</div>
```

For simplicity's sake, the templates look rather skeletal and only provide the required functionality without adding unnecessary cosmetics.

To give the templates an improved, eye-catching look without spending too much time on HTML/CSS, we can easily use a free Twitter Bootstrap UI kit, such as Shards.

7. Running the Application

Finally, let's define the application's entry point.

Like most Spring Boot applications, we can do this with a plain old *main()* method:

```
@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

Now let's hit "Run" in our IDE and then open up our browser and point it to *http://localhost:8080*.

If the build has successfully completed, **we should see a basic CRUD user dashboard with links for adding new entities and for editing and removing existing ones.**

8. Conclusion

In this article, we learned how to build a basic CRUD web application with Spring Boot and Thymeleaf.

As usual, all the code samples shown in the article are available over on GitHub.