Spring Boot | Spring MVC

Thymeleaf

Spring Boot CRUD Application with Thymeleaf

Written by: Alejandro Ugarte



Last updated: May 11, 2024



The implementation of DAO layers that provide CRUD functionality on JPA entities can be a repetitive, time-

1. Overview

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
                                          Changing the Thymeleaf Template Directory in
Thymeleaf
                                          Spring Boot
Learn how to use request parameters with Spring
                                          Learn about Thymeleaf template locations.
and Thymeleaf.
                                          Read more →
Read more →
```

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

<dependencies>

2. The Maven Dependencies

overriding the Java version:

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

<parent> <groupId>org.springframework.boot <artifactId>spring-boot-starter-parent</artifactId> </parent>

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

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

```
@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.
```

Hibernate Validator for validating the constrained fields before persisting or updating an entity in the database.

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*

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

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.

5. The Controller Layer

@Controller

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:

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";
```

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.

return "update-user";

if (result.hasErrors()) {

userRepository.save(user);

return "redirect:/index";

return "update-user";

user.setId(id);

```
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);
```

Within the UserController, we will also have the showUpdateForm() method, which is responsible for fetching

@PostMapping("/update/{id}") public String updateUser(@PathVariable("id") long id, @Valid User user, BindingResult result, Model model) {

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

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

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

```
@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.
```

 <label for="email">Email</label> <input type="text" th:field="*{email}" id="email" placeholder="Email"> <input type="submit" value="Add User">

</form>

<form action="#"

method="post">

removing existing ones:

unnecessary cosmetics.

<thead>

use a free Twitter Bootstrap UI kit, such as Shards.

public static void main(String[] args) {

entities and for editing and removing existing ones.

SpringApplication.run(Application.class, args);

</form>

th:object="\${user}"

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

th:action="@{/update/{id}(id=\${user.id})}"

<label for="name">Name</label>

<label for="email">Email</label>

<input type="submit" value="Update User">

<label for="name">Name</label>

<form action="#" th:action="@{/adduser}" th:object="\${user}" method="post">

<input type="text" th:field="*{name}" id="name" placeholder="Name">

and the post-validation errors. Similar to add-user.html, here's how the update-user.html template looks:

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

expressions for embedding dynamic content in the template, such as the values of the *name* and *email* fields

Notice how we've used the @{/adduser} URL expression to specify the form's action attribute and the \${} variable

<div th:switch="\${users}"> <h2 th:case="null">No users yet!</h2> <div th:case="*"> <h2>Users</h2>

<input type="text" th:field="*{name}" id="name" placeholder="Name">

<input type="text" th:field="*{email}" id="email" placeholder="Email">


```
Name
            Email
           Edit
            Delete
         </thead>
       <a th:href="@{/edit/{id}(id=${user.id})}">Edit</a>
         <a th:href="@{/delete/{id}(id=${user.id})}">Delete</a>
     </div>
 <a href="/signup">Add a new user</a>
</div>
```

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

To give the templates an improved, eye-catching look without spending too much time on HTML/CSS, we can easily

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 {

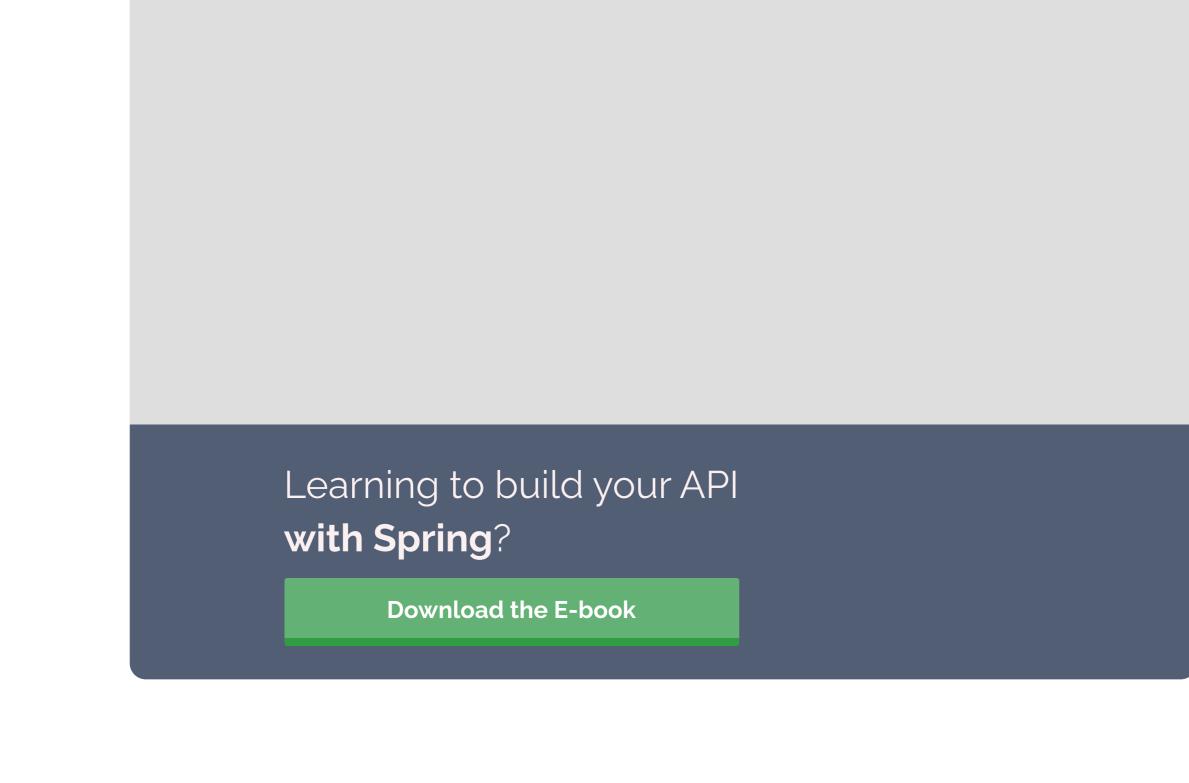
8. Conclusion

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

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

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

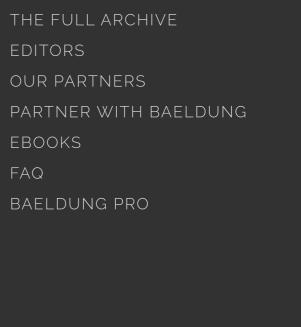
If the build has successfully compiled, we should see a basic CRUD user dashboard with links for adding new



In addition, we've constrained the *name* and *email* fields with the @NotBlank constraint. This implies that we can use 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 functionality through a powerful layer of abstraction placed on top of a JPA implementation. This abstraction layer interface: @Repository public interface UserRepository extends CrudRepository<User, Long> {}

SERIES

TERMS OF SERVICE | PRIVACY POLICY | COMPANY INFO | CONTACT



ABOUT

ABOUT BAELDUNG