

Exception Handling

Exception Handling

- Let's do some exception handling in our ProductInventory App.
- If you observe the “edit-product.html” it sends request to “/save” on form submission.
- It (/save) is the same mapping which we are also using to create our products.
- Can you think of a way to create products from our “edit-products” page ?



Exception Handling

- There is a loophole in our system where user would be able to create more products from “edit-product” page.
- Let’s see how.



Exception Handling

- Click on “Edit” button of the product you wish to edit.
- ‘edit-product’ page appears with filled in form details.
- Now, if you try to modify the ‘id’ field, you will notice that it’s not editable.
- So far so good.



Exception Handling

- Now right click on the 'id' field and click inspect.
- You should see something like below.

```
▼ <tr>
  <td>Product ID:</td>
  ▼ <td>
    <input type="text" readonly="readonly" id="id" name="id" value="1">
  </td>
</tr>
▶ <tr>...</tr>
```

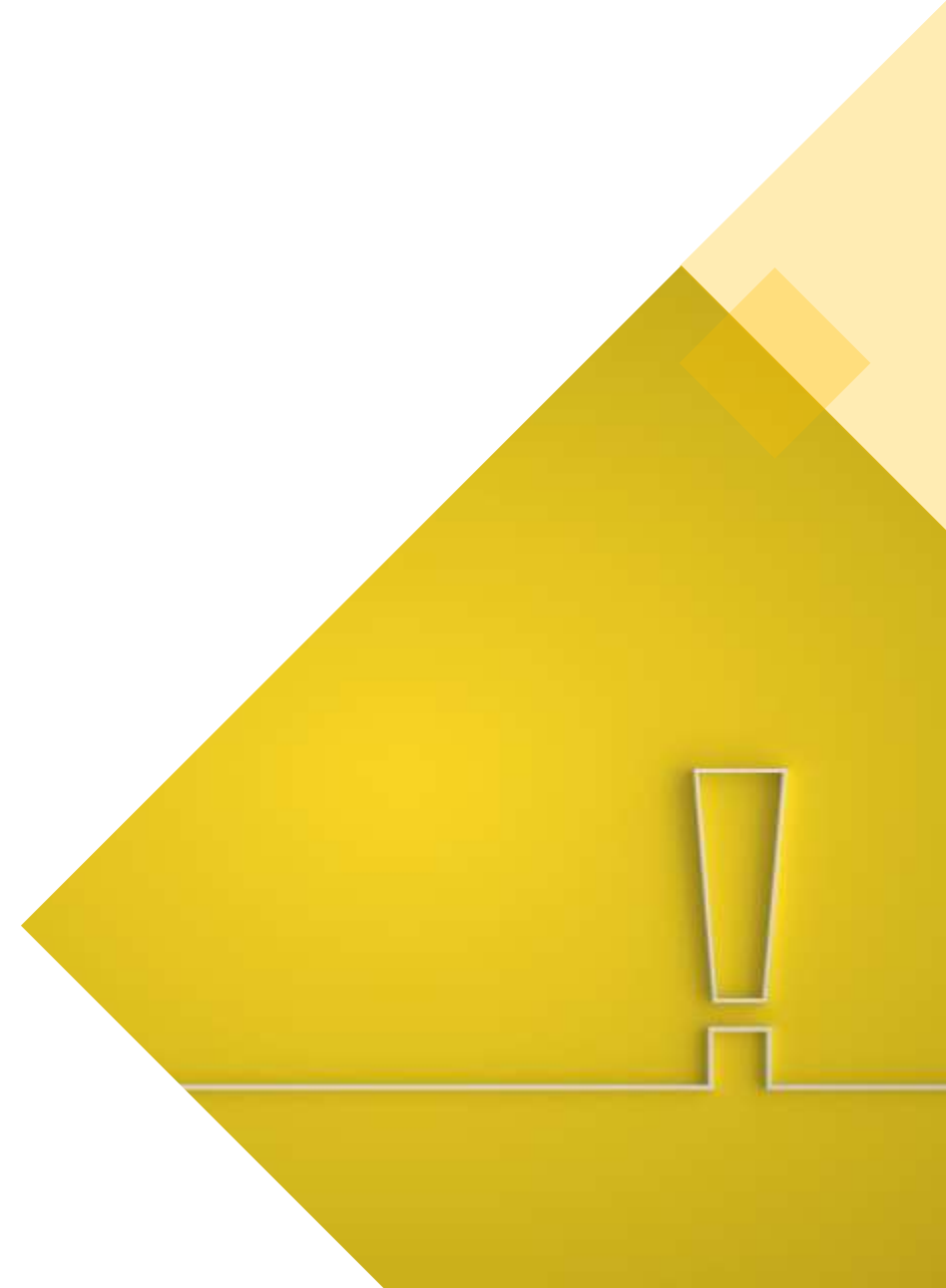
Exception Handling

- Delete ***readonly="readonly"*** .
- Now try to edit the 'id' field.
- You will notice that you are able to change the value of 'id' field now.
- Imagine this can enable a user to create a new product or edit an unintended product.



Exception Handling


- Let's create another mapping to edit our products instead of using the one for create products.
- Create new mapping “/edit” with *PostMapping* in Controller.



Exception Handling

```
@PostMapping("/edit")
public ModelAndView editProduct(@Valid @ModelAttribute("product") Product product, BindingResult result) {

    ModelAndView modelAndView = new ModelAndView();
    if (product.getId() == null || productRepository.findById(product.getId()).isEmpty()) {
        throw new NoSuchElementException("Product with id: " + product.getId() + " does not exist.");
    }
    if (result.hasErrors()) {
        modelAndView.setViewName("edit-product");
        return modelAndView;
    }
    productRepository.save(product);
    return modelAndView;
}
```





Exception Handling

- In the above code. We are throwing exception in two cases:
 - If the passed id is *null*
 - OR if there is no product in our database corresponding to passed id.
- We are also passing a message to be shown to the user.



Exception Handling

- Let's create a `ExceptionHandler` which gets called when an exception is thrown in controller.

```
@ExceptionHandler({Exception.class})
public ModelAndView handleException(Exception ex) {
    ModelAndView modelAndView = new ModelAndView("error");
    modelAndView.addObject("exception", ex.getMessage());
    return modelAndView;
}
```

- Here we are setting view as *error.html*.
- Also, we adding the *exception* object to the model.

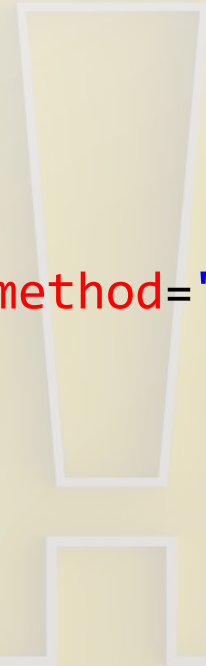


Exception Handling

- Two more things to do now.
 - Create error.html page
 - Change mapping from “/save” to “/edit” in edit-product.html

- Updated edit-product.html :-

```
<form action="#" th:action="@{/edit}" th:object="${product}" method="post">
```



Exception Handling – Error.html

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
  <!-- Bootstrap CSS file -->
  <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.5.0/css/bootstrap.min.css">
</head>
<body>
  <div class="container-fluid" th:text="{exception}">
  </div>
  <!-- JS files: jQuery first, then Popper.js, then Bootstrap JS -->
  <script src="https://code.jquery.com/jquery-3.5.1.min.js"></script>
  <script src="https://cdn.jsdelivr.net/npm/popper.js@1.16.0/dist/umd/popper.min.js"></script>
  <script src="https://stackpath.bootstrapcdn.com/bootstrap/4.5.0/js/bootstrap.min.js"></script>
</body>
</html>
```



Exception Handling

- Now restart the App.
- Try to edit the 'id' field and hit 'save'.
- You should be thrown to error page.



LAB - Exception Handling

- Try to find out areas in our App where there could be unexpected behavior.
- Add appropriate Exception Handling at identified area.
- Note:- Whatever mappings are in the Controller can be accessed through tools like Postman Or RestClient etc. And the user may try to exploit your Apis (controller mappings) in all possible ways. So, while thinking of loopholes, do not restrict your thinking to UI specific loopholes.



Spring Security



Spring Security

- Let's implement spring security in our App.
- To-dos :
 - Registration Page For new Users.
 - Login Page.
 - Logout Feature.

Spring Security

- Add spring-security-starter dependency in pom.xml

```
<dependency>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-starter-security</artifactId>  
</dependency>
```

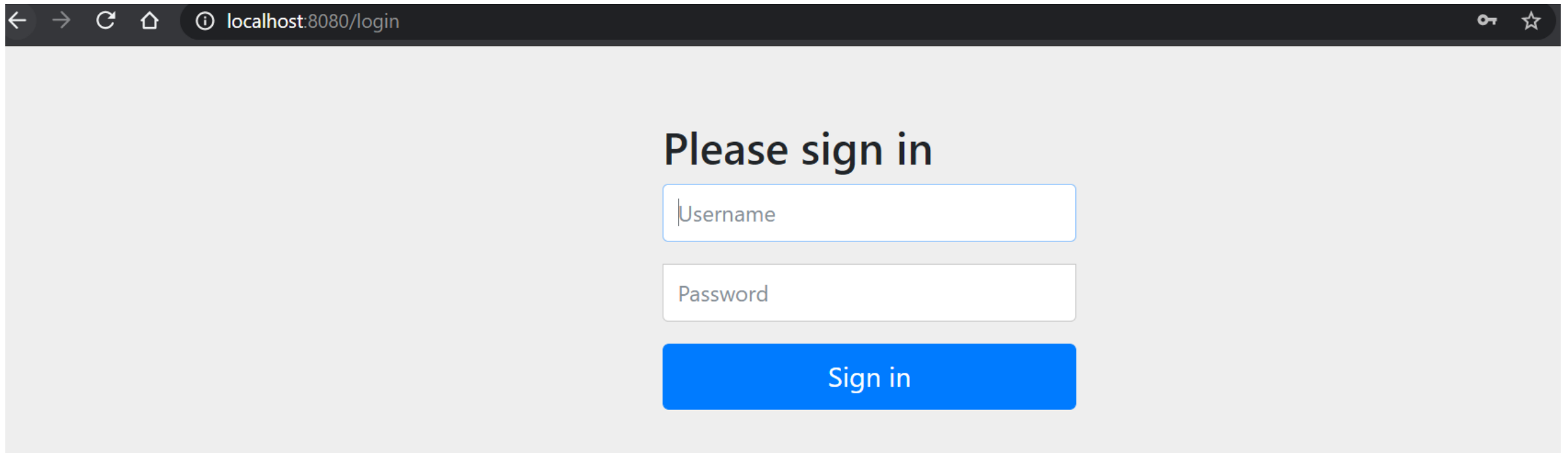
- Now restart the App and open the app in browser (localhost:8080).

Spring Security

- Instead of showing the product list, app is redirected to 'login' page.
- Try some other Urls which were accessible earlier.
 - E.g localhost:8080/edit/1
- All the urls you try to access will redirect you to login page.
- Try to login with some random username and password.
- You will see Bad Credentials Message.
- Great!! We have secured our App.
- But how do we get inside the App?

Spring Security

The Login Page on your screen is default login page provided by Spring.



A screenshot of a web browser window displaying the default Spring Security login page. The browser's address bar shows the URL `localhost:8080/login`. The page has a light gray background and features the heading "Please sign in" in a large, bold, black font. Below the heading are two input fields: the first is labeled "Username" and the second is labeled "Password". Both fields are white with a thin blue border. Below these fields is a prominent blue button with the text "Sign in" in white. The browser's navigation bar at the top includes back, forward, and refresh icons, along with a lock icon and a star icon on the right.

Please sign in

Username

Password

Sign in

Spring Security

- We will need some valid users to be able to login.
- So, basically, we would need a table to store our users. Let's call it 'user' table.
- For simplicity let's consider our User has following fields:
 - Id
 - Username
 - Password
- Also, we would need a Java representation of this user. And we learned in our previous lectures that we can achieve this by creating Entities.

Spring Security

- Script to create user table.

```
CREATE TABLE `user` (  
  `id` int(11) NOT NULL AUTO_INCREMENT,  
  `username` varchar(45) NOT NULL,  
  `password` VARCHAR(200) NOT NULL,  
  PRIMARY KEY (`id`)  
) ENGINE=InnoDB AUTO_INCREMENT=25 DEFAULT  
CHARSET=UTF8
```

Spring Security


- Let's create User entity in 'entities' package.

```
@Entity
@Table(name = "user")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Length(min = 5, message = "*Your user name must have at least 5 characters")
    @NotEmpty(message = "*Please provide a user name")
    private String username;

    @Length(min = 5, message = "*Your password must have at least 5 characters")
    @NotEmpty(message = "*Please provide your password")
    private String password;

    @Transient //This means, we don't want to save this in db
    private String passwordConfirm;
    //getters setters
}
```



Oh, wait!
@GeneratedValue

- We talked about that there are four strategies for @GeneratedValue, but what are they?
 - AUTO
 - IDENTITY
 - SEQUENCE
 - TABLE



@GeneratedValue - AUTO

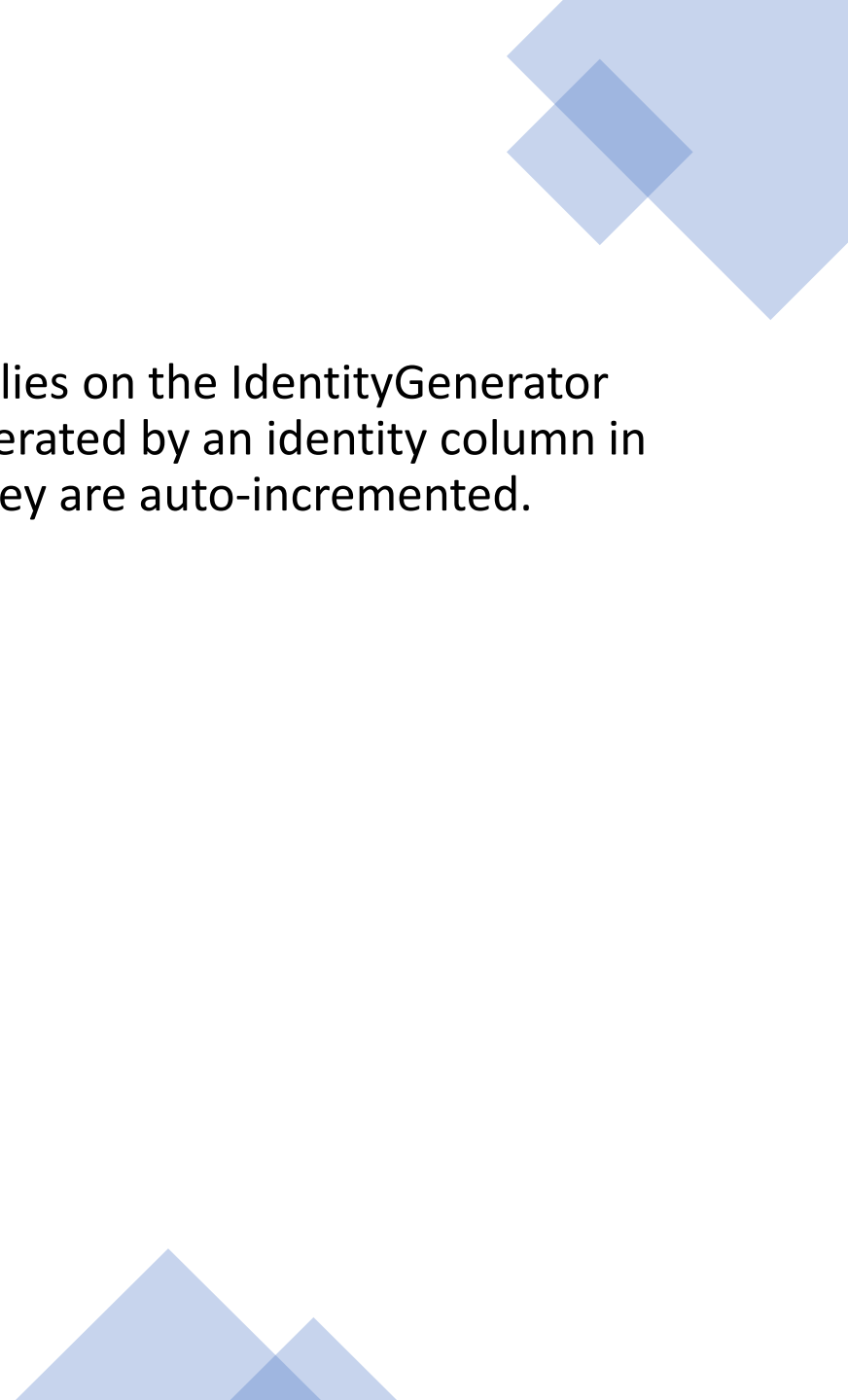
If we're using the default generation type, the persistence provider will determine values based on the type of the primary key attribute.

This type can be numerical or UUID.

- UUID is a unique string provided by the database.
 - Example: 8dd5f315-9788-4d00-87bb-10eed9eff566



@GeneratedValue - IDENTITY

- This type of generation relies on the IdentityGenerator which expects values generated by an identity column in the database, meaning they are auto-incremented.
- 

@GeneratedValue - SEQUENCE

- To use a sequence-based id, Hibernate provides the `SequenceStyleGenerator` class.
- This generator uses sequences if they're supported by our database, and switches to table generation if they aren't.
- To customize the sequence name, we can use the `@GenericGenerator` annotation with `SequenceStyleGenerator` strategy
- SEQUENCE is the generation type recommended by the Hibernate documentation.

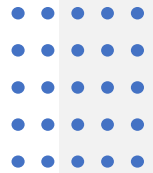
```
@Entity
public class User {
    @Id
    @GeneratedValue(generator = "sequence-generator")
    @GenericGenerator(
        name = "sequence-generator",
        strategy = "org.hibernate.id.enhanced.SequenceStyleGenerator",
        parameters = {
            @Parameter(name = "sequence_name", value = "user_sequence"),
            @Parameter(name = "initial_value", value = "4"),
            @Parameter(name = "increment_size", value = "1")
        }
    )
    private long userId;

    // ...
}
```

@GeneratedValue

- TABLE

- The TableGenerator uses an underlying database table that holds segments of identifier generation values.



@GeneratedValue – own implementation

If we don't want to use any of the out-of-the-box strategies, we can define our custom generator by implementing the IdentifierGenerator interface.

Read more at
<https://www.baeldung.com/hibernate-identifiers>

Now, let's get back to our lab.

Spring Security

- Now, for database operations, let's create a UserRepository in "repositories" package.
- Create a method to find the user by name. Make sure method name is same.
- These are query methods. Spring translates these methods into queries behind the scenes.

Spring Security

- UserRepository.java

```
public interface UserRepository extends JpaRepository<User, Long> {  
    User findByUsername(String username);  
}
```

Spring Security

- Great! Now, to implement login/authentication with Spring Security, we need to implement *org.springframework.security.core.userdetails.UserDetailsService* interface.
- This interface has a method 'loadUserByUsername(String username)'. It basically checks whether user with 'username' exists in database or not.
 - If exists it returns the user details.
 - If not, it throws an exception.

Spring Security

- Let's create a package 'service' and create UserDetailsServiceImpl.java

```
@Service
public class UserDetailsServiceImpl implements UserDetailsService{
    @Autowired
    private UserRepository userRepository;

    @Override
    @Transactional()
    public UserDetails loadUserByUsername(String username) {
        User user = userRepository.findByUsername(username);
        if (user == null) throw new UsernameNotFoundException(username);
        return new org.springframework.security.core.userdetails.User(user.getUsername(), user.getPassword(), Collections.emptyList());
    }
}
```


Spring Security

- Let's create another service 'SecurityService.java' in the same package.
- Let's put some helper methods in this service, which we will need when defining controller mappings.

Spring Security

- SecurityService.java

```
@Service
public class SecurityService {

    @Autowired
    private AuthenticationManager authenticationManager;

    @Autowired
    private UserDetailsService userDetailsService;

    private static final Logger logger = LoggerFactory.getLogger(SecurityService.class);

    public boolean isAuthenticated() {
        Authentication authentication = SecurityContextHolder.getContext().getAuthentication();
        if (authentication == null || AnonymousAuthenticationToken.class.
            isAssignableFrom(authentication.getClass())) {
            return false;
        }
        return authentication.isAuthenticated();
    }
}
```

Spring Security

- SecurityService.java (continued...)

```
public void autoLogin(String username, String password) {  
    UserDetails userDetails = userDetailsService.loadUserByUsername(username);  
    UsernamePasswordAuthenticationToken usernamePasswordAuthenticationToken = new UsernamePass  
wordAuthenticationToken(userDetails, password, userDetails.getAuthorities());  
  
    authenticationManager.authenticate(usernamePasswordAuthenticationToken);  
  
    if (usernamePasswordAuthenticationToken.isAuthenticated()) {  
        SecurityContextHolder.getContext().setAuthentication(usernamePasswordAuthenticationTok  
en);  
        logger.debug(String.format("Auto login %s successfully!", username));  
    }  
}
```

Spring Security

- Let's define one more service, UserService.java, which is basically wrapper for UserRepository.
- We don't want the passwords of users to be saved as plain text. (Why?)
- We will encrypt the password and then save to database.



Spring Security

```
@Service
public class UserService {
    @Autowired
    private UserRepository userRepository;
    @Autowired
    private BCryptPasswordEncoder bCryptPasswordEncoder;

    public void save(User user) {
        user.setPassword(bCryptPasswordEncoder.encode(user.getPassword()));
        userRepository.save(user);
    }

    public User findByUsername(String username) {
        return userRepository.findByUsername(username);
    }
}
```

Spring Security

- Let's define our controller mappings.
- We would need following mappings:
 - /registration – GET : For registration form
 - /registration – POST : For saving user
 - /login – GET : For login form
 - /login – POST : For submitting credentials.

Spring Security

- /registration – GET

```
@GetMapping("/registration")
public String registration(Model model) {
    if (securityService.isAuthenticated()) {
        return "redirect:/";
    }

    model.addAttribute("userForm", new User());

    return "registration";
}
```

Spring Security

- In the previous slide we are using the helper method which we created in `SecurityService.java`.
- We can use this service in our controller class by Autowiring it.

```
@Autowired
```

```
private SecurityService securityService;
```


Spring Security

- We are doing two things here:
 - We are checking if the user who is trying to open the registration page is already logged in then redirect him/her to home page.
 - If the user is not logged in, we show him/her the registration form.

Spring Security

- /registration – POST

```
@PostMapping("/registration")
public String registration(@Valid @ModelAttribute("userForm") User userForm, BindingResult bindingResult) {
    User userExists = userService.findByUsername(userForm.getUsername());
    if (userExists != null) {
        bindingResult
            .rejectValue("username", "error.user",
                "There is already a user registered with the username provided");
    }
    if (bindingResult.hasErrors()) {
        return "registration";
    }
    userService.save(userForm);
    securityService.autoLogin(userForm.getUsername(), userForm.getPasswordConfirm());
    return "redirect:/";
}
```

Spring Security

- Here we are first finding the *user* in db with *username* submitted by from registration form.
- For this we are using *UserService*. **Autowire** it to use it in controller.
- If we find the *user* then we show error on registration form.
- If *user* does not exist, we create a new user.
- Then we do auto login of the user and redirect him/her to home page.
- For auto login we have used helper method from `SecurityService.java`

Spring Security

- Let's create controller mapping for /login – GET method.

```
@GetMapping("/login")
public String login(Model model, String error, String logout) {
    if (securityService.isAuthenticated()) {
        return "redirect:/";
    }
    if (error != null)
        model.addAttribute("error", "Your username and password is invalid.");


    if (logout != null)
        model.addAttribute("message", "You have been logged out successfully.");
};
return "login";
}
```

Spring Security

- Here we again check if the user is already authenticated, if yes, we redirect her to home page.
- Otherwise, we show her login form.
- Also, on the basis of query parameters (error and logout) we show relevant messages to user.
- ***We don't define /login POST controller, it is provided by Spring Security***



Spring Security

- Great!
 - Let's create our views.
 - Create 'login.html' and 'registration.html' in templates folder.
- 

Spring Security – login.html

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
  <head>...</head>

  <body>
    <div class="container">
      <form class="form-signin" method="POST" th:action="@{/login}">
        <h2 class="form-heading">Log in</h2>
        <div class="form-group">
          <span th:text="${message}"></span>
          <input name="username" type="text" class="form-control" placeholder="Username"
            autofocus="true"/>

          <input name="password" type="password" class="form-control" placeholder="Password"/>
          <span class="has-error" th:text="${error}"></span>

          <button class="btn btn-lg btn-primary btn-block" type="submit">Log In</button>
          <h4 class="text-center"><a href="/registration">Create an account</a></h4>
        </div>
      </form>
    </div>
  </body>
</html>
```

Spring Security – registration.html

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
  <head>...</head>
  <body>
    <div class="container">
      <form method="POST" class="form-signin" th:object="${userForm}" th:action="@{/registration}">
        <h2 class="form-signin-heading">Create your account</h2>
        <div class="form-group">
          <input type="text" th:field="*{username}" class="form-control" placeholder="Username"
            autofocus="true">
          <span class="has-error" th:if="${#fields.hasErrors('username')}" th:errors="*{username}"></span>
        </div>
        <div class="form-group">
          <input type="password" th:field="*{password}" class="form-control" placeholder="Password">
          <span class="has-error" th:if="${#fields.hasErrors('password')}" th:errors="*{password}"></span>
        </div>
        <div class="form-group">
          <input type="password" th:field="*{passwordConfirm}" class="form-control"
            placeholder="Confirm your password">
          <span class="has-error" th:if="${#fields.hasErrors('passwordConfirm')}" th:errors="*{passwordConfirm}"></span>
        </div>
        <button class="btn btn-lg btn-primary btn-block" type="submit">Submit</button>
      </form>
    </div>
  </body>
</html>
```




Spring Security

- Let's create a *main.css* under 'resources/static/css' and include it in 'login.html' and 'registration.html'.

Spring Security – main.css

```
body {
  padding-top: 40px;
  padding-bottom: 40px;
  background-color: #eee;
}

.form-signin {
  max-width: 330px;
  padding: 15px;
  margin: 0 auto;
}

.form-signin .form-signin-heading,
.form-signin .checkbox {
  margin-bottom: 10px;
}

.form-signin .checkbox {
  font-weight: normal;
}

.form-signin .form-control {
  position: relative;
  height: auto;
  -webkit-box-sizing: border-box;
  -moz-box-sizing: border-box;
  box-sizing: border-box;
  padding: 10px;
  font-size: 16px;
}

.form-signin .form-control:focus {
  z-index: 2;
}

.form-signin input {
  margin-top: 10px;
  border-bottom-right-radius: 0;
  border-bottom-left-radius: 0;
}

.form-signin button {
  margin-top: 10px;
}

.has-error {
  color: red;
}
```



Spring Security

- Let's create a config file.
- In this file we will be able to tell spring which all mappings we want to secure and which all we want unsecured.
- We will also define that we want to use form login.
- We will also define which implementation of *UserDetailsService* spring should use.



Spring Security

- We will also define which encoder to use.
- Let's create *WebSecurityConfig.java* under 'src/main/java/<groupid>'.
- It should be at same level where *SpringBootApplication.java* is present.


Spring Security

```
@Configuration
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
    @Autowired
    private UserDetailsService userDetailsService;
    @Bean
    public BCryptPasswordEncoder bCryptPasswordEncoder() {
        return new BCryptPasswordEncoder();
    }
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests()
                .antMatchers("/css/**", "/js/**", "/registration").permitAll()
                .anyRequest().authenticated()
                .and().csrf().disable()
            .formLogin()
```

```
                .loginPage("/login")
                .permitAll()
                .and()
            .logout()
                .permitAll();
    }
    @Bean
    public AuthenticationManager customAuthenticationManager() throws Exception {
        return authenticationManager();
    }
    @Autowired
    public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
        auth.userDetailsService(userDetailsService).passwordEncoder(bCryptPasswordEncoder());
    }
}
```

The logo for Spring Security, featuring a large orange circle with the text "Spring Security" in white. A small blue circle is positioned at the bottom-left edge of the orange circle.

Spring Security

- Great! We're all set now!
 - Restart the App.
 - Go to browser and hit localhost:8080
- 
- A decorative yellow dashed arc in the top right corner of the slide.

Spring Security

- You will see a login page.
- Since we don't have any user to login, let's create a user by clicking on 'Create an account'.

Log in

Log In

Create an account

Spring Security

- You will see a registration page.
- You can also verify the validations we have applied to form fields.

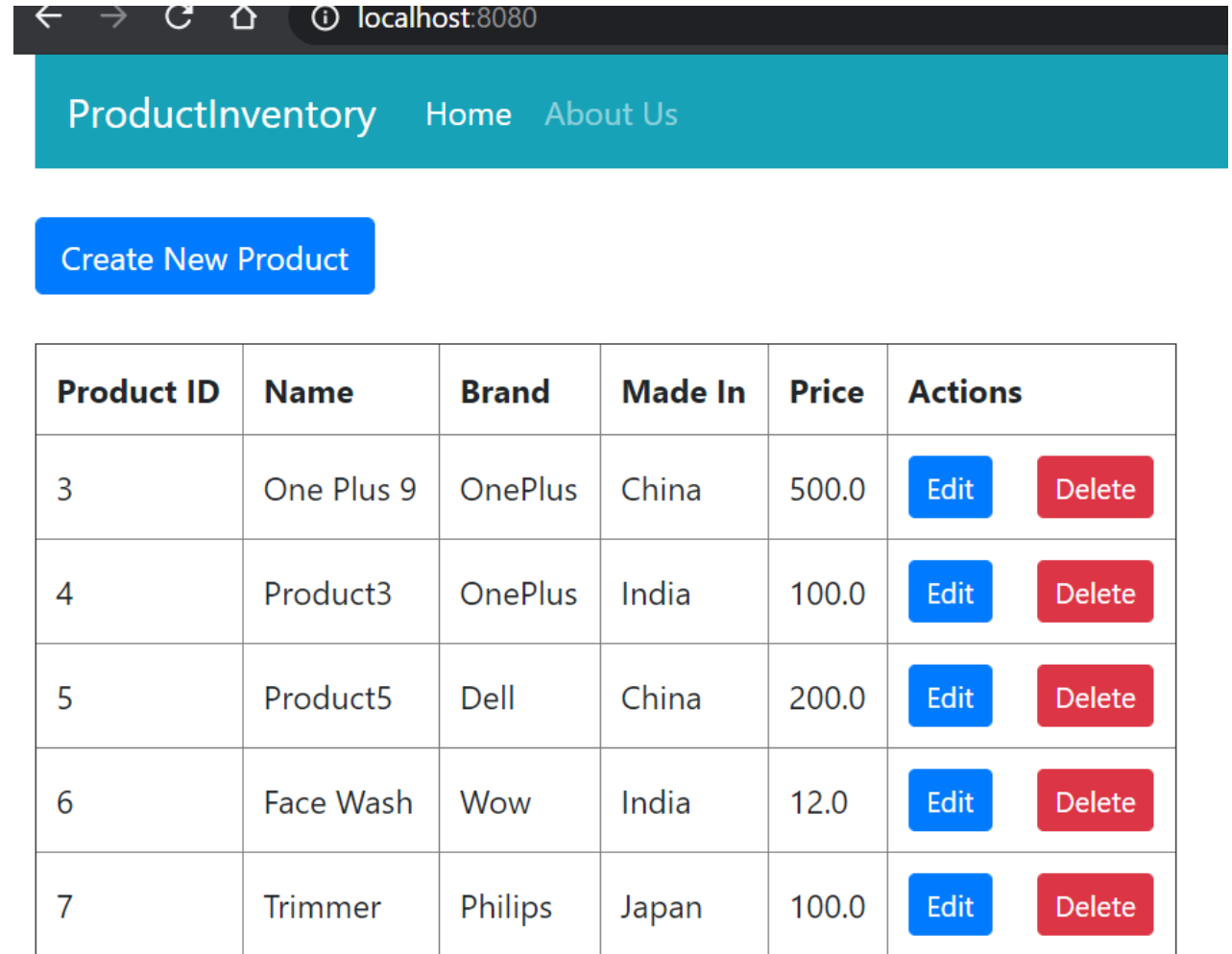
Create your account

*Your user name must have at least 5 characters

*Your password must have at least 5 characters

Spring Security

- Fill up the registration form and click on Submit.
- On successful registration, you will be auto logged in to the application.
- You should see the product list page.



The screenshot shows a web browser at localhost:8080 displaying the 'ProductInventory' application. The page has a teal header with 'ProductInventory', 'Home', and 'About Us' links. Below the header is a blue button labeled 'Create New Product'. The main content area features a table with product details and actions.

Product ID	Name	Brand	Made In	Price	Actions	
3	One Plus 9	OnePlus	China	500.0	Edit	Delete
4	Product3	OnePlus	India	100.0	Edit	Delete
5	Product5	Dell	China	200.0	Edit	Delete
6	Face Wash	Wow	India	12.0	Edit	Delete
7	Trimmer	Philips	Japan	100.0	Edit	Delete



Spring Security

- Go to database and verify whether a user is created or not.
- Also observe the password field in database.
- It should be encrypted.



Spring Security

- Great!
- Let's check some more flows.
- Let's close the browser and open it again.
- Let's go to registration link
<http://localhost:8080/registration>
- Since you are logged in, you should be redirected to product list page.
- Try same for login link.
<http://localhost:8080/login>



Spring Security

- Let's do a logout now.
- We have already configured logout in our security config.
- Let's Add Logout button in our View.

Spring Security

- In our products.html lets replace following div tag.

```
<div class="navbar-nav ml-auto">  
  <a href="#" class="nav-item nav-link">Login</a>  
</div>
```

- Replace above with:

```
<div class="navbar-nav ml-auto">  
  <a onclick="document.forms['logoutform'].submit()" class="nav-item nav-link">Logout</a>  
  <form id="logoutform" method="POST" action="/logout"></form>  
</div>
```

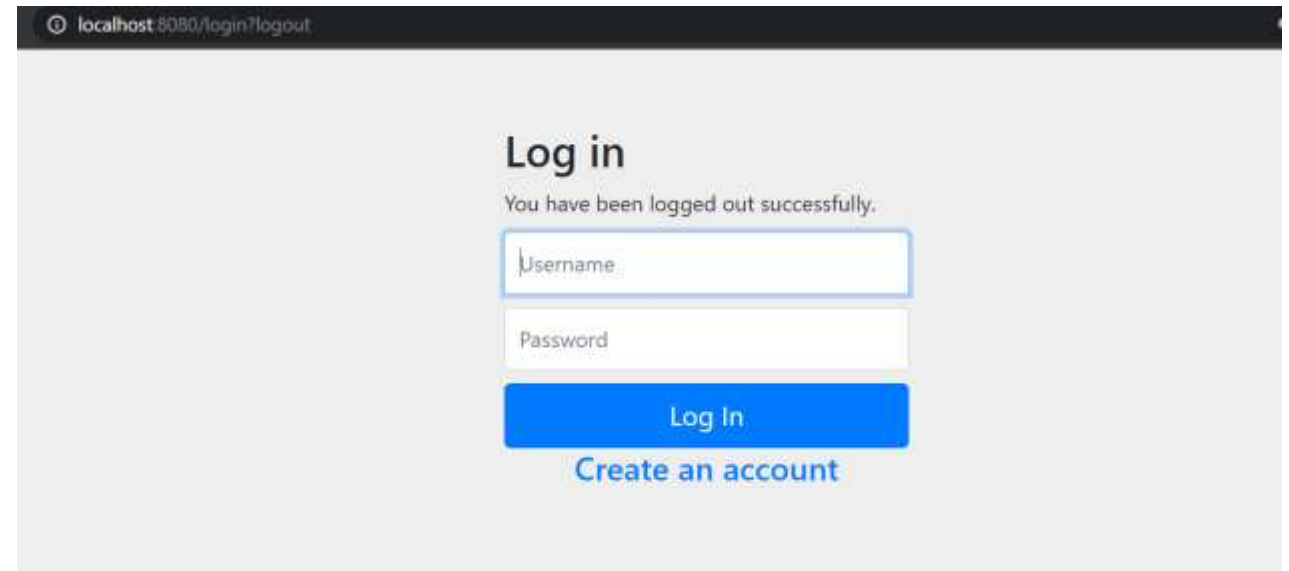


Spring Security

- That's it!
- Let's restart the app, head over to browser and hit localhost:8080.
- You will be shown a login page.
- Let's login with the user we created earlier.
- You should be able to login successfully with correct credentials of course.
- Observe the top right corner of Navbar.
- You should be able to see Logout link.

Spring Security

- Click on Logout. You will be logged out of app.
- Notice the url. It's the same login url with query param logout.
- Now again try to open different urls.
- System should not let you in unless you login again.



localhost:8080/login?logout

Log in

You have been logged out successfully.

Log In
[Create an account](#)



Spring Security

- While working on the web application, we may come into a situation where the same attributes referred to in multiple pages.
- In an online shopping, we may need to refer to a user shopping cart at multiple pages.
- To make sure the availability of the attributes , we need to store/persist this information so as we can pull the same information on the next page.
- In a web application, a good place to store those attributes is in the user's *session*.



Spring Security

- Let's learn how we can save a variable in session and access it across the pages.
- We will use spring's `@SessionAttributes` along with `@ModelAttribute` to achieve this.



Spring Security

- `@SessionAttributes` annotation :
 - It is used on the `@Controller` class level.
 - Its 'value' element is of type `String[]` whose values are the matching names used in `@ModelAttribute` either on method level or on handler's method parameter level.



Spring Security

- `@ModelAttribute` annotation:
 - We will use `@ModelAttribute` annotation to support our `@SessionAttributes` annotation.
 - This annotation binds a method parameter or method return value to a named model attribute, exposed to a web view.
 - When our controller is accessed for the first time, Spring instantiate an instance and place it in the Model.

Spring Security

- Let's create define @sessionattribute and @modelattribute in our App.
- In productController.java define session attributes at class level like below :

```
@Controller
@SessionAttributes("cart")
public class ProductController {
```

Spring Security

- Now define a model attribute in the same class as follows :

```
@ModelAttribute("cart")
public Set<Product> initializeCart() {
    return new HashSet<>();
}
```

Spring Security

- Now Let's add a button in products.html which will allow users to add product to cart.

```
<td>  
    <a class="btn btn-primary btn-sm" th:href="@{'/edit/' + ${product.id}}">Edit</a>  
    &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&~  
    <a class="btn btn-danger btn-sm" th:href="@{'/delete/' + ${product.id}}">Delete</a>  
    ~~~~~~  
    <form style="display: inline;" action="#" th:action="@{'/addToCart/' + ${product.id}}" method="post">  
        <button class="btn btn-info btn-sm" type="submit">Add To Cart</button>  
    </form>  
</td>
```

Spring Security

- Now add a controller mapping for “/addToCart” in ProductController.

```
@PostMapping(value = "/addToCart/{id}")
public String addToCart(@PathVariable(name = "id") long id,
    @ModelAttribute("cart") Set<Product> cart) {
    Product product = productRepository.findById(id).get();
    boolean isItemPresent = cart.stream()
        .anyMatch(item -> item.getId() == product.getId());
    if(!isItemPresent) {
        cart.add(product);
    }
    return "redirect:/";
}
```

Spring Security

- Notice use of `@ModelAttribute("cart") Set<Product> cart` in method parameters.
- Also, we are checking if the product already exist in the cart or not.
- We don't add the product if it's already present.

Spring Security

- Let's see this in action.
- In products.html add following code in nav bar.
- Basically, we are showing number of items in cart.

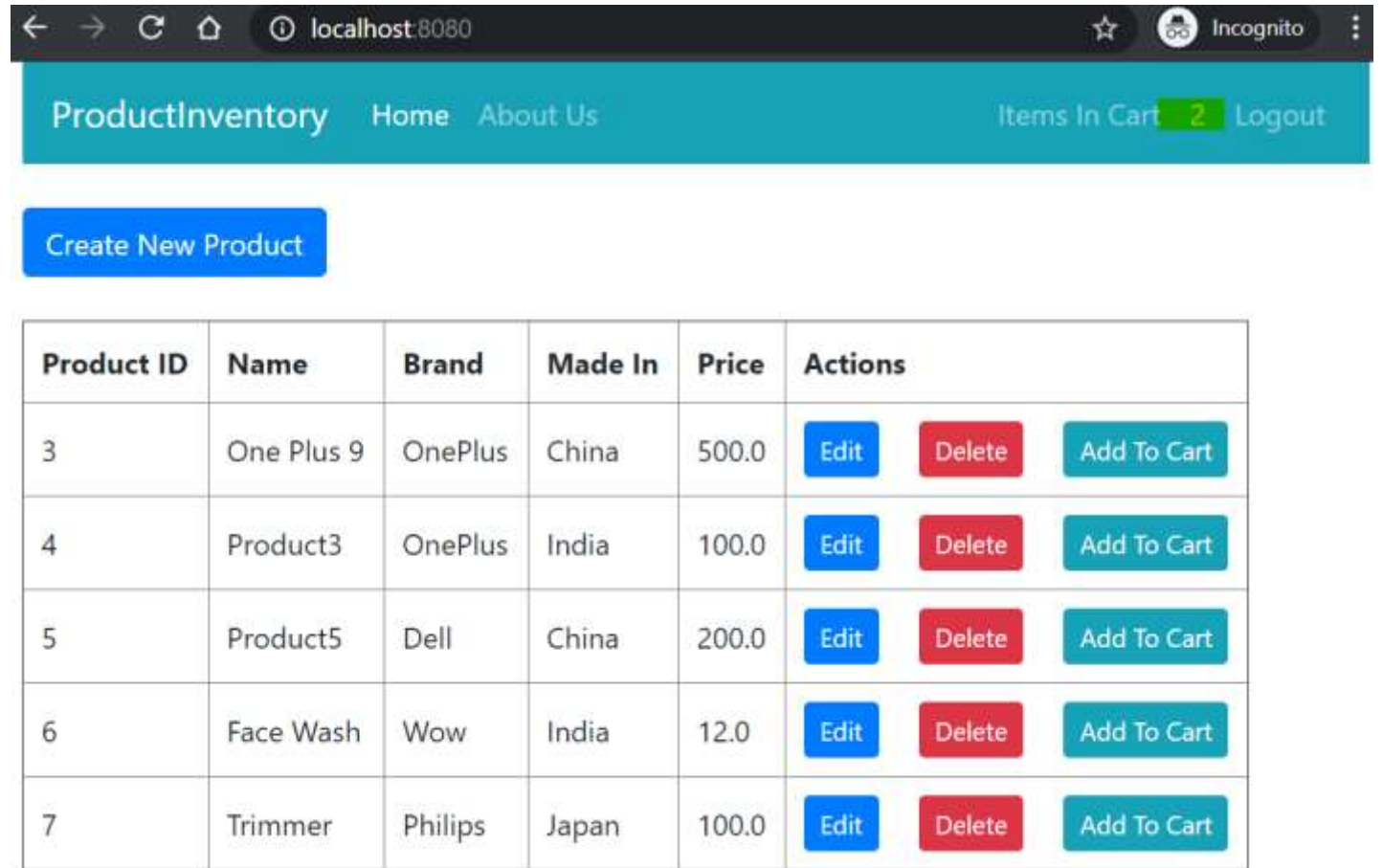
```
<div class="navbar-nav ml-auto">
  <!-- add from here -->
  <span class="nav-item nav-link">Items In Cart</span>
  <span class="nav-item nav-link" th:text="${cart.size()}"></span>
  <!-- to here -->
  <a onclick="document.forms['logoutform'].submit()" class="nav-item nav-link">Logout</a>
  <form id="logoutform" method="POST" action="/logout"></form>
</div>
```



Spring Security

- Now restart the app and open in browser.
- Login using valid credentials.
- On product list page Click on 'Add To Cart' button.
- You should see the count of items should change on top right corner of navbar.

Spring Security



Product ID	Name	Brand	Made In	Price	Actions
3	One Plus 9	OnePlus	China	500.0	Edit Delete Add To Cart
4	Product3	OnePlus	India	100.0	Edit Delete Add To Cart
5	Product5	Dell	China	200.0	Edit Delete Add To Cart
6	Face Wash	Wow	India	12.0	Edit Delete Add To Cart
7	Trimmer	Philips	Japan	100.0	Edit Delete Add To Cart



Spring Security

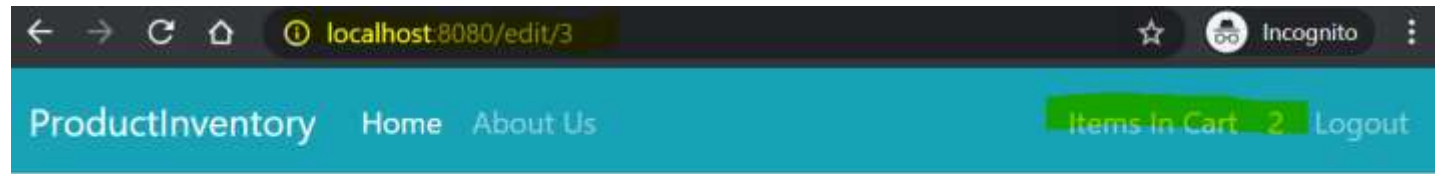
- Great! Number of items gets updated on the products.html page.
- Let's see if we can see the count of cart items on other pages/screens.
- Let's copy the whole Navbar from products.html and paste it to 'new-product.html' and 'edit-product.html'.



Spring Security

- Restart the App and login.
- Add some items to cart.
- Click on Edit button, you should be able to see the same count.
- Try same by clicking 'Create New Product' button.

Spring Security



Edit Product

Product ID:	<input type="text" value="3"/>
Product Name:	<input type="text" value="One Plus 9"/>
Brand:	<input type="text" value="OnePlus"/>
Made In:	<input type="text" value="China"/>
Price:	<input type="text" value="500.0"/>

Is there a way
of getting data
from several
entities in JPA,
like an INNER
JOIN in SQL?

Yes, there is, but it takes some setting up.

Here's a couple of guides:

<https://roytuts.com/spring-boot-data-jpa-left-right-inner-and-cross-join-examples/>

<https://www.baeldung.com/jpa-join-types>

If you're interested, we can try to get through one of the guides together. (I haven't tried any of them beforehand, though.)