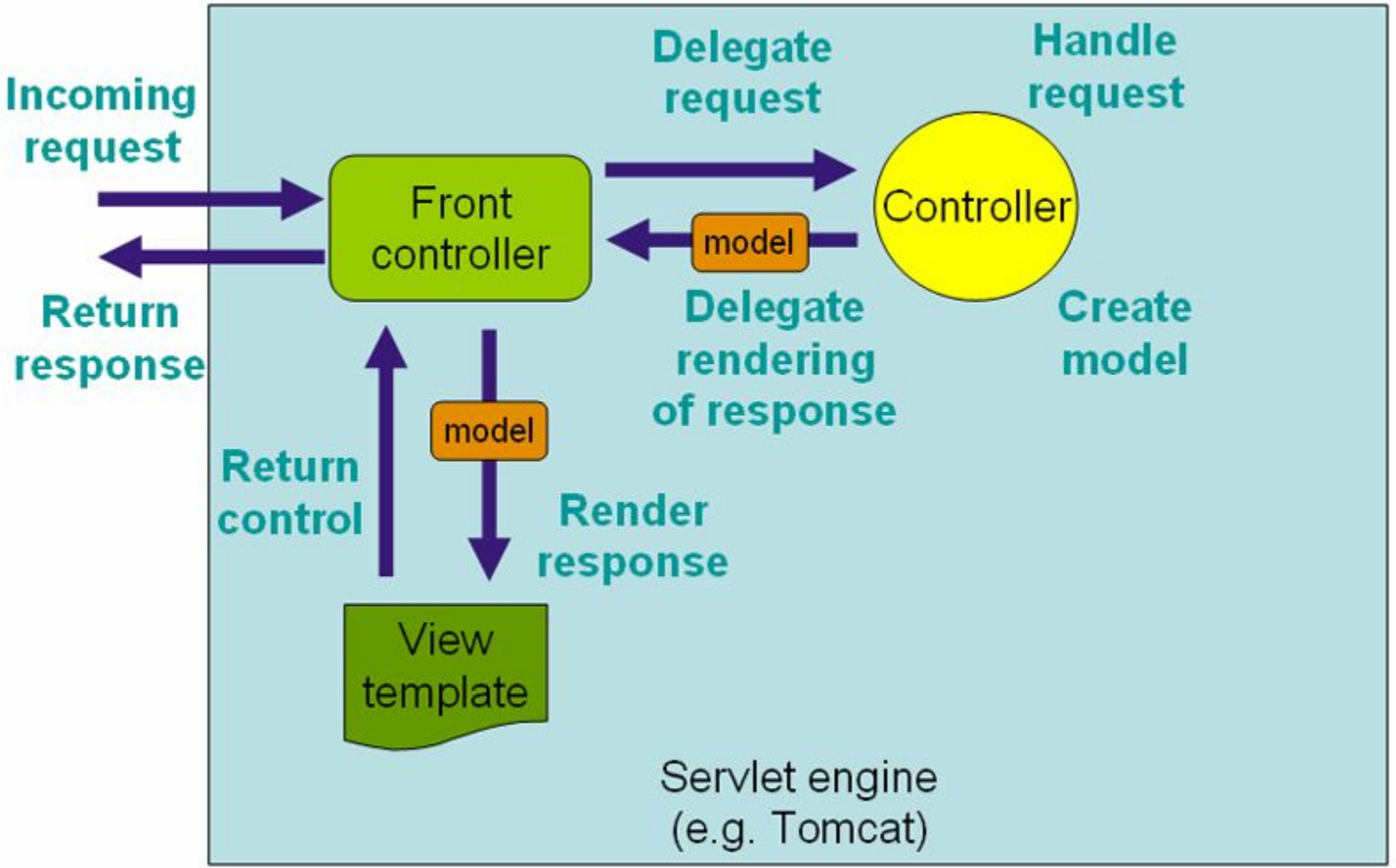
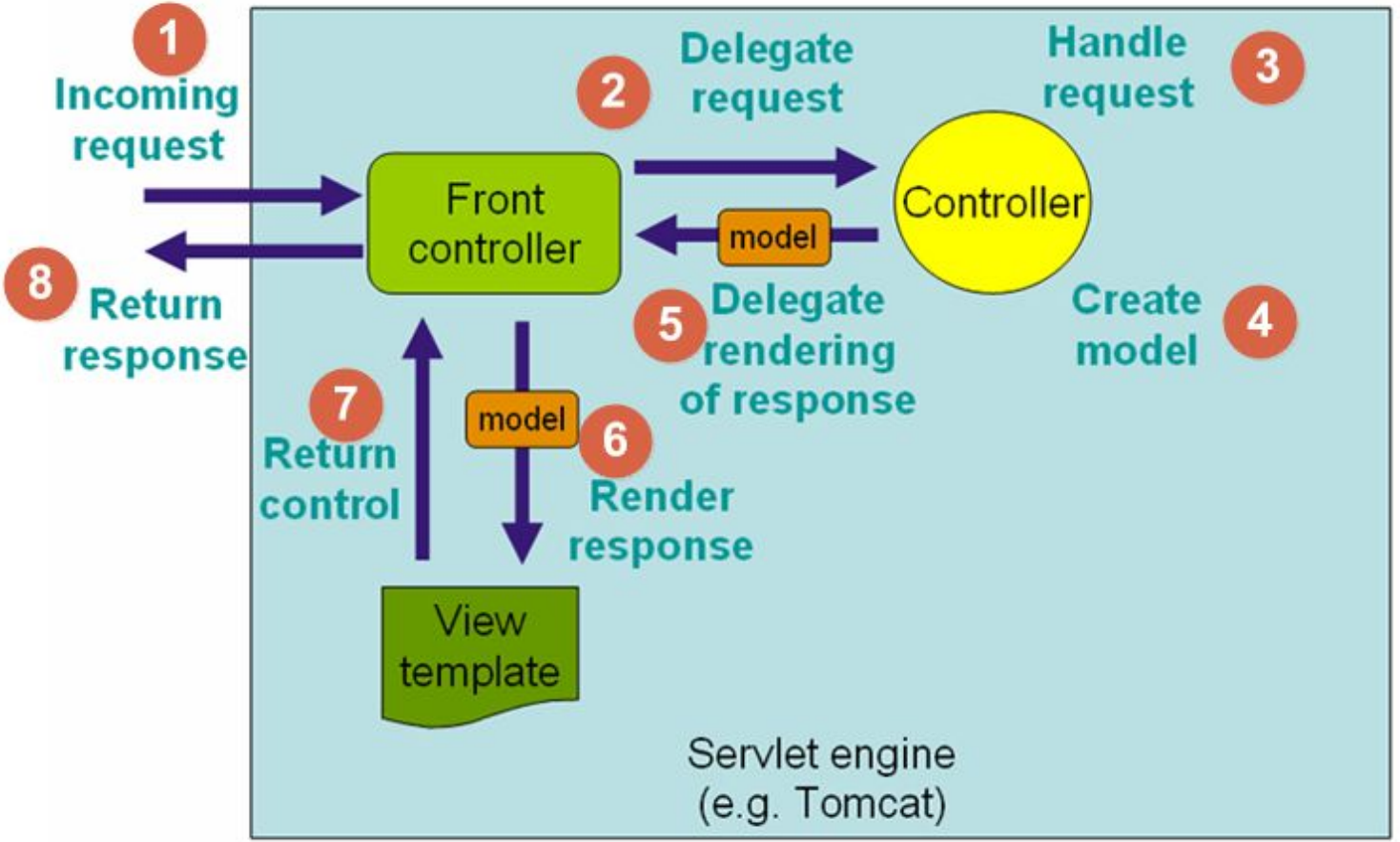
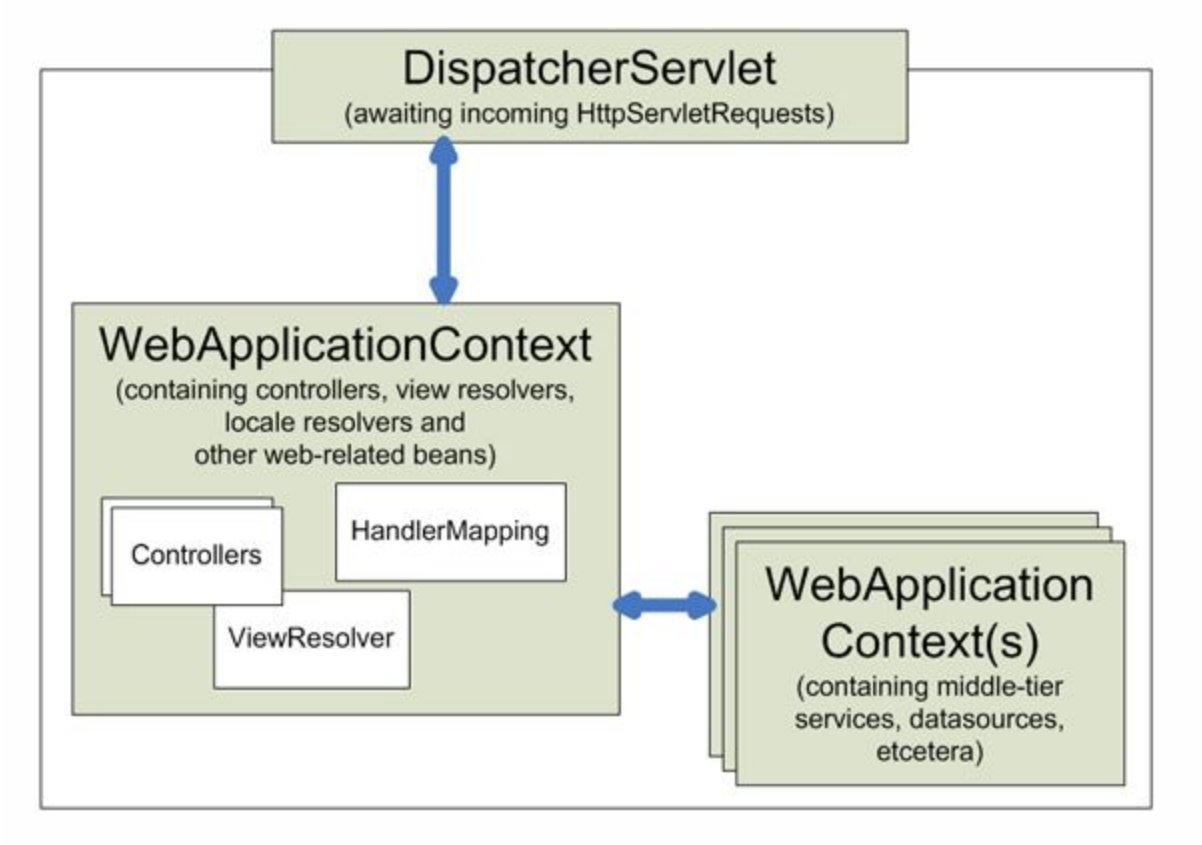
Securing Spring Applications

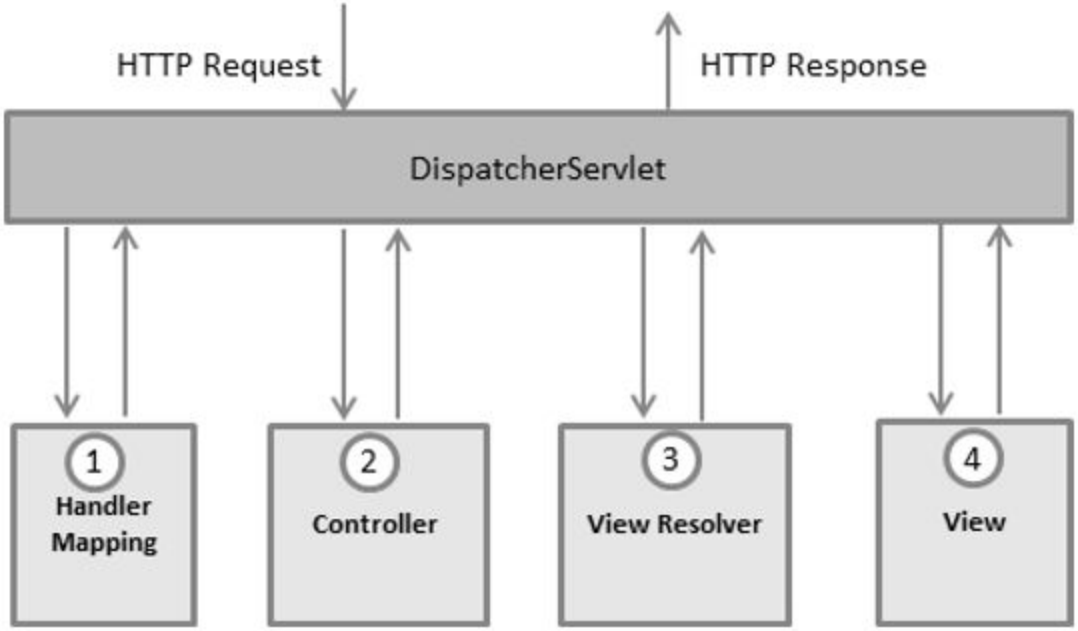
Spring MVC Summary

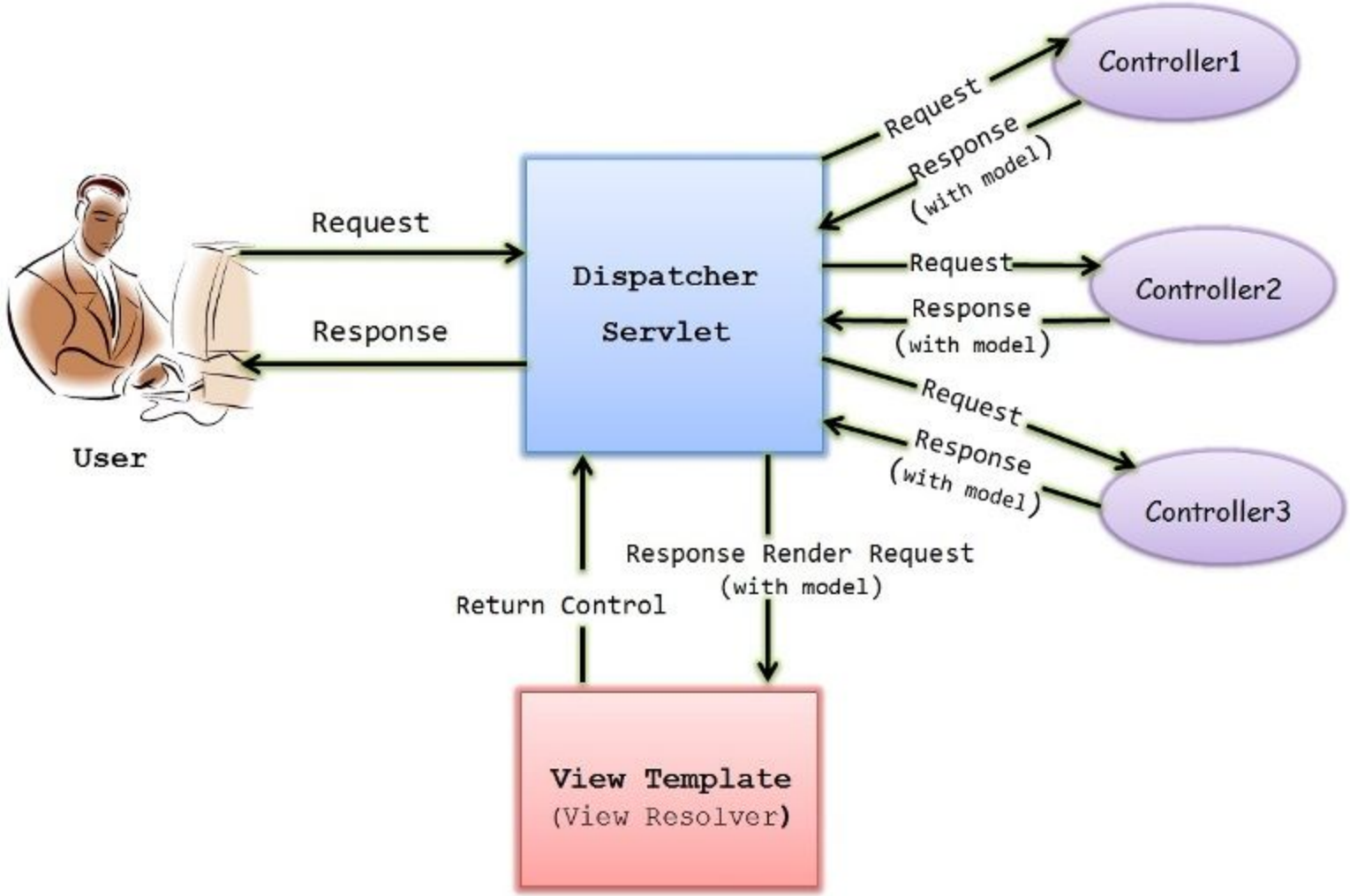
```
@Controller
public class DesignController {

    @GetMapping("/design")
    public String design(Model model) {
        model.addAttribute("newTaco", "Design Taco");
        return "design";
    }

}
```


Incoming request
Return response
Front controller
Delegate request
Handle request
Controller
model
Delegate rendering of response
Create model
model
Return control
Render response
View template
Servlet engine (e.g. Tomcat)


1
Incoming request
2
Delegate request
Handle request
3
Front controller
Controller
model
8
Return response
5
Delegate rendering of response
Create model
4
7
Return control
model
6
Render response
View template
Servlet engine
(e.g. Tomcat)


DispatcherServlet
(awaiting incoming HttpServletRequests)
WebApplicationContext
(containing controllers, view resolvers,
locale resolvers and
other web-related beans)
Controllers
HandlerMapping
ViewResolver
WebApplication
Context(s)
(containing middle-tier
services, datasources,
etcetera)


HTTP Request
HTTP Response
DispatcherServlet
1
Handler Mapping
2
Controller
3
View Resolver
4
View


Controller1
Request
Response
(with model)
Request
Response
Dispatcher
Servlet
Request
Response
(with model)
Controller2
Request
Response
(with model)
Controller3
User
Response Render Request
(with model)
Return Control
View Template
(View Resolver)

Topics

Introduction to Security

Auto Configuring Spring Security

Defining custom user storage

Customizing the login page

Securing against CSRF attacks

The Goal of Security

The goal of software security is to maintain the **confidentiality**, **integrity**, and **availability** of information resources in order **to enable successful business operations**

This goal is accomplished through the implementation of **security controls**

The Goal of Security

Information is probably the most valuable item we now have

Malicious users are looking for ways to steal users' data and identities by sneaking into insecure applications

Security Attack Categories

Spoofing impersonating something or someone else

Tampering modifying something you're not supposed to modify. It can include packets on the wire (or wireless), bits on disk, or the bits in memory

Repudiation claiming you didn't do something

Denial of Service attacks designed to prevent a system from providing service, including by crashing it, making it unusably slow, or filling all its storage

Security Attack Categories

Information Disclosure exposing information to people who are not authorized to see it

Elevation of Privilege when a program or user is technically able to do things that they're not supposed to do

Threat Mitigation Approach

What can you do to prevent these attacks?

Threat Type	Property Violated	Mitigation Approach
Spoofing	Authentication	
Tampering	Integrity	
Repudiation	Non-repudiation	
Information Disclosure	Confidentiality	
Denial of Service	Availability	
Elevation of Privilege	Authorization	

Threat Mitigation Approach

Threat Type	Property Violated	Mitigation Approach
Spoofing	**Authentication**	Passwords, Multi-Factor Authentication, Digital Signature
Tampering	**Integrity**	Permissions/ACLs, Digital Signature
Repudiation	**Non-repudiation**	Secure Logging and Auditing, Digital Signature
Information Disclosure	**Confidentiality**	Encryption, Permissions/ACLs
Denial of Service	**Availability**	Quotas, Permissions/ACLs
Elevation of Privilege	**Authorization**	Permissions/ACLs, Input Validation

Open Web Application Security Project (OWASP)

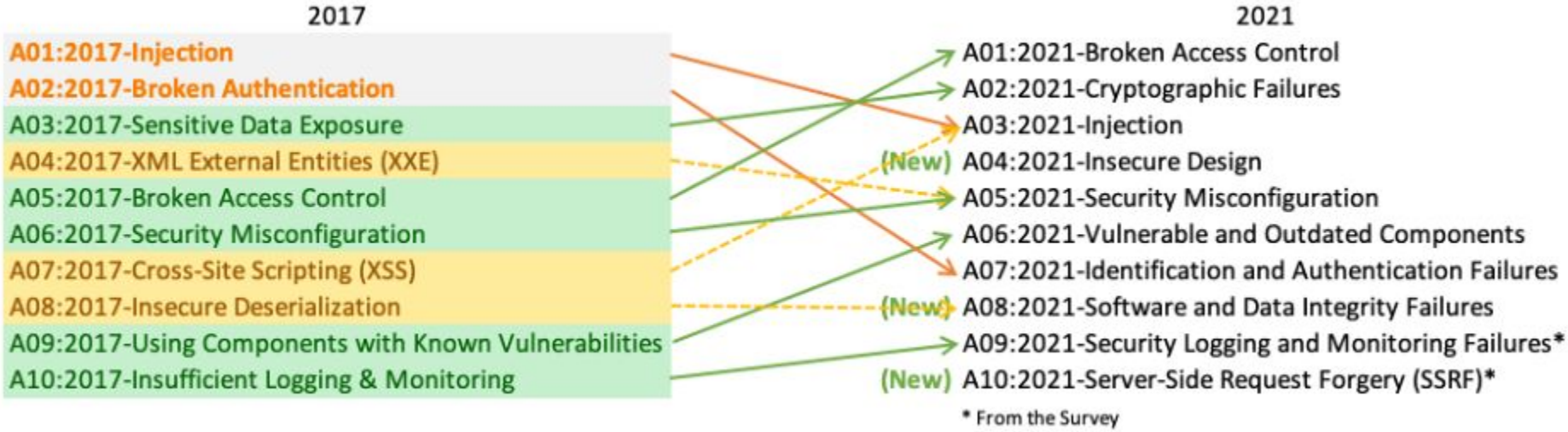
The **Open Web Application Security Project® (OWASP)**

- is a nonprofit foundation that works to improve the security of software
- OWASP Foundation is the source for developers and technologists to secure the web
- OWASP provides
  - tools and resources
  - community and networking
  - education & training

Top 10 Web Application Security Risks

Comparing top 10 during 2017 and 2021

There are new risks in 2021



https://owasp.org/www-project-top-ten/

OWASP Secure Coding Checklist

Input Validation	Data Protection
Output Encoding	Communication Security
Authentication & Password Management	System Configuration
Session Management	Database Security
Access Control	File Management
Cryptographic Practices	Memory Management
Error Handling & Logging	

Enabling Spring Security

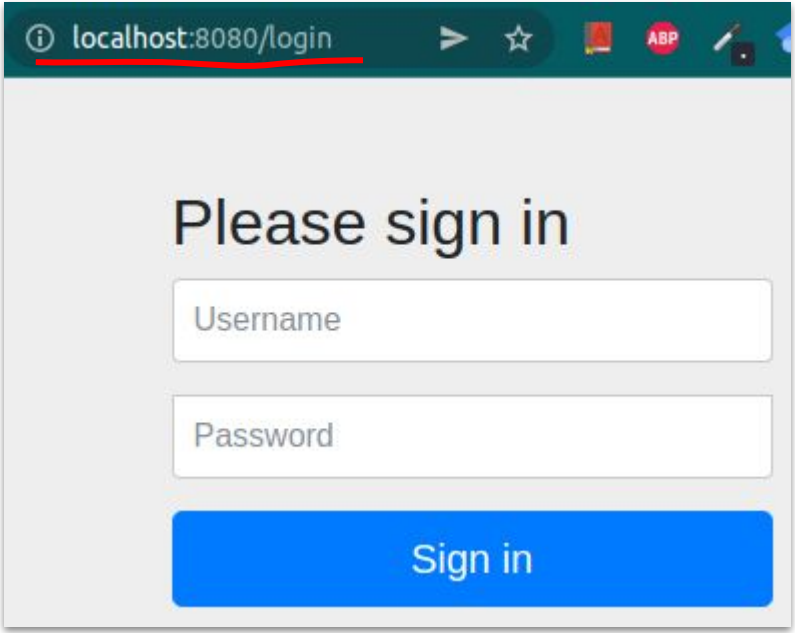
Add the following dependency in your **`pom.xml`** file

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

Enabling Spring Security

If you **(re)start the application** now, autoconfiguration will detect that **Spring Security** is in the classpath and will set up some basic security configuration

If you try to access the application, **it will redirect you to `/login` url** and ask you to provide **`username`** and **`password`** with **default login page** as shown in the right

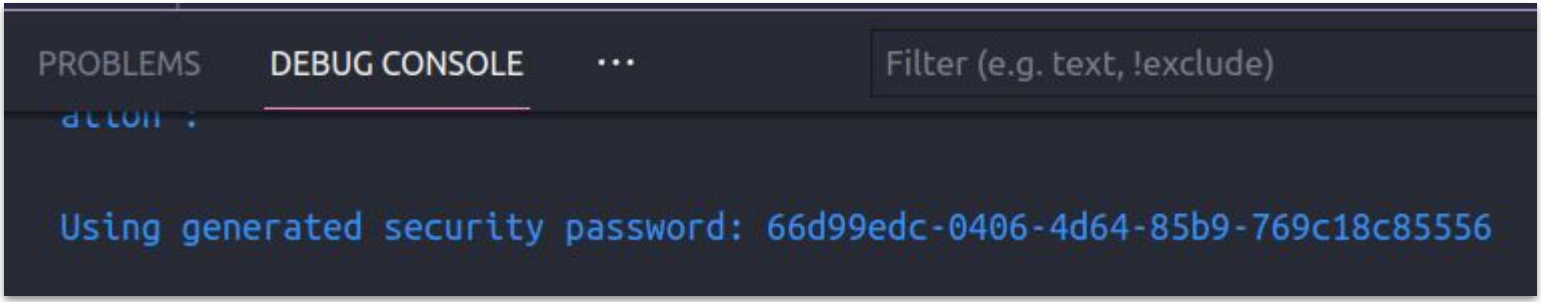


Enabling Spring Security

The default **`username`** is **`user`**

The **`password`** is **randomly generated** and written to the application console

Check your IDE console log to get the password



Enabling Spring Security

By adding the security starter to the project build, you get the following autoconfigured security features

- **All** HTTP request **paths require authentication**
- **No specific roles or authorities** are required
- Authentication is prompted with a **simple login page**
- There's only one user, its `username` is `user`

Securing Taco Cloud

The autoconfigured security features are not adequate for securing Taco Cloud application

Securing Taco Cloud

Below are security requirement of Taco Cloud Application

- provide **custom login page** that is designed to match the website
- **multiple users with** different username should be able to access the system, and **new Taco Cloud customers** should be able to **sign up**
- Apply **different security rules for different request paths**
  - the **homepage** and **registration** pages, for example, **shouldn't require authentication**

Configuring Spring Security

We can secure spring application by configuring **`Authentication`** and **`Authorization`**

Configuring Authentication

To authenticate users, we need to configure

- **Password Encoder**
- **User Store**
- **Configure Security using `HttpSecurity`**

Configuring Password Encoder

Spring Security provides the following password encoders

- **`BCryptPasswordEncoder`** — Applies **`bcrypt`** strong hashing encryption
- **`NoOpPasswordEncoder`** — Applies **`no`** encoding
- **`Pbkdf2PasswordEncoder`** — Applies **`PBKDF2`** encryption
- **`SCryptPasswordEncoder`** — Applies **`scrypt`** hashing encryption
- **`StandardPasswordEncoder`** — Applies **`SHA-256`** hashing encryption

Configuring Password Encoder

To use any of the password encoders, you need to **provide instances** of them as a bean in a **configuration class** (a class annotated with `@Configuration` annotation)

Create a `package` called `security` in the Taco Cloud Application project and then put the code shown in the next slide into a file named `SecurityConfig.java`

Creating a password encoder bean/instance

```
@Configuration
public class SecurityConfig {


    @Bean
    PasswordEncoder bcryptPasswordEncoder() {
        return new BCryptPasswordEncoder();
    }


}
```

Configuring Password Encoder

No matter which password encoder you use, the password in **the database is never decoded**

Instead, the password that the user enters at login is encoded using the same algorithm, and it's then compared with the encoded password in the database

> the comparison is performed in the `PasswordEncoder`'s `matches()` method

Configuring User Store

In order to configure a **user store** for authentication purposes, we'll need to declare a **`UserDetailsService`** bean

The **`UserDetailsService`** interface includes only one method - **`loadUserByUsername()`** that must be implemented

```
public interface UserDetailsService {
   UserDetails loadUserByUsername(String username)
                                throws UsernameNotFoundException;
}
```

Configuring User Store

The **`loadUserByUsername()`** method accepts a **`username`** and uses it to look up a **`UserDetails`** object

> if no user can be found for the given **`username`**, then it will throw a **`UsernameNotFoundException`**

```
public interface UserDetailsService {

    UserDetails loadUserByUsername(String username)
                                throws UsernameNotFoundException;
}
```

Configuring User Store

Spring Security offers several out of the box implementations of **`UserDetailsService`**, including:

- An **`in-memory`** user store
- A **`JDBC`** user store
- An **`LDAP`** user store

You can also create your own implementation to suit your application's specific security needs (we will use this approach)

Defining and Persisting `User` Domain

Suppose that Taco Cloud customers need to provide the following information to get registered as a customer

`username`, **`password`**, **`full name`**, and **`phone number`**

Create a file named **`User.java`** inside the **`security`** package and then add the **`User`** entity class shown in the next slide

```java
@Entity
@Data
@NoArgsConstructor
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @Column(unique = true)
    private String username;
    private String password;
    private String fullName;
    private String phone;
}
```

Defining and Persisting `User` Domain

The **`User`** class should implement the **`UserDetails`** interface as that is what the **`UserDetailsService`** expects

```
public interface UserDetailsService {
    UserDetails loadUserByUsername(String username)
                                throws UsernameNotFoundException;
}
```

Modify the **`User`** class as shown in the next slide so as to make it implement the **`UserDetails`** interface

```java
@Entity
@Data
@NoArgsConstructor
public class User implements UserDetails {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    // ...

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return Arrays.asList(new SimpleGrantedAuthority("ROLE_USER"));
    }
     // ...
}
```

Defining and Persisting `User` Domain

`UserDetails` interface declares methods that the inheriting class - **`User`** must implement/override

- one of them - **`getAuthorities()`** is shown in the previous slide

The **`getAuthorities()`** method is where we tell Spring Security about the authorities that the **`User`** have

The remaining methods are - **`isAccountNonExpired()`**, **`isAccountNonLocked()`**, **`isCredentialsNonExpired()`**, **`isEnabled()`**

- For now, you can simply return **true**

User Entity

If you (re)run the application now, a corresponding user table should be created in the database

```
mysql> describe user;
+--------------+--------------+------+
| Field        | Type         | Null |
+--------------+--------------+------+
| id           | bigint       | NO   |
| full_name    | varchar(255) | YES  |
| password     | varchar(255) | YES  |
| phone_number | varchar(255) | YES  |
| username     | varchar(255) | YES  |
+--------------+--------------+------+
```

```
mysql> show tables;
+----------------------+
| Tables_in_taco_cloud |
+----------------------+
| hibernate_sequence   |
| ingredient           |
| taco                 |
| taco_ingredients     |
| taco_order           |
| user                 |
+----------------------+
6 rows in set (0.01 sec)
```

Defining the `UserRepository`

Create a file named **`UserRepository.java`** inside **`security`** package and add the following code in it

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

Creating `UserDetailsService` Bean

Recall that to use Spring Security, we need to create a **`UserDetailsService`** bean

Since we already have a configuration class - **`SecurityConfig`**, we can modify it to provide a **`UserDetailsService`** bean as shown in the next slide

```java
@Configuration
public class SecurityConfig {
    // ...
   @Bean
   public UserDetailsService userDetailsService(UserRepository userRepo) {
       return username -> {
           User user = userRepo.findByUsername(username);
           if (user != null)
               return user;

           throw new UsernameNotFoundException(
                                   "User '" + username + "' not found");
       };
   }
}
```

Registering Users

Create a controller named **`RegistrationController`** shown in the next slide in a file named **`RegistrationController.java`** inside the **`security`** package

We want the controller to handle the following requests

`GET /register` - to display the registration form

`POST /register` - to process the registration using the supplied User information

RegistrationController

```
@Controller
@RequiredArgsConstructor
@RequestMapping("/register")
public class RegistrationController {

    private final UserRepository userRepository;
    private final PasswordEncoder passwordEncoder;

}
```

RegistrationController

```
@Controller
@RequiredArgsConstructor
@RequestMapping("/register")
public class RegistrationController {

    private final UserRepository userRepository;
    private final PasswordEncoder passwordEncoder;

}
```

Allows us to access `save` method that we can use to persist the `User` object

Allows us to `encode` the password the user provided. Spring will inject the bean we defined earlier

`GET /register` handler method

Modify **`RegistrationController`** class as shown in the next slide

```java
@Controller
@RequiredArgsConstructor
@RequestMapping("/register")
public class RegistrationController {

    private final UserRepository userRepository;
    private final PasswordEncoder passwordEncoder;

    @GetMapping
    public String registerForm() {
      return "registration";
    }
}
```

`RegistrationForm` helper class

Let us create a **`RegistrationForm`** helper class which helps us map the user information provided in the registration form to the **`User`** class

It also **`encode`** the password the user provided using the password encoder bean provided in the **`SecurityConfig`** class

Create a file named **`RegistrationForm.java`** and add the code shown in the nest slide

```java
@Data
public class RegistrationForm {
    private String username;
    private String password;
    private String fullName;
    private String phone;

    User toUser(PasswordEncoder encoder) {
        User user = new User();
        user.setUsername(this.username);
        user.setPassword(encoder.encode(this.password));
        user.setFullName(this.fullName);
        user.setPhone(this.phone);
        return user;
    }
}
```

`POST /register` handler method

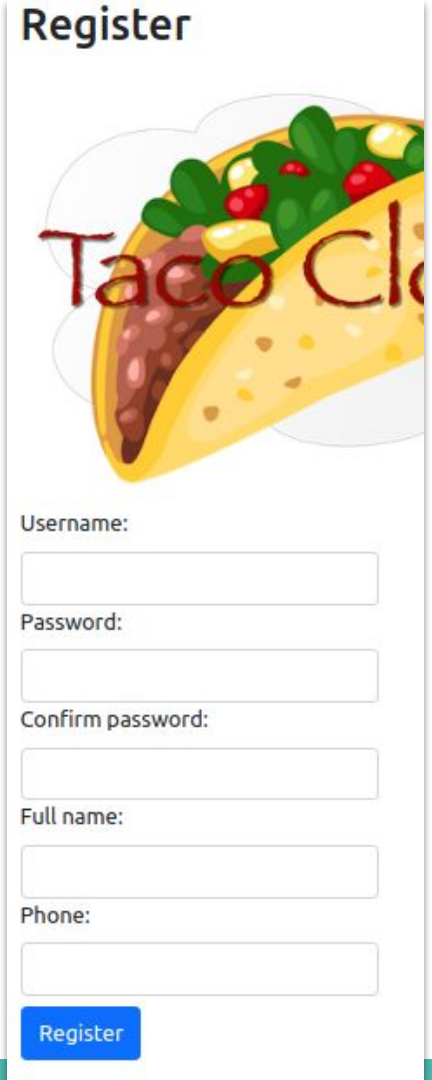
Add **`processRegistration()`** method in the **`RegistrationController`** class as shown in the next slide

```java
@Controller
@RequiredArgsConstructor
@RequestMapping("/register")
public class RegistrationController {
    private final UserRepository userRepository;
    private final PasswordEncoder passwordEncoder;
        // ...
    @PostMapping
    public String processRegistration(RegistrationForm form) {
        userRepository.save(form.toUser(passwordEncoder));
        return "redirect:/login";
    }
}
```

Registration View

The registration view contains a **`form`** with **`Username`**, **`Password`**, **`Confirm password`**, **`Full name`**, and **`Phone`** fields as shown on the right

The form part of the **`HTML`** code is shown in the next slide



```
<form method="POST" th:action="@{/register}" id="registerForm">
  <label class="form-label" for="username">Username: </label>
  <input class="form-control" type="text" name="username" id="username" />
  <label class="form-label" for="password">Password: </label>
  <input class="form-control" type="password" name="password" id="password" />
  <label class="form-label" for="confirm">Confirm password: </label>
  <input class="form-control" type="password" name="confirm" id="confirm"/>
  <label class="form-label" for="fullName">Full name: </label>
  <input class="form-control" type="text" name="fullname" id="fullName" />
  <label class="form-label" for="phone">Phone: </label>
  <input class="form-control" type="text" name="phone" id="phone" />
  <input class="btn btn-primary my-2 " type="submit" value="Register" />
</form>
```

Securing Web Requests

As per Taco Cloud Application security requirement, `homepage`, `login page`, and `registration page` should be available to **unauthenticated** users

To configure these kind of security rules, we'll need to declare a **`SecurityFilterChain`** bean

A minimal **`SecurityFilterChain`** is shown in the next slide

Securing Web Requests

Add the following method inside **`SecurityConfig`** class found in **`SecurityConfig.java`** file

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) {
    return http.build();
}
```

The **`filterChain()`** method accepts an **`HttpSecurity`** object, which acts as a builder that can be used to configure how security is handled at the web level

Securing Web Requests

Among the many things you can configure with `HttpSecurity` are these:

- Requiring that certain security conditions be met before allowing a request to be served
- Configuring a custom login page
- Enabling users to log out of the application
- Configuring cross-site request forgery protection

Securing Requests

Intercepting requests to ensure that the user has proper authority is one of the most common things you'll configure `HttpSecurity` to do

Let us update the `HttpSecurity` configuration as shown in the next slide to

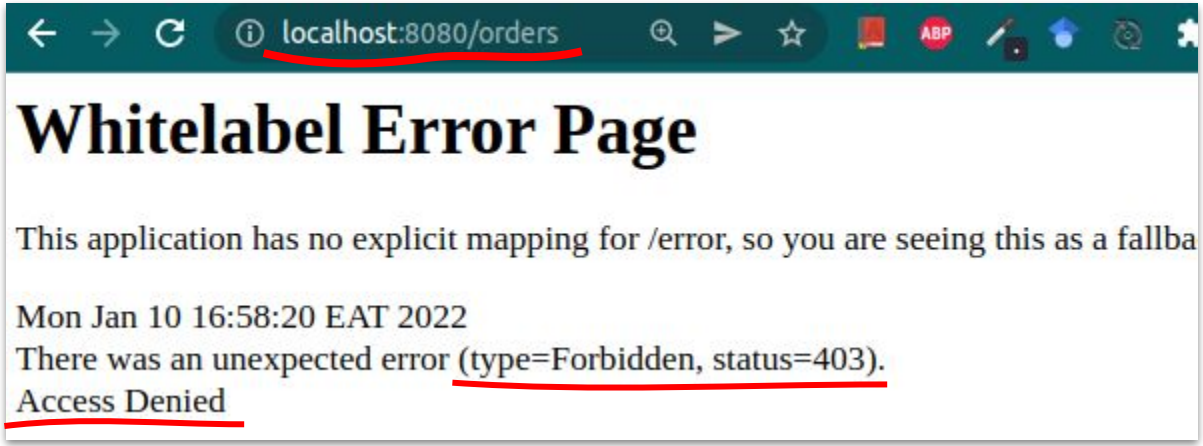
- ensure that requests for `/design` and `/orders` are **only available to authenticated users** and
- all other requests should be permitted for all users

The updated configuration is shown in the next slide

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http)
                                                throws Exception {
    return http
            .authorizeRequests()
            .antMatchers("/design", "/orders").hasRole("USER")
            .antMatchers("/", "/**").permitAll()
            .and()
            .build();
}
```

Securing Requests

If you now try to send **`GET /design`** or **`GET /order`** request, you will get **`403 Forbidden`** (**`Access Denied`**) error as shown below



Securing Requests

If you try to send `GET /` for the home page or `GET /register` requests for the registration page, you should be able to access them as they requests are permitted

Securing Requests

The call to `authorizeRequests()` returns an object on which you can specify `URL` **paths** and **patterns** and the **security requirements** for those paths

Note that, requests for `/design` and `/orders` should be for users with a granted authority of `ROLE_USER`

> don’t include the "`ROLE_`" **prefix** on roles passed to `hasRole()`; it will be assumed by `hasRole()`

Securing Requests

Note that, the order of the rules is important

- security rules declared first take precedence over those declared lower down
- if you were to swap the order of those two security rules, all requests would have `permitAll()` applied to them; the rule for `/design` and `/orders` requests would have no effect

Methods for Declaring Security requirements

Method	What it does
`access(String)`	Allows access if the given Spring Expression Language (SpEL) expression evaluates to `true`
`anonymous()`	Allows access to anonymous users
`authenticated()`	Allows access to authenticated users
`denyAll()`	Denies access unconditionally
`fullyAuthenticated()`	Allows access if the user is fully authenticated (not remembered)

Methods for Declaring Security requirements

Method	What it does
`hasAnyAuthority(String...)`	Allows access if the user has any of the given authorities
`hasAnyRole(String...)`	Allows access if the user has any of the given roles
`hasAuthority(String)`	Allows access if the user has the given authority
`hasIpAddress(String)`	Allows access if the request comes from the given IP address

Methods for Declaring Security requirements

Method	What it does
`hasRole(String)`	Allows access if the user has the given role
`not()`	Negates the effect of any of the other access methods
`permitAll()`	Allows access unconditionally
`rememberMe()`	Allows access for users who are authenticated via remember-me

Methods for Declaring Security requirements

You can use the `access()` method to provide a SpEL expression to declare richer security rules

Using the `access()` method along with the `hasRole()` and `permitAll` expressions, we can rewrite the `SecurityFilterChain` configuration as shown in the next slide

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws
Exception {
 return http
    .authorizeRequests()
      .antMatchers("/design", "/orders").access("hasRole('USER')")
      .antMatchers("/", "/**").access("permitAll()")
    .and()
    .build();
}
```

More Complex Security Requirements

Suppose that you only wanted to allow users with `ROLE_USER` authority to create new tacos on **Tuesdays**

- you could rewrite the expression using `access()` method as shown in the next slide

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http)
                                                throws Exception {
 return http
    .authorizeRequests()
      .antMatchers("/design", "/orders")
        .access("hasRole('USER') && " +
          "T(java.util.Calendar).getInstance().get("+
          "T(java.util.Calendar).DAY_OF_WEEK) == " +
          "T(java.util.Calendar).TUESDAY")
      .antMatchers("/", "/**").access("permitAll")
    .and().build();
}
```

Creating a Custom Login Page

To replace the built-in login page, you first need to tell Spring Security what path your custom login page will be at

that can be done by calling `formLogin()` on the `HttpSecurity` object as shown in the next slide

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http)
                                                throws Exception {
    return http
            .authorizeRequests()
            .antMatchers("/design", "/orders").hasRole("USER")
            .antMatchers("/", "/**").permitAll()
            .and()
            .formLogin()
            .loginPage("/login")
            .and()
            .build();
}
```

Optional Login Configuration

By default, Spring Security listens for login requests at **`/login`** and expects that the **`username`** and **`password`** fields be named **`username`** and **`password`**

You can change this as shown in the next slide, if you want

```
.and()
 .formLogin()
    .loginPage("/login")
    .loginProcessingUrl("/authenticate")
    .usernameParameter("user")
    .passwordParameter("pwd")
```

Optional Login Configuration

By default, a successful login will take the user directly to the page that they were navigating to when Spring Security determined that they needed to log in

> If the user were to directly navigate to the login page, a successful login would take them to the root path. But you can change that by specifying a **default success page**:

```
.and()
 .formLogin()
   .loginPage("/login")
   .defaultSuccessUrl("/design")
```

Working with View Controllers

When a controller is simple enough that it doesn’t populate a model or process input you can define the controller as shown in the next slide

Create a WebConfig.java file and put the code shown in the next slide into it

```java
@Configuration
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        registry.addViewController("/").setViewName("home");
        registry.addViewController("/login").setViewName("login");
    }

}
```

`WebMvcConfigurer`

The **`WebConfig`** class implements the **`WebMvcConfigurer`** interface

`WebMvcConfigurer` defines several methods for configuring **`Spring MVC`**

Even though it's an interface, it provides default implementations of all the methods, so you only need to override the methods you need

In the our case, we override **`addViewControllers()`**

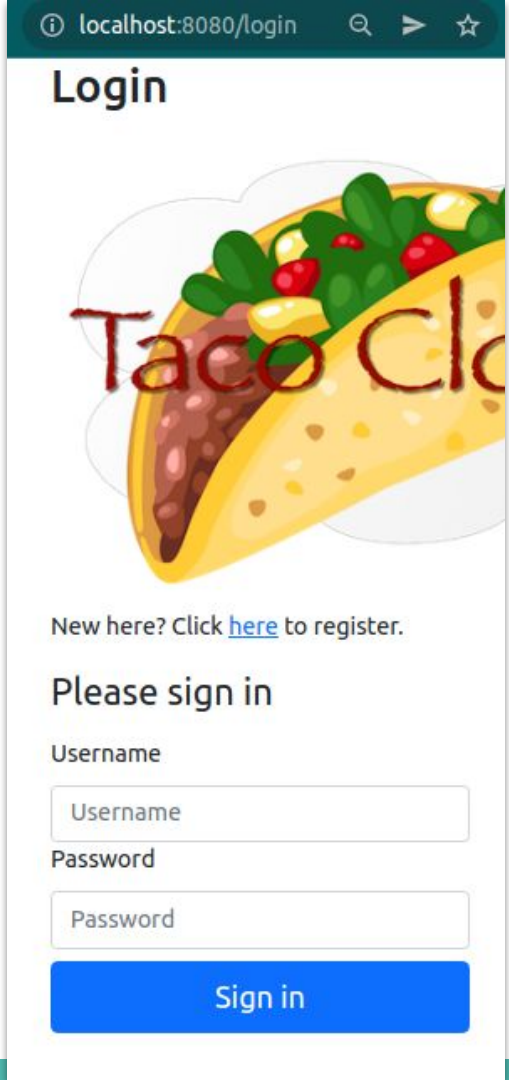
Login Page

Now if you make a **`GET /login`** request you will see the login page

if you try to also access protected pages such as **`/order`** or **`/design`** you will be redirected to the login page

You will also be redirected to the login page after registration

The partial **form** part of the HTML code is shown in the next slide



```
<form method="POST" th:action="@{/login}">
    <h1 class="h3 mb-3 fw-normal">Please sign in</h1>

    <label class="form-label" for="username">Username</label>
    <input type="text"  id="username"  name="username"/>
    <label class="form-label" for="password">Password</label>
    <input type="password" id="password" name="password"/>

    <button type="submit">Sign in</button>
</form>
```

Logging Out

Just as important as logging into an application is logging out

To enable **`logout`**, you simply need to call **`logout`** on the **`HttpSecurity`** object:

```
and()
.logout()
```

This sets up a security filter that intercepts **`POST`** requests to **`/logout`**

Logging Out

To provide logout capability, you need to add a logout form and button to the views in your application

```
<form class="d-flex" method="POST" th:action="@{/logout}">
 <input type="submit" value="Logout"/>
</form>
```

Logging Out

When the user clicks the **`logout`** button, their session will be cleared, and they will be logged out of the application

By default, they'll be redirected to the **`login`** page where they can log in again

But if you'd rather they be sent to a different page, you can call **`logoutSuccessUrl()`** to specify a different page

```
.and()
 .logout()
   .logoutSuccessUrl("/")
```

Preventing Cross-Site Request Forgery (CSRS)

Cross-site request forgery (CSRF) is a common security attack

Assume that your bank's website provides a form that allows transferring money from the currently logged in user to another bank account

for example, the HTTP request might look like

```
POST /transfer HTTP/1.1
Host: bank.example.com
Cookie: JSESSIONID=randomid; Domain=bank.example.com; Secure; HttpOnly
Content-Type: application/x-www-form-urlencoded

amount=100.00&account=9876
```

Preventing Cross-Site Request Forgery (CSRS)

Now pretend you authenticate to your bank's website and then, without logging out, visit an evil website

The evil website contains an HTML page with the following form

```
<form action="https://bank.example.com/transfer" method="post">
 <input type="hidden" name="amount" value="100.00" />
 <input type="hidden" name="account" value="evilsAccountNum" />
 <input type="submit" value="Win Money!" />
</form>
```

Preventing Cross-Site Request Forgery (CSRS)

```
<form action="https://bank.example.com/transfer" method="post">
 <input type="hidden" name="amount" value="100.00" />
 <input type="hidden" name="account" value="evilsAccountNum" />
 <input type="submit" value="Win Money!" />
</form>
```

You like to win money, so you click on the submit button

in the process, you have unintentionally transferred $100 to a malicious user

Preventing Cross-Site Request Forgery (CSRS)

The attack happens because, while the evil website cannot see your cookies, **the cookies associated with your bank are still sent along with the request**

This whole process could have been automated using JavaScript

this means you didn't even need to click on the button

So how do we protect ourselves from such attacks?

Preventing Cross-Site Request Forgery (CSRS)

To protect against such attacks

- applications can generate a CSRF token upon displaying a form
- place that token in a hidden field, and then stow it for later use on the server
- when the form is submitted, the token is sent back to the server along with the rest of the form data

Preventing Cross-Site Request Forgery (CSRS)

To protect against such attacks

- the request is then intercepted by the server and compared with the token that was originally generated
- if the token matches, the request is allowed to proceed
- otherwise, the form must have been rendered by an evil website without knowledge of the token generated by the server

Preventing Cross-Site Request Forgery (CSRS)

Spring Security has built-in CSRF protection

It’s enabled by default and you don’t need to explicitly configure it

You only need to make sure that any forms your application submits include a field named `_csrf` that contains the `CSRF` token

Preventing Cross-Site Request Forgery (CSRS)

Spring Security even makes that easy by placing the CSRF token in a request attribute with the name **`_csrf`**

Therefore, you could render the **`CSRF`** token in a hidden field with the following in a Thymeleaf template

```
<input type="hidden" name="_csrf" th:value="${_csrf.token}"/>
```

Preventing Cross-Site Request Forgery (CSRS)

In case of Thymeleaf with the Spring Security dialect,the hidden field will be rendered automatically for you

you only need to make sure that one of the attributes of the **<form>** element is prefixed as a **Thymeleaf attribute**.

```
<form method="POST" th:action="@{/login}" id="loginForm">
 <!-- ... -->
</form>
```

Check CSRF Token

Check the page source code for the login form, for example

```
<p>New here? Click <a href="/register">here</a> to register.</p>
<form method="POST" action="/login"><input type="hidden" name="_csrf" value="686ef08b-6fc9-4897-b746-938d4d72ffd9"/>
```

The Source Code

https://github.com/betsegawlemma/taco-cloud-sample/tree/security

Further References

https://www.owasp.org/index.php/OWASP_Guide_Project

https://www.owasp.org/index.php/Category:OWASP_Code_Review_Project

https://www.owasp.org/index.php/OWASP_Secure_Coding_Practices_-_Quick_Reference_Guide

https://www.owasp.org/images/b/ba/Web_Application_Development_Dos_and_Donts.ppt