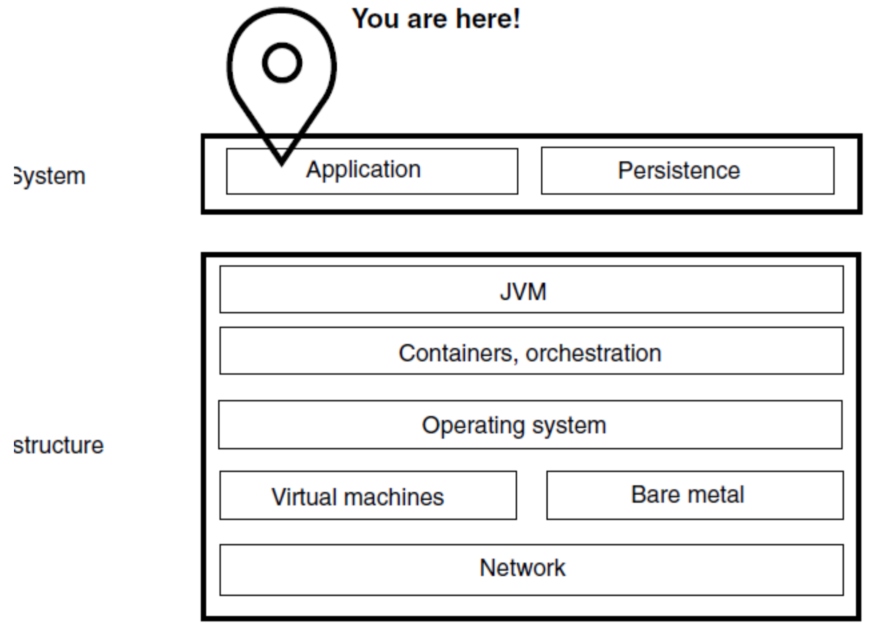
SPRING PART III

Chandreyee chowdhury

SOFTWARE SECURITY

- ❑Software systems today manage large amounts of data, of which a significant part can be considered sensitive, especially given the current General Data Protection Regulations (GDPR) requirements.
- ❑ In India THE DIGITAL PERSONAL DATA PROTECTION BILL, 2022 is proposed to regulate the data protection
- ❑Any information that you, as a user, consider private is sensitive for your software application
- ❑GDPR created a lot of buzz globally after its introduction in 2018.
- ❑It generally represents a set of European laws that refer to data protection and gives people more control over their private data.
- ❑GDPR applies to the owners of systems having users in Europe.
- ❑The owners of such applications risk significant penalties if they don’t respect the regulations imposed



TYPES OF DATA

- We classify data as “at rest” or “in transition.”
- In this context, *data at rest refers* to data in computer storage or, in other words, persisted data.
- *Data in transition* applies to all the data that’s exchanged from one point to another.
- Different security measures should, therefore, be enforced, depending on the type of data.

❑*Application-level security* refers to everything that an application should do to protect the environment it executes in, as well as the data it processes and store
❑An application might contain vulnerabilities that allow a malicious individual to affect the entire system

COMMON VULNERABILITIES IN WEB APPLICATIONS

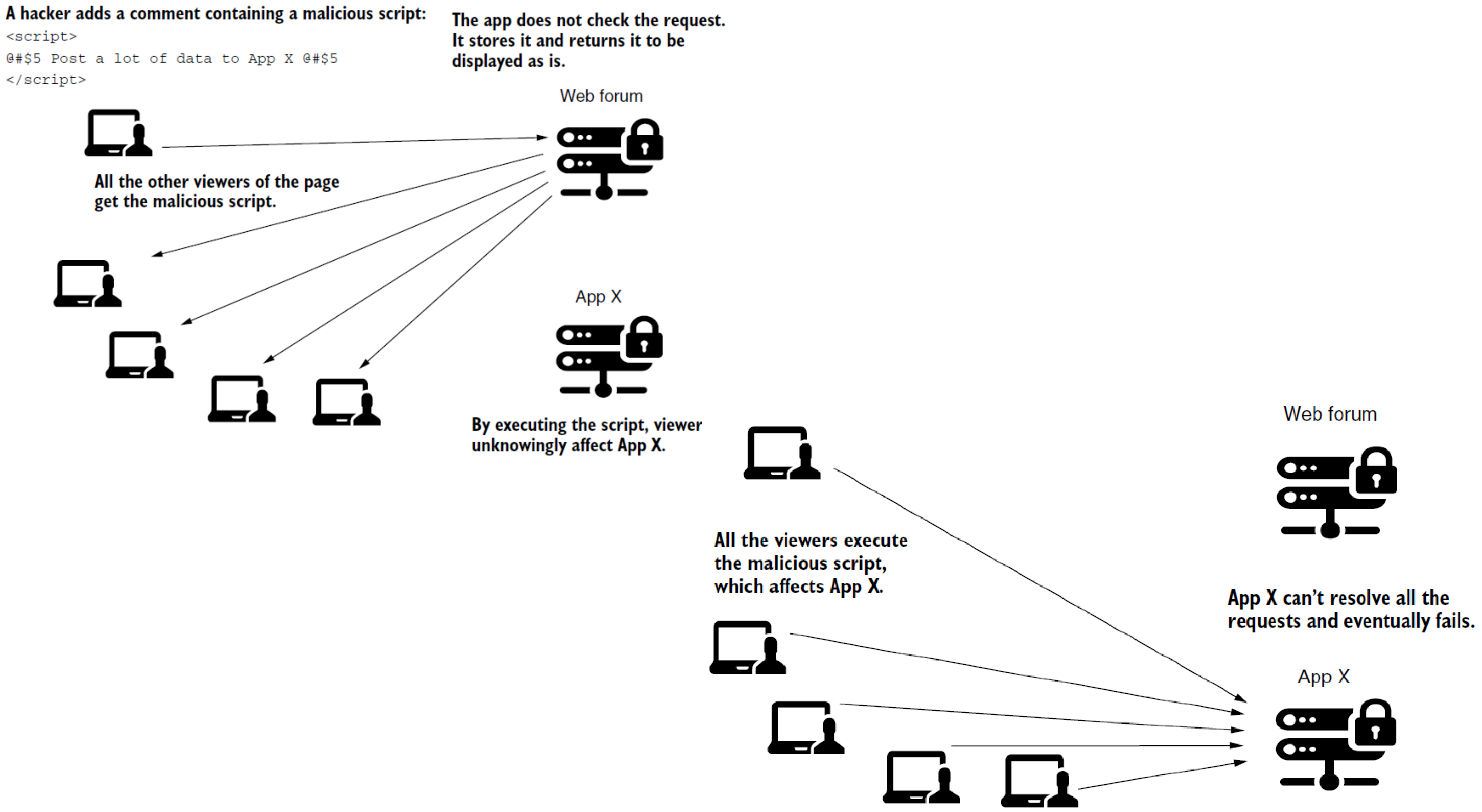
- Broken authentication
- Session fixation
- Cross-site scripting (XSS)
- Cross-site request forgery (CSRF)
- Injections
- Sensitive data exposure

AUTHENTICATION AND AUTHORIZATION

- *Authentication*
- represents the process in which an application identifies someone trying to use it
- *Authorization* is the process of establishing if an authenticated caller has the privileges to use specific functionality and data
- We have a broken authorization if an individual with bad intentions somehow gains access to functionality or data that doesn’t belong to them
- For instance, a user can view other users’ products

VULNERABILITIES

- *Session fixation* vulnerability is a more specific, high-severity weakness of a web application.
- If present, it permits an attacker to impersonate a valid user by reusing a previously generated session ID.
- There are various ways an individual can use this vulnerability.
- For example, if the application provides the session ID in the URL, then the victim could be tricked into clicking on a malicious link
- If the application stores the value of the session in a cookie, then the attacker can inject a script and force the victim's browser to execute it
- *Cross-site scripting*, also referred to as XSS, allows the injection of client-side scripts into web services exposed by the server, thereby permitting other users to run these.
- Before being used or even stored, you should properly "sanitize" the request to avoid undesired executions of foreign scripts.
- The potential impact can relate to account impersonation (combined with session fixation) or to participation in distributed attacks like DDoS.



A hacker adds a comment containing a malicious script:

```
<script>
@#$5 Post a lot of data to App X @#$5
</script>
```

The app does not check the request. It stores it and returns it to be displayed as is.

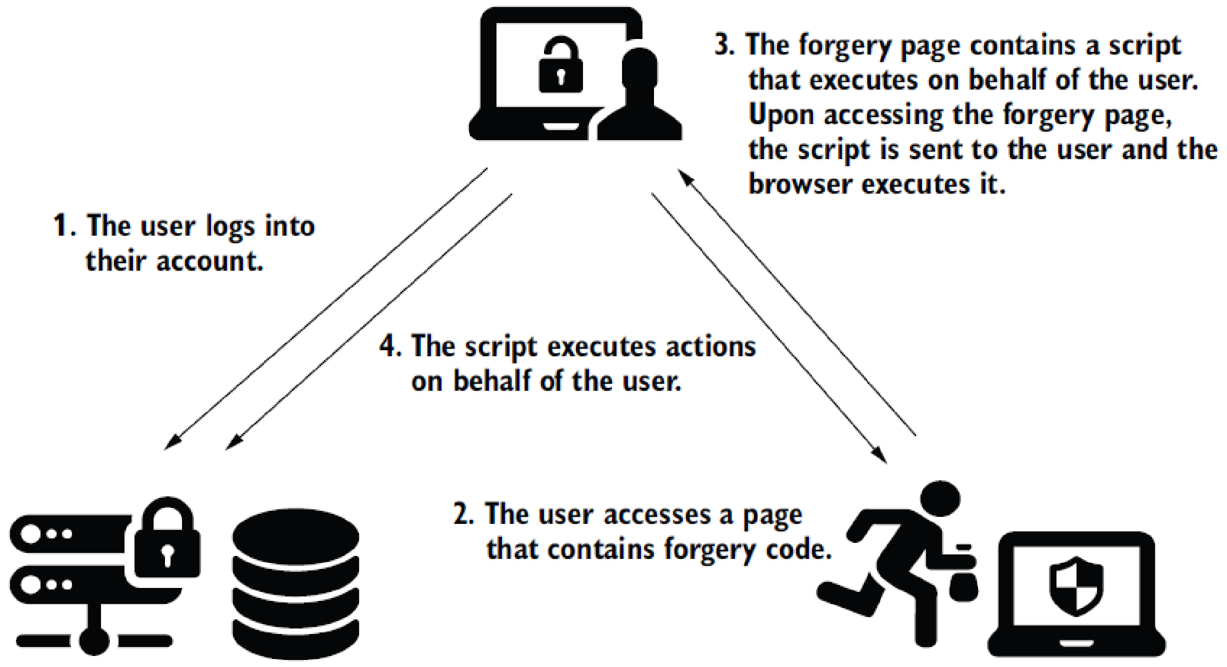
All the other viewers of the page get the malicious script.

By executing the script, viewer unknowingly affect App X.

All the viewers execute the malicious script, which affects App X.

App X can't resolve all the requests and eventually fails.

Cross-site request forgery (CSRF) vulnerabilities are also common in web applications.
CSRF attacks assume that a URL that calls an action on a specific server can be extracted and reused from outside the application


3. The forgery page contains a script that executes on behalf of the user. Upon accessing the forgery page, the script is sent to the user and the browser executes it.
1. The user logs into their account.
4. The script executes actions on behalf of the user.
2. The user accesses a page that contains forgery code.

One of the ways of mitigating this vulnerability is to use tokens to identify the request or use cross-origin resource sharing (CORS) limitations. In other words, validate the origin of the request

INJECTION ATTACKS

Injection types of vulnerabilities are important, and the results of exploiting these can be change, deletion, or access to data in the systems being compromised

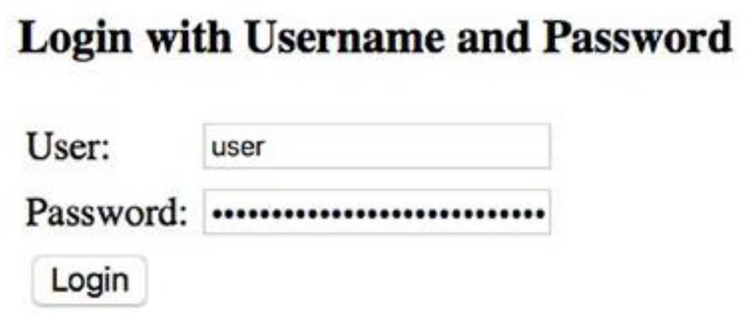
One of the oldest and perhaps well-known types of injection vulnerability is SQL injection.

If your application has an SQL injection vulnerability, an attacker can try to change or run different SQL queries to alter, delete, or extract data from your system.

CORE SECURITY SERVICES

Security services

- Confidentiality
- Authentication
- Authorization



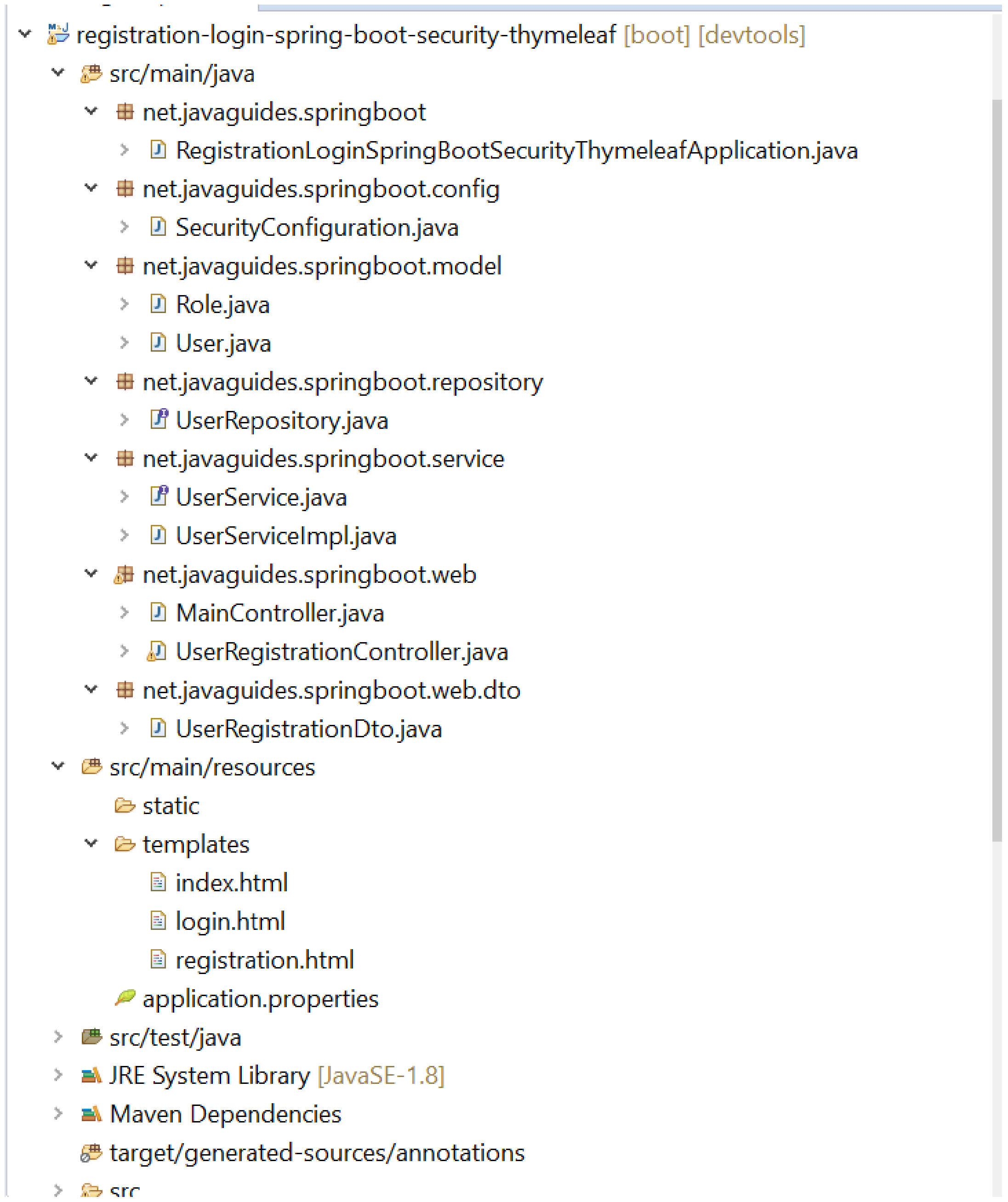
Login with Username and Password

User: user

Password: ••••••••••••••••••••••••••••••

SPRING SECURITY

FILE ORGANIZATION



```
registration-login-spring-boot-security-thymeleaf [boot] [devtools]
  src/main/java
    net.javaguides.springboot
      RegistrationLoginSpringBootSecurityThymeleafApplication.java
    net.javaguides.springboot.config
      SecurityConfiguration.java
    net.javaguides.springboot.model
      Role.java
      User.java
    net.javaguides.springboot.repository
      UserRepository.java
    net.javaguides.springboot.service
      UserService.java
      UserServiceImpl.java
    net.javaguides.springboot.web
      MainController.java
      UserRegistrationController.java
    net.javaguides.springboot.web.dto
      UserRegistrationDto.java
  src/main/resources
    static
    templates
      index.html
      login.html
      registration.html
    application.properties
  src/test/java
  JRE System Library [JavaSE-1.8]
  Maven Dependencies
  target/generated-sources/annotations
  src
```

WEBMVC

- The `addViewControllers()` method (which overrides the method of the same name in `WebMvcConfigurer`) adds four view controllers
- If visitors click the link on the home page, they see the greeting with no barriers to stop them
- You need to add a barrier that forces the visitor to sign in before they can see that page
- You do that by configuring Spring Security in the application
- If Spring Security is on the classpath, Spring Boot automatically secures all HTTP endpoints with “basic” authentication
- However, you can further customize the security settings

AUTHENTICATION AND AUTHORIZATION

WebSecurityConfigurer Adapter

SPRING SECURITY

The first step is to add `spring-boot-starter-security` dependency

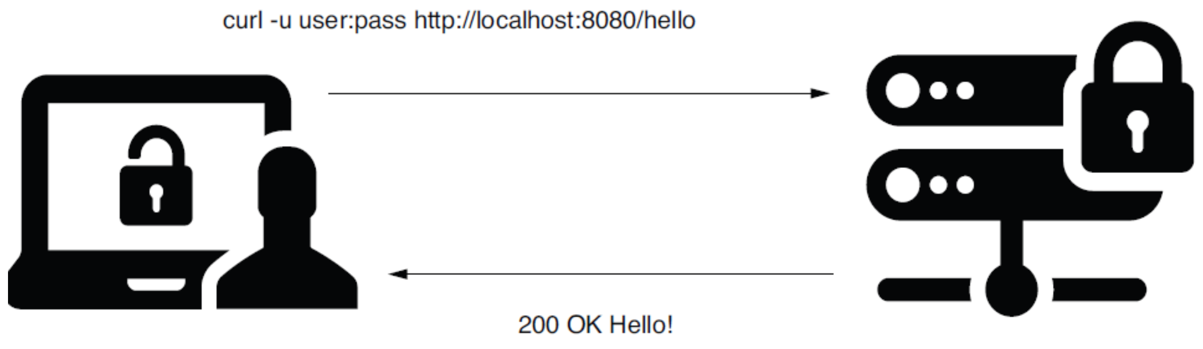
Declarative security

- spring.security.user.name=apress
- spring.security.user.password=springboot2
- spring.security.user.roles=ADMIN

Programmatic security

- Extend WebSecurityConfigureAdapter class

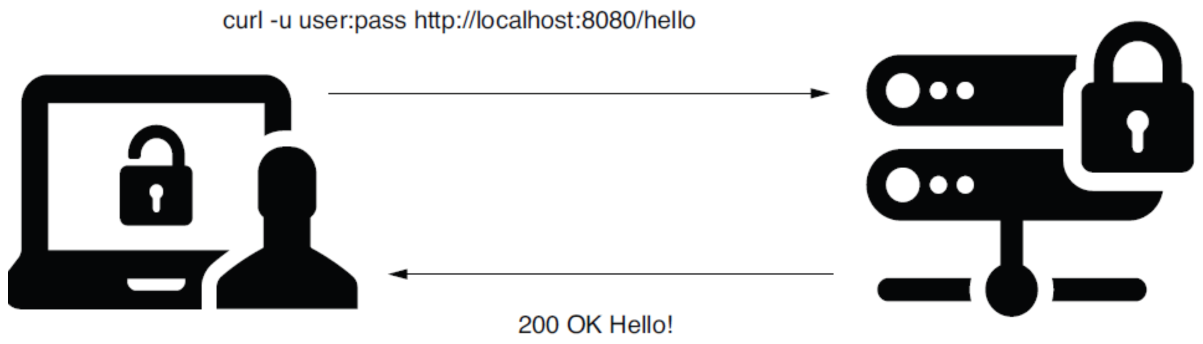
BASIC EXAMPLE FOR AUTHENTICATION



- ❑ Spring Security secures this endpoint using HTTP Basic authentication
- ❑ Just by creating the project and adding the correct dependencies, Spring Boot applies default configurations, including a username and a password when you start the application
- ❑ The only dependencies you need to select for our first project are `spring-boot-starter-web` and `spring-boot-starter-security`
- ❑ `POM.XML` contains the following

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

BASIC EXAMPLE FOR AUTHENTICATION



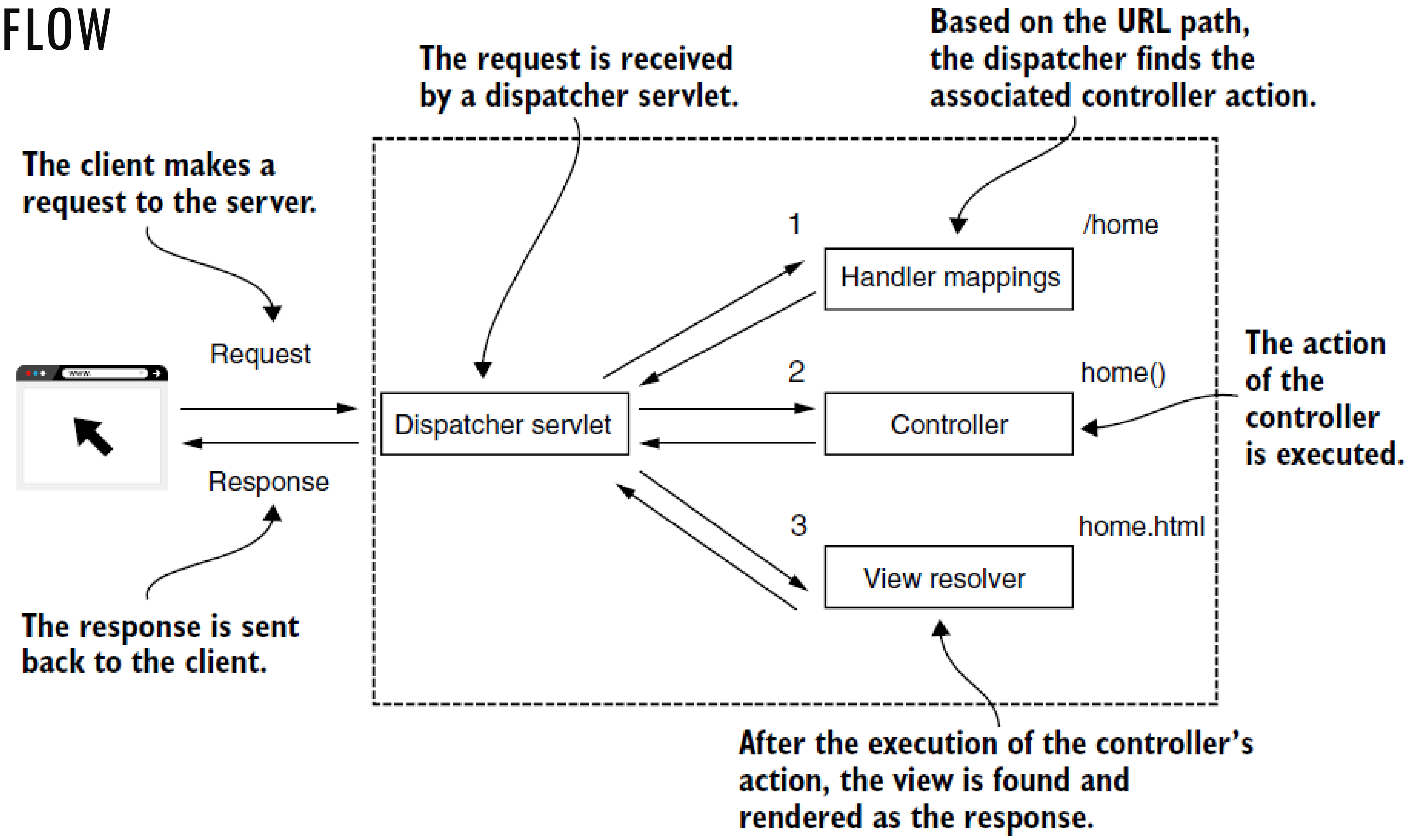
Spring Boot applies the default configuration of the Spring context for us based on which dependencies we wouldn't be able to learn much about security if we don't have at least one endpoint that's secured

```
@RestController
public class HelloController {

  @GetMapping("/hello")
  public String hello() {
    return "Hello!";
  }
}
```

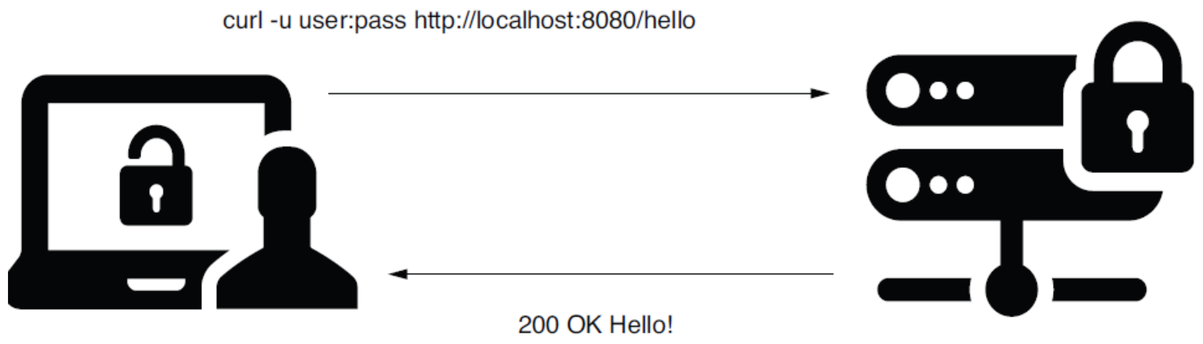
The `@RestController` annotation registers the bean in the context and tells Spring that the application uses this instance as a web controller.

Also, the annotation specifies that the application has to set the returned value as the body of the HTTP response.

OVERALL WORKFLOW


The request is received by a dispatcher servlet.
Based on the URL path, the dispatcher finds the associated controller action.
The client makes a request to the server.
1
/home
Handler mappings
Request
2
home()
The action of the controller is executed.
Dispatcher servlet
Controller
Response
3
home.html
View resolver
The response is sent back to the client.
After the execution of the controller's action, the view is found and rendered as the response.

BASIC EXAMPLE FOR AUTHENTICATION



Once you run the application, besides the other lines in the console, you should see something that looks similar to this:

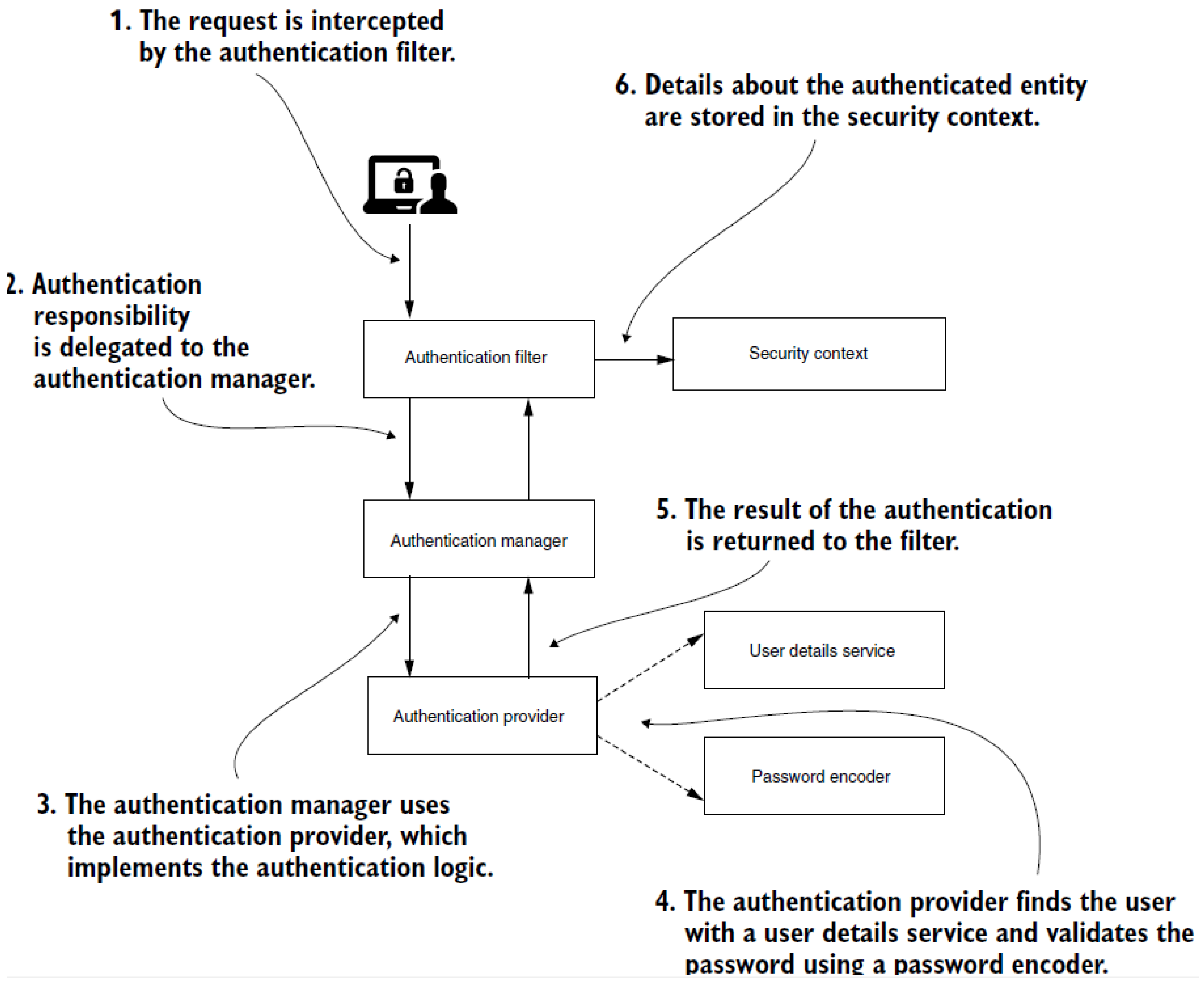
```
Using generated security password: 93a01cf0-794b-4b98-86ef-
54860f36f7f3
```

Each time you run the application, it generates a new password and prints this password in the console as presented in the previous code snippet.

You must use this password to call any of the application’s endpoints with HTTP Basic authentication

BIG PICTURE

The authentication provider uses two beans to find users and to check their passwords


1. The request is intercepted by the authentication filter.
6. Details about the authenticated entity are stored in the security context.
2. Authentication responsibility is delegated to the authentication manager.
Authentication filter
Security context
Authentication manager
5. The result of the authentication is returned to the filter.
User details service
Authentication provider
Password encoder
3. The authentication manager uses the authentication provider, which implements the authentication logic.
4. The authentication provider finds the user with a user details service and validates the password using a password encoder.

USER AND PASSWORD DETAILS

- A `UserDetailsService` contract with Spring Security manages the details about users.
- Until now, we used the default implementation provided by Spring Boot.
- This implementation only registers the default credentials in the internal memory of the application.
- These default credentials are “user” with a default password that’s a universally unique identifier (UUID).
- This password is randomly generated when the Spring context is loaded.
- The `PasswordEncoder` does two things:
- ⍰ Encodes a password
- ⍰ Verifies if the password matches an existing encoding

PASSWORD ENCODING

Spring Boot also chooses an authentication method when configuring the defaults, HTTP Basic access authentication

Basic authentication only requires the client to send a username and a password through the HTTP `Authorization` header.

In the value of the header, the client attaches the prefix `Basic`, followed by the Base64 encoding of the string that contains the username and password, separated by a colon (:)

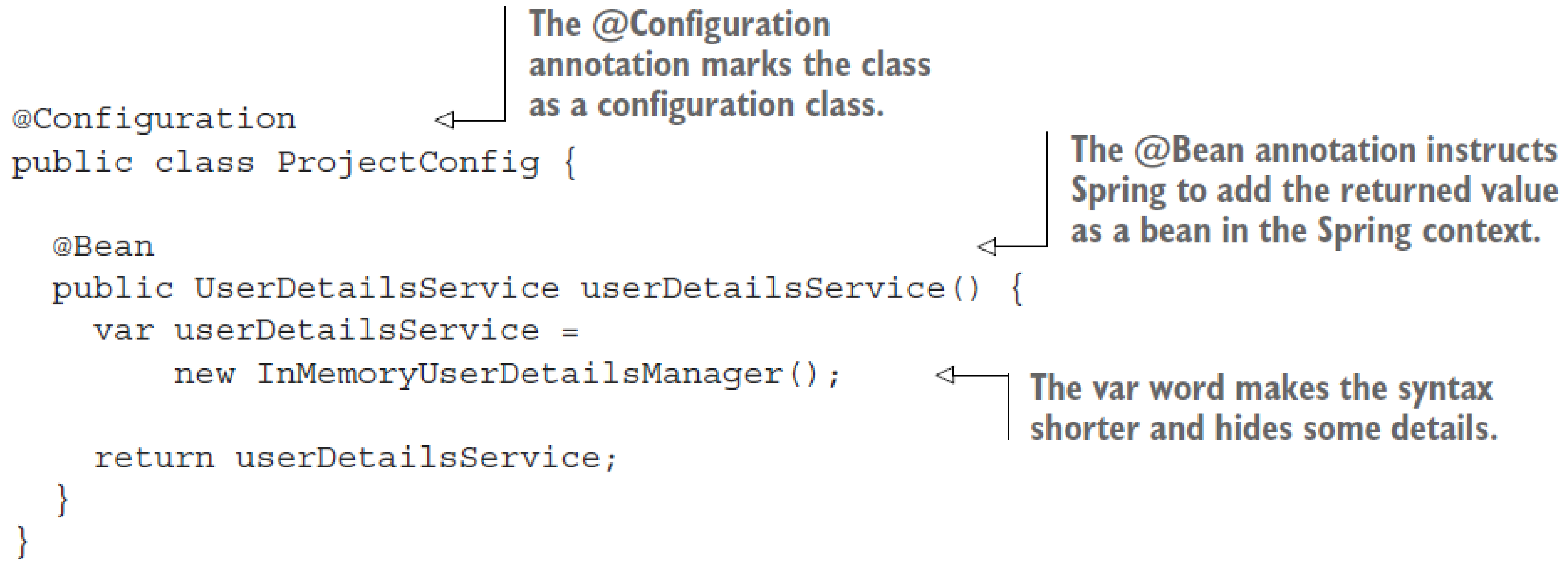
The `AuthenticationProvider` defines the authentication logic, delegating the user and password management.

A default implementation of the `Authentication-Provider` uses the default implementations provided for the `UserDetailsService` and the `PasswordEncoder`.

Implicitly, your application secures all the endpoints

IMPLEMENTING USERDETAILSERVICE

- We use the `InMemoryUserDetailsManager` implementatation of `UserDetailService`
- This implementation stores credentials in memory, which can then be used by Spring Security to authenticate a request
- An `InMemoryUserDetailsManager` implementation isn't meant for production-ready applications, but it's an excellent tool for examples or proof of concepts.



```
@Configuration
public class ProjectConfig {

  @Bean
  public UserDetailsService userDetailsService() {
    var userDetailsService =
        new InMemoryUserDetailsManager();

    return userDetailsService;
  }
}
```

The @Configuration annotation marks the class as a configuration class.

The @Bean annotation instructs Spring to add the returned value as a bean in the Spring context.

The var word makes the syntax shorter and hides some details.

- The application now uses the instance of type `UserDetailsService` you added to the context instead of the default autoconfigured one.
- We need to
- 1 Create at least one user who has a set of credentials (username and password)
- 2 Add the user to be managed by our implementation of `UserDetailsService`
- 3 Define a bean of the type `PasswordEncoder` that our application can use to verify a given password with the one stored and managed by `UserDetailsService`
- When building the instance, we have to provide the username, the password, and at least one authority.
- The *authority* is an action allowed for that user, and we can use any string for this

```java
@Configuration
public class ProjectConfig {

  @Bean
  public UserDetailsService userDetailsService() {
    var userDetailsService =
        new InMemoryUserDetailsManager();

    var user = User.withUsername("john")
            .password("12345")
            .authorities("read")
            .build();

    userDetailsService.createUser(user);

    return userDetailsService;
  }
}
```

Builds the user with a given username, password, and authorities list

Adds the user to be managed by UserDetailsService

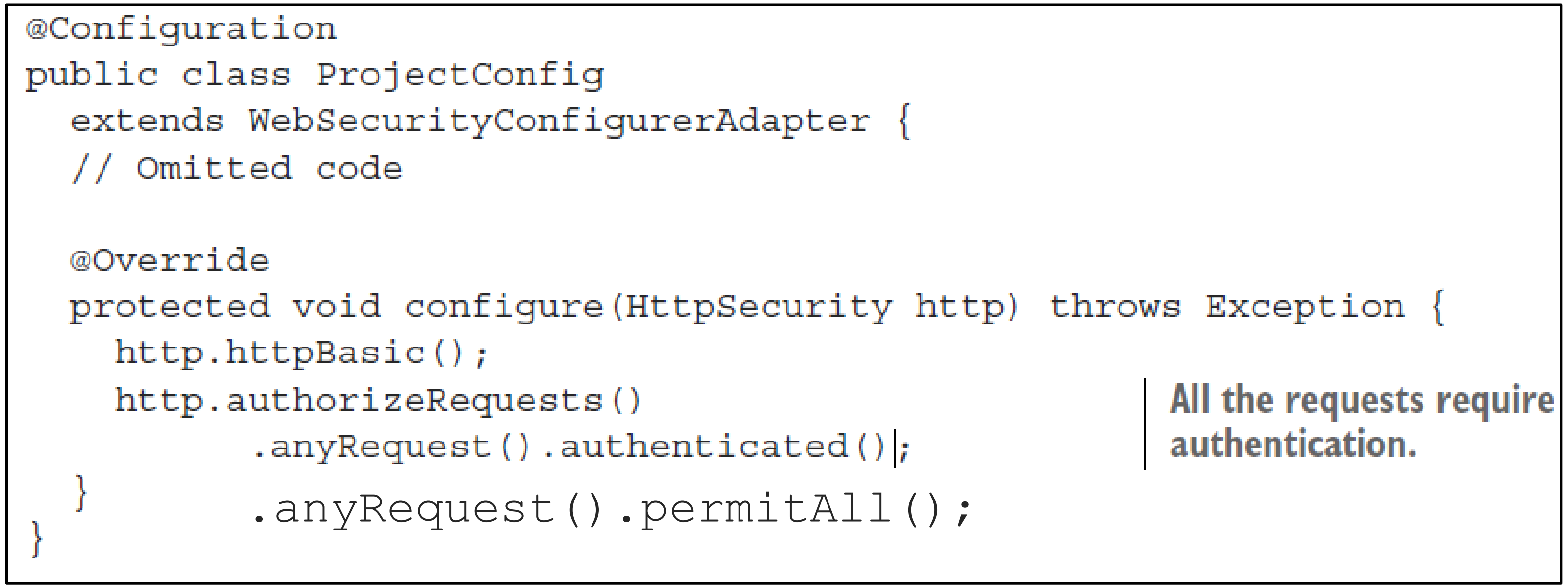
```java
  @Bean
  public PasswordEncoder passwordEncoder() {
    return NoOpPasswordEncoder.getInstance();
  }
}
```

A new method annotated with @Bean to add a PasswordEncoder to the context

CONFIGURING END POINTS

We start by extending the `WebSecurityConfigurerAdapter` class.

Extending this class allows us to override the `configure(HttpSecurity http)` method



```
@Configuration
public class ProjectConfig
  extends WebSecurityConfigurerAdapter {
  // Omitted code

  @Override
  protected void configure(HttpSecurity http) throws Exception {
    http.httpBasic();
    http.authorizeRequests()
          .anyRequest().authenticated();
  }
          .anyRequest().permitAll();
}
```

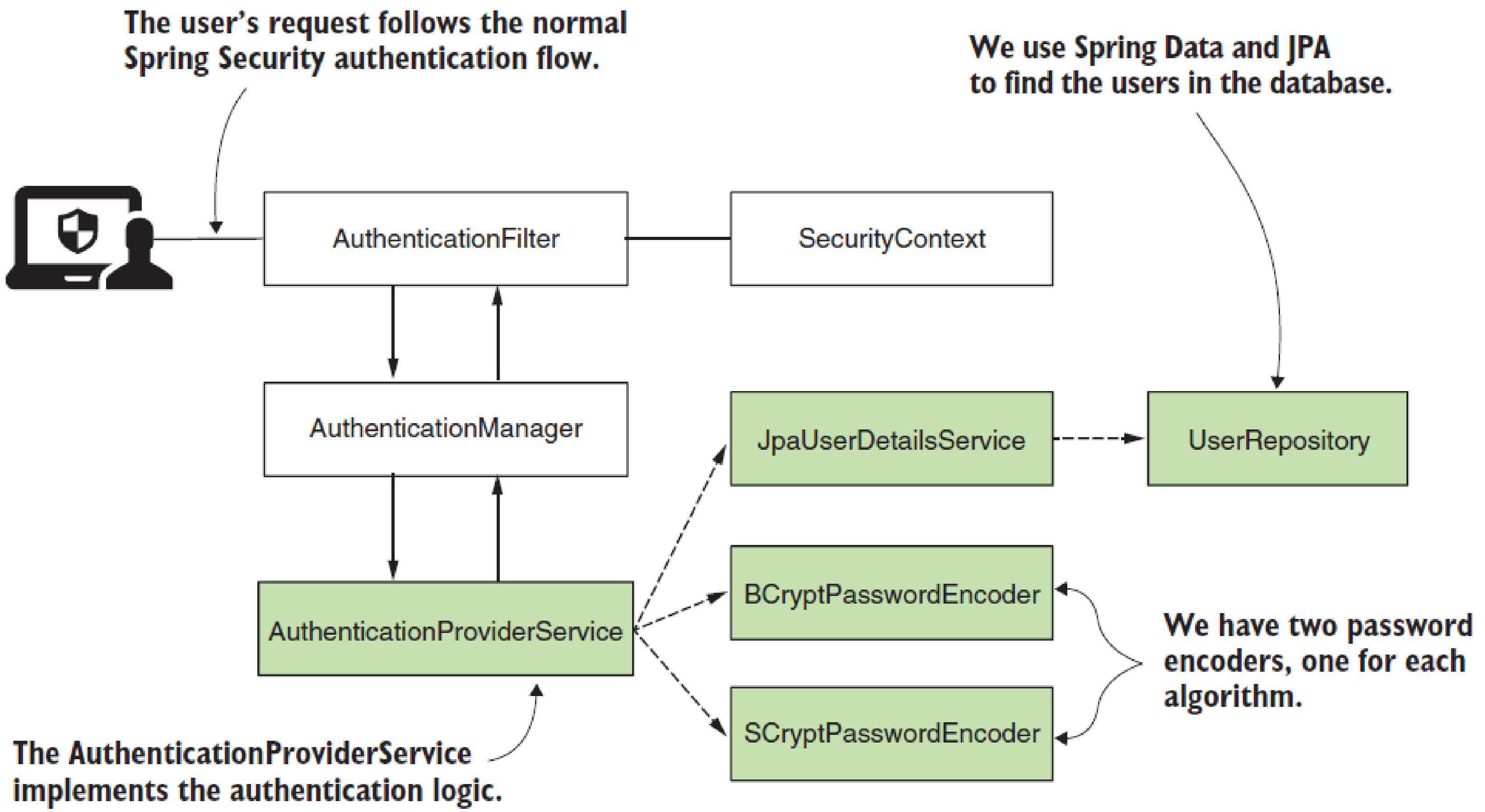
All the requests require authentication.

```
@Configuration          @EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
        @Override
        protected void configure(HttpSecurity http) throws Exception {
                http
                        .authorizeRequests()
                                .antMatchers("/").permitAll()
                                .anyRequest().authenticated()
                                .and()
                        .formLogin()
                                //.loginPage("/login")
                                .permitAll()
                                .and()
                        .logout()
                                .permitAll()
                                .and()
                                .httpBasic();
                http.csrf().disable();

        }
        @Bean
        @Override
        public UserDetailsService userDetailsService() {
                UserDetails user = User.withDefaultPasswordEncoder().username("user").password("password").roles("USER").build();
                return new InMemoryUserDetailsManager(user);
        }}
```

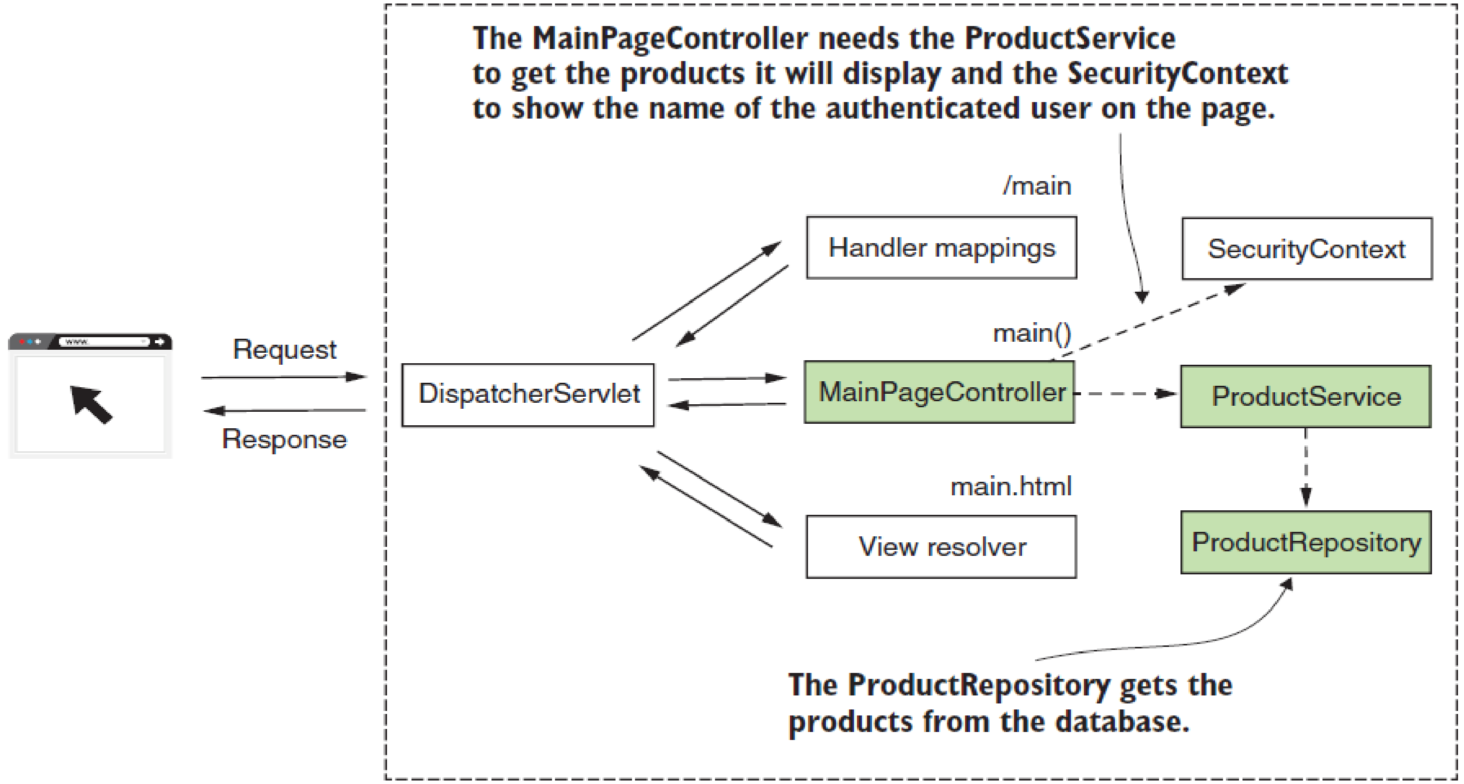
The HttpSecurity class allows you to configure web-based security for specific HTTP requests. By default, it is applied to all requests, but can be restricted using requestMatcher(RequestMatcher) or similar methods

```
@Configuration
 @EnableWebSecurity
 public class LogoutSecurityConfig {
 @Bean
 public SecurityFilterChain securityFilterChain(HttpSecurity
http) throws Exception {
http.authorizeRequests().requestMatchers("/**").hasRole("USER
").and().
formLogin().and().
logout().deleteCookies("remove").
invalidateHttpSession(false)
.logoutUrl("/customlogout"). logoutSuccessUrl("/logout-
success");
    return http.build();    }
```

HANDLING USER DETAILS



The user's request follows the normal Spring Security authentication flow.
We use Spring Data and JPA to find the users in the database.
AuthenticationFilter
SecurityContext
AuthenticationManager
JpaUserDetailsService
UserRepository
BCryptPasswordEncoder
AuthenticationProviderService
SCryptPasswordEncoder
We have two password encoders, one for each algorithm.
The AuthenticationProviderService implements the authentication logic.


The MainPageController needs the ProductService to get the products it will display and the SecurityContext to show the name of the authenticated user on the page.
Request
Response
DispatcherServlet
/main
Handler mappings
SecurityContext
main()
MainPageController
ProductService
main.html
View resolver
ProductRepository
The ProductRepository gets the products from the database.

- `AuthenticationProviderService` class, which implements the `AuthenticationProvider` interface.

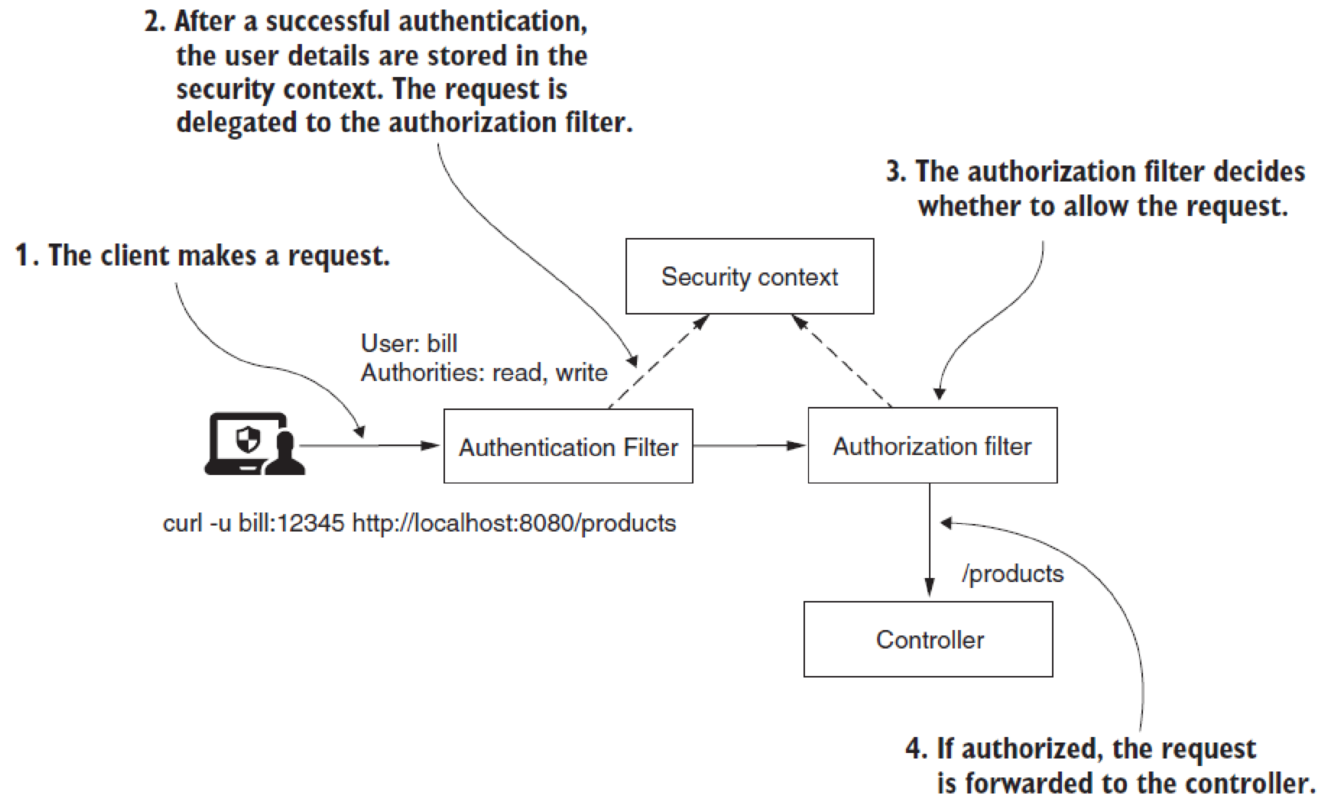
- This implementation defines the authentication logic where it needs to call a `UserDetailsService` to find the user details from a database and the `PasswordEncoder` to validate if the password is correct

For this application, a `JpaUserDetailsService` that uses Spring Data JPA to work with the database.

For this, we configure `formLogin` as the authentication method.

AUTHENTICATION AND AUTHORIZATION

- ❑Gain an understanding of what an authority is and apply access rules on all endpoints based on a user’s authorities.
- ❑ Learn how to group authorities in roles and how to apply authorization rules based on a user’s roles.


2. After a successful authentication, the user details are stored in the security context. The request is delegated to the authorization filter.
3. The authorization filter decides whether to allow the request.
1. The client makes a request.
Security context
User: bill
Authorities: read, write
Authentication Filter
Authorization filter
curl -u bill:12345 http://localhost:8080/products
/products
Controller
4. If authorized, the request is forwarded to the controller.

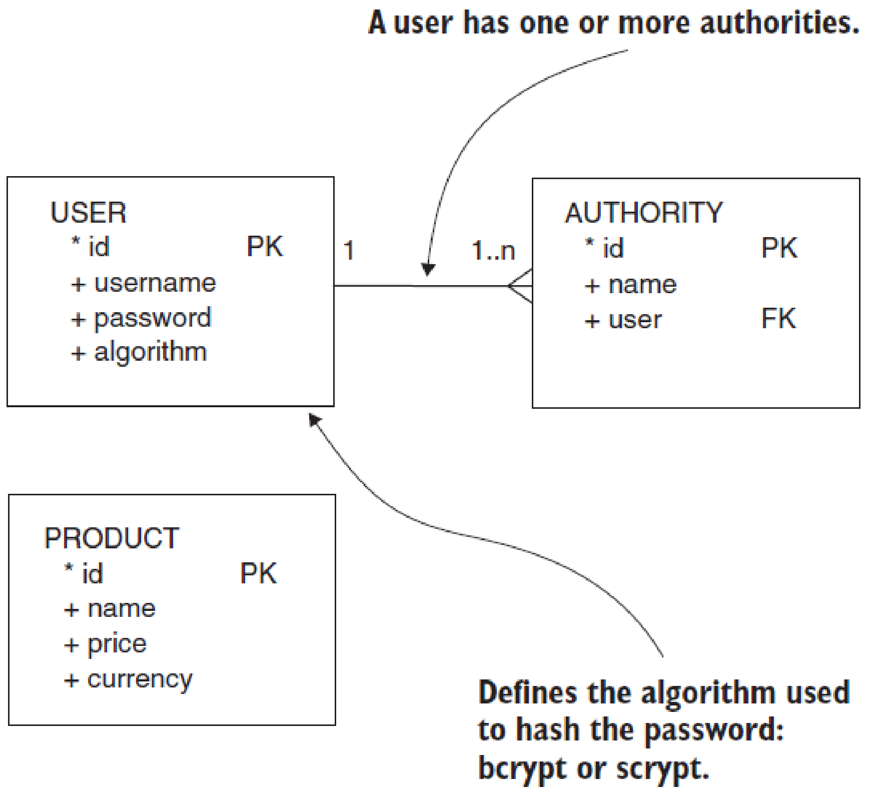
AUTHENTICATION

We need a `MainPageController` that defines the action that the application executes upon the request for the main page.

The `MainPageController` displays the name of the user on the main page, so this is why it depends on the `SecurityContext`.

It obtains the username from the security context and the list of products to display from a service that I call `ProductService`.

The `ProductService` gets the list of products from the database using a `ProductRepository`, which is a standard Spring Data JPA repository.



MAIN STEPS

1 Set up the database

2 Define user management

3 Implement the authentication logic

4 Implement the main page

5 Run and test the application

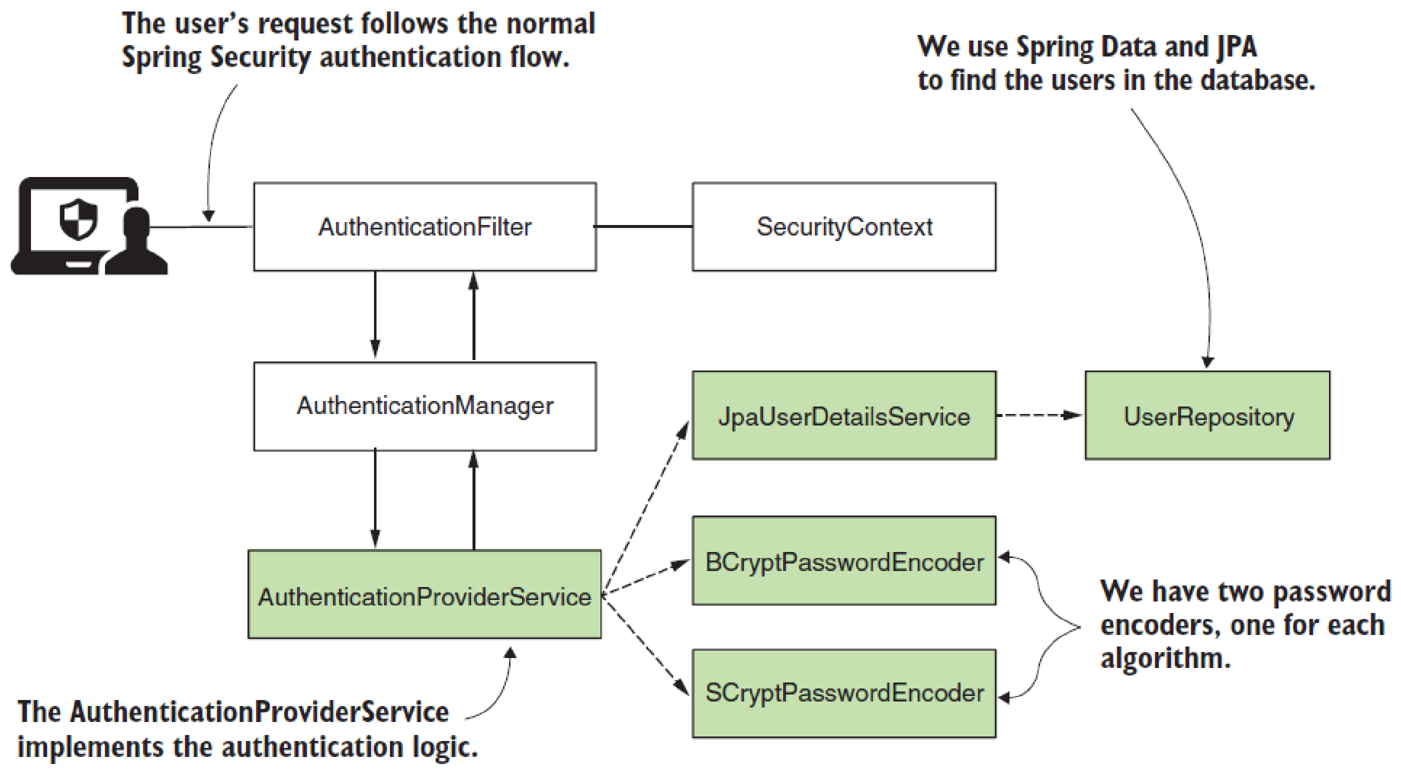
```
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
@Entity
public class User implements UserDetails{
@Id @GeneratedValue(strategy = GenerationType.AUTO)
private Integer id;
private String firstName;
private String lastName;
private String email;
private String password;
@Enumerated(EnumType.STRING)
private Role role;
@Override
public Collection<? extends GrantedAuthority>
getAuthorities() {
    return List.of(new
    SimpleGrantedAuthority(role.name()));
}
```

LOMBOK

```xml
<dependency>
<groupId>org.projectlombok</groupId>
<artifactId>lombok</artifactId>
<version>1.18.26</version>
<optional>true</optional>
</dependency>
<dependency>
<groupId>io.jsonwebtoken</groupId>
<artifactId>jjwt-api</artifactId>
<version>0.11.5</version>
</dependency>
<dependency>
<groupId>io.jsonwebtoken</groupId>
<artifactId>jjwt-impl</artifactId>
<version>0.11.5</version>
</dependency>
<dependency>
<groupId>io.jsonwebtoken</groupId>
<artifactId>jjwt-jackson</artifactId>
<version>0.11.5</version>
</dependency>
```

USER MANAGEMENT

We need to implement at least this `UserDetailsService` contract to instruct Spring Security how to retrieve the details of your users.


The user's request follows the normal Spring Security authentication flow.
We use Spring Data and JPA to find the users in the database.
AuthenticationFilter
SecurityContext
AuthenticationManager
JpaUserDetailsService
UserRepository
BCryptPasswordEncoder
AuthenticationProviderService
SCryptPasswordEncoder
We have two password encoders, one for each algorithm.
The AuthenticationProviderService implements the authentication logic.

1 Set up the database
2 Define user management
3 Implement the authentication logic
4 Implement the main page
5 Run and test the application

STEPS TO FOLLOW

1. Define the password encoder objects for the two hashing algorithms.

2 Define the JPA entities to represent the user and authority tables that store the details needed in the authentication process.

3 Declare the `JpaRepository` contracts for Spring Data. In this example, we only need to refer directly to the users, so we declare a repository named `UserRepository`.

4 Create a decorator that implements the `UserDetails` contract over the `User` JPA entity. Here, we use the approach to separate responsibilities

5 Implement the `UserDetailsService` contract. For this, create a class named `JpaUserDetailsService`.

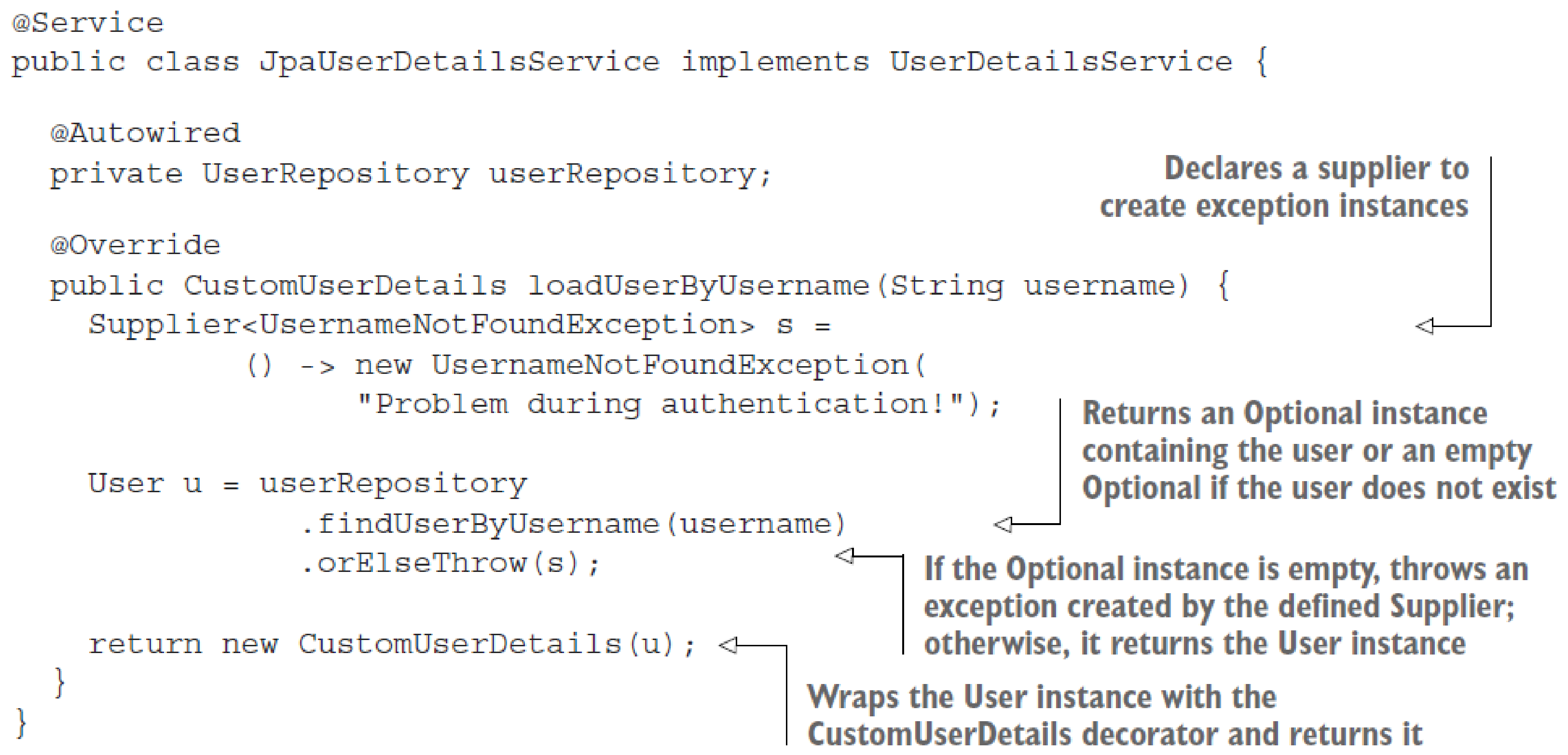
This class uses the `UserRepository` we create in step 3 to obtain the details about users from the database. If `JpaUserDetailsService` finds the users, it returns them as an implementation of the decorator we define in step 4.

@CONFIGURATION

```
@Configuration
@RequiredArgsConstructor
public class ApplicationConfig {

private final UserRepository repository;

@Bean
public UserDetailsService userDetailsService() {
return username-> repository.findByEmail(username)
.orElseThrow(()-> new UsernameNotFoundException("User not
found")); }

@Bean
public PasswordEncoder passwordEncoder() {
// TODO Auto-generated method stub
return new BCryptPasswordEncoder();}
}
```

1 Set up the database
2 Define user management
3 Implement the authentication logic
4 Implement the main page
5 Run and test the application

- For user management, we need to declare a `UserDetailsService` implementation, which retrieves the user by its name from the database.
- It needs to return the user as an implementation of the `UserDetails` interface, and we need to implement two JPA entities for authentication: `User` and `Authority`.



```
@Service
public class JpaUserDetailsService implements UserDetailsService {

  @Autowired
  private UserRepository userRepository;

  @Override
  public CustomUserDetails loadUserByUsername(String username) {
    Supplier<UsernameNotFoundException> s =
            () -> new UsernameNotFoundException(
                  "Problem during authentication!");

    User u = userRepository
               .findUserByUsername(username)
               .orElseThrow(s);

    return new CustomUserDetails(u);
  }
}
```

Declares a supplier to create exception instances

Returns an Optional instance containing the user or an empty Optional if the user does not exist

If the Optional instance is empty, throws an exception created by the defined Supplier; otherwise, it returns the User instance

Wraps the User instance with the CustomUserDetails decorator and returns it

- Having completed user and password management, we can begin writing custom authentication logic.
- To do this, we have to implement an `AuthenticationProvider` and register it in the Spring Security authentication architecture.
- The dependencies needed for writing the authentication logic are the `UserDetailsService` implementation and the two password encoders.

```
@Service
public class AuthenticationProviderService
  implements AuthenticationProvider {

  @Autowired
  private JpaUserDetailsService userDetailsService;

  @Autowired
  private BCryptPasswordEncoder bCryptPasswordEncoder;

  @Autowired
  private SCryptPasswordEncoder sCryptPasswordEncoder;

  @Override
  public Authentication authenticate(
    Authentication authentication)
      throws AuthenticationException {
        // ...
  }

  @Override
  public boolean supports(Class<?> aClass) {
    return return UsernamePasswordAuthenticationToken.class
        .isAssignableFrom(aClass);
  }
}
```

Injects the necessary dependencies, which are the UserDetailsService and the two PasswordEncoder implementations

The `authenticate()` method first loads the user by its username and then verifies if the password matches the hash stored in the database

CONNECTING TO THE MAIN APPLICATION

We need to register the `AuthenticationProvider` within the configuration class.

```
@Configuration
@EnableWebSecurity
@RequiredArgsConstructor
public class SecurityConfiguration {


private final AuthenticationProvider
authenticationProvider;
```

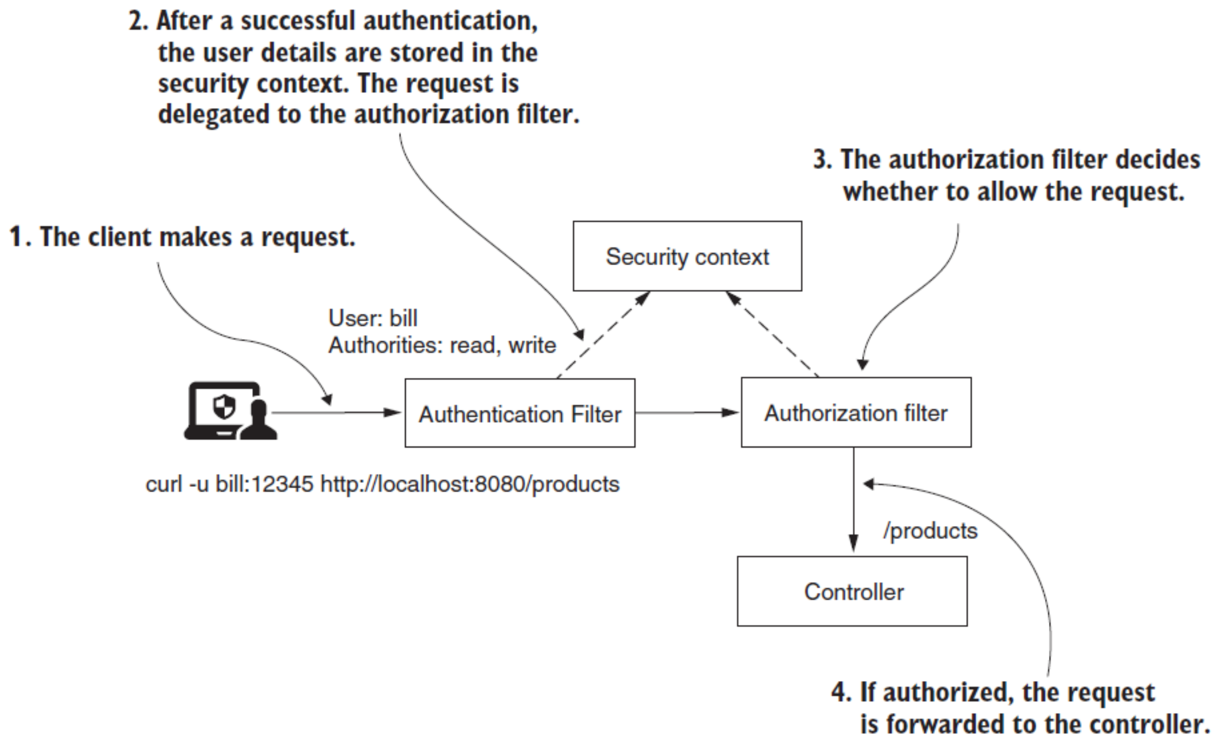
```
@Configuration
public class ProjectConfig extends
WebSecurityConfigurerAdapter {
@Autowired
private AuthenticationProviderService
authenticationProvider;
```

In the configuration class, we want to set both the authentication implementation to the `formLogin` method and the path /main as the default success URL

```
@Override
protected void configure(HttpSecurity
http)
throws Exception {
http.formLogin()
.defaultSuccessUrl("/main", true);
http.authorizeRequests()
.anyRequest().authenticated();
}
```

```
@Controller
public class MainPageController {
@Autowired
private ProductService productService;
@GetMapping("/main")
public String main(Authentication a, Model model) {
model.addAttribute("username", a.getName());
model.addAttribute("products", productService.findAll());
return "main.html";
}
}
```

AUTHORIZATION



2. After a successful authentication, the user details are stored in the security context. The request is delegated to the authorization filter.
3. The authorization filter decides whether to allow the request.
1. The client makes a request.
Security context
User: bill
Authorities: read, write
Authentication Filter
Authorization filter
curl -u bill:12345 http://localhost:8080/products
/products
Controller
4. If authorized, the request is forwarded to the controller.

- ❑A user has one or more authorities (actions that a user can do).
- ❑ During the authentication process, the **`UserDetailsService`** obtains all the details about the user, including the authorities.
- ❑The application uses the authorities as represented by the **`GrantedAuthority`** interface for authorization after it successfully authenticates the user

@CONFIGURE

```
@Override
protected void configure(HttpSecurity
http)
throws Exception {
http.formLogin()
.defaultSuccessUrl("/main", true);
http.authorizeRequests()
.anyRequest().authenticated();
}
```

```
@Configuration
public class ProjectConfig {
@Bean
public UserDetailsService
userDetailsService() {
var manager = new
InMemoryUserDetailsManager();
var user1 = User.withUsername("john")
.password("12345")
.authorities("READ")
.build();
```

```
http.authorizeRequests()

.anyRequest()

.hasAuthority("WRITE");
```

SETTINGS FOR 8443

```
keytool -genkey -noprompt -alias tomcat-localhost -keyalg RSA -keystore
C:\Users\chand\localhost-rsa.jks -keypass 123456 -storepass 123456 -dname
"CN=tomcat-cert, OU=JU, O=JU, L=WB, ST=WB, C=IN“
```

```
<Connector
        protocol="org.apache.coyote.http11.Http11NioProtocol"
        port="8443" maxThreads="200"
        scheme="https" secure="true" SSLEnabled="true"
        keystoreFile="C:\my-cert-dir\localhost-rsa.jks"
        keystorePass="123456"
        clientAuth="false" sslProtocol="TLS"/>
```

SPRING

```
 keytool -genkey -alias skipper -keyalg RSA -keystore
c:\User\user1\skipper.keystore   -validity 3650 -storetype JKS
-dname "CN=localhost, OU=Spring, O=Pivotal, L=Holualoa, ST=HI,
C=IN" -keypass skipper -storepass skipper
```

- ❑ This method generates the key needed for HTTPS.
- ❑ Excute this command from jdk/bin of your machine
- ❑ Move the generated keystore file to the “resources” folder of your application

CONFIDENTIALITY

Keep the following methods in the file where the main method is present

```
 @Bean
public ServletWebServerFactory servletContainer() {
TomcatServletWebServerFactory tomcat = new
TomcatServletWebServerFactory() {
@Override
 protected void postProcessContext(Context context) {
SecurityConstraint securityConstraint = new
SecurityConstraint();
securityConstraint.setUserConstraint("CONFIDENTIAL");
SecurityCollection collection = new SecurityCollection();
collection.addPattern("/*");
securityConstraint.addCollection(collection);
context.addConstraint(securityConstraint);            }
};
tomcat.addAdditionalTomcatConnectors(redirectConnector());
return tomcat;    }
```

CONFIDENTIALITY

Keep the following methods in the file where the main method is present

```
 private Connector redirectConnector() {
        Connector connector = new 
Connector("org.apache.coyote.http11.Http11NioProtocol");
        connector.setScheme("http");
        connector.setPort(8080);
        connector.setSecure(false);
        connector.setRedirectPort(8443);
        return connector;
    }
```

APPLICATION STARTUP

Spring is going to look at our application and the configuration that we're expressing in it, and then automatically, in this case, create a web container put a dispatcher servlet in that web container, and then auto discover all of our controllers in our application

An application where configuration is stated

- ☐ Controllers
- ☐ POJOs
- ☐ Views

@SpringBootApplication

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

@SPRINGBOOTAPPLICATION

- @EnableAutoConfiguration: enable Spring Boot’s auto-configuration mechanism
- @ComponentScan: enable@Componentscan on the package where the application is located
- @Configuration: allow to register beans in the context or import additional configuration classes

All of your application components (@Component,@Service,@Repository, @Controlleretc.) are automatically registered as Spring Beans

@ENABLEAUTOCONFIGURATION

- ❑ Spring Boot auto-configuration attempts to automatically configure our Spring application based on the jar dependencies added by the programmer
- ❑ Spring Boot adds @EnableWebMvc automatically when it sees spring-webmvcon the classpath. This flags the application as a web application and activates key behaviors such as setting up a DispatcherServlet.

- ❑Auto-configuration is non-invasive.
  - ❑At any point, you can start to define your own configuration to replace specific parts of the auto-configuration.

https://docs.spring.io/spring-boot/docs/2.0.x/reference/html/using-boot-auto-configuration.html

@ENABLEAUTOCONFIGURATION

- these four annotations on this configuration class go and set up an entire web container
  - they create the DispatcherServletthat we need to route requests to our controllers.
  - They automatically scan the appropriate packages that we want, and discover our controllers,
  - they'll automatically configure our controllers with any dependencies that we want them to have.

APPLICATIONCONTEXT

- If Dependency Injection is the core concept of Spring, then the ApplicationContextis its core object.
  - The interface `org.springframework.context.ApplicationContext`represents the Spring IoC container
  - It extends the`BeanFactory`interface, in addition to extending other interfaces to provide additional functionality in a more*application framework-oriented style*
  - The container gets its instructions on what objects to instantiate, configure, and assemble by reading configuration metadata
  - Several implementations of the`ApplicationContext`interface are supplied out-of-the-box with Spring
  - such asContextLoaderthat automatically instantiates an`ApplicationContext`as part of the normal startup process

https://docs.spring.io/spring-framework/docs/3.0.0.M4/spring-framework-reference/html/ch15s02.html