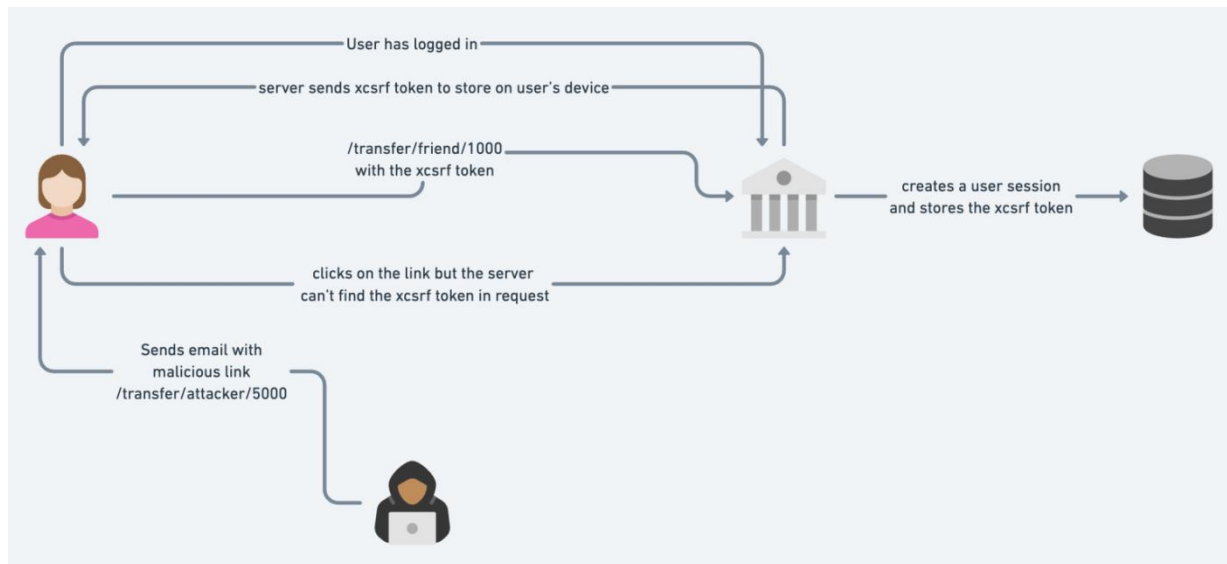


➤ **CSRF**

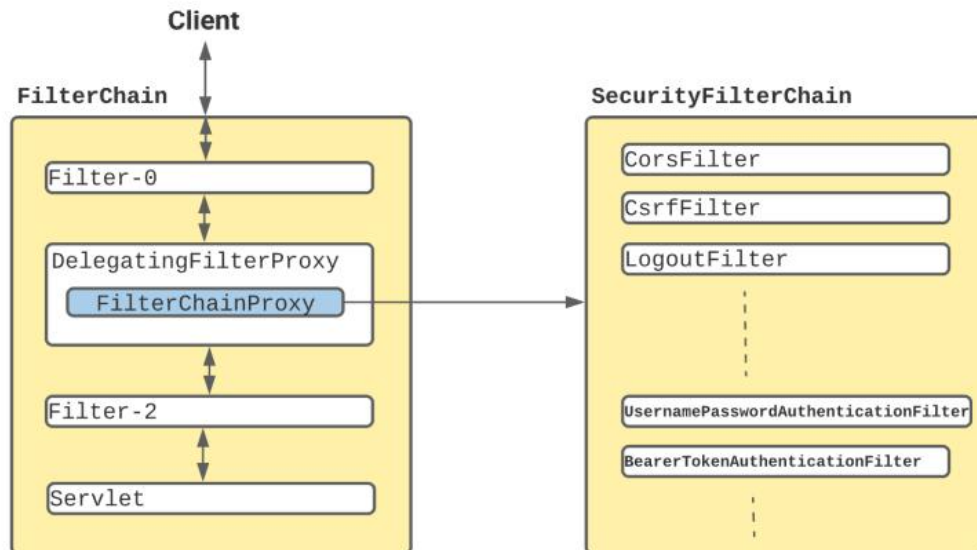
- ⌘ CSRF (Cross-Site Request Forgery) is an attack where a malicious website tricks a logged-in user's browser into sending an unauthorized request to a trusted website using the user's existing session/cookies.
- ⌘ So basically, **CSRF** attacks doesn't steal password; It uses user's cookies stored in the browser to steal the data/money.
- ⌘ So, to prevent CSRF, either you need to pass some unique token in the request headers or make the request bind with a particular website (means other website cannot send that request)
- ⌘ Someone can steal your cookies, but they don't have your headers.

➤ With **csrf** token, it can be prevented.

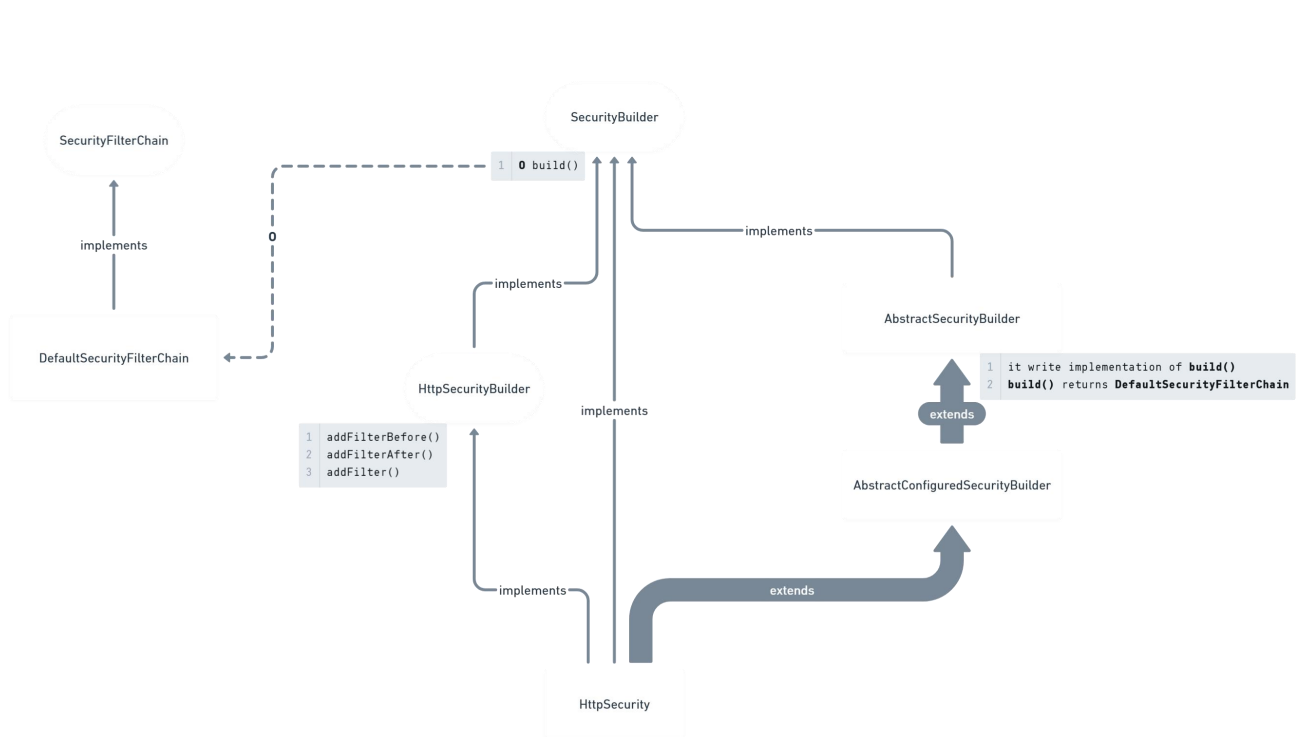


Internal Working Of Spring Security

- The dependency **spring-boot-starter-security** has to be added in pom.xml which groupId is **org.springframework.boot**
- After that spring-boot will auto-configure the security with sensible defaults defined in **WebSecurityConfiguration** class.



- - ⌘ In Spring Boot application, **SecurityFilterAutoConfiguration** automatically registers the **DelegatingFilterProxy** filter with the name **springSecurityFilterChain**.
 - ⌘ Once the request reaches to **DelegatingFilterProxy**, Spring delegates the processing to **FilterChainProxy** bean that utilizes the **SecurityFilterChain** to execute the list of all filters to be invoked for the current request.
- Default behaviour of Spring-Security
 - ⌘ Creates a bean named **springSecurityFilterChain** & registers the **filter** with a bean named **springSecurityFilterChain** with the Servlet container for every request.
 - ⌘



Made with Whimsical

- This is the flow of Security Filters.
- In the class **WebSecurityConfiguration**, the beans are created.
 - ⌘ The bean of **HttpSecurity** is created.

```

@Autowired(
    required = false
)
private HttpSecurity httpSecurity;

```

- ⌘ One **HttpSecurity** type of object can build only one **SecurityFilterChain**.

- **HttpSecurity** contains **build()** method that returns object of type **DefaultSecurityFilterChain**. Then why Spring needs to create a bean of **SecurityFilterChain** ?

- ⌞ **WebSecurityConfiguration** has a list of **SecurityFilterChain**

- ⌞

```
private List<SecurityFilterChain> securityFilterChains = Collections.emptyList();
```

- ⌞ Here one method is there to create the bean named as **springSecurityFilterChain**.

- ⌞

```
@Bean(  
    name = {"springSecurityFilterChain"}  
)  
public Filter springSecurityFilterChain() throws Exception {
```

- ⌞ But here, **Filter** is the return type; confusing; will discuss at the end :)

- ⌞ If there is no filter chains, it'll add the default chain from *HttpSecurity* build() method.

- ⌞ it'll not add the default chain to the list i.e. **securityFilterChains**, rather it is being added to **webSecurity** object.

- ⌞

```
public final class WebSecurity extends AbstractConfiguredSecurityBuilder<Filter, WebSecurity> implements SecurityBuilder {  
    private final Log logger = LogFactory.getLog(this.getClass());  
    private final List<RequestMatcher> ignoredRequests = new ArrayList();  
    private final List<SecurityBuilder<? extends SecurityFilterChain>> securityFilterChainBuilders = new ArrayList();
```

- ⌞ It is being added here (WebSecurity class).

- ⌞

```
if (!hasFilterChain) {  
    this.webSecurity.addSecurityFilterChainBuilder(() -> {  
        this.httpSecurity.authorizeHttpRequests(( AuthorizationManager) authorizationManager -> {  
            this.httpSecurity.formLogin(Customizer.withDefaults());  
            this.httpSecurity.httpBasic(Customizer.withDefaults());  
            return (SecurityFilterChain) this.httpSecurity.build();  
        });  
    });
```

- ⌞ If filter chains are already there, then it add those filter chains to **webSecurity** object.

- ⌞

```
for (SecurityFilterChain securityFilterChain : this.securityFilterChains) {  
    this.webSecurity.addSecurityFilterChainBuilder(() -> securityFilterChain);  
}
```

- ⌞ At the end it'll return an object of type **Filter**

- ⌞

```
return (Filter) this.webSecurity.build();
```

➤ Spring Filter Behind the Scene

- What we see, **chain of filters** are being executed in between **Servlet Container** and **Servlet**.
 - ⌘ One **Servlet** containers can have many filters, many servlets; but in case of spring we have only one Servlet which is **DispatcherServlet**.
 - ⌘ And, its not by limitation, but by design spring make sure only one **Filter** should be there in between **Servlet Container** (tomcat in our case) and **Servlet** (DispatcherServlet).
 - ⌘ So, only **one Filter** should be able to handle **multiple Filter Chains** where **each chain contains multiple Filters**.
 - ⌘ its like **one object** is equivalent to **list of list of the same object**.
 - ⌘ So, **FilterChainProxy** was introduced which implements **Filter**. And it contains the list of **SecurityFilterChain** objects.
 - ⌘ And the thing is, this **SecurityFilterChain** class contains a method **getFilters()** which returns a list of **Filter** objects.
 - ⌘ So now, we can return **FilterChainProxy** object instead of **Filter** because **FilterChainProxy** is nothing but the child of **Filter**.
 - ⌘ And also it contains *list of SecurityFilterChain* which means **list of (list of (Filter))**.
- If you want to create your own **filter chains**.
 - ⌘ Don't override the bean creation of the bean **springSecurityFilterChain**, otherwise spring will only create your bean and all the necessary steps like the below will be skipped.
 - ⌘ Adding the filter chain to **webSecurity**.
 - ⌘ **customize** using the **Customizer** object..
 - ⌘ This all will have to be implemented in your own bean creation method.
 - ⌘ So, create beans of type **SecurityFilterChain**

```
@Bean
SecurityFilterChain apiChain(HttpSecurity http) { ... }

@Bean
SecurityFilterChain webChain(HttpSecurity http) { ... }
```

- ⌘ Now you might be thinking, if we are creating **multiple beans of same type**, spring will be confused which bean has to be created; but in this case we

spring has **list of SecurityFilterChain** not a single object of *SecurityFilterChain*, so all the beans will be added to that list.

```
void setFilterChains(List<SecurityFilterChain> securityFilterChains) {  
    this.securityFilterChains = securityFilterChains;  
}
```

*

* This method is inside **WebSecurityConfiguration** class.

* All the **SecurityFilterChain** objects that you created beans of, will be passed to this *setFilterChains* method.

- Now, when the list i.e. **securityFilterChains** (present inside *WebSecurityConfiguration.class*) has already some elements present, so the default **filter chain** will not be added to this.

```
if (!hasFilterChain) {  
    this.webSecurity.addSecurityFilterChainBuilder(() -> {  
        this.httpSecurity.authorizeHttpRequests(( AuthorizationMana  
        this.httpSecurity.formLogin(Customizer.withDefaults());  
        this.httpSecurity.httpBasic(Customizer.withDefaults());  
        return (SecurityFilterChain) this.httpSecurity.build();  
    });  
}
```

*

*

* F

* F

* F

* F

*

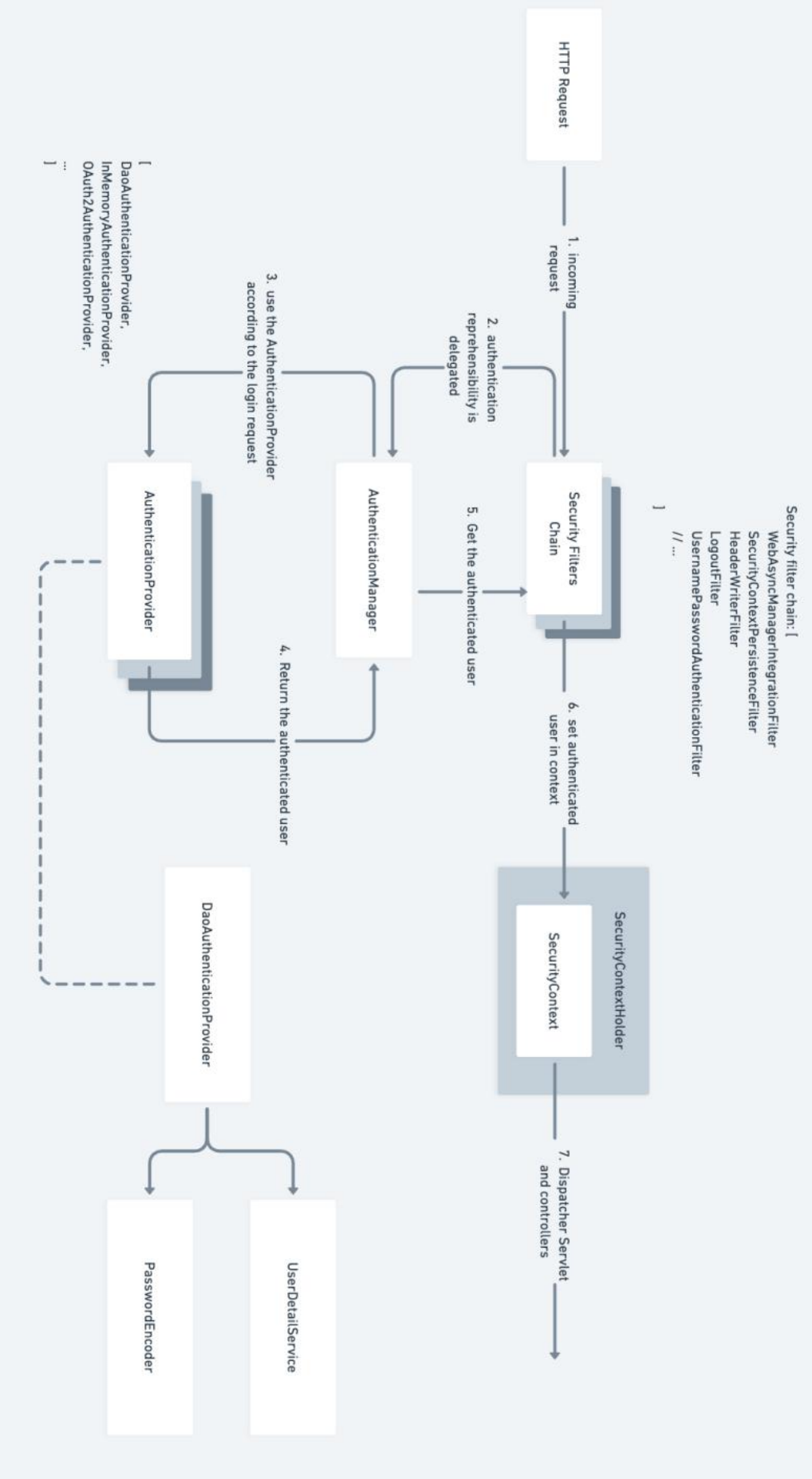
- When you run the application after including the dependency **spring-boot-starter-security**, by default one login page will appear to authenticate you.
 - ⌘ If you inspect that, you'll find one *hidden input* field containing the csrf token as its value which is being attached to request.

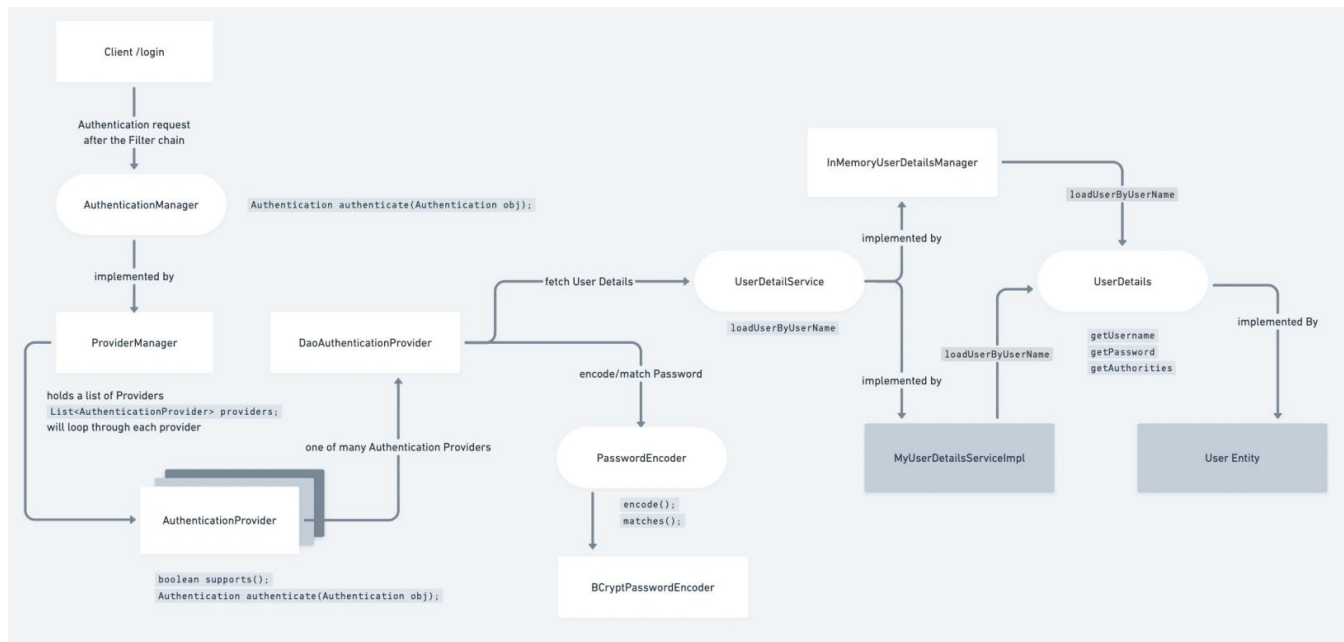
```

▶ <p>⋮</p>
  <input name="_csrf" type="hidden" value="DmL2Ac
  <button type="submit" class="primary">Sign in</button>

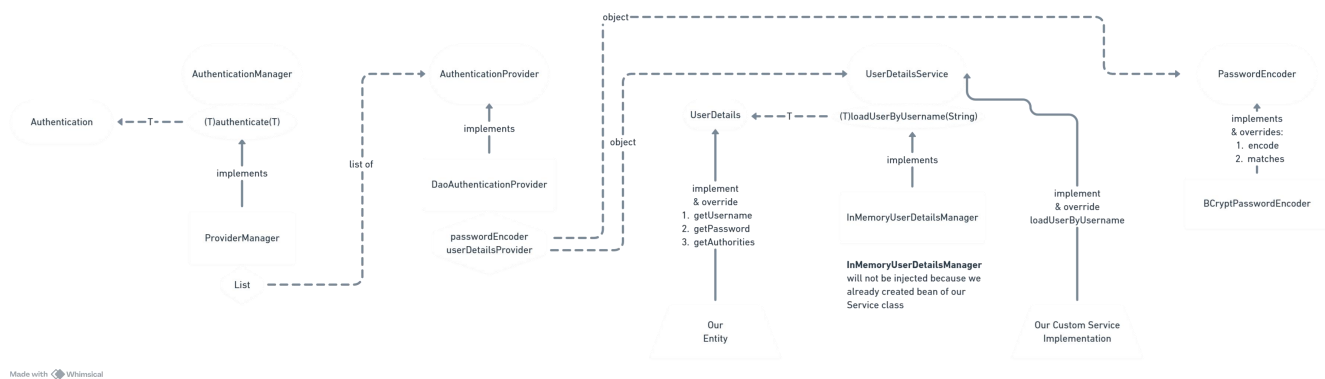
```

- ⌘ Means every time any request being sent, **session id** (cookie) along with **csrf token** (request header) are also sent.
- Default Security Filters configurations
 - ⌘ **SecurityFilterChain** is an interface containing list of filters.
 - ⌘ **DefaultSecurityFilterChain** implements *SecurityFilterChain* and initializes the filters inside its constructor.
 - ⌘ But someone needs to call this constructor passing the **filters** as argument to initialize the filters.
 - ⌘ **HttpSecurity** class extends **AbstractSecurityBuilder** class which contains **build()** method which returns an object of type **DefaultSecurityFilterChain**.
 - ⌘





(rectangles: classes, ellipse: interfaces)



Flowchart of Authentication

- **AuthenticationManager** is an interface.

```
public interface AuthenticationManager {  
    Authentication authenticate(Authentication authentication)  
}
```

- ⌘ **Authentication** is also an interface containing necessary methods

```
public interface Authentication extends Principal, Serializable {  
    Collection<? extends GrantedAuthority> getAuthorities(); 1 imp  
  
    Object getCredentials(); 10 implementations  
  
    Object getDetails(); 1 implementation  
  
    Object getPrincipal(); 10 implementations  
  
    boolean isAuthenticated(); 1 implementation  
  
    void setAuthenticated(boolean isAuthenticated) throws IllegalA  
}
```

- ⌘ After the authentication done,
- ⌘ **isAuthenticated** will be marked as true.
- Now, so many classes implement **AuthenticationManager**, one of those is **ProviderManager**
 - ⌘ It holds a *list* of **AuthenticationProvider** (which is also an interface)
- **DaoAuthenticationProvider** implements **AuthenticationProvider**
 - ⌘ It holds the **UserDetailsService** and **PasswordEncoder** type of objects.
- **UserDetailsService**
 - ⌘ We will implement this interface from our custom **Service** class and implement the **loadUserByUsername** method.
 - ⌘ This method use **UserDetails** type of object.
- **UserDetails**
 - ⌘ Our entity should implement this interface.
 - ⌘

➤ Writing custom Filter Chains

- We'll create the custom filters like the below

```
@Bean no usages
SecurityFilterChain securityFilterChain(HttpSecurity httpSecurity)
    httpSecurity
        .formLogin(Customizer.withDefaults());
    return httpSecurity.build();
}
```

⌘

- ⌘ You might be thinking, while creating the bean, how this **HttpSecurity** type of objects is being coming; as according to what we were doing earlier, it didn't have any argument in the `@Bean` method.

- ⌘ You can mention any type of argument you want in the `@Bean` method, if that type of **bean** is created, then Spring will automatically inject that bean to your `@Bean` method's argument.

➤

- ⌘ F

Udemy Part

- **Client --- Servlet Container --- filter-1 --- filter-2 --- --- filter-N --- Servlet**
 - ⌘ 2 way (request and response)
- If you include **spring-boot-starter-security**, whenever you'll try to hit any api endpoint, it'll redirect you to the login page;
 - ⌘ You need to authenticate yourself with username and password then one **session id** will be generated and will be stored in the *browser cookies*.
 - ⌘ whenever you'll hit any end-point, that **session id** will also be attached with the request.
 - ⌘ By default username id **user** and password will be generated when you'll run the spring boot application.
 - ⌘ For customized username and password, you can write those in the **application.properties** file.

```
spring.security.user.name=alok
spring.security.user.password=1234
```

- In the controller, you can get **HttpServletRequest** type of object which contains details like session id and all.

```
@GetMapping("hello") no usages
public String greet(HttpServletRequest request) {
    System.out.println(request);
    return "Hello " + request.getSession().getId();
}
```

- Get request working; post not working because we are not sending csrf token;
- When you trigger get request, **csrf** token is not required because its read-only;
 - ⌘ But, for remaining operations, you need to pass **csrf** token in the *headers*.

```
@GetMapping("csrf-token") no usages
public CsrfToken getCsrfToken(HttpServletRequest request) {
    return (CsrfToken) request.getAttribute(s: "_csrf");
}
```

- ⌘ I created this end-point to get the **csrf** token.

```
<p>⋮</p>
<input name="_csrf" type="hidden" value="DmL2Ac
<button type="submit" class="primary">Sign in</
```

- ⌘ If you see the default login page generated by **spring security**, the name will be **_csrf** here.

- So the attribute name is `_csrf`

```
{  
  "parameterName": "_csrf",  
  "token": "6aJLzII0QYJGEYALvxXL",  
  "headerName": "X-CSRF-TOKEN"  
}
```

- While triggering POST request via *postman*, you need to pass **X-CSRF-TOKEN** in the headers.

- It is one way of solving CSRF issue; other way is “Don’t allow any other website to use your session ID”

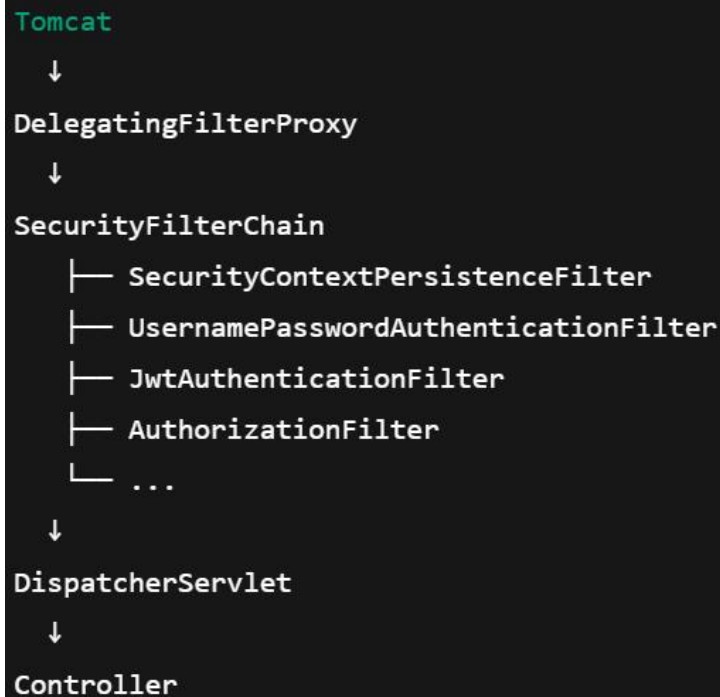
```
server.servlet.session.cookie.same-site=strict
```

- Add this in your **application.properties** file and it’ll restrict other website to use your session id.
- There are 2 types of application: **stateful** and **stateless**.
 - The one we were using now was **stateful**, because it was using same *session ID* for all the requests.
 - In case of **stateless** we need to pass the **username & password** in each request; so there is no need of **csrf token** here.



➤ Spring Security Working

- The filters are present in between the **Tomcat** and **DispatcherServlet**



➤ @EnableWebSecurity

- ♣ It tells Spring “Activate Spring Security’s filter chain for web requests”.
- ♣ Without it, no security filters are applied.
- In **Spring Boot**, spring security filters are auto-configured if the dependency **spring-boot-starter-security** is present in the pom.xml
- If you create a **bean** of type **SecurityFilterChain** then Spring will not create bean; means basically you did override the bean creation.

```
@Configuration no usages
@EnableWebSecurity
public class SecurityConfig {

    @Bean no usages
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        return http.build();
    }
}
```

- ♣ Now the filters will not be executed; you need to mention those filters.
-
- F
- F
- F

➤ **F**

➤ **F**

► **F**

► **F**

➤ **F**

► **F**

➤ **vf**