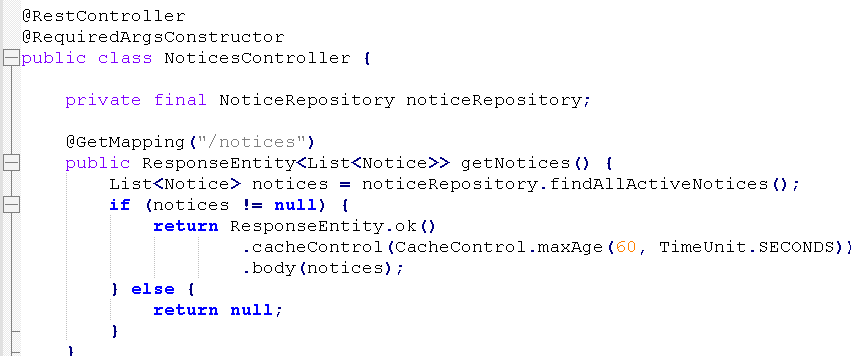
**7.cors, csrf - section8**

These attacks, will be possible only through the browsers.

If you have a backend API and these backend APA is being invoked

by another backend API, then these attacks, they're not possible.

Since these attacks can only be demoed using an UI application.



So within 60 seconds, if the end user trying to refresh the notices page multiple times, the backend API invocation is not going to happen.

Instead the browser, it's going to show the notices data from the cache itself.

Whereas if the end user try to refresh the notices page after the 60 seconds, then the UI application is going to make an actual backend API call to load the notice details from the database.

If you have a scenario where the response data may not change very frequently in such scenarios you can tell to the client applications to cache that data

for a specific period of time.



So let me go to the ProjectSecurityConfig files.

So here I'll try to mention /user as authenticated(), the same kind of change I'll try to do inside ProjectSecurityProdConfig as well.

**With this, you may have a question which is, if you are making the user API as authenticated, then how come the UI application can try to invoke**

**this API during the login operation?**

Let's try to understand the same.

Since I made this API as authenticated, the Spring Security framework will try

to authenticate the request by looking into the RequestHeader for an authorization header.

As long as a UI application is sending the user credentials inside the authorization header in the format of HttpBasic, the Spring Security framework is going to take care of authenticating the end user automatically.

Since UI application is trying to invoke and secure API. Once the authentication is successful, this API is going to be invoked automatically by the Spring Security framework because the client is actually trying to invoke this API.

In between, Spring Security tried to intercept it and tried to perform the authentication operation.

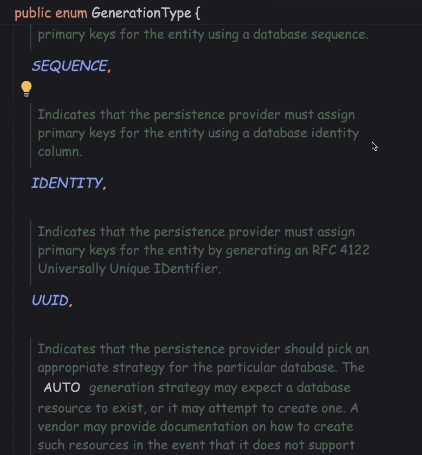
Once the authentication operation is successful, Spring Security will realize

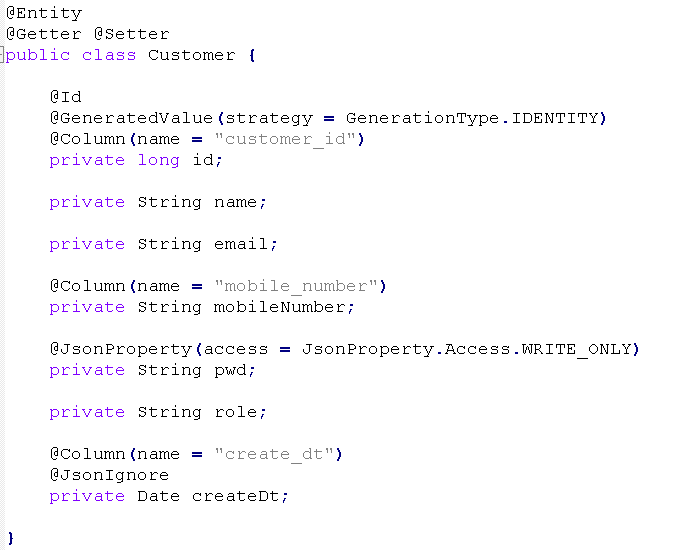
that the client application originally tried to invoke the user API.

That's why it is going to forward the request to the user API.

When this method is invoked, we are trying to automatically inject the authentication object. Inside this authentication object, we are going to have what is the name

of the current authenticated user.





Here, if you see, we have used this @Id annotation.

With this, what is happening behind the scenes is my JPF framework is giving the responsibility of generating the customerId to the database server completely.

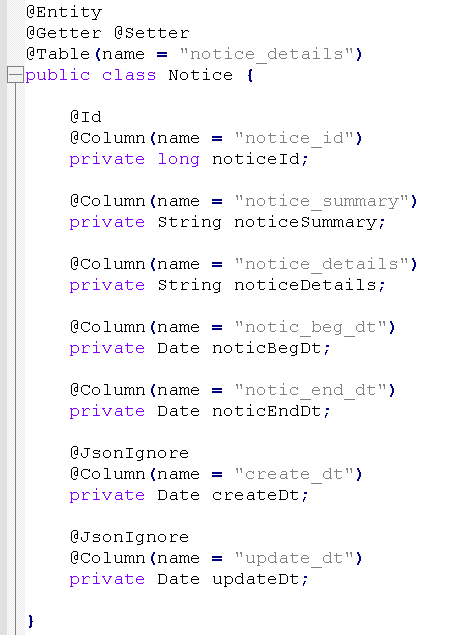
But the database server, it is not communicating the same back to the JPF framework about the customerId.

So to resolve it we just have to add one more annotation, which is @GeneratedValue with the strategy as GenerationType.IDENTITY.

So with this strategy, what we're telling to the Spring Data JPA framework

is it should assign the primary keys for the entity using a database identity column.

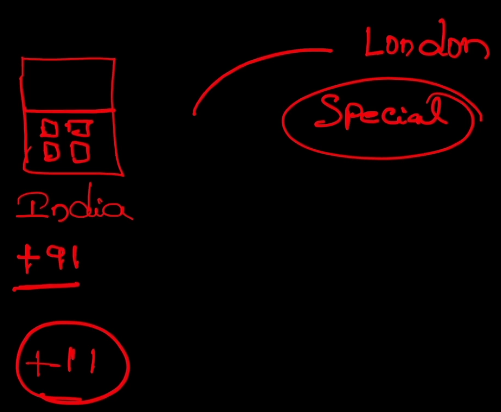
Now the Spring data JPA framework is going to take the responsibility of loading the customerId value by connecting with that database server.



we just have to mention an annotation, which is JsonIgnore.

With these, whenever this object is getting converted into a JSON object, these fields are going to be ignored. And with that, the client application, they're never going to receive these fields.

Introduction to CORs



Suppose think like you are staying in a country, for example, I'm staying inside the India. So all the phone numbers that are registered in India,

they're going to have a common code which is plus nine one.

So this is the country code of India. If I receive from any call from an Indian member,

there's a good chance that I would always lift the call without any thought.

It can be a marketing call or it can be my friend.

But let's imagine I'm receiving a call from some other country which has a country code as plus 11 inside the mobile number. So in these kind of scenarios, I would be very cautious to lift the call because there is a good chance that one

of the scammer might be calling me to scam me.

Let's imagine that my brother is staying inside London. Definitely whenever my brother want to call to me, there's a good chance that I might have already saved

his number inside my contacts. So without any thought, I will be lifting the call,

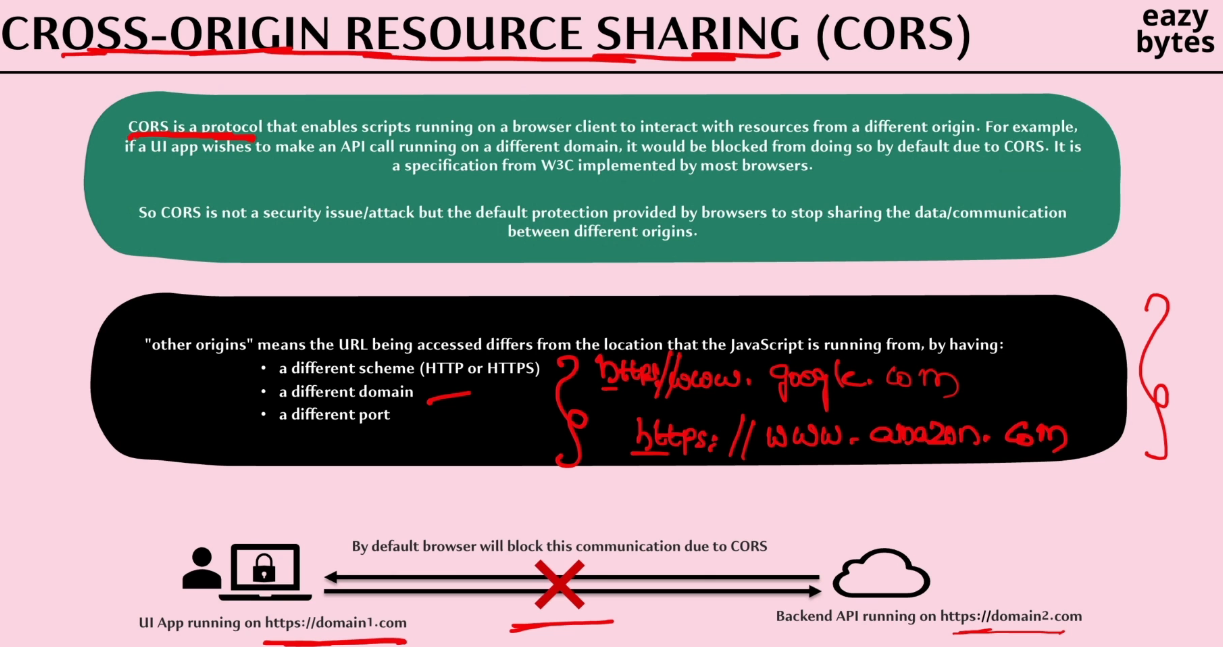
even though the call is coming from the another country.

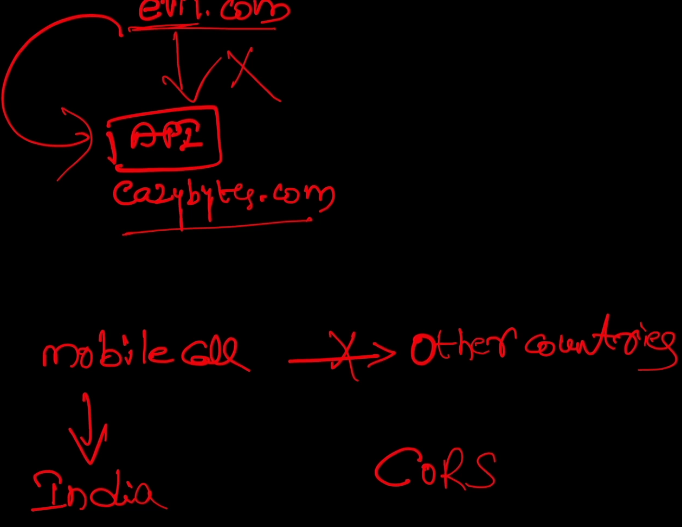
Even if I don't save his contact details, I know by looking at the country code

that this call is coming from the London and there is a good chance that my brother

or his family members are trying to reach out to me. So this is a special scenario where I will try to attend the call without any thought, though the call is coming from another country, whereas in the remaining scenarios where I'm getting a call from all other different countries, then I would be very cautious and I will not lift the call.

So this is a real life scenario. I'm going to compare this with the CORS in few minutes.





Inside the mobile scenario, I'm trying to always accept the mobile call

if it is coming from the same country. So if it is coming from the India,

I'm always trying to accept the call, even the browser is always going to accept the communication between a UI application and the backend application

if they're deployed at the same origin.

Whereas if the calls are coming from the other countries, I'm trying to be over cautious and I will not accept any calls.

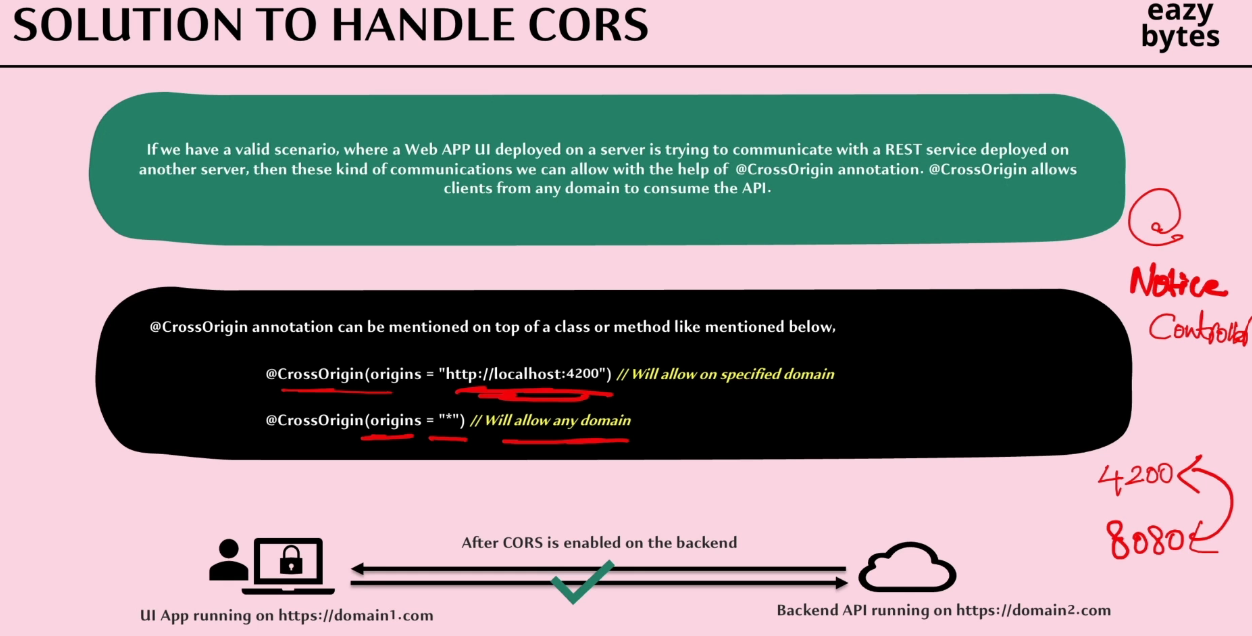
The same is happening inside the CORS as well. So the browser is trying to be cautious here, and by default, it'll stop the communication between the origins.

So if anyone says that CORS is a security attack, please stop them and let them know that CORS is not a security threat or a security attack.

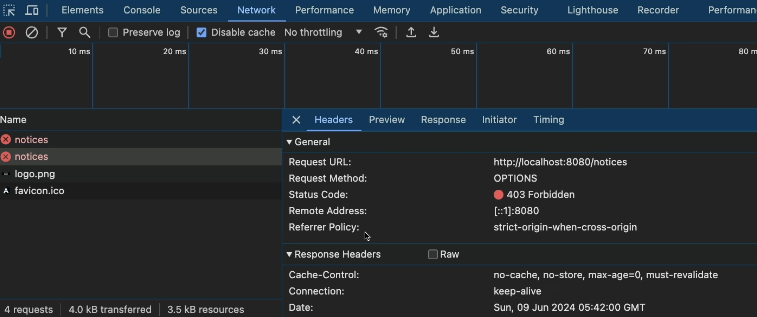
It is a protection provided by the browser by blocking the communication

between the different origins. This way, most of the security threats from the hackers will be blocked by the browsers itself.

Possible options to fix the CORs issue







Whenever a browser detects that the traffic is going to the other origin,

then in such scenarios, it is going to make a preflight request.

So you can see here there is a preflight request.

So what is this preflight request?

It is the request that is going to send by the browser to the backend server before the actual API request. As part of this preflight request, the browser is going to look

for the CORS related configurations from the backend server.

If the backend server replied saying that, "Okay, I'm fine accepting the traffic

from so and so origin." Then only the actual API call is going to be sent by the browser. Otherwise, it is going to block the traffic with the CORS related error.

You'll be able to see there is a preflight type request. If you try to click on this preflight request, you'll be able to see all the requests that is sent by the browser

as part of the preflight request.

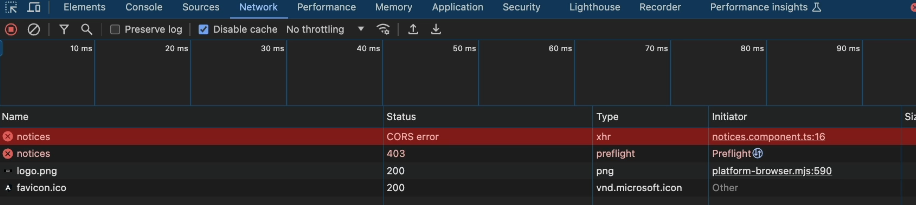
To this requests, the backend server has to send the proper response headers

indicating all the accepted origin details. Since right now our backend server

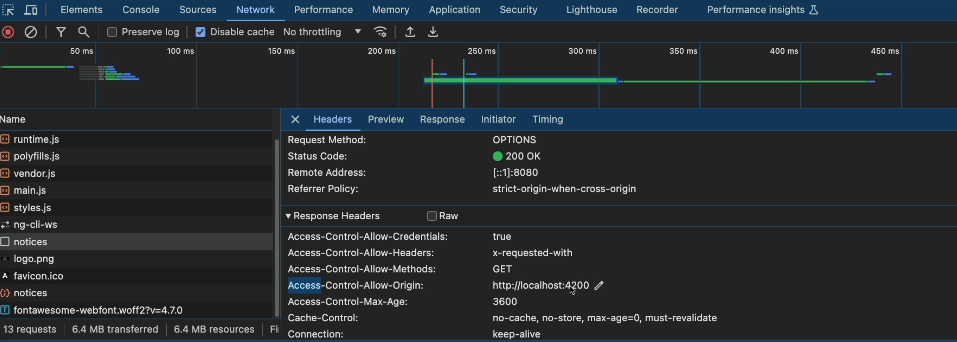
is not providing all these details, the preflight requests failed with the 403 errors.

Since the preflight request fail, the actual request is also going to fail

the status CORS error.



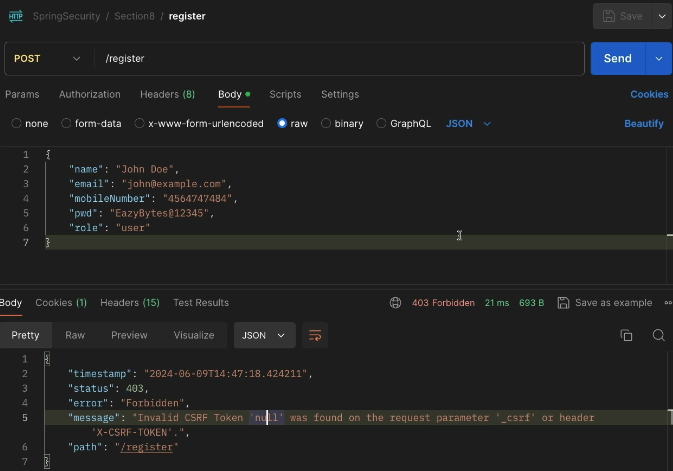
Fixing CORs issue using Spring Security



Demo of default CSRF protection inside Spring Security

Unlike CORS, CSRF is a proper security attack that leveraged by the hackers to steal the data inside your applications or to perform some privileged actions

inside your application.



We are getting this error because we are trying to create a new data

with the help of post method, but we are not passing any CSRF token

as part of our request.

Whenever the CSRF token is missing, the Spring Security will assume that someone is trying to attack our application with the help of CSRF attack.

That's why it stopped processing this request by throwing the 403 error.

The same kind of behavior we can also observe from the UI application as well.

Spring Security gave a flexibility by allowing all the GET API methods,

even though if you don't implement any CSRF related solution.

But coming to the POST, PUT methods, or DELETE methods, it is going to stop.

Introduction to CSRF attack

The full description of CSRF is Cross-Site Request Forgery.

So, what is the meaning of forgery?

If someone tried to exploit you without your knowledge, we call such kind of activity as forgery. Think like someone got you a checkbook.

If they try to forge you a signature and try to withdraw the funds from your bank account, then it is going to be considered as a forgery because some bad person is trying to exploit your money without your knowledge.

The same kind of concept works inside the CSRF attack as well.

So inside this CSRF attack, what is going to happen is the hacker, he will try to perform an operation in a web application on behalf of a user without their explicit consent. The end user will never know that hacker is performing CSRF attack on him or her.

Inside this attack, the hacker will never steal the username, password of the end user

or the JSESSIONID or the cookie of the end user. Instead, he will trick the end user

or he will exploit the end user to carry out an action without their will.

Example

So, initially the end user, he will try to log in into the netflix.com with his own credentials. As soon as the end user enter his credentials, the backend server of the netflix.com is going to authenticate the user if the credentials are valid.

It is also going to provide a cookie which will store inside the browser

against that domain name of netflix.com.

Why netflix.com is doing this?

So that it can leverage this cookie for further actions that are going to be performed

by the end user without asking any credentials.

The beauty of cookies are they're usually bound to a domain name.

In this scenario, the domain name will be netflix.com.

So, think like one of the cookie got created with the value as abc123.

So, this a cookie value. Let's imagine this value.

So, this cookie is mapped to the netflix.com.

So, the job of the browser is to attach this cookie whenever a request is going to the netflix.com.

For example, inside the same browser, if I try to make a request to some other website like amazon.com or facebook.com, then in all such scenarios,

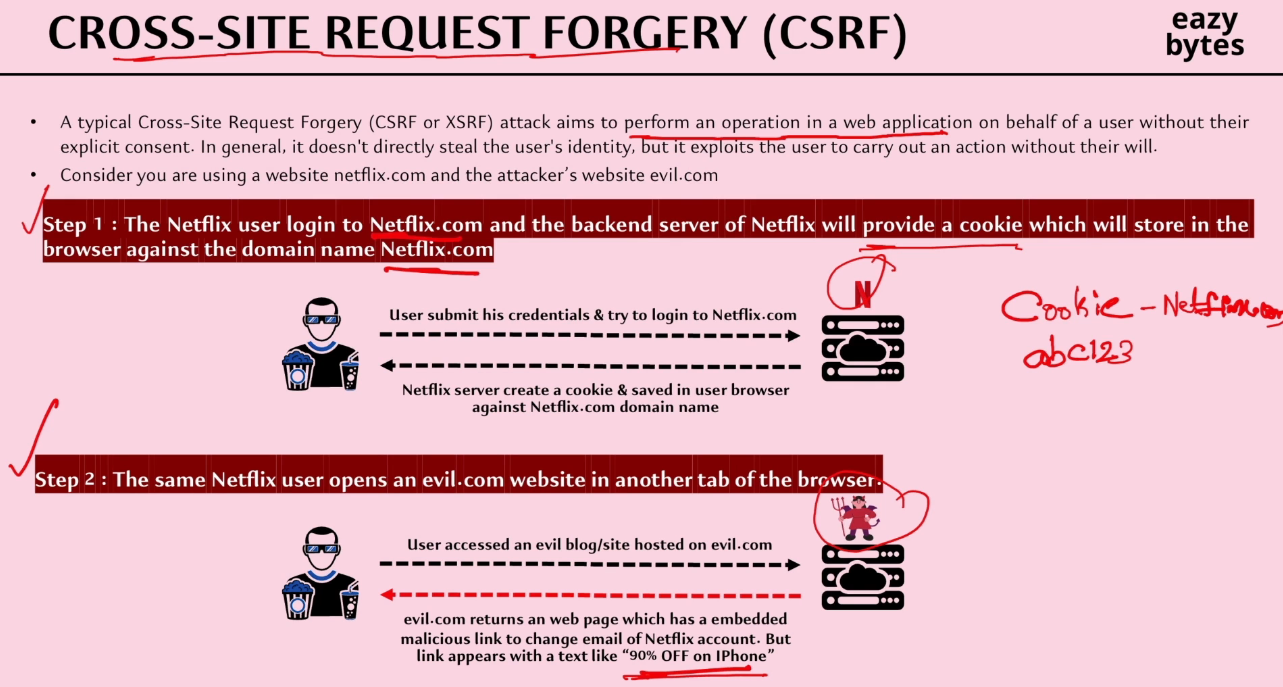
this cookie will never be attached to the request that is going to the other domains.

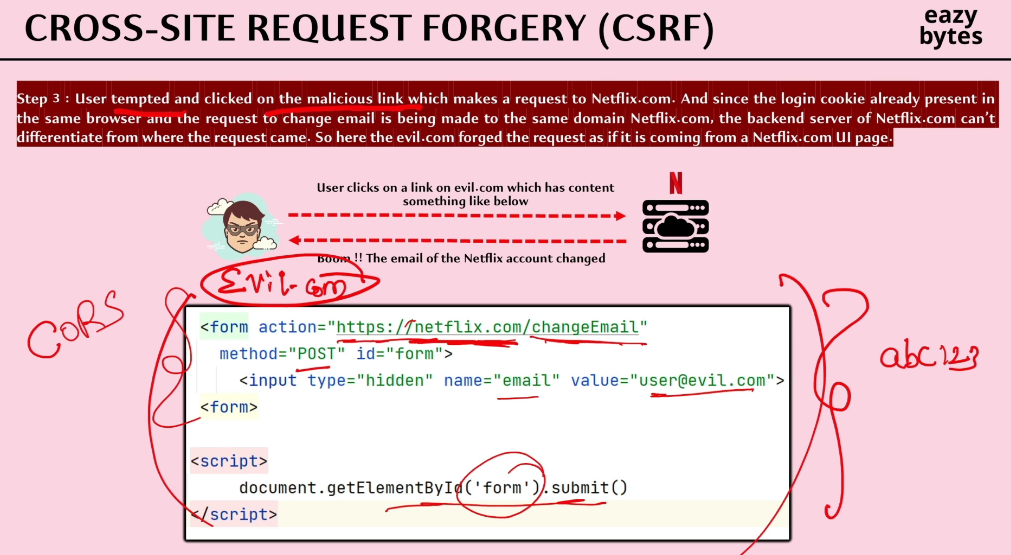
So, that's the beauty of cookie.

So inside this step one, nothing wrong as an end user.

You just used your account and you're trying to watch a movie.

But behind the scenes, Netflix created a cookie and stored inside your browser.





Now, let's try to see what is happening inside the step two.

Maybe after watching the movie, the end user decided to browse the internet.

He did not close the entire browser. Instead, he opened the new tab inside the same browser and opened one of his favorite website, which is evil.com.

So, this evil.com is maintained by the hacker.

So inside the step two, when the end user trying to access the evil.com website,

he will get that response from the evil.com backend server. And with the help of the response, a beautiful HTML page is going to be displayed on the browser.

Inside the page of the evil.com, there is a hyperlink or there is some banner

highlighting that "90% off on iPhone." So the hacker, what he's trying to do here

is he's trying to tempt the end user to click on this link or on the banner by providing some outstanding offers.

Most of the users, they will end up clicking on this banner or on this link

because everyone want iPhone at 90% off discount.

Let's consider our end user, our innocent user decided to click on this link.

And with that, let's see what is going to happen.

Inside the step three, like I said, the user tempted and clicked on the malicious link

provided by the hacker inside their website, evil.com. So behind this link, what hacker might have done is he might have built a embedded form

which can't be seen by the end user. So, as soon as someone click on that link,

this script is going to be executed. So inside his HTML page, he will have some hidden form or he will have a form which can't be seen by the normal end user

until unless they open the source code of the page and look for the HTML code, they can't see these form that the hacker is maintaining inside their website.

So inside this form, what is happening?

The hacker is trying to make a request to that netflix.com backend server

with the path netflix.com change email. And this is a POST method.

And to this change email path, the hacker is also trying to send a hidden parameter value, which is email with the value as [user@evil.com](mailto:user@evil.com).

So what the hacker is trying to do here?

The hacker, he's trying to change the email of your account and the email value that he provided belongs to the hacker. So, as soon as the end user click on the malicious link, this form is going to be submitted. And since this form is making a request to the netflix.com, what browser will do, my innocent browser will attach the cookie

which has value abc123 to this request.

And with that, what is going to happen?

The Netflix backend, it is going to accept these requests, process the request

because it receives a proper cookie value**. Here, the problem is that netflix.com backend server is not able to differentiate whether the request is coming from a evil website or whether it is coming from their own website.**

This is because even the hacker also is trying to send the request to the netflix.com only. He's not sending to some other domain. He's sending the request to the same domain. Since this form is getting invoked from the same browser

but from the different tab, the browser is going to still attach the cookie

of the netflix.com to this form while it is making a request to the backend of netflix.com.

Here, you may have a question which is won't CORS will help us in this kind of scenario?

Because this form is present inside the evil.com website and from the evil.com website, the request will go to the netflix.com website. So, this might be your assumption.

But let me highlight you that regardless whether the CORS is implemented or not,

still the CSRF attack is possible in this scenario.

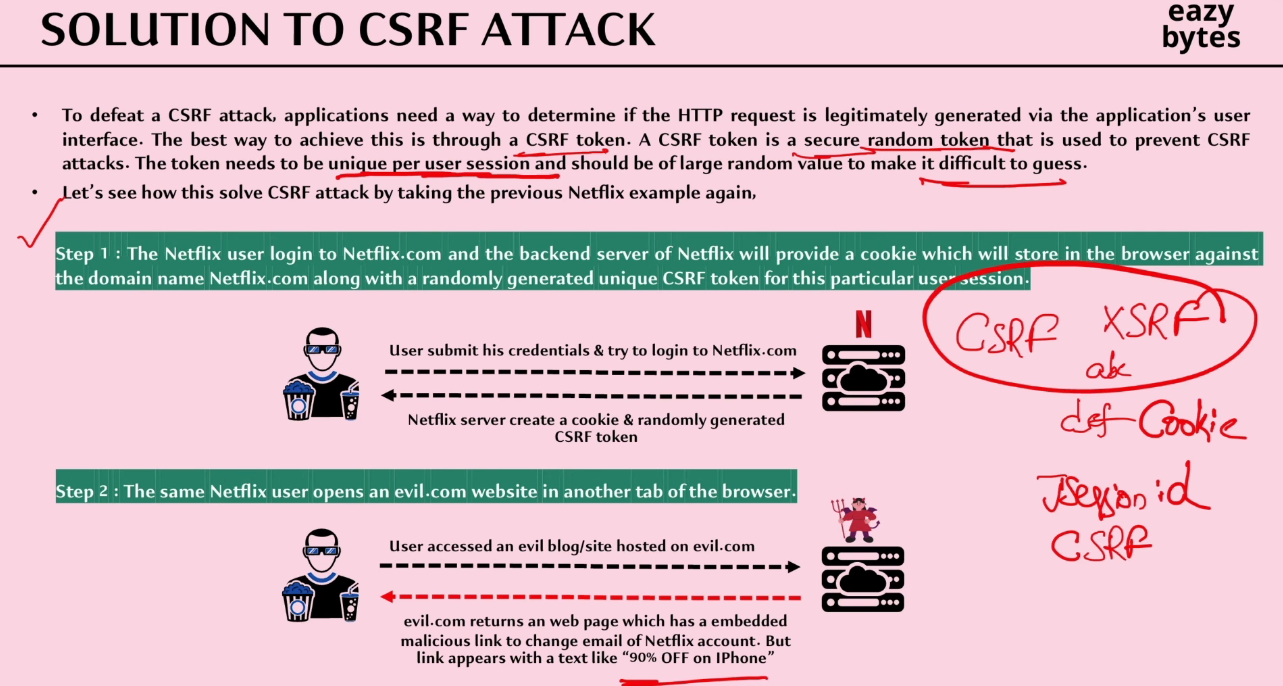
The reason is this is an embedded form that is maintained by the hacker inside their website. The advantage with the embedded form sees whenever someone is making a request to an action using an embedded form, the browser will think the request

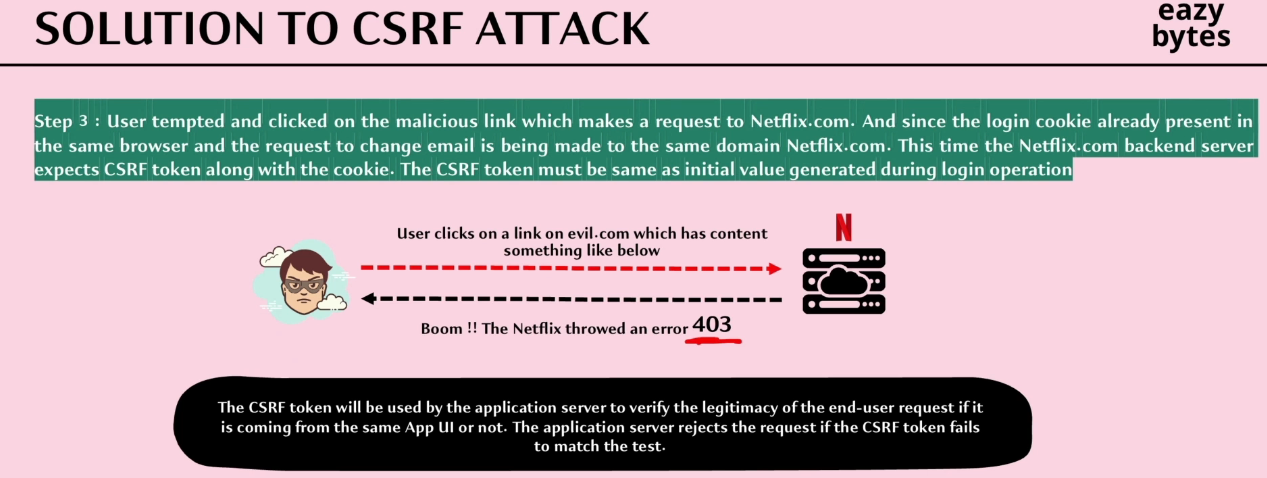
is originated from this domain itself. It'll never know that the embedded form

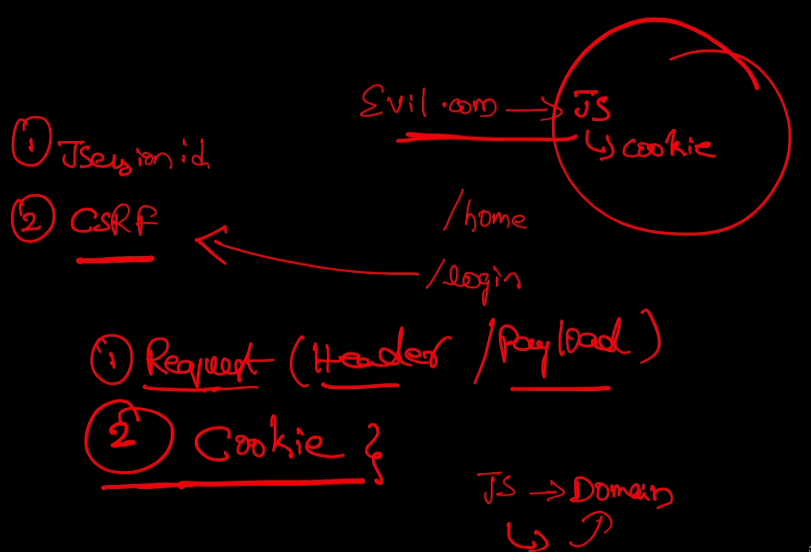
is present inside another domain, which is evil.com.

That's why CORS is not going to be helpful in this scenario.

Solution to handle CSRF attacks







As of now, the problem is the backend server of netflix.com or any other backend system, they don't have a mechanism to identify if the given request is coming from their own UI website or from the evil.com website.

Because both these approaches, they are attaching the JSESSIONID cookie

or login-related cookie as part of the request they are making, and with that, the backend server simply processing the request.

To defeat these CSRF attack, we should make our backend server smart enough

to detect the request, whether it is coming from the original website

or from the evil.com website.

To help our backend server, in this scenario, we need to implement a solution with the help of CSRF token. With the help of this CSRF token, my backend server will be able to differentiate whether the request is coming from the original website

or from the eil.com website.

I'll explain you the scenario in few minutes, but before that, let me highlight

that the CSRF token is a secure random token value that is used to prevent the CSRF attack.

The CSRF token value needs to be unique by user session and should be of large random value to make it difficult to guess for the hackers.

Whenever we are generating a CSRF token, it should be at least 12-length character,

and it should have some random text value so that it is going to be super tough for the hackers to guess the CSRF token value to perform the CSRF attack.

So let's try to replay the same Netflix scenario one more time, but this time

we'll try to include the CSRF token solution as part of the scenario.

In the very first step, the end user entered his credentials on the UI of the netflix.com;

with that, the netflix.com, it is going to validate the credentials.

And this time, along with the cookie related to the authentication or login operation,

this time, the backend server, it is also going to generate one more extra value,

which is CSRF token value.

A few people, they also call this token as **XSRF** token value. So this token will have some randomly generated value, so only the backend server will know at this point of time what is this value.

Since this token has to be sent back to the UI application, what backend server it's going to do is, it is going to attach this token value as a cookie to the UI application.

So just like how it is trying to attach the JSESSIONID cookie or the login-related cookie, similarly, it is also going to attach the XSRF token value as a cookie to the UI application.

So now my UI application received two tokens. The very first one, think like it is JSESSIONID, which is going to help to avoid the authentication again and again.

And the next cookie is CSRF token itself. So this is what happened in the step 1.

Now, in the step 2, the same Netflix user is going to open the evil.com website

in another tab of the user. And inside the evil.com website, he's going to see a malicious link, which is climbing 90% off on the iPhone.

Let's imagine our end user tempted and clicked on that malicious link,

which make a request to the netflix.com to change the email by using an embedded form.

So when this embedded form making a request to the netflix.com backend,

the browser is also going to attach the two cookies that got generated.

The very first one is login-related cookie and the second one is CSRF cookie.

This time, even though the browser is attaching both the cookies from the evil.com website, but still my backend server, it is going to throw the 403 error.

And with that, the CSRF attack is going to be stopped.

Now, you may have a million dollar question here which is how my backend server is differentiating even though the hacker request has two cookies attached.

So this can be your question.

So let's try to understand these in detail.

So we have two cookies. The very first one is JSESSIONID and the second one is CSRF cookie. So whenever a CSRF solution is implemented by a backend application, we know it is going to generate the CSRF token value during the very first initial GET() operation.

It can be a public GET() operation like loading the homepage, or it can be during the login operation. So if your login operation uses GET(), then behind the scenes,

these GET() operations, they can process the request, and as part of the response they can send the CSRF cookie.

But next time when the UI application is trying to make a POST() request or a PUT() request or any other delete request, the backend server, it is going to expect the **CSRF token value in two places.**

**The very first place is it has to send as part of the request, so it can be inside the Header or it can be inside the Payload.**

And the next place is as part of the cookie. Sending the CSRF token value inside the Cookie is going to take care by the browser.

We know the browser is always going to attach the cookies related to your domain automatically,

so the hacker will be able to send that token value inside the Cookie

by leveraging the automatic attaching of the cookies by the browser.

But the hacker will not be able to send the CSRF token value as part of the request.

The reason is very simple.

In order to send the cookie value inside the RequestHeader or inside the RequestBody, the client application, at the UI application,

they have to manually read the cookie value using a JavaScript.

Once the cookie value is read by the JavaScript code, inside the same JavaScript code, they're going to populate the CSRF token value inside the Header or inside a Payload.

The same kind of operation the hacker can't perform from the JavaScript code inside the evil.com. Because whenever a JavaScript is being executed from a given domain,

the JavaScript code can read only the cookies related to the same domain.

So inside the evil.com, what is going to happen, though hacker can manage to build a client-side code with the help of JavaScript code to read the cookie value inside the browser of the end user, but this JavaScript to code is not going to work,

because the browser is going to stop evil.com JavaScript code from reading the cookie belongs to the netflix.com.

So this way the hacker will not be able to send the cookie value inside the RequestHeader or RequestPayload.

With that, the backend server will easily identify, it is going to simply throw the 403 error, whereas in the proper, genuine scenario of netflix.com,

the JavaScript code available inside the netflix.com UI website,

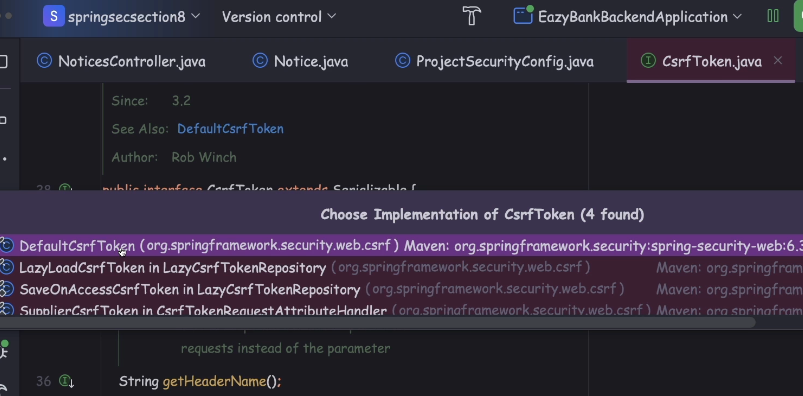
it will be able to easily read the Cookie value present inside the browser,

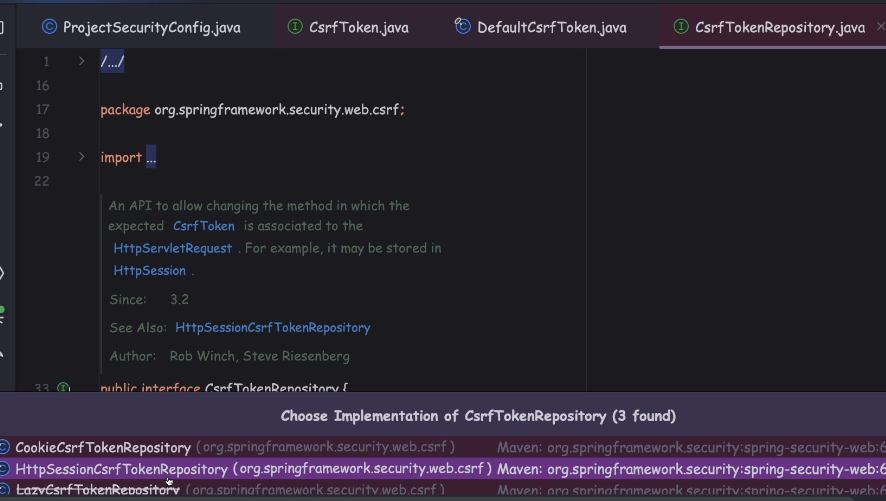
because the JavaScript code is being executed from the same domain, which is netflix.com.

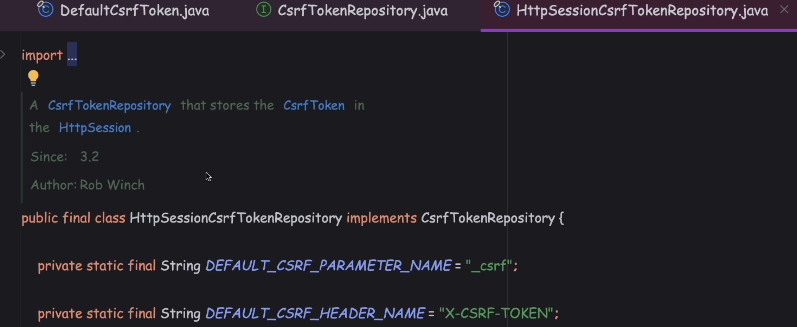
So this is the most common approach that is used inside the industry to handle the CSRF attack.

Implementing CSRF token solution inside backend application



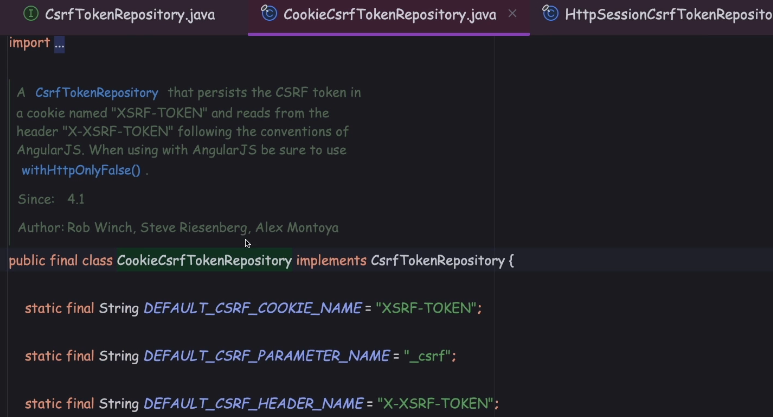


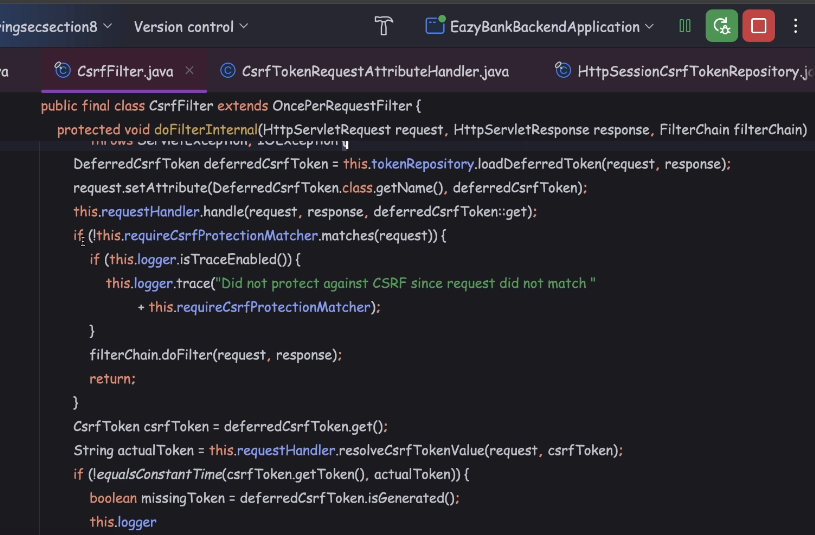




But in general, storing the token details inside the HttpSession is not recommended.

Instead we should try to generate the tokens and store them inside the cookies of the application.





That's why let me open the other implementation, which is CookieCsrfTokenRepository. Inside this class, you'll be able to see lot,

many methods like generateToken(), saveToken(), loadToken().

So all these methods, they're going to help to generate a CSRF token value during the login operation.

So think like we are leveraging one of these Repository classes,

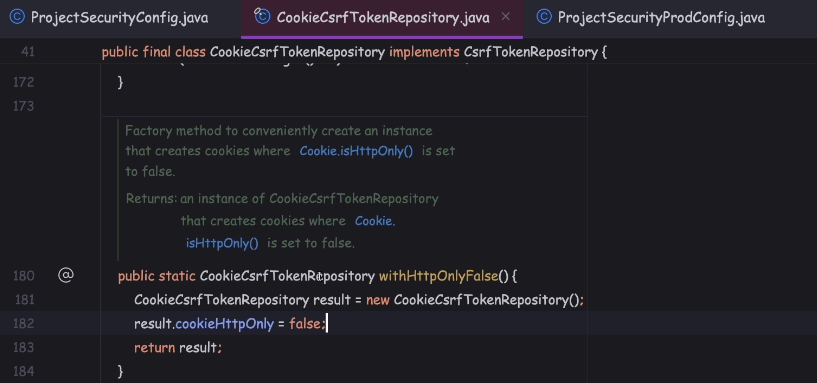
and we have generated the token, and we send the token to the UI application,

from the next subsequent request onwards, if the request is of type POST, PUT or DELETE, the Spring Security framework, it has to validate whether it receives the same CSRF token or not.

To handle the subsequent request and validate the CSRF token provided by the UI application, we have CsrfFilter is a filter name. Inside this filter, under the doFilterInternal() method, you'll be able to see all the logic

of loading the token from the request and comparing with the original token.

For some reason, if the tokens are not valid, this filter, it is going to throw the AccessDeniedException.

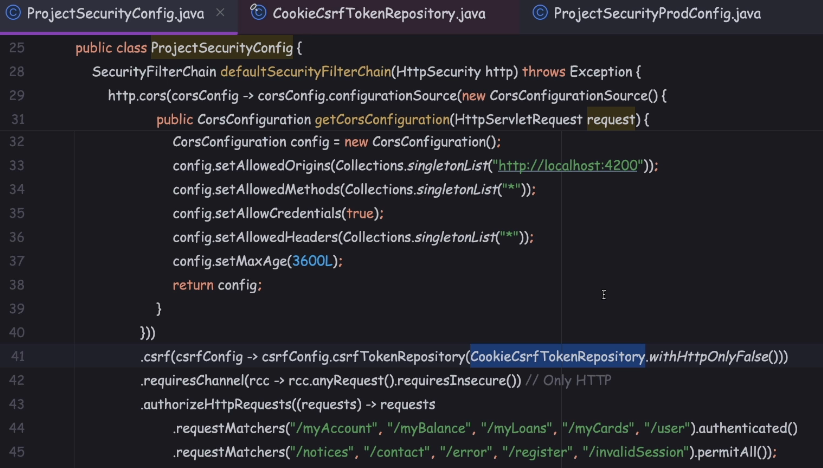


Since the Spring Security framework is going to expect the CSRF token value, both as part of the cookie and as part of the RequestHeader, we need to make sure we are setting these property as false.

Otherwise, the Angular UI code or the JavaScript code on the client side,

they will not be able to read the cookie value.





but whenever we are generating the CSRF token value with the help of this class, behind the scenes, the token is going to be generated lazily, which means until unless someone is trying to read the token value manually, the token is never going to be generated and the cookie also is never going to be generated.

So this has been done intentionally by the Spring Security framework,

just to achieve some performance.

Since the token is getting generated lazily, the CSRF cookie, that we are expecting

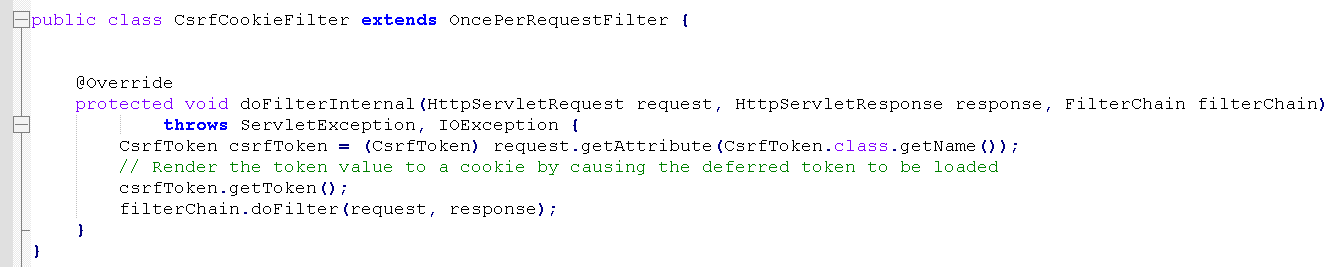
from the backend to the UI, is never going to be generated automatically.

We need to write a piece of logic to read the token manually.

Once the token is read manually, behind the scenes, Spring Security is going to take care of generating the token value, and generating the cookie,

and sending the same to the UI framework. So to read that token value manually,

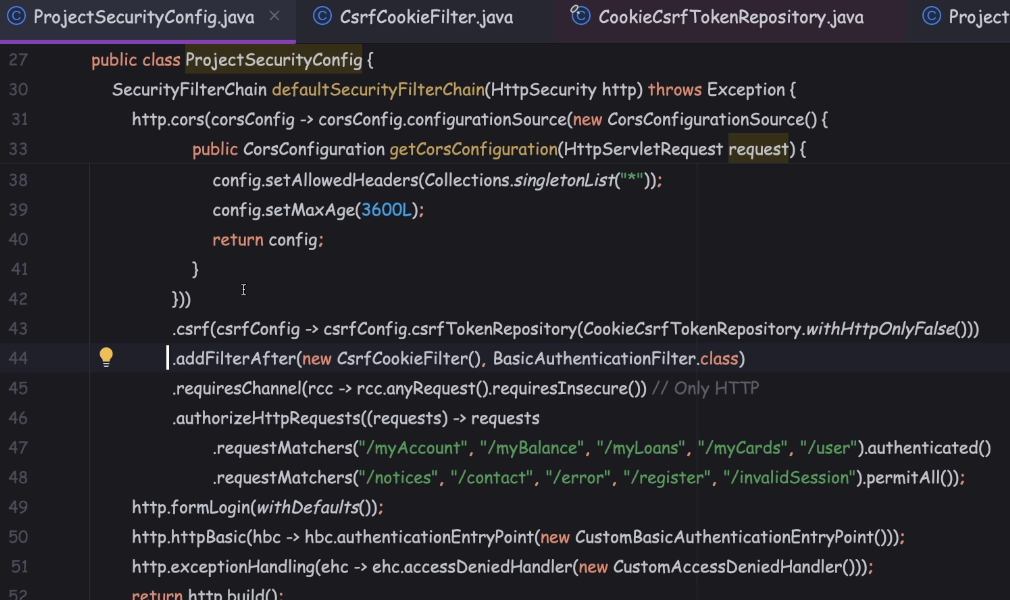
what we have to do is we have to define a filter.



So when I invoke the getToken() method during the execution of this piece of code only, the actual token generation is going to happen.

And once the token is generated, Spring Security framework is going to take care

of sending the same as part of the cookie as well.



So whenever we are sending the credentials using the httpBasic format,

this filter is going to take care of performing the actual authentication

by extracting the credentials. So once the authentication is completed,

Spring Security framework is going to generate the CSRF token with the help

of this CookieCsrfTokenRepository class.

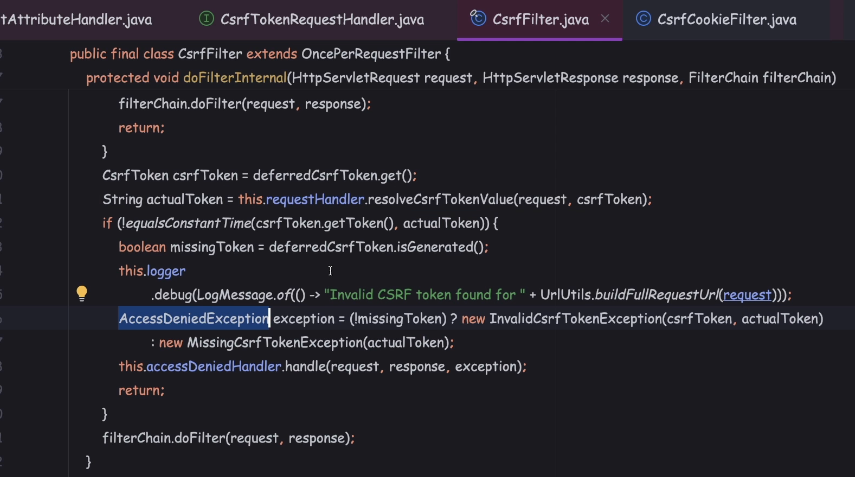
So once all this is completed, I want my filter to be executed to read that deferred token and load the actual value. So that's why I'm trying to mention this addFilterAfter() method

Implementing CSRF token solution inside backend application - Part 2





Inside this method we have the logic to read the CSRF token value from the request to header. Once that token value is read, the same is going to be returned as an output from this method.



So if you go to the CSRF filter and at this doFilter() internal method, we have a logic

to resolve the csrfTokenValue() from the request received by the UI application.

Once the actual token is loaded, this filter is going to take care of comparing the actual token with the previously generated token and accordingly

it'll throw the access denied exception if the token values are different.

To make sure that this filter is doing its job correctly, we need to make sure we are creating the object of CsrfTokenRequestAttributeHandler inside the ProjectSecurityConfig. The same AttributeHandler object we need

to set under the CSRF configurations.



From now onwards we are going to use and separate UI application to log in the user. There'll be a separate login page. From this login page we're going to accept the credentials and these credentials we are going to send to the backend server in the format of Httpbasic in this kind of scenarios Spring Security by default it is not going to generate the jsessionId.

We need to explicitly tell to the Spring Security framework, please generate the jsessionId so that all the subsequent request I can invoke with the help

of jsessionId without passing the credentials.

So how to do the same? It's very easy.

We just have to invoke the sessionManagement() method.

So whatever jsessionId that is going to be generated based upon the session configurations, it is not going to be automatically stored inside the SecurityContextHolder.

So by default being security framework, whenever we are making these configurations, it is not going to take the headache of storing the jsessionId

inside the SecurityContextHolder. Either we need to manually store the jsessionId

inside the SecurityContextHolder or we need to make a configuration so

that the Spring Security framework can take care for us.

So how to do that, it's very simple.

We need to invoke the securityContext() method.

To this method also, we need to invoke the method require explicit save

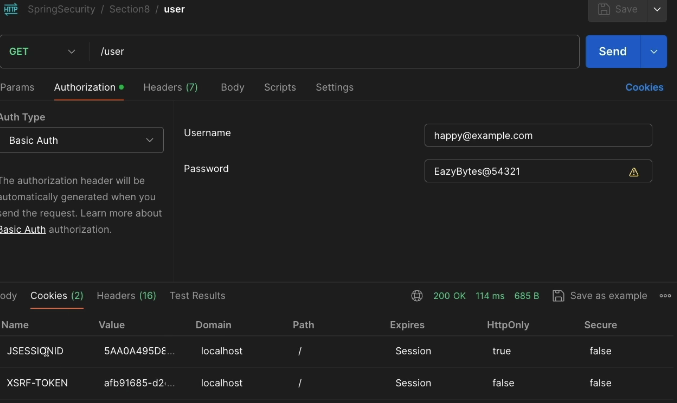
and to this I'm going to pass the input which is false.

With this configuration I'm telling to the Spring Security framework, I'm not going

to store any jsessionId details or the logged in authenticated details

inside the SecurityContextHolder. Instead, I want the Spring Security framework

to take care of the same.



So if you can open this user API, this is http GET API to this API, I'm trying

to pass the credentials using the Basic Auth.

So when I invoke this method, what is going to happen?

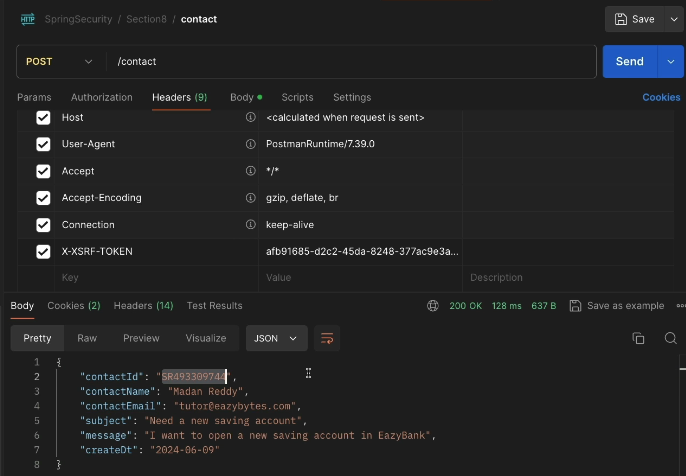
Since this is a security API behind the scenes, the Spring Security framework, it is going to invoke the BasicAuthenticationFilter and inside the same filter,

the actual authentication is going to happen.

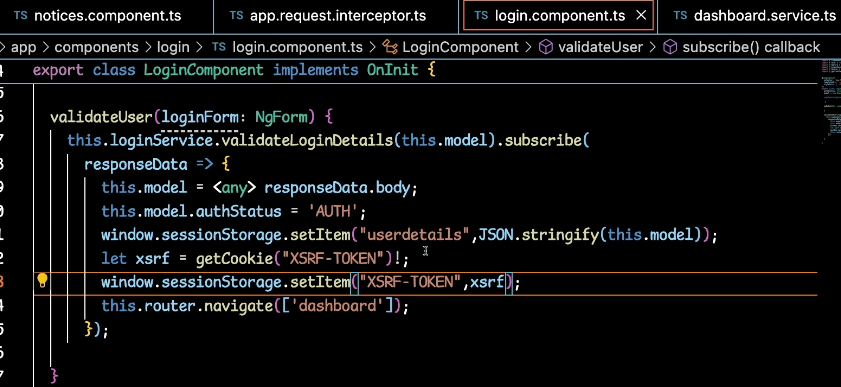
Once the actual authentication completed, the user API present inside the user control is going to be invoked. Inside this API, we load all the customer details from the database and return the same as a response.

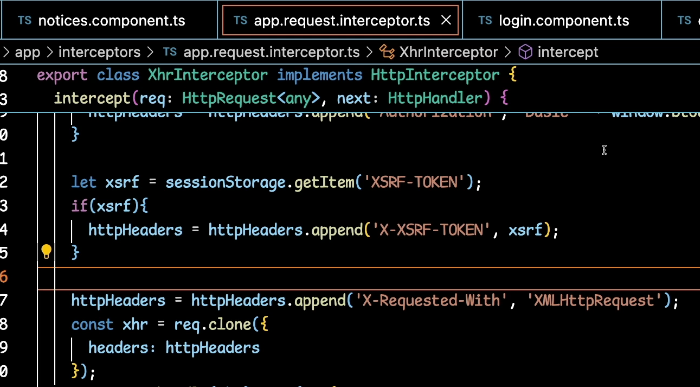
So during these login operation, only the CSRF token value is going to be generated.

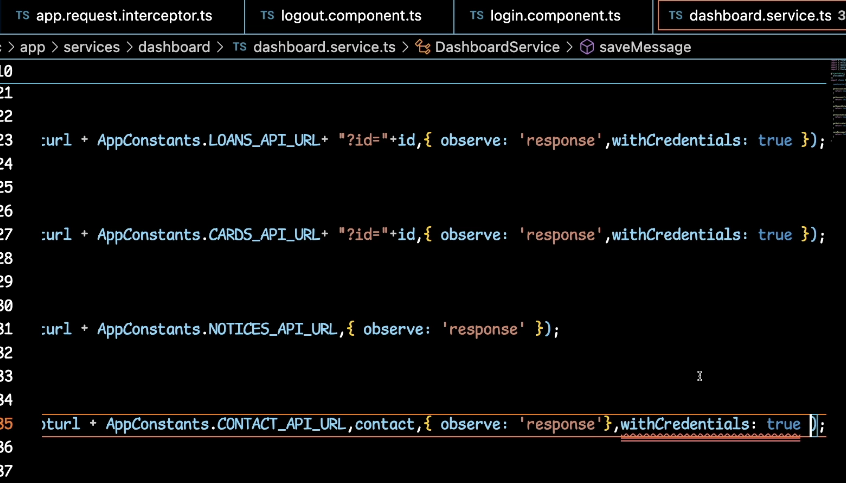
Not only this secured API if you try to invoke any secured API with the HTTP basic credentials behind the scenes. Since the authentication is going to happen very first time, the CSRF token is also going to be generated.



Implementing CSRF token solution inside UI application







Spring Security always expects the CsrfToken value in two different places.

The very first one is inside the cookie, and the second one is inside the RequestHeader. So just to make sure that my Angular

is sending the cookie details in the scenario of contact API,

I'm trying to mention these with the credentials as true.

Ignoring CSRF protection for public APIs

To this "ignoringRequestMatchers()" method, we can pass the list of APIs

for which we want ignore the CSRF protection.

Summary

First, we're trying to configure the "csrfTokenRequestAttributeHandler()" object

into the CSRF configurations with the help of this method. So, this is going to help me to read the CSRF token value that I'm receiving inside the RequestHeader from the UI. Once the token value is read, the same is going to be populated

as part of the request attribute. So, whatever we populated inside the request attribute, the same is going to be leveraged by the CsrfFilter to perform the token related validations.

Next, with the help of ignoringRequestMatches(), we are trying to ignore the CSRF protection for these APIs.

Next, with the help of the csrfTokenRepository() method, we are telling to the Spring Security Framework to leverage the CookieCsrfTokenRepository to generate the CSRF token value and store the same inside the cookie.

At last, we're trying to configure our own custom filter, which is responsible to read the CSRF token. With the help of this getToken() method.

When we try to manually read this token at this point of time,

the token value is going to be rendered to a cookie by causing that deferred token to be loaded.

Top of Form

Which of the following is a mechanism that allows a server to indicate any origins (domain, scheme, or port) other than its own from which a browser should permit loading of resources?

**Cross-Origin Resource Sharing (CORS)**

Top of Form

Which of the following parameters are considered to identify "other origins" as part of CORs policy?

**Domain & Port & scheme (HTTPS/HTTP)**

Top of Form

If you have a scenario where a front end application try to communicate with a backend application which is deployed in a different domain/port, how to resolve CORs issue that will arise in these kind of scenarios?

**Backend server can be configured to allow cross-origin resource sharing by including some special headers like Access-Control-Allow-Origin, Access-Control-Allow-Methods etc. Web browsers can use these headers to determine whether a request should continue or fail.**

Top of Form

Which of the following explains CROSS-SITE REQUEST FORGERY(CSRF) issue correctly?

**A typical Cross-Site Request Forgery (CSRF or XSRF) attack aims to perform an operation in a web application on behalf of a user without their explicit consent. In general, it doesn't directly steal the user's identity, but it exploits the user to carry out an action without their will.**

Bottom of Form

Bottom of Form