Programming 6

(Frontend) Security

Image Designed by vectorjuice / Freepik

Contents

Security is a very broad topic…
We limit ourselves to aspects related to the development of single page applications (with React…) in combination with an IDP (like Keycloak…) and a backend (like Spring…)

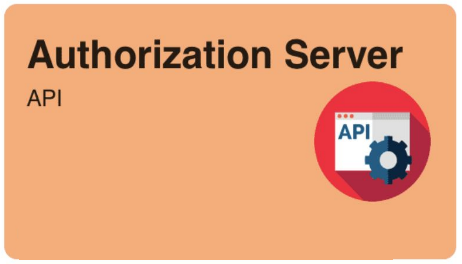
Authentication

CORS

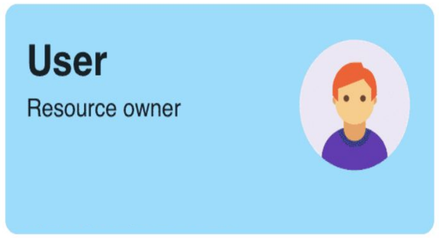
XSS

Authentication

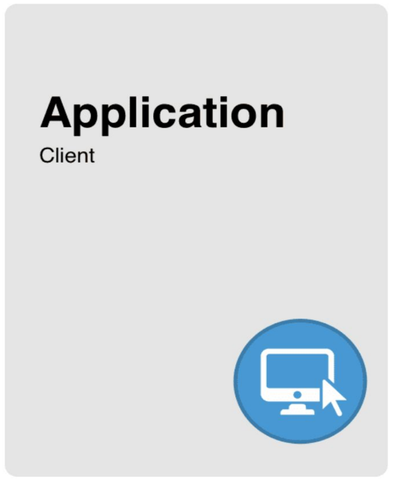
Who’s dancing?



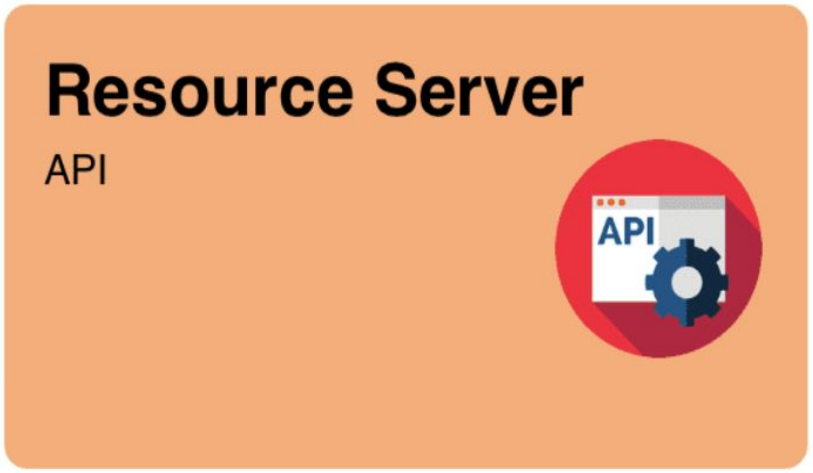
Keycloak IDP



Human, service,..



React client



Spring backend

Using which tokens?

Identity (ID) token

Contains identity information about the resource owner (name, profile pic,...).

It proves that a user has been authenticated.

Can be used to show user info in the UI.

Access token

Provides access to protected resources. Powerful token! Short lifespan!

Stored in Authorization header (‘bearer’ token)

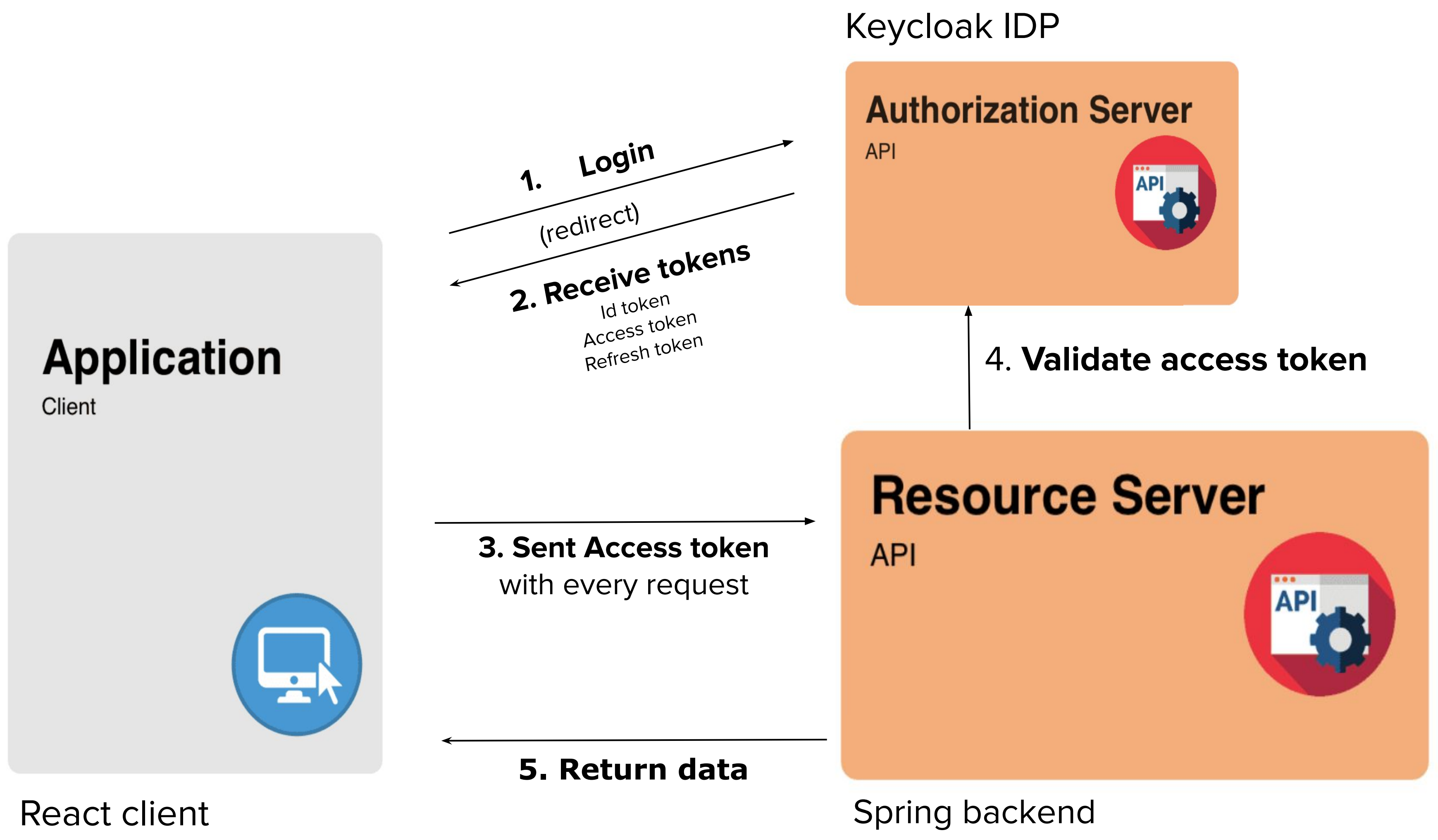
Can contain a role (admin,...) that determines which resources the token grants access to

Refresh token

Used to retrieve new access tokens without requiring the user to perform a new login.

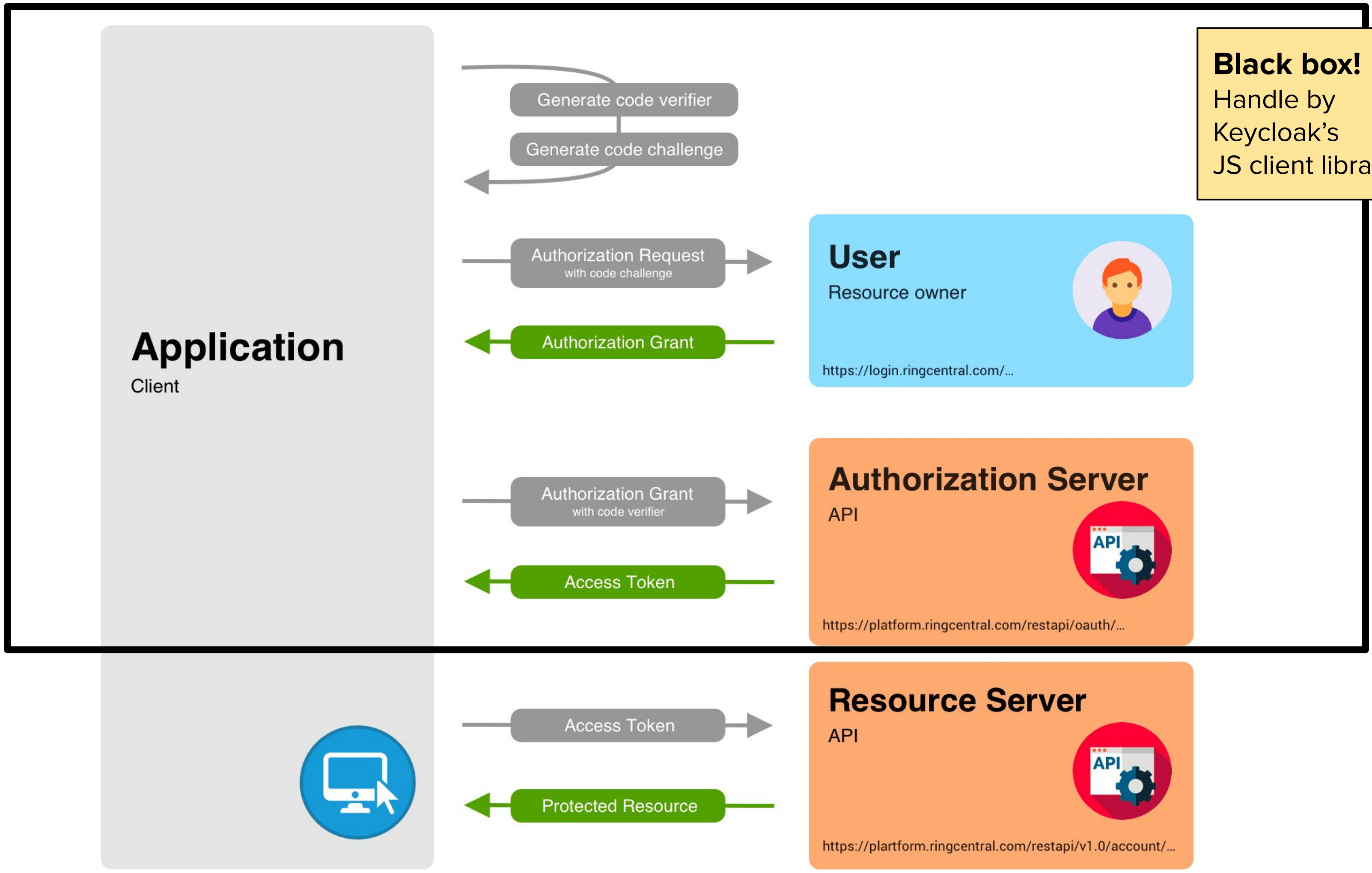
Managed by the IDP (keycloak)!

Token flow


Keycloak IDP
Authorization Server
API
API
1. Login
(redirect)
2. Receive tokens
Id token
Access token
Refresh token
Application
Client
4. Validate access token
Resource Server
API
API
3. Sent Access token
with every request
5. Return data
React client
Spring backend

Complete flow

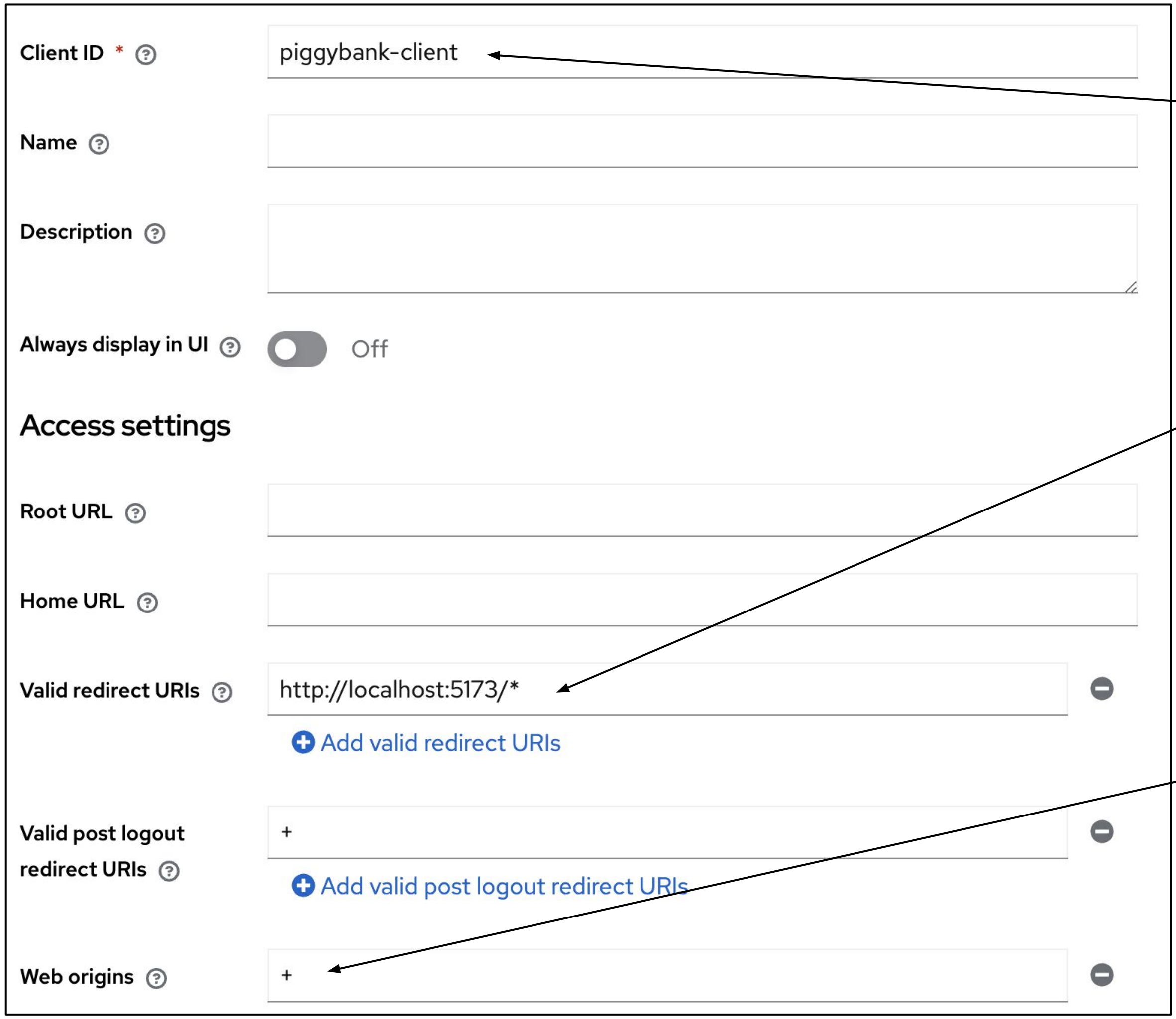
= Authorization Code Flow with PKCE (“pixee”)



Black box!
Handle by
Keycloak’s
JS client library

Keycloak configuration

Configure a client in de KC admin panel under the Realm for the application (1/2)



Client ID * : piggybank-client

Name :

Description :

Always display in UI : Off

Access settings

Root URL :

Home URL :

Valid redirect URIs : http://localhost:5173/*

Valid post logout redirect URIs : +

Web origins : +

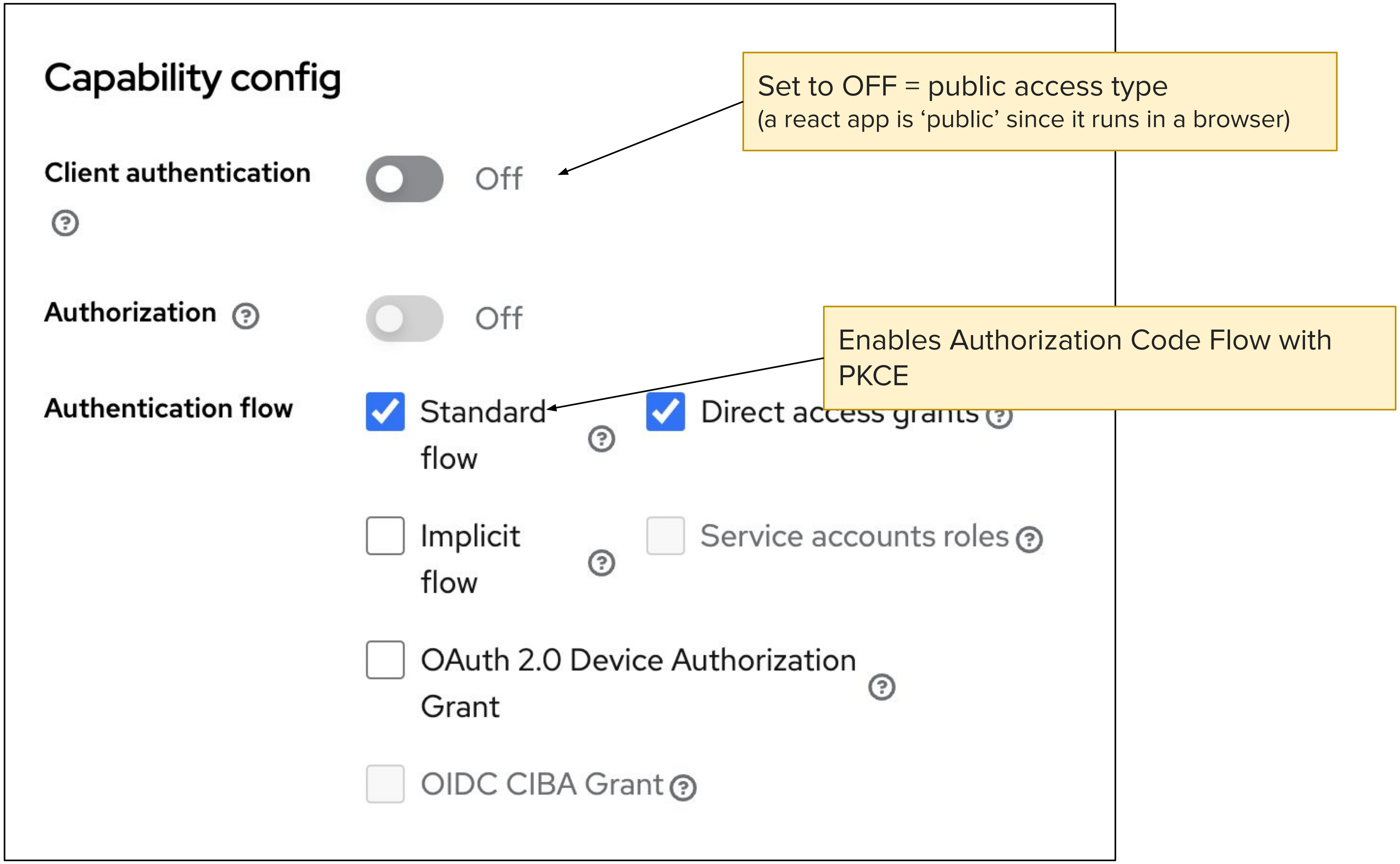
Client ID, see further

URL of React app

Enables CORS between React and KC
(+ = the valid redirect URI’s)

Keycloak configuration

Configure a client in de KC admin panel under the Realm for the application (1/2)



Capability config

Client authentication: Off

Authorization: Off

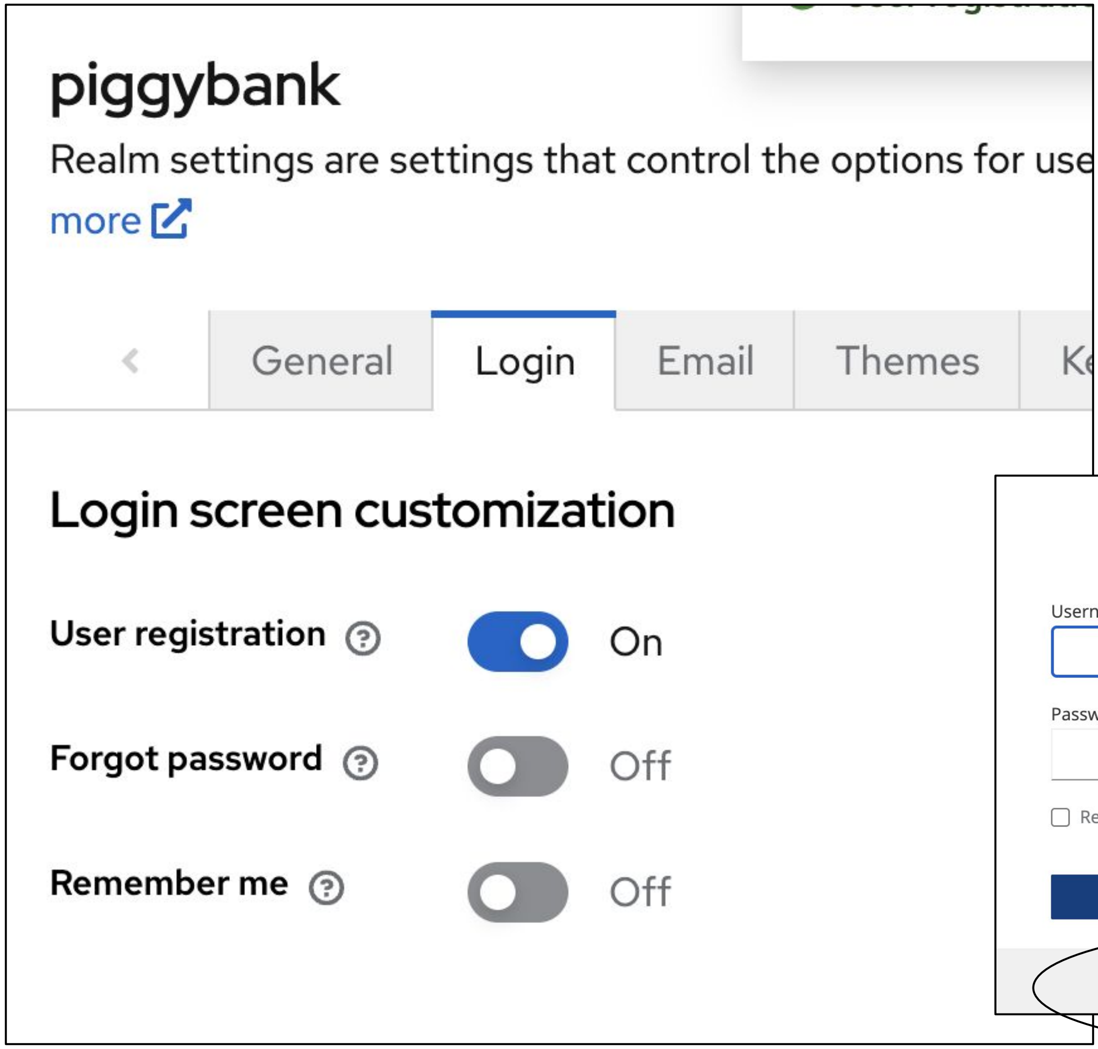
Authentication flow:
- [✓] Standard flow
- [✓] Direct access grants
- [ ] Implicit flow
- [ ] Service accounts roles
- [ ] OAuth 2.0 Device Authorization Grant
- [ ] OIDC CIBA Grant

Set to OFF = public access type
(a react app is ‘public’ since it runs in a browser)

Enables Authorization Code Flow with PKCE

Keycloak configuration

Optionally add user registration and other facilities



piggybank

Realm settings are settings that control the options for use

more

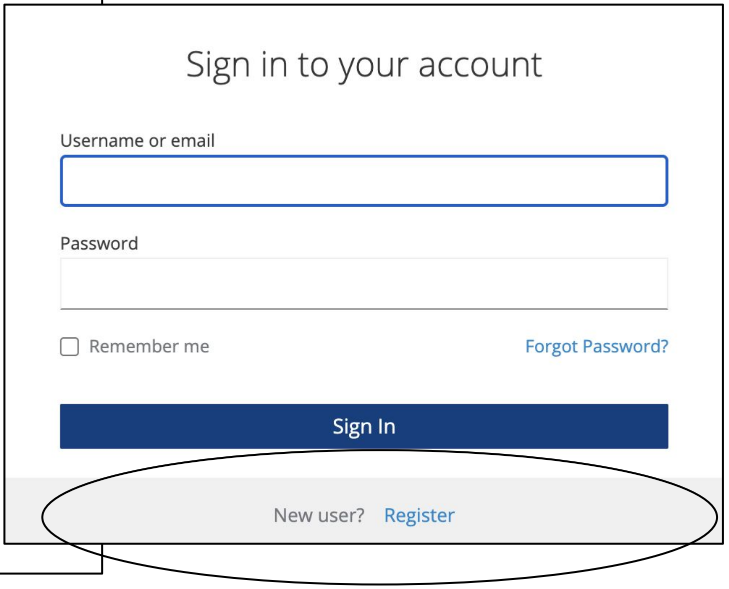
General | Login | Email | Themes | Ke

Login screen customization

User registration: On

Forgot password: Off

Remember me: Off



Sign in to your account

Username or email

Password

Remember me

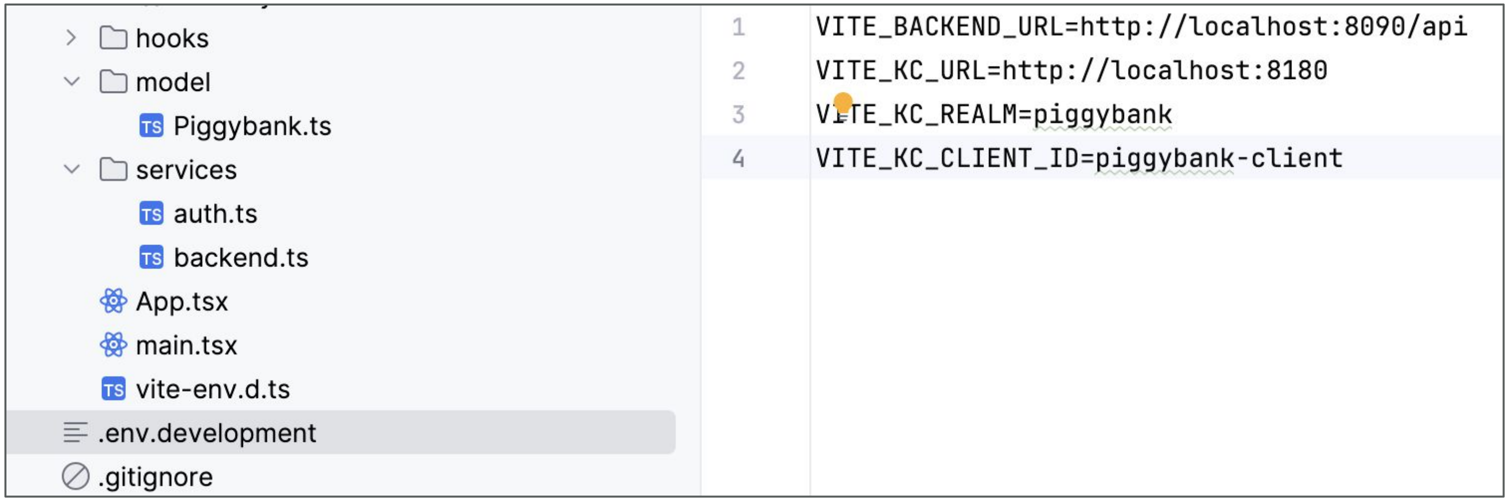
Forgot Password?

Sign In

New user? Register

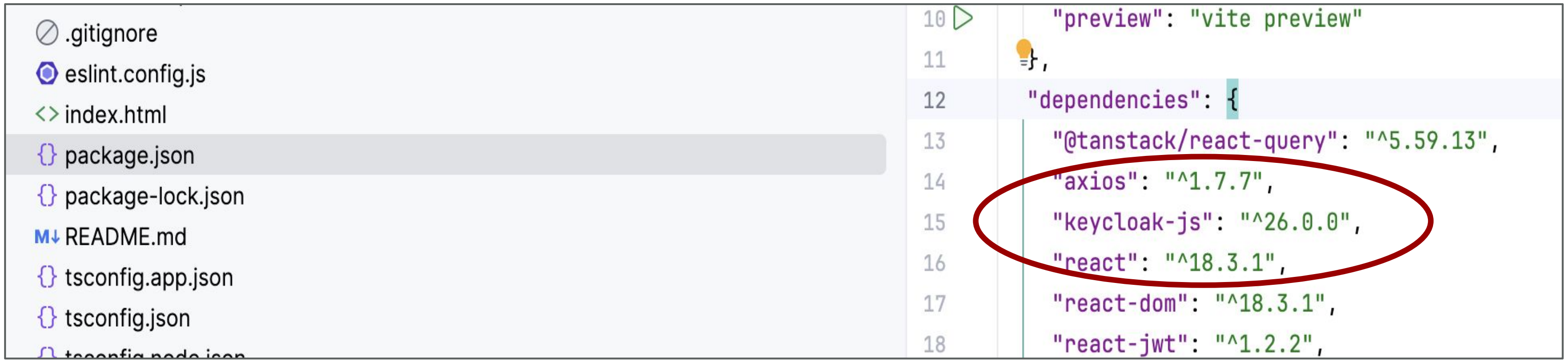
React

Setup the necessary variables pointing to your KC instance, backend, realm and client-id. You can use vite's .env files for this…



```
VITE_BACKEND_URL=http://localhost:8090/api
VITE_KC_URL=http://localhost:8180
VITE_KC_REALM=piggybank
VITE_KC_CLIENT_ID=piggybank-client
```

Install the KC Javascript adapter

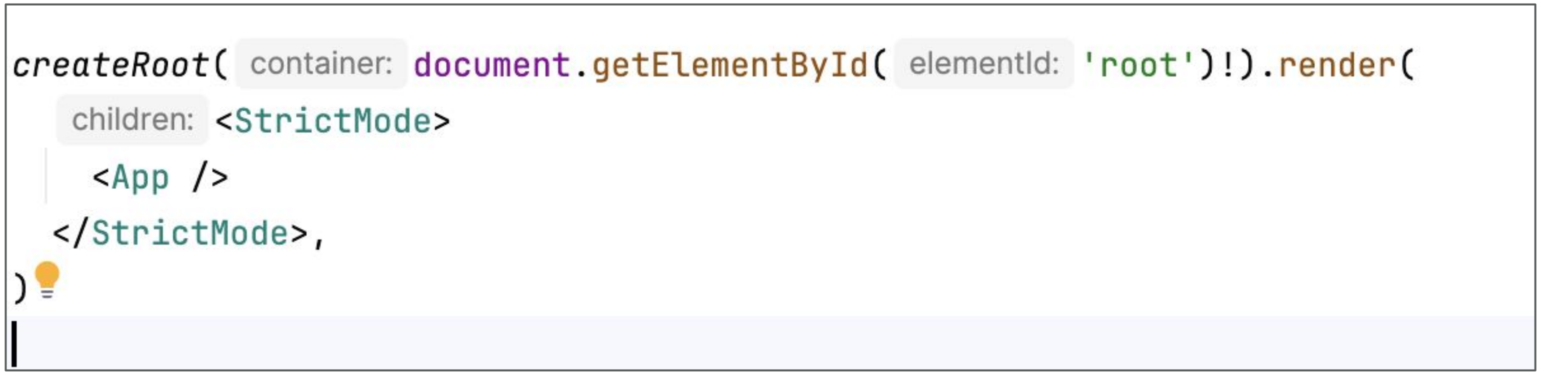


```
    "preview": "vite preview"
  },
  "dependencies": {
    "@tanstack/react-query": "^5.59.13",
    "axios": "^1.7.7",
    "keycloak-js": "^26.0.0",
    "react": "^18.3.1",
    "react-dom": "^18.3.1",
    "react-jwt": "^1.2.2",
```

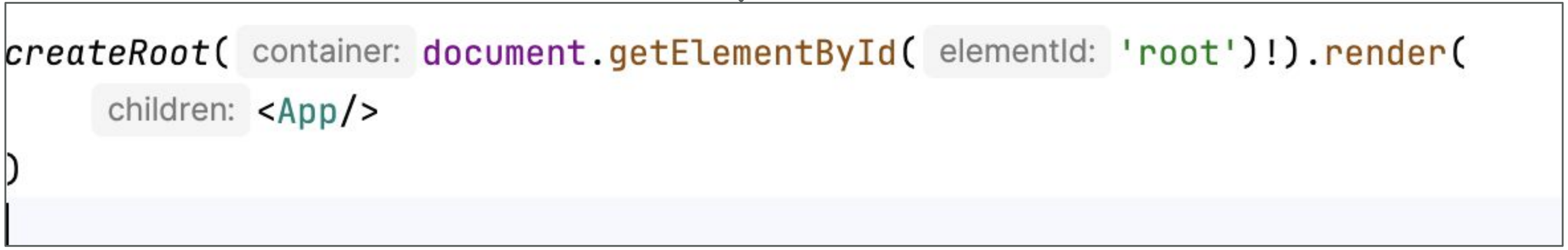
React

In **main.tsx**, remove strict mode

In strict mode, all components are initialised/rendered twice to detect possible bugs. The init method of Keycloack.js cannot cope with this…



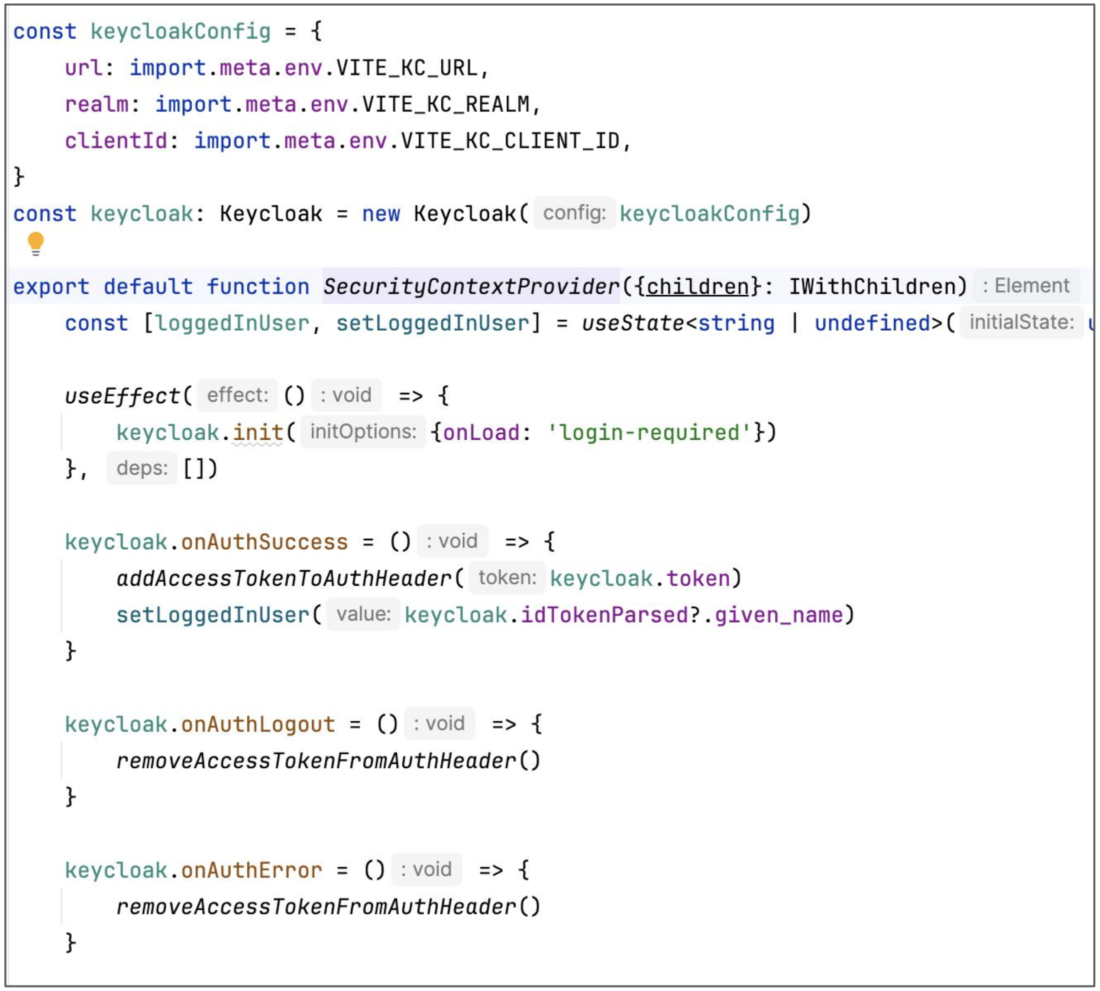
```
createRoot( container: document.getElementById( elementId: 'root')!).render(
  children: <StrictMode>
    <App />
  </StrictMode>,
)
```



```
createRoot( container: document.getElementById( elementId: 'root')!).render(
    children: <App/>
)
```

React

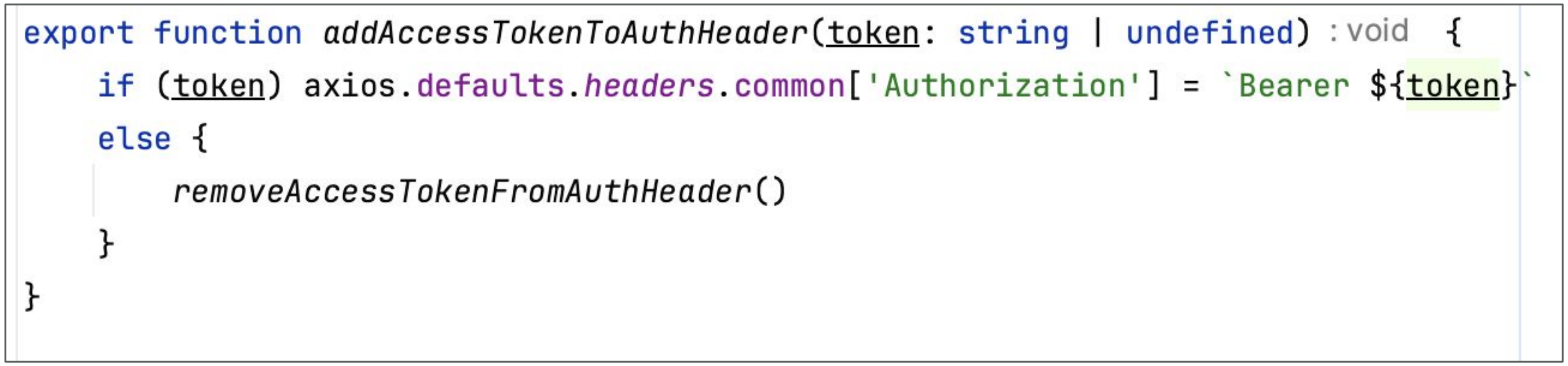
Typically a **context** is used to init the client library and provide security info to your components
(see example code on Canvas)



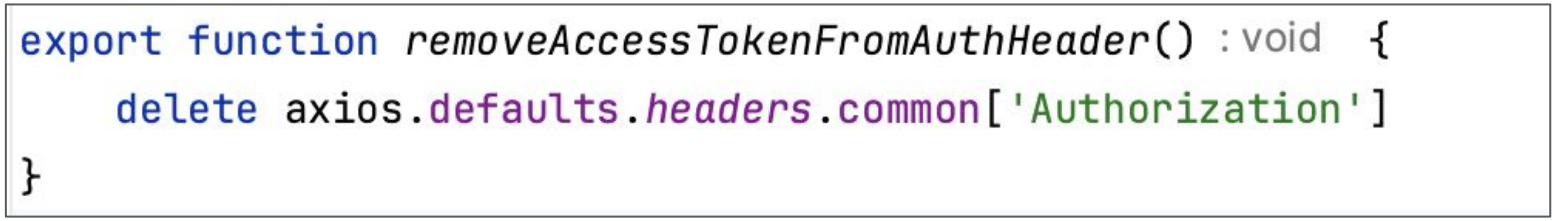
```
const keycloakConfig = {
    url: import.meta.env.VITE_KC_URL,
    realm: import.meta.env.VITE_KC_REALM,
    clientId: import.meta.env.VITE_KC_CLIENT_ID,
}
const keycloak: Keycloak = new Keycloak( config: keycloakConfig)

export default function SecurityContextProvider({children}: IWithChildren) : Element
    const [loggedInUser, setLoggedInUser] = useState<string | undefined>( initialState:

    useEffect( effect: () : void  => {
        keycloak.init( initOptions: {onLoad: 'login-required'})
    }, deps: [])

    keycloak.onAuthSuccess = () : void  => {
        addAccessTokenToAuthHeader( token: keycloak.token)
        setLoggedInUser( value: keycloak.idTokenParsed?.given_name)
    }

    keycloak.onAuthLogout = () : void  => {
        removeAccessTokenFromAuthHeader()
    }

    keycloak.onAuthError = () : void  => {
        removeAccessTokenFromAuthHeader()
    }
```

React

Pass the received access token as a bearer to all outgoing HTTP calls



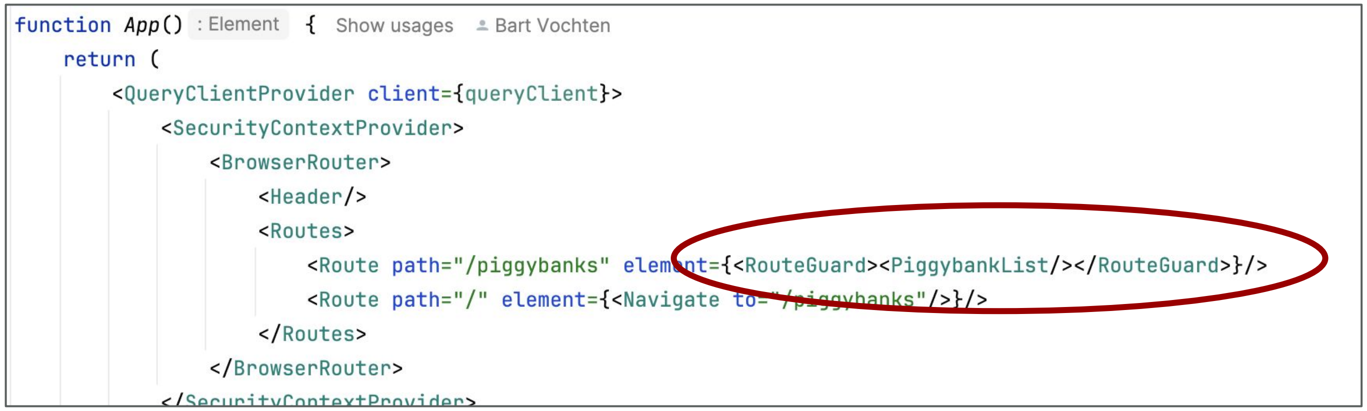
```
export function addAccessTokenToAuthHeader(token: string | undefined) :void  {
    if (token) axios.defaults.headers.common['Authorization'] = `Bearer ${token}`
    else {
        removeAccessTokenFromAuthHeader()
    }
}
```



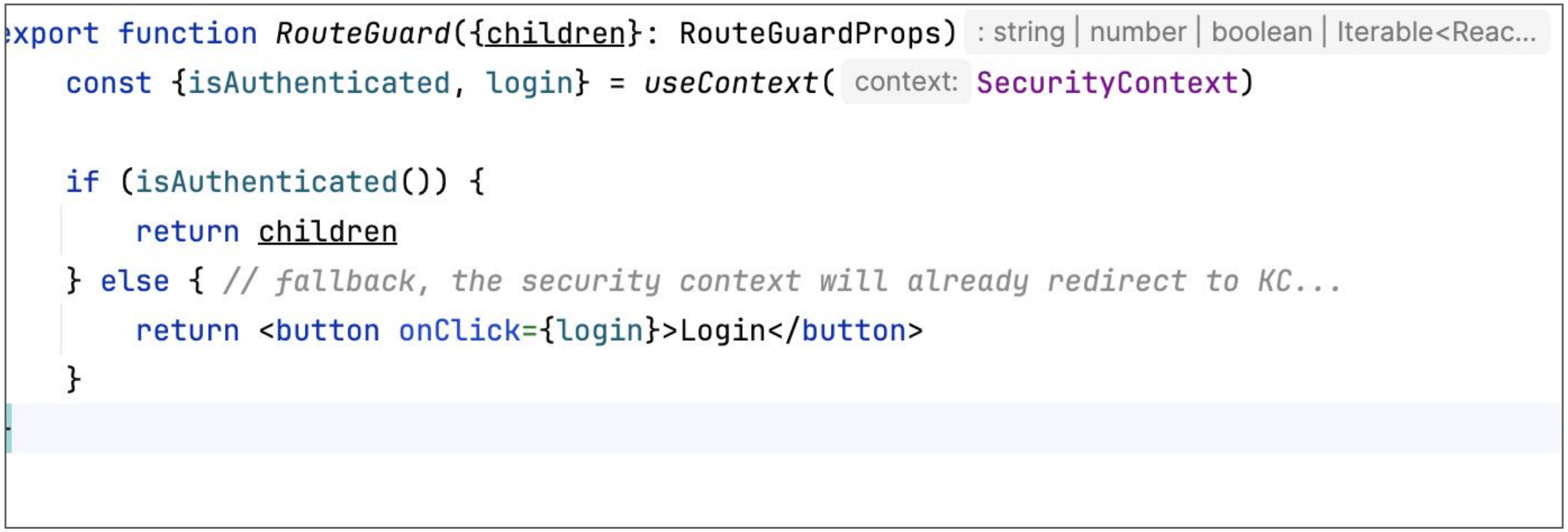
```
export function removeAccessTokenFromAuthHeader() :void  {
    delete axios.defaults.headers.common['Authorization']
}
```

React

Protect your routes by (for instance) wrapping them in a guard



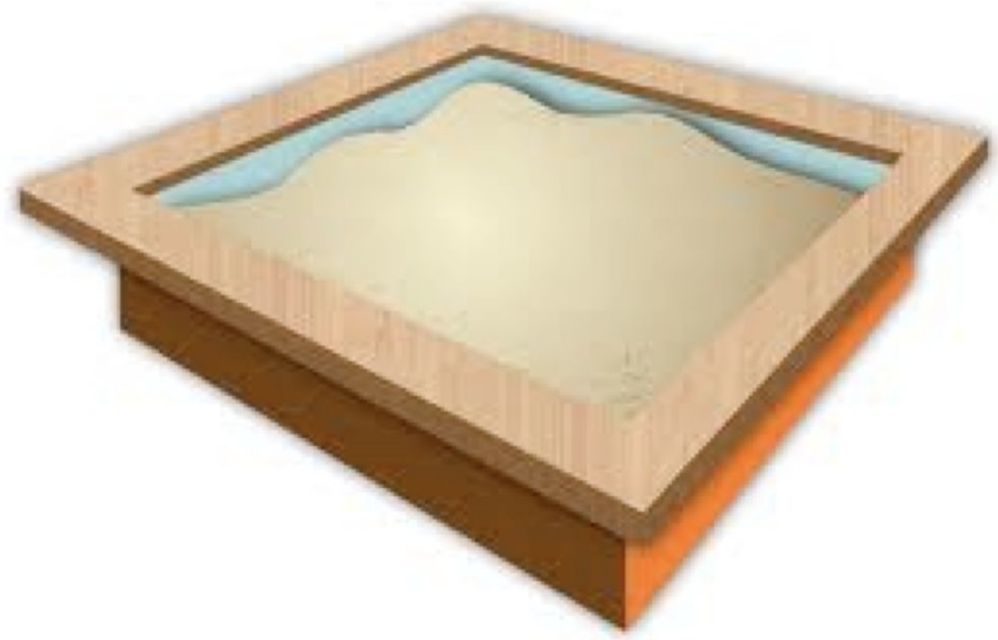
```
function App() : Element {  Show usages  Bart Vochten
    return (
        <QueryClientProvider client={queryClient}>
            <SecurityContextProvider>
                <BrowserRouter>
                    <Header/>
                    <Routes>
                        <Route path="/piggybanks" element={<RouteGuard><PiggybankList/></RouteGuard>}/>
                        <Route path="/" element={<Navigate to="/piggybanks"/>}/>
                    </Routes>
                </BrowserRouter>
            </SecurityContextProvider>
```



```
xport function RouteGuard({children}: RouteGuardProps) : string | number | boolean | Iterable<Reac...
    const {isAuthenticated, login} = useContext( context: SecurityContext)

    if (isAuthenticated()) {
        return children
    } else { // fallback, the security context will already redirect to KC...
        return <button onClick={login}>Login</button>
    }
```

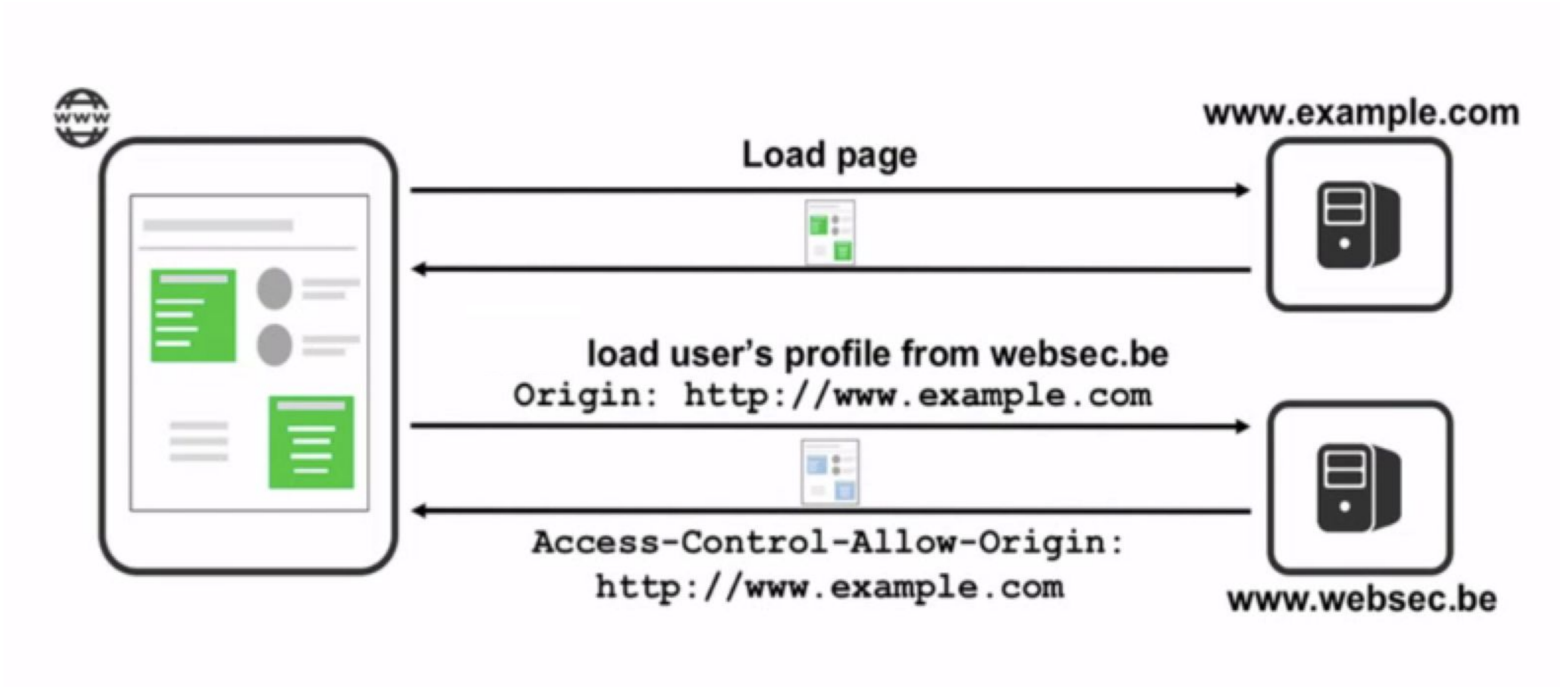
CORS

Browsers run web pages in a **security sandbox**



- Shielded from **local resources** (disk, camera,...)
- Shielded from **remote resources** that come from a **different domain** ('origin') than the domain from which the page was loaded
  = **SAME ORIGIN POLICY (SOP)**

How can you use data from another domain in your web app?
(or even from your domain but on a different subdomain or port)



www
www.example.com
Load page
load user's profile from websec.be
Origin: http://www.example.com
Access-Control-Allow-Origin:
http://www.example.com
www.websec.be

www.websec.be needs to respond with a **Access-Control-Allow-Origin** header (set to 'www.example.com' or to '*' to allow all domains)

Should be **configured in the backend** (Spring,...)

CSP

The src en href properties of <img>, <a>, <audio>,… tags are not restricted by CORS.

<img src="url to another domain"> is perfectly valid HTML (although you can not access the pixels in the loaded image)

If you want to restrict this further, you can use [Content Security Policy (CSP)]()

For example if you set this header, images can be loaded from any domain but media and scripts only from specific domains

```
Content-Security-Policy: default-src 'self'; img-src *; media-src media1.com media2.com;
script-src userscripts.example.com
```

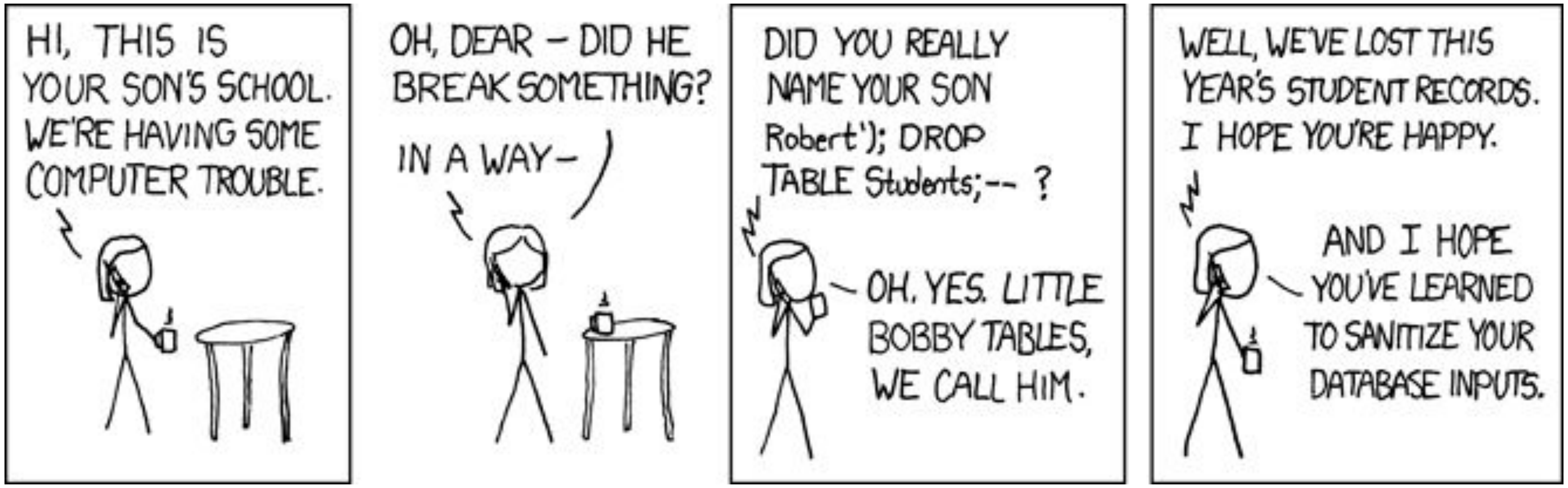
This helps you to reduce the risk on XSS attacks (see further)

XSS

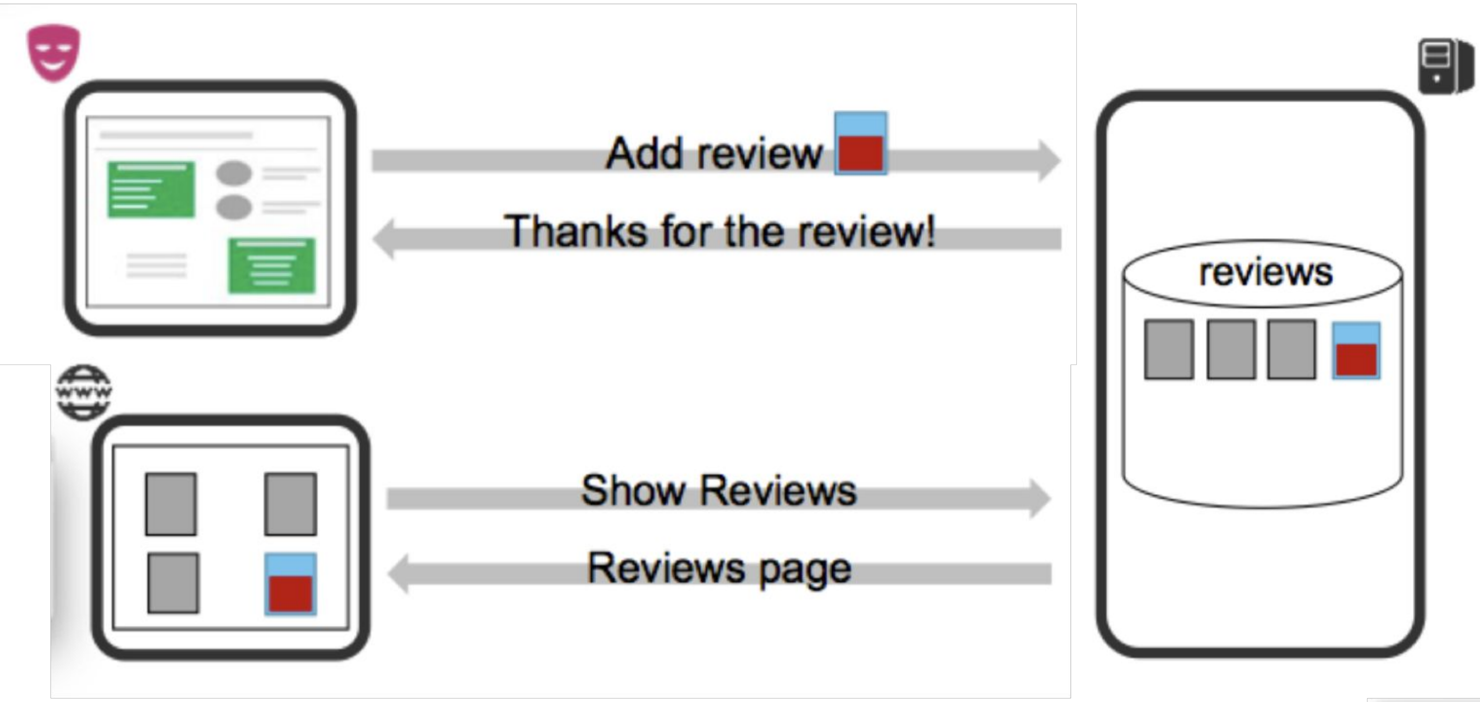
XSS

Users are security risks!

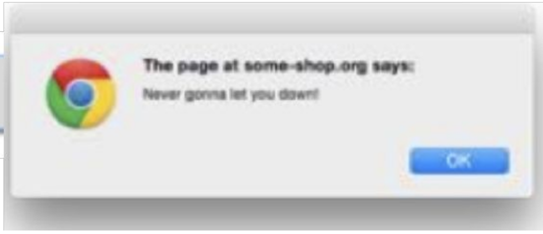
- Each ‘input’ is a possible attack vector
- What if a user inserts JS?
  - JS gets inserted where it wasn’t expected
  - A render of the unexpected input triggers the execution of the JS embedded in the input
  - XSS attack!


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

```
I can really recommend product X. It is awesome!
<script>alert('Never gonna let you down!')</script>
```



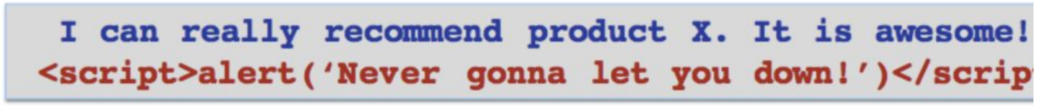
```
<html><body>...                ...</body></html>
```


The page at some-shop.org says:

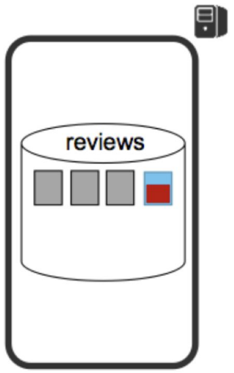
Never gonna let you down!

XSS

1. Attacker adds this review



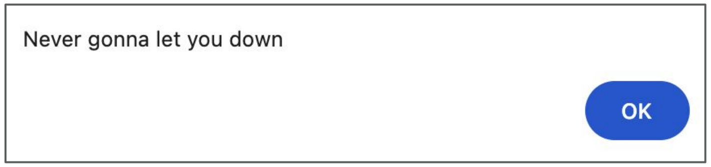
```
I can really recommend product X. It is awesome!
<script>alert('Never gonna let you down!')</scrip
```



2. Review gets stored unsanitized

3. Innocent user asks for reviews

4. Script code is executed in the context of the user



Never gonna let you down

Attacker could use this security hole to steal user information or perform actions on their behalf

XSS and React

JSX protects against XSS attacks by default: it escapes values embedded in JSX → It is **safe to render user input in JSX**.

&lt;script&gt;alert("XSS Attack")&lt;/script&gt

There is one exception!
Sometimes we need to render HTML defined by users or external systems (Rich Text Editor, API that returns HTML,...)

React has the '**dangerouslySetInnerHtml()**' function for that, but this loses the React DOM escaping!

Use this with care!

dangerouslySetInnerHtml

- The name is explicitly chosen to indicate danger
- React's replacement for 'innerHtml()' (don't use this)
- This is NOT a XSS safe method!
- Docs
- If you do need to use this method, use a library to **sanitize you HTML!**
    - DOMPurify is a good one
- Best practice:
    - Create a component that is responsible for sanitizing and rendering the HTML
    - When rendering user input that might contain HTML, use only this component!
    - That way, there is only one call to dangerouslySetInnerHtml in your code

Check the **XSS** demo on Canvas!

dangerouslySetInnerHtml

```
import DOMPurify from 'dompurify'

export function SanitizedText({ input }: { input: string }) {
    return (
        <span
            dangerouslySetInnerHTML={{
                __html: DOMPurify.sanitize(input),
            }}
        />
    )
}
```

That's all Folks!