Front-end security with HATEOAS

Plovdiv, 01.04.2023

Software Security

Software security is the protection of software applications and digital solutions **from unauthorized access, use, or destruction**.

Software security is set of measures, practices and techniques incorporated in the software development life cycle (SDLC) and testing processes.

Authentication

Who you are? - verifying identity

Relies on a factor to establish trust:

- Something you know (like passwords)
- Something you have (using your phone for OTPs)
- Something you are (biometrics, such as your face).

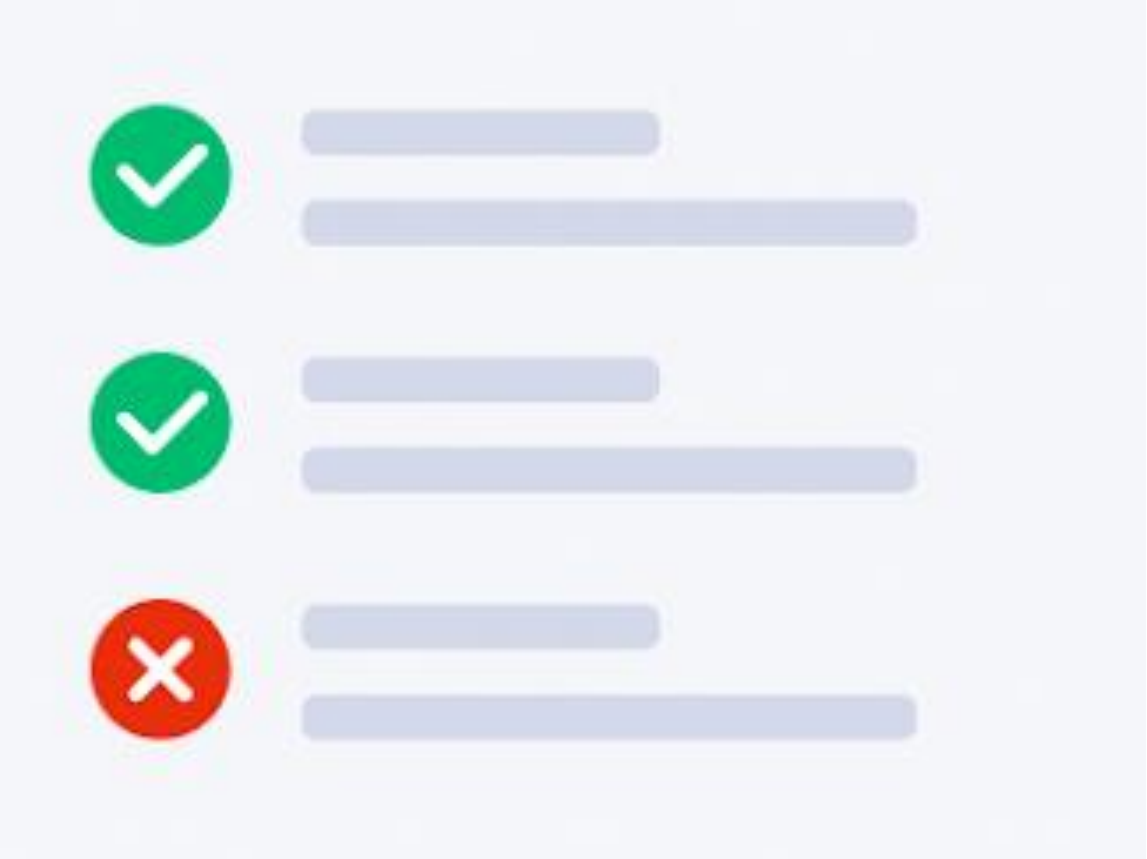
Authorization

Can you do that? verifying access to specific features or resources

Allow or deny the access to a specific resource or action.

Other terms with same meaning:

- access control/management
- client privileges



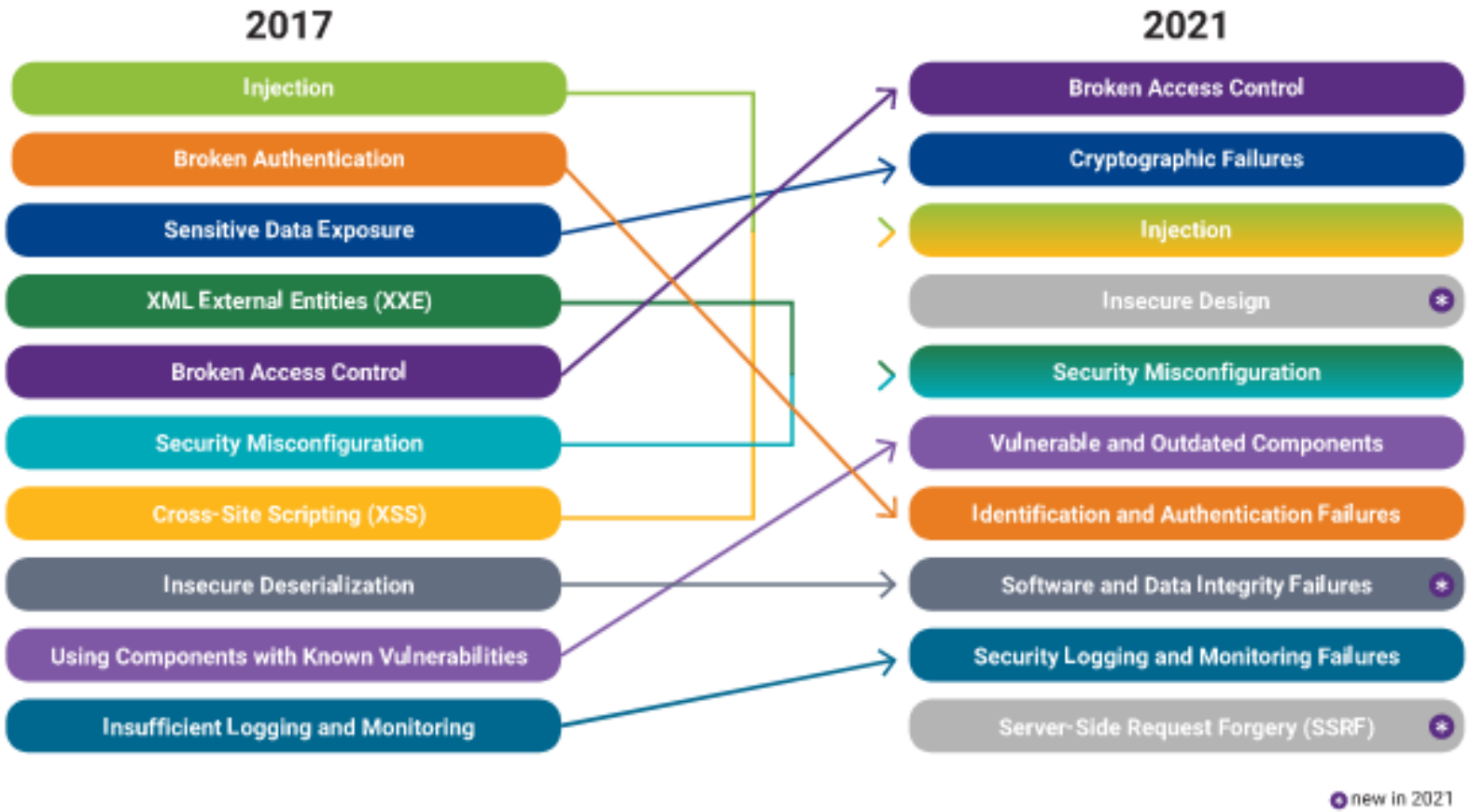
OWASP

The Open Worldwide Application Security Project® (OWASP) is a nonprofit foundation that works to improve the security of software.

Definitive source of security tools and resources:

- OWASP Top 10
- OWASP Cheat Sheet Series
- OWASP Dependency Check
- …

OWASP Top 10 (Vulnerabilities)

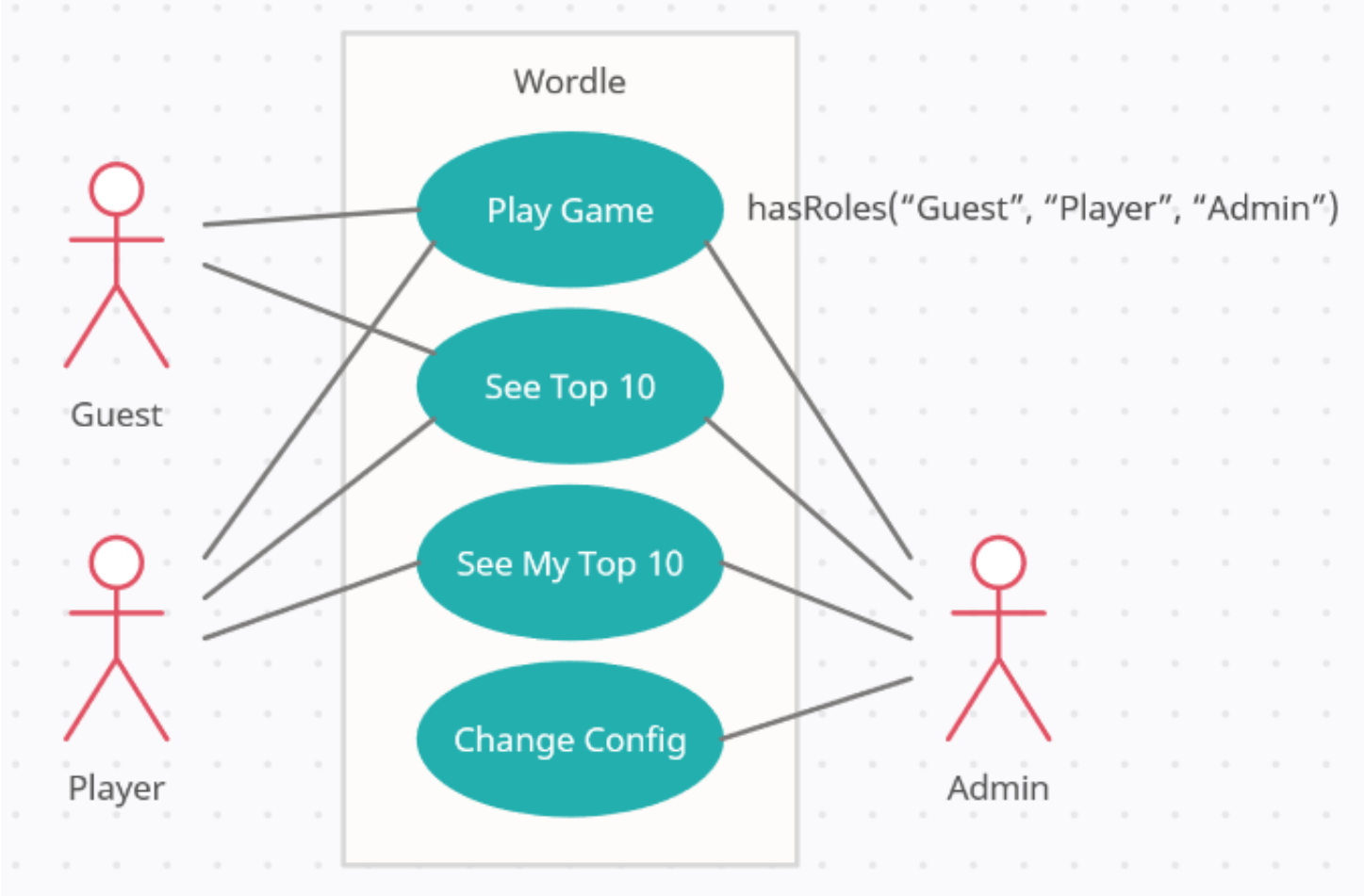


Access Control models

- Role-Based Access Control (RBAC)
- Resource-Based Access Control (RBAC)
- Attribute-Based Access Control (ABAC)
- Relations-Based Access Control (ReBAC)
  Suitable for Social Networks -> Only friends can see the posts

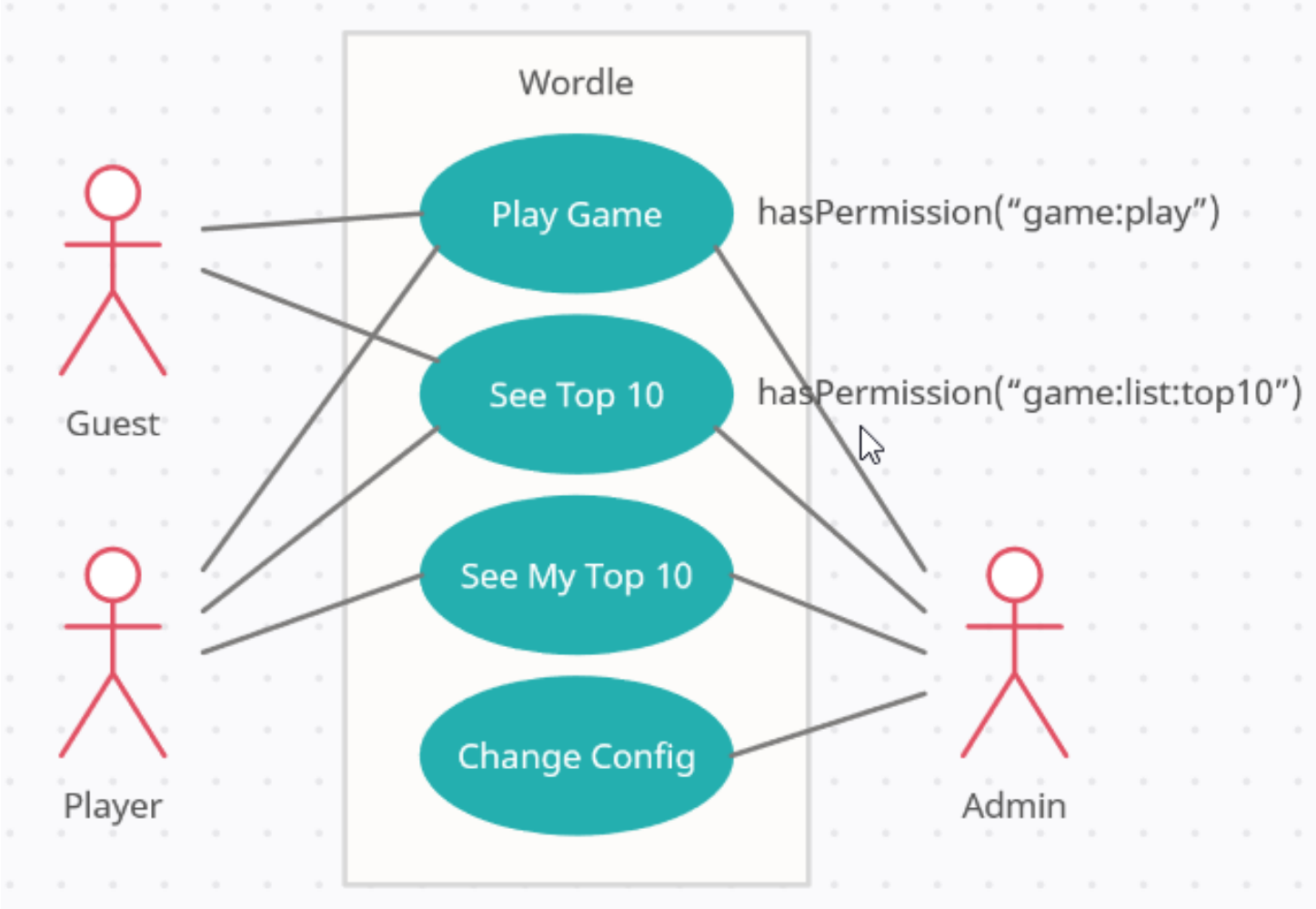
Role-Based Access Control

- Static checks against hard-coded roles
- User's roles are defined dynamically
- Any change in what a role can do requires coding



Resource-Based Access Control

- Static checks against hard-coded permissions
- Role's permissions are defined dynamically
- User's roles are defined dynamically

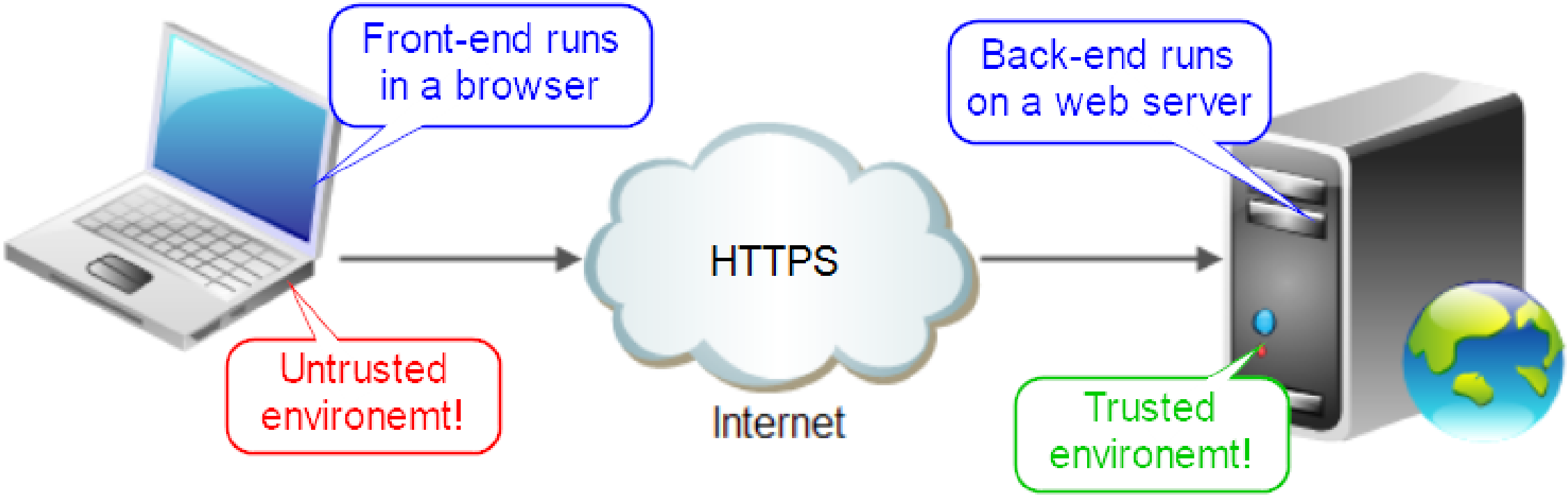


Attribute-Based Access Control (ABAC)

- Every Subject, Resource, Action or Environment could have attributes.
- Access control is defined using policies defining if access is grated/denied for a set of attributes
  - Subject's "job role" = "communications"
  - Subject's "business unit" = "marketing"
  - Action = "edit"
  - Resource "type" = "media strategy document"
  - Resource "business unit" = "marketing"
- More flexible than RBAC, but more complex to implement

Web Application Architecture

Nowadays applications use Client-Side Rendering, i.e., JavaScript running in a Browser (aka front-end) calling APIs on the back-end



OWASP Application Secuirty Verification Standard

- Verify that Authorization Checks are Performed in the Right Location
  - 1.4.1 Verify that trusted enforcement points, such as access control gateways, servers, and serverless functions, enforce access controls. Never enforce access controls on the client.
  - 4.1.1 Verify that the application enforces access control rules on a trusted service layer, especially if client-side access control is present and could be bypassed

Back-end authorization

Spring Security

Simple. Java. Security.

Back-end authz with Shiro

- Organized arround permissions, i.e. what a user can do in the application
- A permission depends on 3 parts devided by colon (last 1 is optional): **Resource, Action, Instance**

```
"permissions": [
  "game:play",
  "game:query:myTop10",
  "config:read",
  "config:update"
],
```

instance

resource

action

Back-end authz with Shiro

- Declarative way with Java annotations at class or method level
- Programmatic (imperative) way streight in the code

```
@GetMapping
@RequiresPermissions("config:read")
public ConfigModel getConfig() {
  return configService.getConfig();
}

@GetMapping()
public CollectionModel<GameModel> listLast10(@RequestParam String filter) {
  return switch (filter) {
    case "myTop10" -> {
      SecurityUtils.getSubject().checkPermission("game:query:myTop10");
      yield gameModelAssembler.toCollectionModel(gameService.listLast10());
    }
```

Front-end authorisation

Front-end authz with CASL

- Organized arround abilities, i.e. what a user can do in the application
- An ability depends on 4 parameters (last 3 are optional): **User Action, Subject, Fields, Conditions**

```
export default defineAbility((can, cannot) => {
  can('read', 'post');
  can('update', 'post');
  can('read', 'comment');
  cannot('update', 'comment');
});
```

action

subject (resource)

Front-end authz with CASL

- CASL React – React components for CASL
- DSL like syntax, easy to read and understand
- Show/hide or pass allowed value

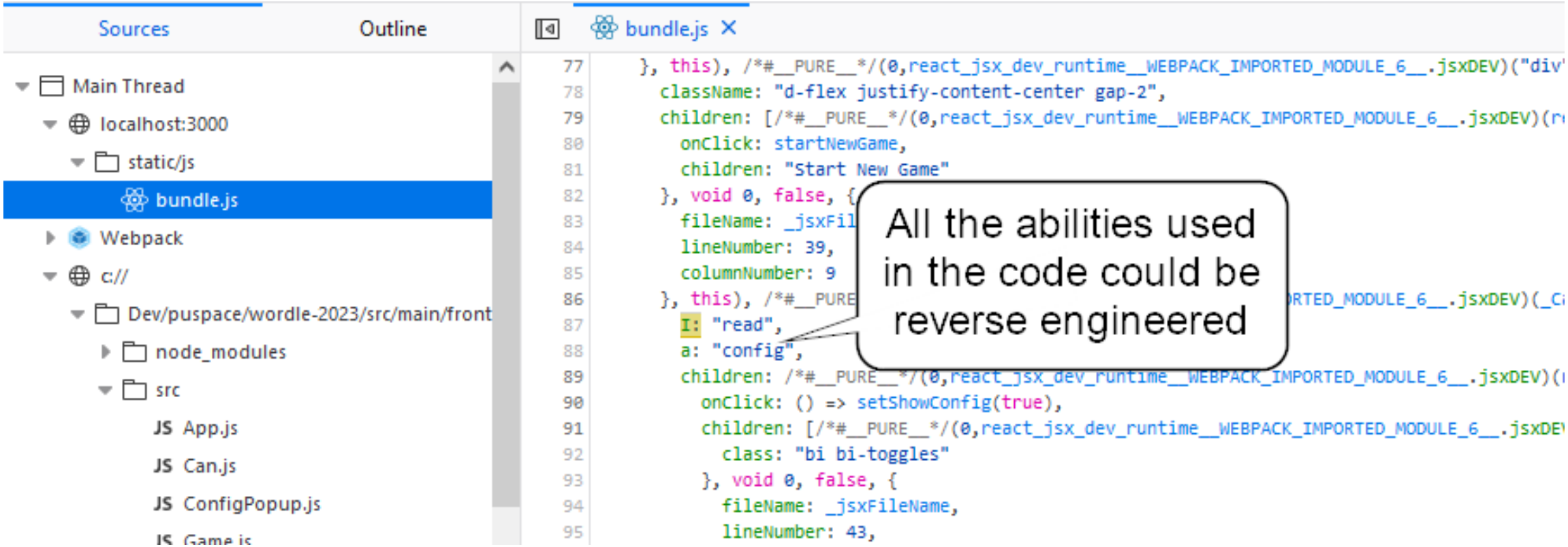
```
<!-- show/hide -->
<Can I="create" a="post">
  <button>Save</button>}
</Can>

<!-- pass allowed value -->
<Can I="create" a="post" passThrough>
  {allowed => <button disabled={!allowed}>Save</button>}
</Can>
```

What is wrong?

- If your JavaScript is prepared to handle all use cases then you have all REST endpoints hard-coded in the front-end
- The result: front-end source code becomes like Swagger UI for the back-end REST API

What is wrong?

- If your JavaScript is prepared to handle all use cases then you have all abilities listed there
- The result: front-end source code becomes like documentation for ALL permissions used in the back-end



How to prevent exposing sensitive information to the browser that is not intended for the current user?

HATEOS

Hypertext As The Engine Of Application State

Authorization with HATEOS

Replace passing the granted permissions with passing links to the resources with granted access

```
{
  "id": 2,
  "name": "Wordle Admin",
  "permissions": [
    "game:play",
    "game:query:myTop10",
    "config:read",
    "config:update"
  ]
}
```

```
{
  "id": 2,
  "name": "Wordle Admin",
  "_links": {
    "logout": { "href": "http://localhost:8080/logout" },
    "startNewGame": { "href": "http://localhost:8080/api/games" },
    "top10": { "href": "http://localhost:8080/api/games?filter=top10" },
    "myTop10": { "href": "http://localhost:8080/api/games?filter=myTop10" },
    "readConfig": { "href": "http://localhost:8080/api/config" },
    "self": {
        "href": "http://localhost:8080/api/users/{userId}",
        "templated": true
    }
  }
}
```

Authorization with HATEOS

Get rid of all hard-coded URI in the front-end, except one entry point

```
12 function App() {
13   const [filter, setFilter] = useState("top10");
14   const { data: user } = useSWR("/api/users/current");
15   const { data: top10Model } = useSWR(`/api/games?filter=${filter}`);
16   const top10 = top10Model?._embedded.gameModelList;
17
18   const [showSignIn, setShowSignIn] = useState(false);
19   const [showConfig, setShowConfig] = useState(false);
20
21   const navigate = useNavigate();
22   async function startNewGame() {
23     const response = await fetch("/api/games", { method: "POST" });
24     const game = await response.json();
25     navigate(`/games/${game.id}`);
26   }
27
28   async function signOut() {
29     await fetch("/logout");
30     mutate("/api/users/current");
31   }
```

```
11 function App() {
12   const [filter, setFilter] = useState("top10");
13   const { data: user } = useSWR("/api/users/current");
14   const { data: top10Model } = useSWR(user?._links[filter].href);
15   const top10 = top10Model?._embedded.gameModelList;
16
17   const [showSignIn, setShowSignIn] = useState(false);
18   const [showConfig, setShowConfig] = useState(false);
19
20   const navigate = useNavigate();
21   async function startNewGame() {
22     const response = await fetch("/api/games", { method: "POST" }
23     const game = await response.json();
24     navigate(`/games/${game.id}`);
25   }
26
27   async function signOut(link) {
28     await fetch(link.href);
29     mutate("/api/users/current");
30   }
```

Authorization with HATEOS

Get rid of @casl/react and all hard-coded `<Can I="…" a="…" />`

```
38
39          <Can I="read" a="config">
40            <>
41              <Button onClick={() => setShowConfig(true)}>
42                <i class="bi bi-toggles"></i> Config
43              </Button>
44              <ConfigPopup
45                show={showConfig}
46                onHide={() => setShowConfig(false)}
47              />
48            </>
49          </Can>
50
74            <Can I="query" a="game" field="myTop10">
75              <Nav.Item>
76                <Nav.Link eventKey="myTop10">My Top 10</Nav.Link>
77              </Nav.Item>
78            </Can>
```

```
37
38          {user?._links.readConfig && (
39            <>
40              <Button onClick={() => setShowConfig(true)}>
41                <i class="bi bi-toggles"></i> Config
42              </Button>
43              <ConfigPopup
44                show={showConfig}
45                onHide={() => setShowConfig(false)}
46                link={user?._links.readConfig}
47              />
48            </>
49          )}
50
86            {user?._links.myTop10 && (
87              <Nav.Item>
88                <Nav.Link eventKey="myTop10">My Top 10</Nav.Link> Link>
89              </Nav.Item>
90            )}
```

More secure

- **No hard-coded permissions**
  Nobody can reverse engineer what functionality your application provide over what he is granted to see
- **No hard-coded links**
  Nobody can reverse engineer your application APIs over the APIs he is granted to use
- **The decision what the front-end can see is taken entirely in the trusted back-end**

Q &A