00.14

JAVASCRIPT.JS
Opslag

javascript.js

PASSWORD

password must match the following: "/^(?!.*\s)(?=.{8,256}$)(?=.*[a-z])(?=.*[A-Z])((?=.*[0-9])|(?=.*[~!@#$%^&*()-=_+[\]{}|;:,./<>?])).*$/"

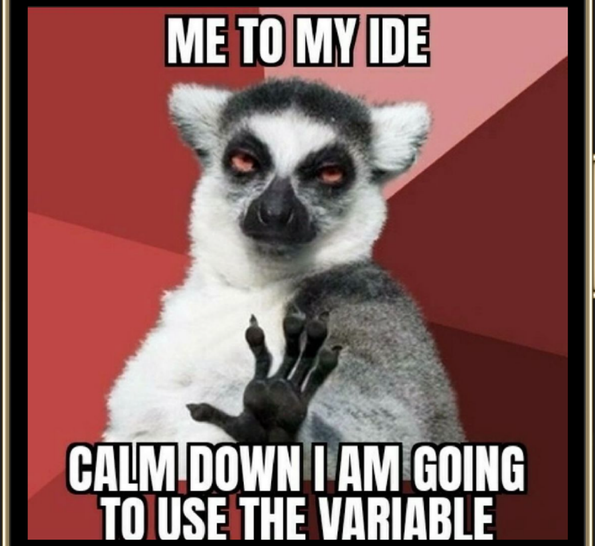
Synes godt om fra **kasialaniecka** og andre
javascript.js 😅... mere
Vis alle 51 kommentarer
for 5 dage siden · **Se oversættelse**

javascript.js

00.15

JAVASCRIPT.JS
Opslag

javascript.js



Synes godt om fra **wolflorena** og andre
javascript.js 🤓... mere
Vis alle 41 kommentarer
31. marts · **Se oversættelse**

javascript.js

00.18

JAVASCRIPT.JS
Opslag

javascript.js

Synes godt om fra **gabriela.rodickova** og andre
javascript.js Good one - JS 🔥🙌... mere
Vis alle 185 kommentarer
7. januar 2023 · **Se oversættelse**

javascript.js

Adam Bertram
@adbertram

Authentication & Authorization

localhost:3000/signin

Sign In

Mail: Type your mail...

Password: Type your password...

Sign In

No account? Sign up here.

localhost:3000/signup

Sign Up

Mail: Type your mail...

Password: Type your password...

Sign Up

Already have an account? Sign in here.

Remix Post App

localhost:3000/posts

POSTS | ADD POST | PROFILE

Posts

Rasmus Cederdorff
Senior Lecturer

Exploring the city center of Aarhus

Dan Okkels Brendstrup
Lecturer

A cozy morning with coffee

Rasmus Cederdorff
Senior Lecturer

Serenity of the forest

Maria Louise Bendixen

Anne Kirketerp
Head of Department

Rasmus Cederdorff
Senior Lecturer

localhost:3000/posts/65d1f502f5b3f142ef417bb8

localhost:3000/profile

POSTS | ADD POST | PROFILE

Profile

Name: Rasmus Cederdorff

Title: Senior Lecturer

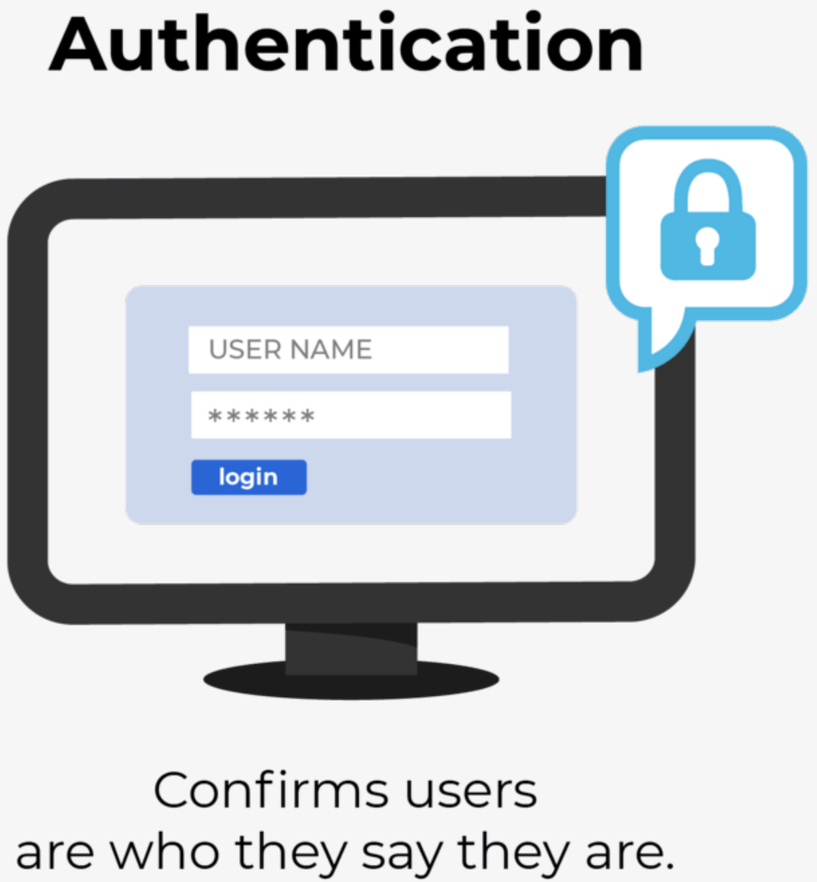
Mail: race@eaaa.dk

Logout

Authentication (godkendelse)

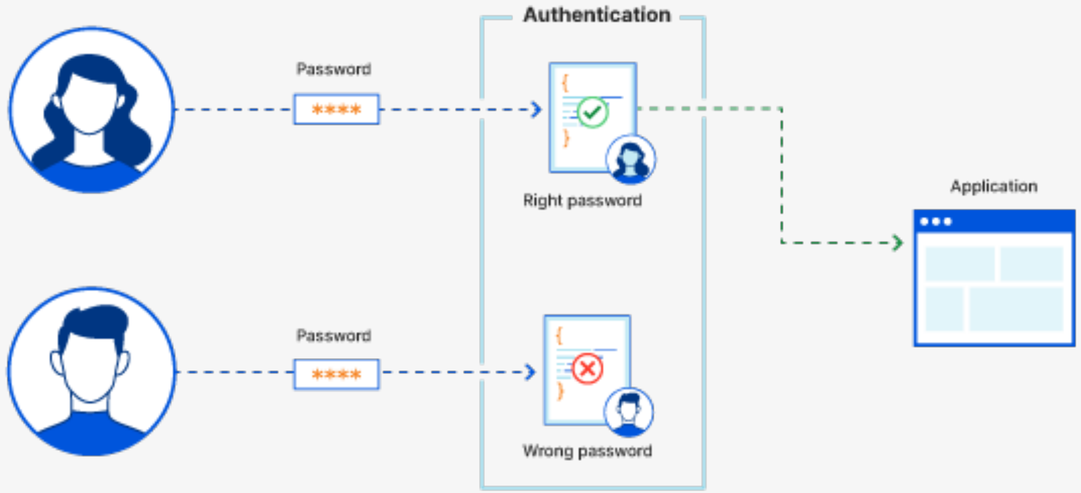
Bekræfter hvem en bruger er.

- Eks.: Logge ind med e-mail og adgangskode, Google-login, Facebook, GitHub osv.
- Hvis godkendelsen lykkes, får brugeren adgang til systemet.



Authentication?

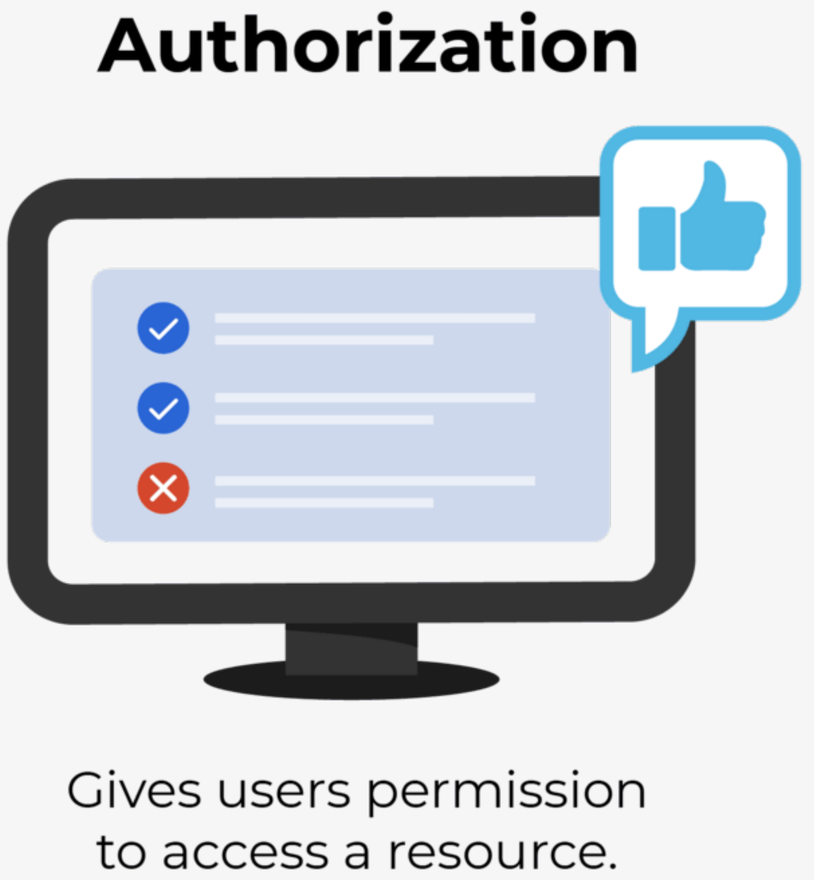
Authentication is the process of **verifying the identity of a user or system**, typically through credentials like **usernames and passwords**, **biometric data**, or security **tokens**, to ensure that individuals are who they claim to be before **granting access** to secure systems or information.



Authorization (autorisering)

Bestemmer hvad en bruger har adgang til.

- Eks.: En admin-bruger kan se og ændre indstillinger, mens en almindelig bruger kun kan se sin egen profil.
- Finder sted **efter** authentication.

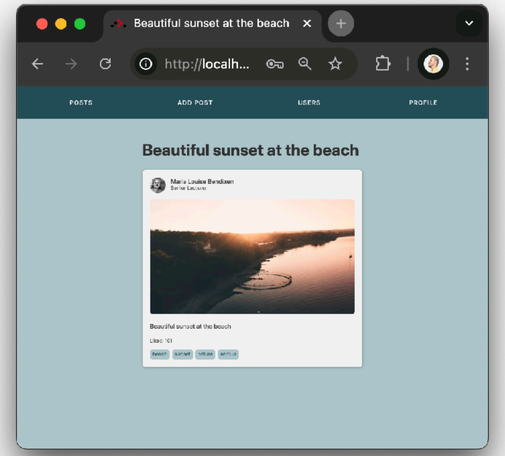


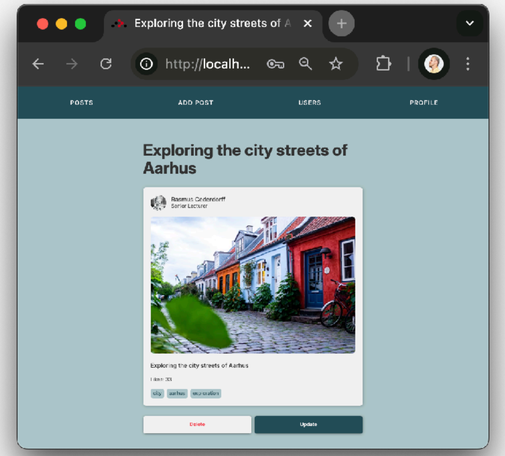
Authentication = "Er du den, du siger, du er?”;

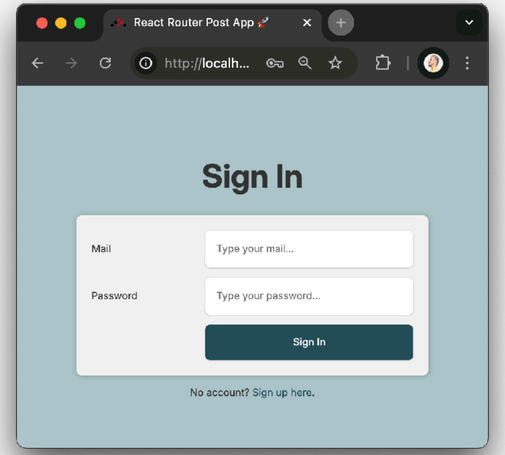
Authorization = "Hvad har du lov til at gøre?”;

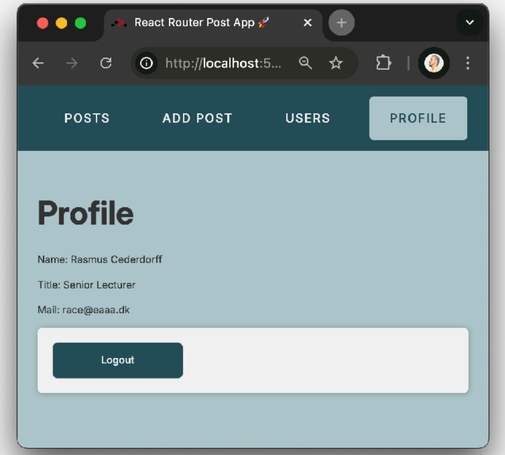
Eksempel 1: Webapp med login

- **Authentication**:
  - Brugeren logger ind med e-mail og adgangskode (eller Google/ Facebook-login).
- **Authorization**:
  - En almindelig bruger kan kun se sin egen konto.
  - En admin-bruger kan redigere alle brugere og ændre systemindstillinger.









Eksempel 2: GitHub

- **Authentication**:
  - Du logger ind på GitHub med dit brugernavn og adgangskode.
- **Authorization**:
  - Du kan læse og ændre dine egne repositories.
  - Du kan kun bidrage til andres repositories, hvis du har fået de rette tilladelser (fx read, write eller admin access).

Eksempel 3:
Firebase Authentication + Firestore Rules

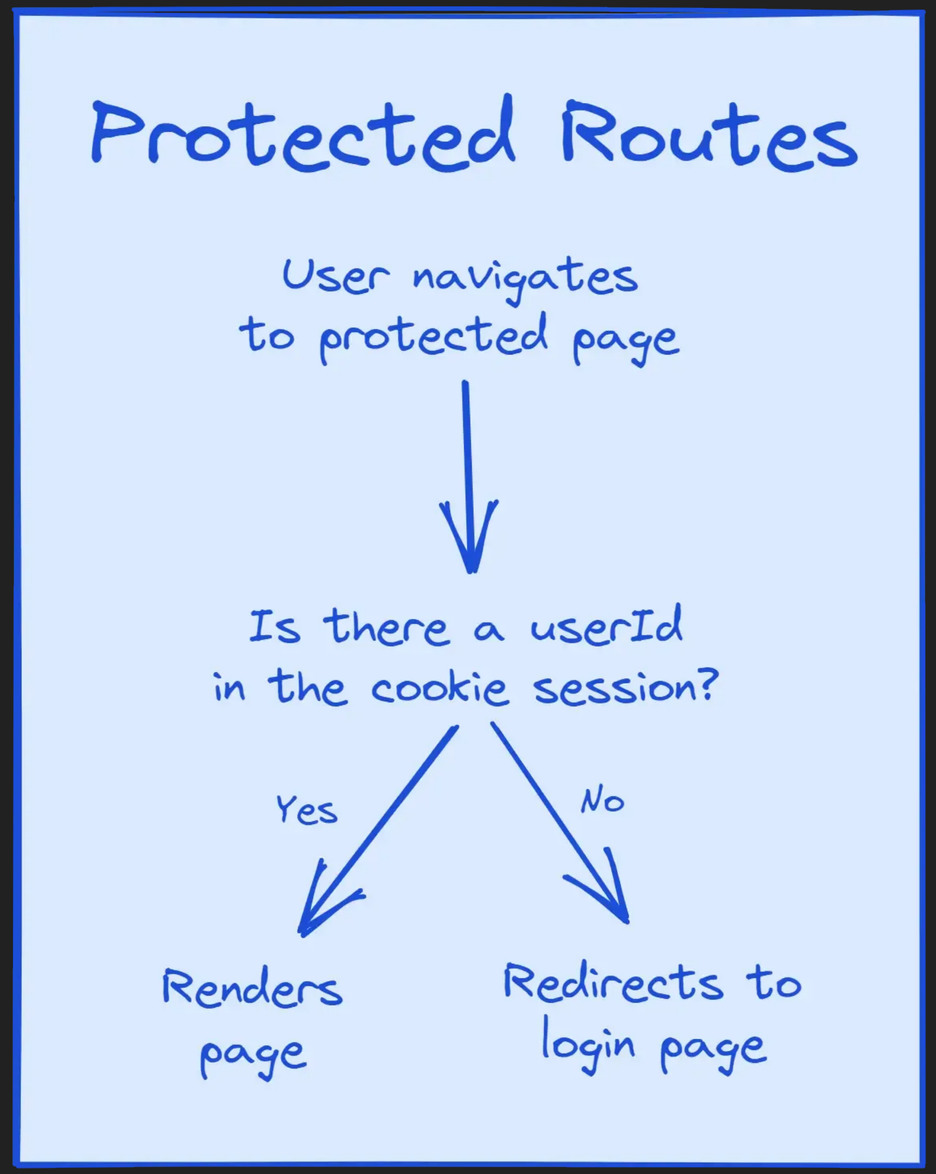
- **Authentication**:
  - Firebase håndterer login med e-mail, Google, Facebook osv.
- **Authorization:**
  - Firestore Security Rules sikrer, at:
    - Brugere kun kan læse/skrive deres egne data.
    - Admins har fuld adgang til databasen.

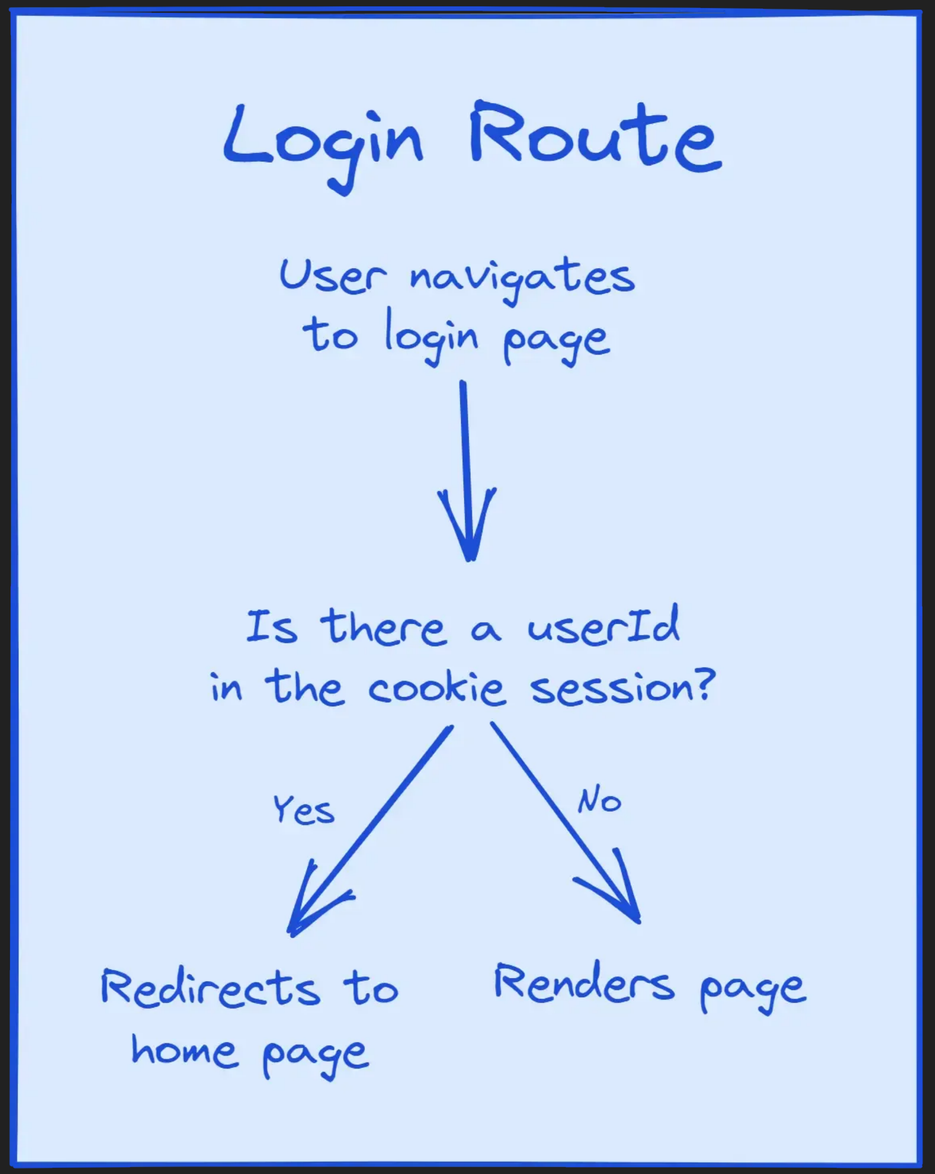

```
{
  "rules": {
    "users": {
      ".read": true,
      "$user":{
        ".write": "auth != null && auth.uid == $user",  // Only authenticated users
        ".validate": "newData.hasChildren(['name', 'mail', 'image'])",
        "name": { ".validate": "newData.val().length > 0" },
        "mail": { ".validate": "newData.val().length > 0" },
        "image": { ".validate": "newData.val().length > 0"}
      },
      ".indexOn": ["name", "mail", ".value"]
    },
    "posts": {
      ".read": true,
      "$post":{
        ".write": "auth !== null && (auth.uid == newData.child('uid').val() || auth
        ".validate": "newData.hasChildren(['caption', 'image', 'uid'])",
        "caption": { ".validate": "newData.val().length > 0" },
        "image": { ".validate": "newData.val().length > 0"},
        "uid": { ".validate": "newData.val().length > 0"}
      },
      ".indexOn": ["uid"]
    }
  }
}
```

Auth

Auth

- Remix Auth is a library for implementing authentication in Remix / RR7 Applications.
- Simplifies the process of adding authentication by providing a structured way to handle user authentication flows, including login, logout, and session management.
- Heavily inspired by Passport.js





https://www.mattstobbs.com/remix-authentication/

Auth

- Full Server-Side Authentication
- Complete TypeScript Support
- Strategy-based Authentication
- Easily handle success and failure
- Implement custom strategies
- Supports persistent sessions

Auth

- **Setup**: You integrate Remix Auth into your Remix project by installing the necessary package and setting up authentication strategies.
- **Strategies**: Remix Auth supports various authentication strategies, such as local (username and password), OAuth2 (Google, GitHub), and more. These strategies define how users will be authenticated.
- **Session Management**: It handles session creation, validation, and destruction, allowing you to manage user sessions securely.
- **Middleware & Hooks**: Remix Auth provides middleware and hooks to protect routes and access user information within your components, ensuring that certain parts of your app are accessible only to authenticated users.

Auth

The core idea is to provide a developer-friendly, secure, and flexible way to add authentication to your Remix applications, leveraging the framework's conventions and the broader ecosystem of authentication providers.

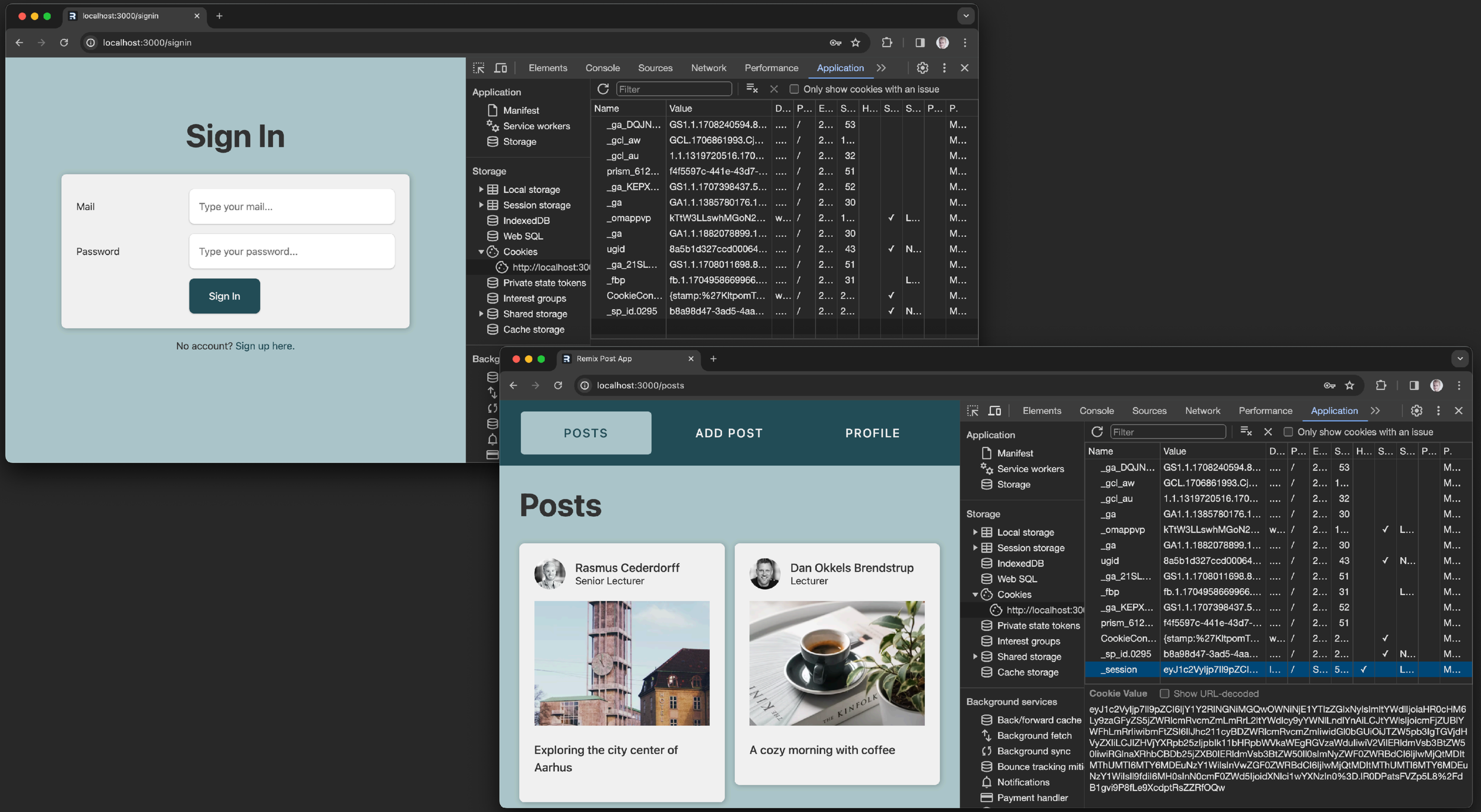
Cookies

- Small piece of data stored on the user's browser.
- The web server uses cookies to store user specific data on the client.
- Sent from a website during browsing.
- Used to remember stateful information or user's browsing activity.
- Can track items in a shopping cart, user preferences, and login status.
- Helps pre-fill form fields like names and passwords.
- Enables personalized user experiences on websites.

Cookies: In the context of Remix

Auth

- Utilized to manage authentication states and user sessions.
- Stores session IDs or tokens to maintain user login across visits.
- Ensures secure transmission of authentication information.
- Facilitates secure, stateful interaction within Remix applications.


localhost:3000/signin
Sign In
Mail
Type your mail...
Password
Type your password...
Sign In
No account? Sign up here.
Elements
Console
Sources
Network
Performance
Application
Application
Manifest
Service workers
Storage
Storage
Local storage
Session storage
IndexedDB
Web SQL
Cookies
Private state tokens
Interest groups
Shared storage
Cache storage
Filter
Only show cookies with an issue
Name
Value
Remix Post App
localhost:3000/posts
POSTS
ADD POST
PROFILE
Posts
Rasmus Cederdorff
Senior Lecturer
Exploring the city center of Aarhus
Dan Okkels Brendstrup
Lecturer
A cozy morning with coffee
Background services
Back/forward cache
Background fetch
Background sync
Bounce tracking miti
Notifications
Payment handler
Cookie Value
Show URL-decoded

But Why?

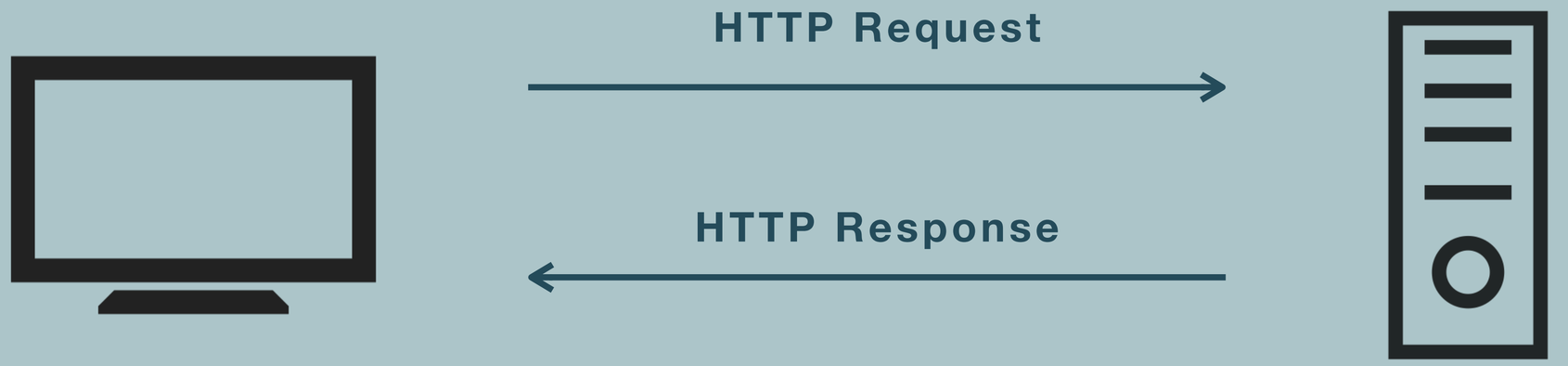
HTTP is stateless

- HTTP is designed to be a stateless protocol, meaning that each request is processed independently of previous requests.
- When a server receives an HTTP request, it is processed based solely on the information contained in the request.

Auth

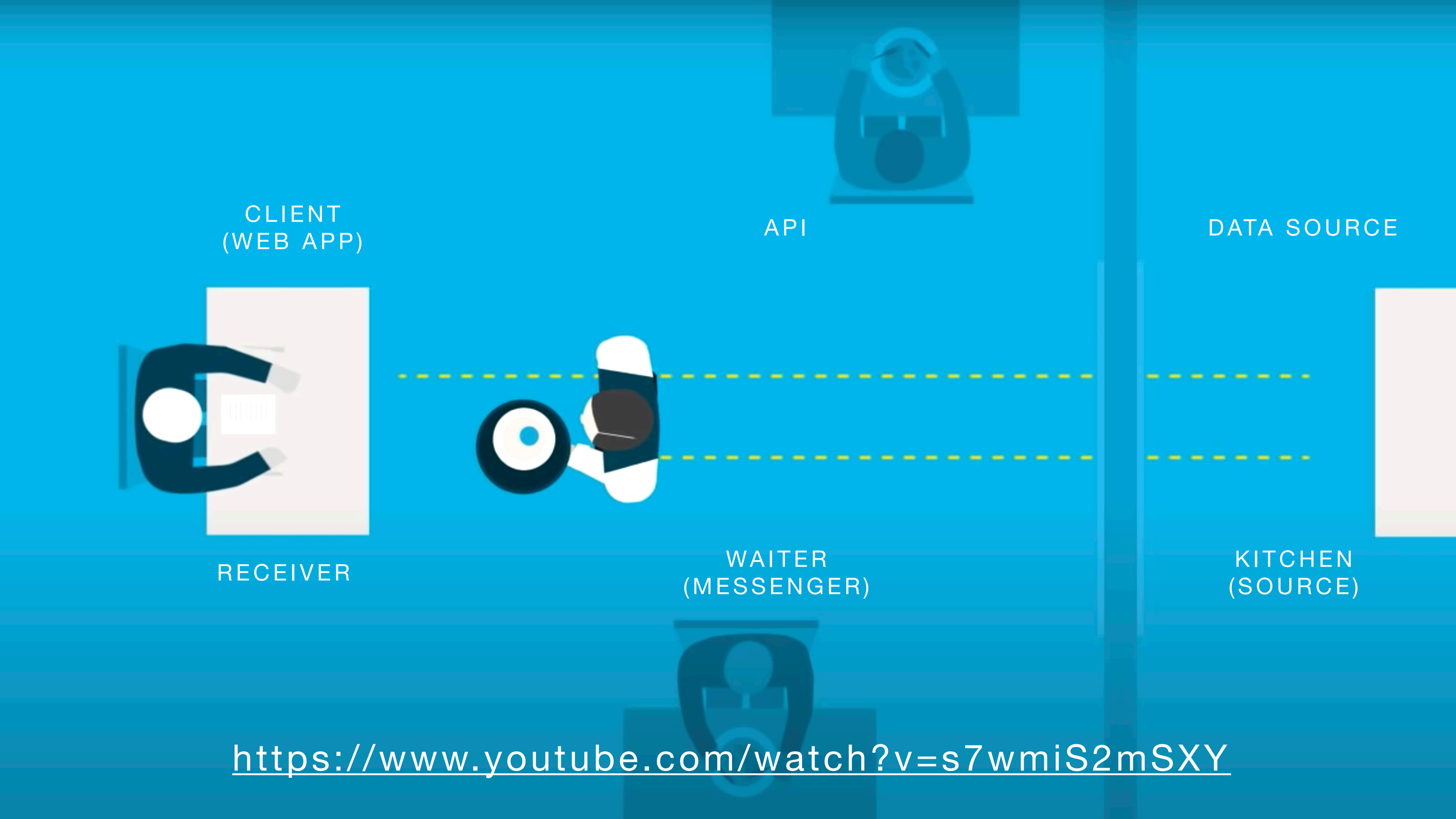
Client-Server Model

Communication between web **clients** and web **servers**.



Client-Server Model

Communication between web **clients** and web **servers**.

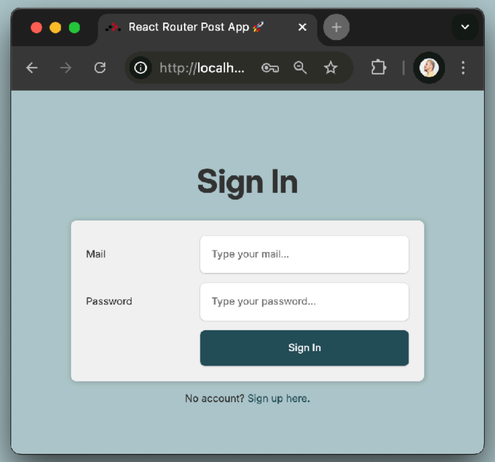


https://www.youtube.com/watch?v=s7wmiS2mSXY

Flow for authentication med remix-auth-form-strategy

1. **Brugeren indtaster loginoplysninger**

    - Brugeren udfylder en login-form (email + password) og sender den til en Remix action.




```tsx
export default function SignIn({ actionData }: Route.ComponentProps) {
  return (
    <div id="sign-in-page" className="page">
      <h1>Sign In</h1>
      <Form id="sign-in-form" method="post">…
      </Form>
      <p>
        No account? <NavLink to="/signup">Sign up here.</NavLink>
      </p>
    </div>
  );
}

// We need to export an action function, here we will use the
// `authenticator.authenticate method`
export async function action({ request }: Route.ActionArgs) {
  try {
    // we call the method with the name of the strategy we want to use and the
    // request object
    let userId = await authenticator.authenticate("user-pass", request);
    let session = await sessionStorage.getSession(request.headers.get("cookie"))
    session.set("authUserId", userId);
    return redirect("/", {
      headers: { "Set-Cookie": await sessionStorage.commitSession(session) }
    });
  } catch (error) {
    if (error instanceof Error) {
      // here the error related to the authentication process
      return data({ error: error.message });
    }
  }
}
```

Flow for authentication med remix-auth-form-strategy

2. Login-handling i en Remix action

- remix-auth-form-strategy tager de indsendte data og forsøger at validere brugeren.
- Hvis login-oplysningerne er korrekte, oprettes en session, og brugeren bliver logget ind.
- Hvis der er fejl (forkert email/ adgangskode), returneres en fejlmeddelelse.

```
export let authenticator = new Authenticator<string>();

// Tell the Authenticator to use the form strategy
authenticator.use(
  new FormStrategy(async ({ form }) => {
    const mail = form.get("mail");
    const password = form.get("password");

    // do some validation, errors are saved in the sessionErrorKey
    if (!mail || typeof mail !== "string" || !mail.trim()) {
      throw new Error("Email is required and must be a string");
    }

    if (!password || typeof password !== "string" || !password.trim()) {
      throw new Error("Password is required and must be a string");
    }

    // verify the user
    return await verifyUser(mail, password);
  }),
  "user-pass"
);

async function verifyUser(mail: string, password: string) {
  const user = await User.findOne({ mail }).select("+password");
  if (!user) {
    throw new Error("No user found with this email.");
    // throw new AuthorizationError("No user found with this email.");
  }

  const passwordMatch = await bcrypt.compare(password, user.password);
  if (!passwordMatch) {
```

Flow for authentication med remix-auth-form-strategy

3. Oprettelse af en session

- Hvis brugeren valideres, gemmes sessionen i cookies eller en database.
- Brugeren bliver derefter redirected til en beskyttet side (se action).

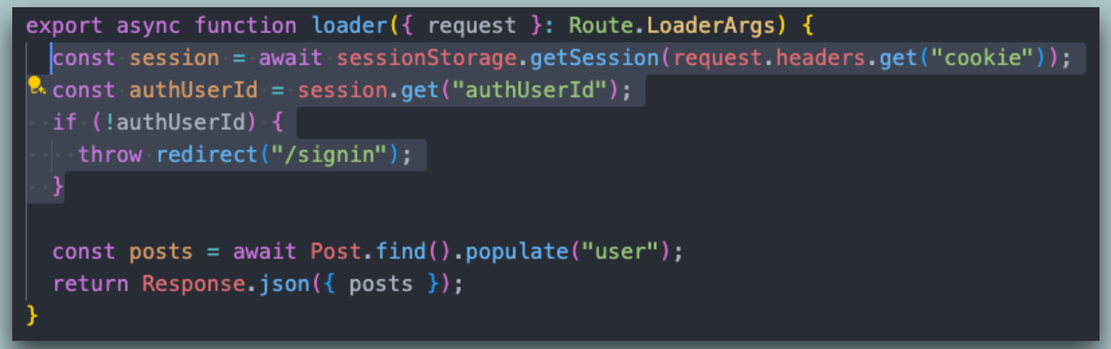
```ts
export const sessionStorage = createCookieSessionStorage({
  cookie: {
    name: "_session", // use any name you want here
    sameSite: "lax", // this helps with CSRF
    path: "/", // remember to add this so the cookie will work in all routes
    httpOnly: true, // for security reasons, make this cookie http only
    secrets: ["s3cr3t"], // replace this with an actual secret
    secure: process.env.NODE_ENV === "production" // enable this in prod only
  }
});

// you can also export the methods individually for your own usage
export const { getSession, commitSession, destroySession } = sessionStorage;
```

```ts
export async function action({ request }: Route.ActionArgs) {
  try {
    // we call the method with the name of the strategy we want to use and the
    // request object
    let userId = await authenticator.authenticate("user-pass", request);
    let session = await sessionStorage.getSession(request.headers.get("cookie"));
    session.set("authUserId", userId);
    return redirect("/", {
      headers: { "Set-Cookie": await sessionStorage.commitSession(session) }
    });
  } catch (error) {
    if (error instanceof Error) {
      // here the error related to the authentication process
      return data({ error: error.message });
    }
  }
}
```

Flow for authentication med remix-auth-form-strategy

4. Beskyttelse af ruter

- Når brugeren besøger en beskyttet rute, tjekkes sessionen.
- Hvis sessionen er gyldig, vises indholdet.
- Hvis sessionen er ugyldig, redirectes brugeren til login-siden.



Sessions

- A session is a way to persist data across multiple HTTP requests, creating a continuous and stateful interaction between a client and a server.
- It typically involves assigning a unique identifier (session ID) to each user's interaction, which is stored on the server and the client (usually within a cookie).
- This session ID allows the server to retrieve the stored session data, such as user authentication status, shopping cart contents, or any other user-specific information, for the duration of the session.
- Sessions help overcome the stateless nature of HTTP by maintaining a stateful context for each user.

Auth

Sessions

- **Persistence Across Requests**: Maintains data across multiple HTTP requests.
- **Unique Identifier**: Assigns a unique session ID to each user interaction.
- **Server and Client Storage**: Stores session ID on the server and in the client's cookie.
- **User-Specific Data**: Holds data like authentication status and shopping cart contents.
- **Stateful Interaction**: Overcomes HTTP's stateless nature by maintaining user context.
- **Duration**: Has a defined lifetime, after which the session expires.

Auth

So Cookies and Sessions?

Are used to achieve statefulness in an inherently stateless HTTP protocol.

- **State Management**: Cookies and sessions enable continuous user experiences by remembering user interactions across requests.
- **User Authentication**: Facilitate login processes, allowing users to stay authenticated as they browse different pages without re-logging in.
- **Personalization**: Store preferences for customized content and experiences.
- **Session Management**: Securely store sensitive data server-side, using session IDs in cookies for linkage, protecting client-side exposure.
- **Efficiency**: Reduce database queries by storing necessary data for ongoing interactions in sessions.
- **Security**: Cookies' secure attributes and server-side sessions help prevent XSS and CSRF threats.

Auth

Session vs Token Authentication in 100 Seconds

https://www.youtube.com/watch?v=UBUNrFtufWo

remix-auth

- Uses cookies and sessions to manage authentication and maintain user sessions in Remix applications.
- Leverages cookies to store session IDs and manage user sessions on the server, enabling secure, authenticated user experiences

```ts
import { createCookieSessionStorage } from "@remix-run/nod

// export the whole sessionStorage object
export let sessionStorage = createCookieSessionStorage({
  cookie: {
    name: "_session", // use any name you want here
    sameSite: "lax", // this helps with CSRF
    path: "/", // remember to add this so the cookie will
    httpOnly: true, // for security reasons, make this coo
    secrets: ["s3cr3t"], // replace this with an actual se
    secure: process.env.NODE_ENV === "production" // enabl
  }
});

// you can also export the methods individually for your c
export let { getSession, commitSession, destroySession } =
```

session.server.ts

Storing Session IDs

- Uses cookies to store a unique session identifier (session ID) on the client's browser.
- This session ID is key to linking the client with its session data stored on the server.

```ts
import { createCookieSessionStorage } from "@remix-run/nod

// export the whole sessionStorage object
export let sessionStorage = createCookieSessionStorage({
  cookie: {
    name: "_session", // use any name you want here
    sameSite: "lax", // this helps with CSRF
    path: "/", // remember to add this so the cookie will
    httpOnly: true, // for security reasons, make this coo
    secrets: ["s3cr3t"], // replace this with an actual se
    secure: process.env.NODE_ENV === "production" // enabl
  }
});

// you can also export the methods individually for your c
export let { getSession, commitSession, destroySession } =
```

session.server.ts

Session Management

- On the server side, Remix Auth manages session data, such as user authentication status or other user-specific information.
- When a request comes in, it checks the session ID from the cookie to retrieve and validate the session data.

```ts
import { createCookieSessionStorage } from "@remix-run/nod

// export the whole sessionStorage object
export let sessionStorage = createCookieSessionStorage({
  cookie: {
    name: "_session", // use any name you want here
    sameSite: "lax", // this helps with CSRF
    path: "/", // remember to add this so the cookie will
    httpOnly: true, // for security reasons, make this coo
    secrets: ["s3cr3t"], // replace this with an actual se
    secure: process.env.NODE_ENV === "production" // enabl
  }
});

// you can also export the methods individually for your c
export let { getSession, commitSession, destroySession } =
```

session.server.ts

Secure Transmission

- It ensures that cookies are set with security options, such as HttpOnly and Secure flags, to prevent access to the cookie via client-side scripts and ensure cookies are transmitted securely over HTTPS.

```ts
import { createCookieSessionStorage } from "@remix-run/nod

// export the whole sessionStorage object
export let sessionStorage = createCookieSessionStorage({
  cookie: {
    name: "_session", // use any name you want here
    sameSite: "lax", // this helps with CSRF
    path: "/", // remember to add this so the cookie will
    httpOnly: true, // for security reasons, make this coo
    secrets: ["s3cr3t"], // replace this with an actual se
    secure: process.env.NODE_ENV === "production" // enabl
  }
});

// you can also export the methods individually for your c
export let { getSession, commitSession, destroySession } =
```

session.server.ts

Authentication Flows

- During the login process, Remix Auth creates a new session for the authenticated user and stores the session ID in a cookie.

auth.server.ts

```ts
// Create an instance of the authenticator, pass a generic with what
// strategies will return and will store in the session
export let authenticator = new Authenticator<string>();

// Tell the Authenticator to use the form strategy
authenticator.use(
  new FormStrategy(async ({ form }) => {…
  }),
  "user-pass"
);
```

route (signin)

```ts
let userId = await authenticator.authenticate("user-pass", request);
let session = await sessionStorage.getSession(request.headers.get("cookie"));
session.set("authUserId", userId);
return redirect("/", {
  headers: { "Set-Cookie": await sessionStorage.commitSession(session) }
});
```

Authentication Flows

- For logout, it destroys the session on the server and clears the cookie on the client side.

```
export async function action({ request }: Route.ActionArgs) {
  // Get the session
  const session = await sessionStorage.getSession(
    request.headers.get("cookie")
  );
  // Destroy the session and redirect to the signin page
  return redirect("/signin", {
    headers: { "Set-Cookie": await sessionStorage.destroySession(session) }
  });
}
```

Access Control

- Remix Auth uses the session data to control access to routes within the application.
- You can check if a user is authenticated and authorize access to protected resources based on the session information.

```
export async function loader({ request }: Route.LoaderArgs) {
  const session = await sessionStorage.getSession(
    request.headers.get("cookie")
  );
  const authUserId = session.get("authUserId");
  if (!authUserId) {
    throw redirect("/signin");
  }
}
```

Auth: Route protection & restriction

Protect routes

- Loaders: protect all your loaders!
- Actions: protect all your actions!

Loaders and actions must return a response

```
export async function loader({ request }: Route.LoaderArgs) {
  const session = await sessionStorage.getSession(
    request.headers.get("cookie")
  );
  const authUserId = session.get("authUserId");
  if (!authUserId) {
    throw redirect("/signin");
  }

  const posts = await Post.find().populate("user");
  return Response.json({ posts });
}
```

```
export async function action({ request, params }: Route.ActionArgs) {
  const session = await sessionStorage.getSession(
    request.headers.get("cookie")
  );
  const authUserId = session.get("authUserId");
  if (!authUserId) {
    throw redirect("/signin");
  }

  const formData = await request.formData();

  await Post.findByIdAndUpdate(params.id, {
    caption: formData.get("caption"),
    image: formData.get("image")
```

Restrictions

- Assigning Ownership: Assign the Authenticated User to a new post.
- Make sure the authenticated user becomes the owner/ creator of a new post.

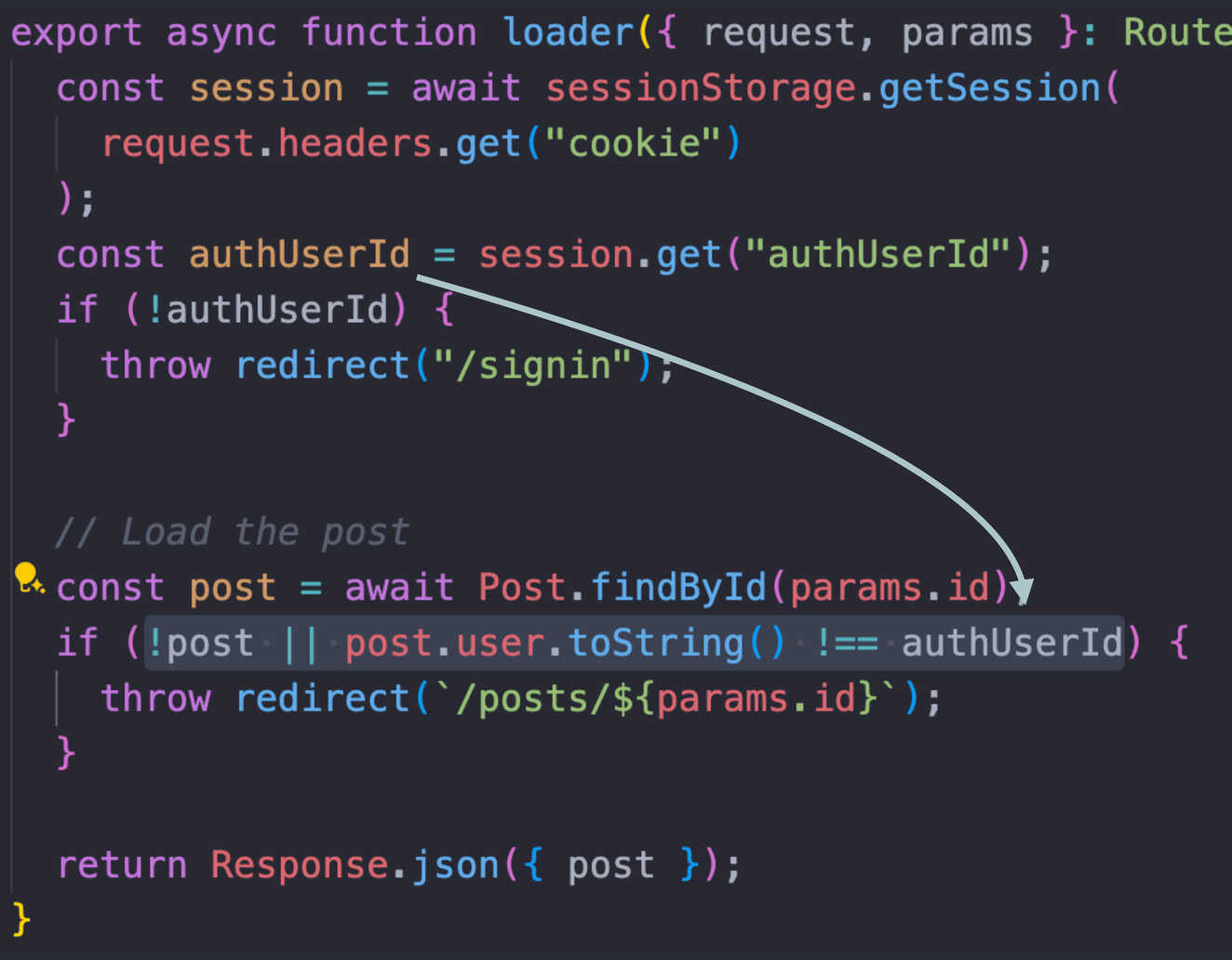
```
export async function action({ request }: Route.ActionArgs) {
  const session = await sessionStorage.getSession(
    request.headers.get("cookie")
  );
  const authUserId = session.get("authUserId");
  if (!authUserId) {
    throw redirect("/signin");
  }

  const formData = await request.formData();

  // Extract and typecast values correctly
  const caption = formData.get("caption");
  const image = formData.get("image");

  try {
    // Create the post and ensure it's awaited
    await Post.create({
      caption,
      image,
      user: authUserId
    });

    return redirect("/");
  } catch (error) {
```

Restrictions

- User Restrictions: a user cannot update and delete other users' posts.
- Make sure only the authenticated/owner/ creator of the post can update and delete.

Restrictions

- Restrict the loader and action handler for update post.
- Ensure only the post's authenticated/ owner/creator can update.
- Check the id of the auth user and the user of the post.

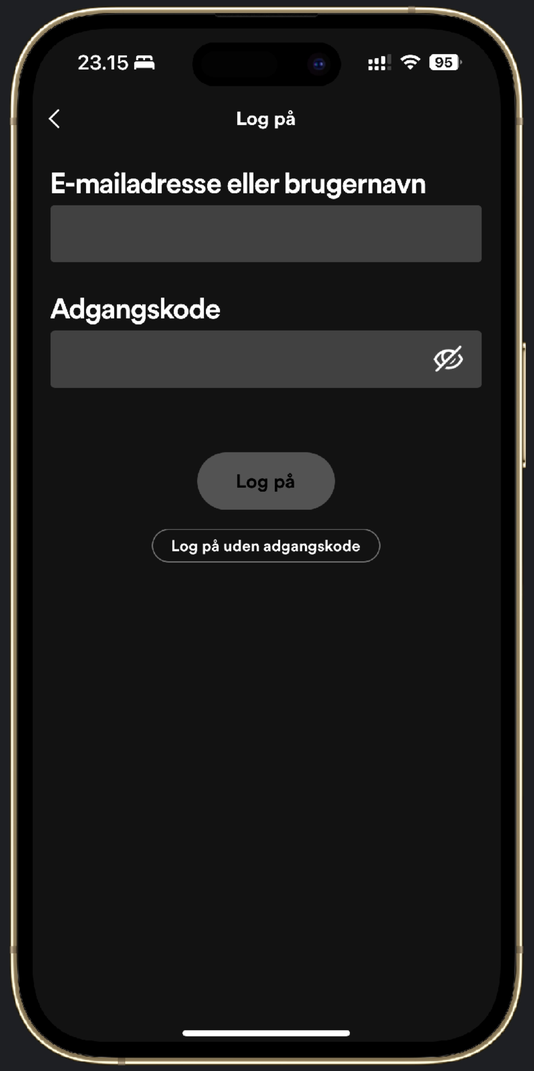


Auth OAuth2 Strategy

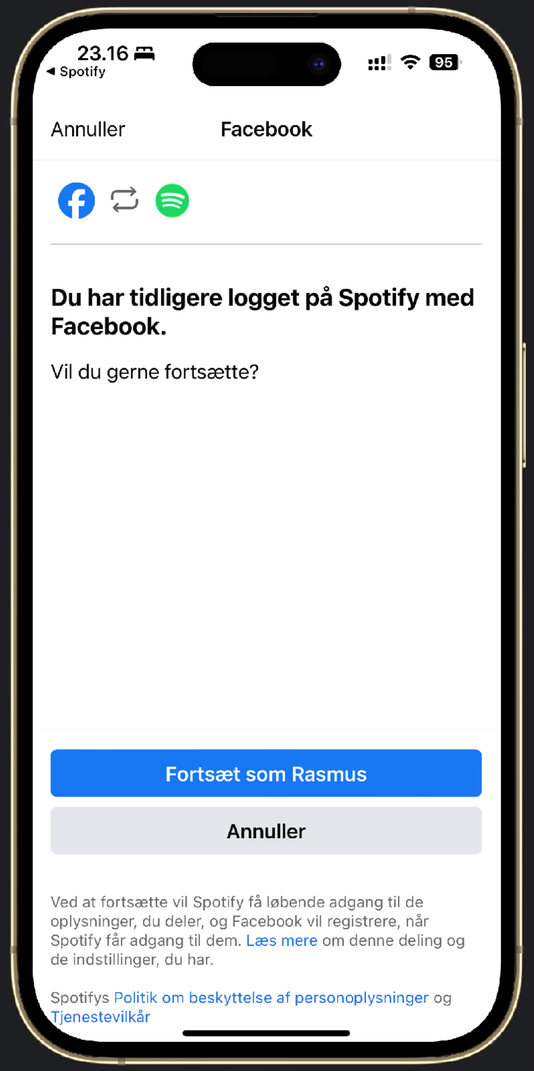
What is OAuth 2.0?

100 SECONDS OF

https://www.youtube.com/watch?v=yufqeJLP1rI&t=398s&ab_channel=Fireship



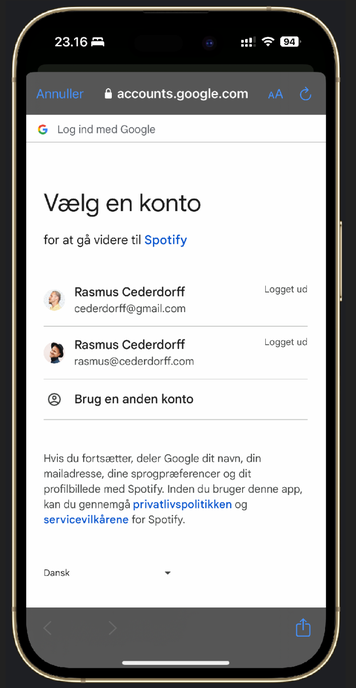
Form Strategy

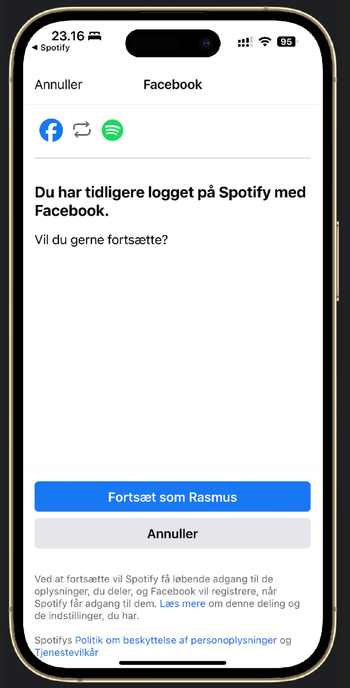


OAuth 2.0 Strategy: Google & Facebook

What is OAuth 2.0?

“OAuth 2.0, which stands for ‘Open Authorization’, is a standard designed to allow a website or application to access resources hosted by other web apps on behalf of a user.”





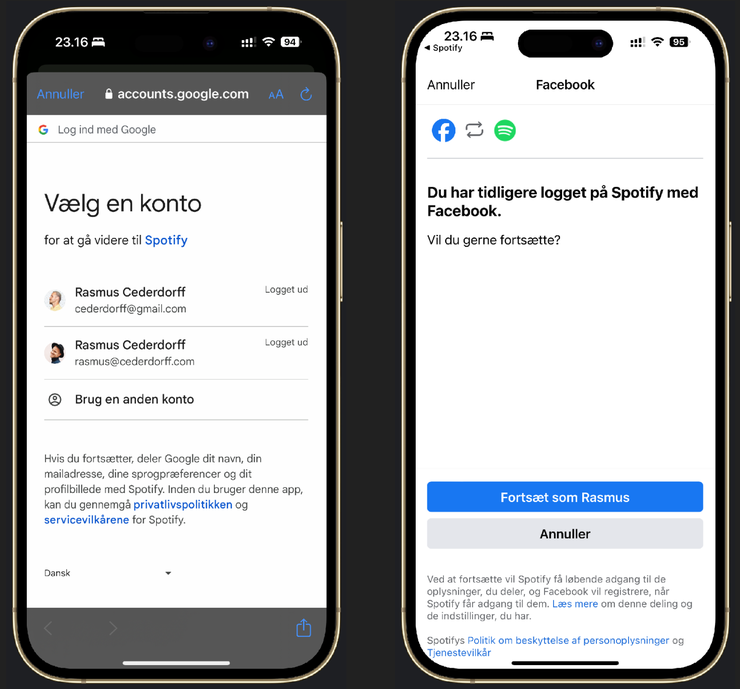
https://auth0.com/intro-to-iam/what-is-oauth-2

What is OAuth 2.0?

... is used to implement authentication with federated services.

What is OAuth 2.0?

- OAuth 2.0 stands for "Open Authorization".
- It is a standard that allows websites or applications to access resources on other web apps on behalf of a user (Access Delegation).
- Replaced OAuth 1.0 in 2012 and is now widely adopted as the industry standard for online authorization.
- Enables consensual access, allowing client apps to perform actions on resources on behalf of the user without sharing the user's credentials.
- Although primarily used on the web, OAuth 2.0 also applies to other client types like browser-based applications, server-side web applications, native/mobile apps, and connected devices.



https://auth0.com/intro-to-iam/what-is-oauth-2

The Pricinples of OAuth 2.0

- OAuth 2.0 is an authorization protocol, **not** an authentication protocol.
- Primarily designed for granting access to resources, such as remote APIs or user data.
- Utilizes Access Tokens to represent authorization to access resources on behalf of the end-user.
- The format of Access Tokens is not specifically defined in OAuth 2.0; however, JSON Web Token (JWT) format is commonly used for including data within the tokens.
- Access Tokens often have an expiration date for enhanced security.

https://auth0.com/intro-to-iam/what-is-oauth-2

Authorization Vs Authentication

Authentication is about verifying identity, while authorization is about granting access to resources based on that verified identity.

- **Authentication** verifies the identity of a user, device, or entity. It confirms that the user is who they claim to be. This process typically involves credentials like usernames, passwords, biometric data, or other identity proofs.
- **Authorization** determines the rights or permissions a user, device, or entity has after being authenticated. It controls access to resources by ensuring that the authenticated user has the appropriate permissions to perform certain actions or access certain data.

https://auth0.com/intro-to-iam/what-is-oauth-2

How Does OAuth 2.0 Work?

“At the most basic level, before OAuth 2.0 can be used, the Client must acquire its own credentials, a _client id _ and client secret, from the Authorization Server in order to identify and authenticate itself when requesting an Access Token.

Using OAuth 2.0, access requests are initiated by the Client, e.g., a mobile app, website, smart TV app, desktop application, etc. The token request, exchange, and response follow this general flow:

1. The Client requests authorization (authorization request) from the Authorization server, supplying the client id and secret to as identification; it also provides the scopes and an endpoint URI (redirect URI) to send the Access Token or the Authorization Code to.
2. The Authorization server authenticates the Client and verifies that the requested scopes are permitted.
3. The Resource owner interacts with the Authorization server to grant access.
4. The Authorization server redirects back to the Client with either an Authorization Code or Access Token, depending on the grant type, as it will be explained in the next section. A Refresh Token may also be returned.
5. With the Access Token, the Client requests access to the resource from the Resource server.”

https://auth0.com/intro-to-iam/what-is-oauth-2

OAuth 2

Client

Provider (Server)

Resource Owner

Authorization Server

Resource Server

Authorization Request

Authorization Grant

Authorization Grant

Access Token

Access Token

Protected Resource

My take: How Does OAuth 2.0 Work?

Before using OAuth 2.0, the Client obtains credentials (`client_id` and `client_secret`) from the Authorization Server to authenticate requests for an Access Token.

OAuth 2.0 access requests proceed as follows:

1. The **Client** sends an **Authorization Request** to the **Resource Owner** including `client_id`, `client_secret` and callback URL.
2. The **Resource Owner** approves the request and issues an **Authorization Grant** to the **Client**.
3. The **Client** exchanges the **Authorization Grant** for an **Access Token** with the **Authorization Server**.
4. The **Authorization Server** sends the **Access Token** to the **Client**.
5. The **Client** uses the **Access Token** to request a **Protected Resource** from the **Resource Server**.
6. The **Resource Server** validates the **Access Token** and, if it's valid, returns the **Protected Resource** to the **Client**.

https://auth0.com/intro-to-iam/what-is-oauth-2

OAuth 2 Roles & Essential Components

Client

The system that requires the access to protected resources. To access resources, the Client must hold an Access Token.

Resource Owner

The system or user that owns the resources and grant access to them.

Authorization Server

Processes the Access Token requests from the Client after authenticating and obtaining consent from the Resource Owner. It has two endpoints: Authorization for user authentication and consent, and Token for machine interactions.

Resource Server

Holds protected resources, validates Access Tokens from the Client, and provides resources.

https://auth0.com/intro-to-iam/what-is-oauth-2

My Take: OAuth 2 Roles & Essential Components

Client

The application that wants to access the user's data.

Resource Owner

The system or user who authorizes an application to access their account.

Authorization Server

The server issuing access tokens to the client after successfully authenticating the resource owner and obtaining authorization.

Resource Server

The server that holds the user's data and is capable of accepting and responding to protected resource requests using access tokens.

https://auth0.com/intro-to-iam/what-is-oauth-2

Auth Libraries

- **Remix Auth**: https://remix.run/resources/remix-auth
- **Remix Auth Form Strategy**: https://remix.run/resources/remix-auth-form-strategy
- **Remix Auth OAuth2 Strategy**: https://remix.run/resources/remix-auth-oauth2-strategy
  - Remix Auth GitHub Strategy: https://remix.run/resources/remix-auth-github-strategy
  - Remix Auth Microsoft Strategy: https://remix.run/resources/remix-auth-microsoft-strategy
  - Remix Auth Google Strategy: https://www.npmjs.com/package/remix-auth-google
  - And more ...

Authentication Flows

- The OAuth 2.0 strategy is defined and then used by the authenticator.
- Remix Auth handles the session and stores the session ID in a cookie.

auth.server.js

```
import { GitHubStrategy } from "remix-auth-github";
import { Authenticator } from "remix-auth";
import { sessionStorage } from "./session.server";

const gitHubStrategy = new GitHubStrategy(
  {
    clientID: process.env.CLIENT_ID,
    clientSecret: process.env.CLIENT_SECRET,
    callbackURL: process.env.CALLBACK_URL
  },
  async ({ accessToken, extraParams, profile }) => {
    console.log("accessToken:", accessToken);
    console.log("extraParams:", extraParams);
    console.log("profile:", profile);
    // Save/Get the user data from your DB or API using the token
    return profile;
  }
);

export const authenticator = new Authenticator(sessionStorage);
authenticator.use(gitHubStrategy);
```

Authentication Flows

- When the client is successfully authorized, the callback route will be “called” with the protected resource (user information).
- The callback route will then redirect to the protected resource.

auth.github.callback.jsx

```
export async function loader({ request }) {
  return authenticator.authenticate("github", request, {
    successRedirect: "/protected",
    failureRedirect: "/login"
  });
}
```

Authentication Flows

- Then we can use the authenticator to protect routes.
- And read the protected resource (user information).

Protected route

```
export async function loader({ request }) {
  const user = await authenticator.isAuthenticated(request, {
    failureRedirect: "/login"
  });

  return json({ user });
}
```

OAuth 2

In pairs

- With your own words, what is OAuth 2.0?
- Explain the flow using the diagram.
- Put it in perspective to the code examples:
  - How is OAuth 2.0 implemented?
  - And how do you see the OAuth 2.0 elements, roles and components in the implementation?
- Discuss whether we should implement OAuth 2.0:
  - Why? Why not?
  - What should we be aware of if we implement OAuth 2.0?

OAuth 2 Strategies

- **GitHub Strategy**
  - How to Implement SSO Authentication in Remix using GitHub and Remix Auth - In Under 10 minutes Links to an external site.
  - https://remix.run/resources/remix-auth-github-strategy
- **Google Strategy**
  - Follow the guide in the Readme: https://github.com/pbteja1998/remix-auth-google#readme
  - Get authorisation details: https://developers.google.com/identity/protocols/oauth2/web-server#creatingcred
- **Microsoft Strategy**
  - Follow the setup here: https://remix.run/resources/remix-auth-microsoft-strategy

OAuth 2
Implementation examples

- remix-auth-github: https://github.com/cederdorff/remix-auth-github
- remix-auth-google: https://github.com/cederdorff/remix-auth-google
- remix-auth-microsoft: https://github.com/cederdorff/remix-auth-microsoft