

# Remix

## Auth

localhost:3000/signin

# Sign In

Mail

Password

**Sign In**

No account? [Sign up here.](#)

localhost:3000/signup

# Sign Up

Mail

Password

**Sign Up**

Already have an account? [Sign in here.](#)

localhost:3000/posts

## Posts

**POSTS**    **ADD POST**    **PROFILE**

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

localhost:3000/posts/65d1f502f5b3f142ef417bb8

Anne Kirketerp  
Head of Department

localhost:3000/profile

## Profile

Name: Rasmus Cederdorff  
Title: Senior Lecturer  
Mail: race@eaaa.dk

**Logout**

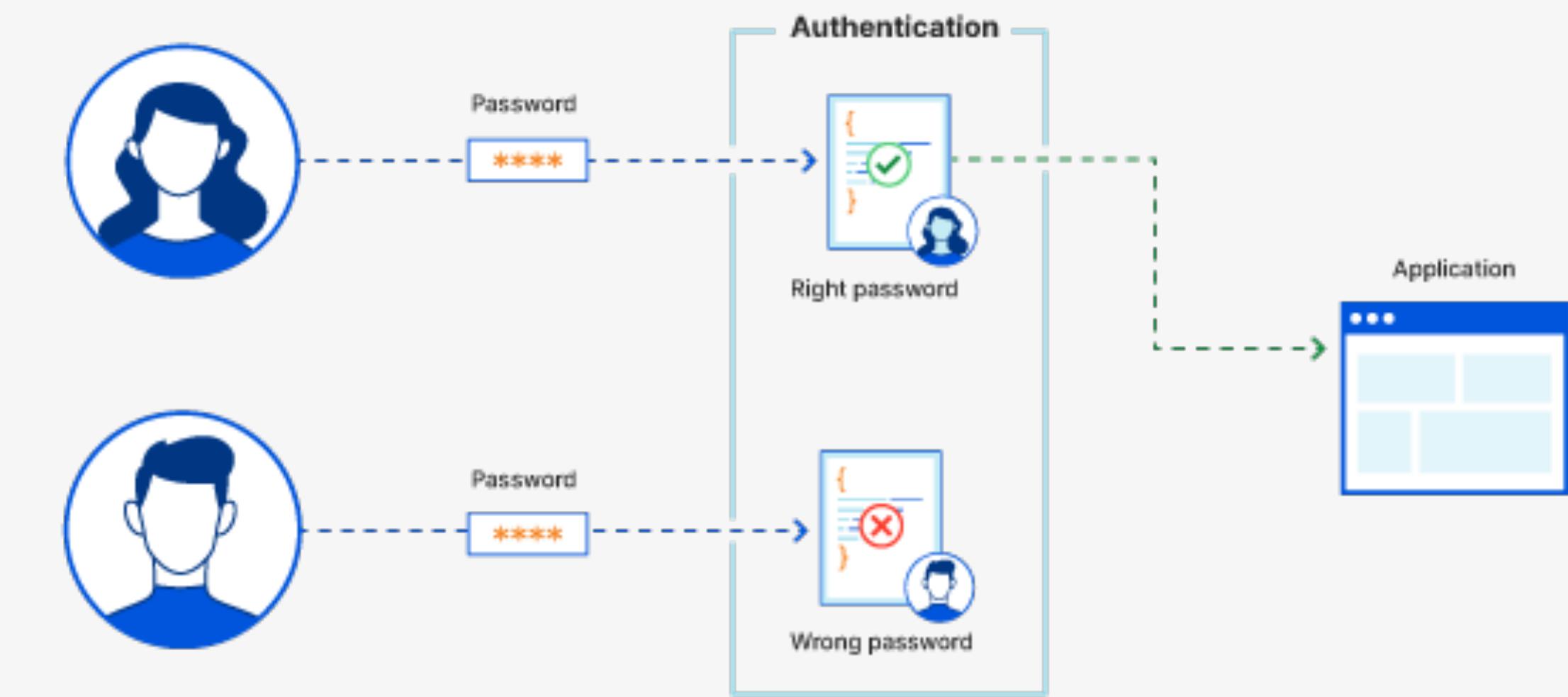
# Remix

## Auth

- Remix Auth is a library for implementing authentication in Remix 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

# 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.



# Protected Routes

User navigates  
to protected page



Is there a userId  
in the cookie session?

Yes

Renders  
page

No

Redirects to  
login page

# Login Route

User navigates  
to login page



Is there a userId  
in the cookie session?

Yes

Redirects to  
home page

No

Renders page



# 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

- 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.

Auth



The screenshot shows a browser window with two tabs open. The active tab is titled "localhost:3000/signin" and displays a sign-in form. The form has fields for "Mail" and "Password", both with placeholder text "Type your mail..." and "Type your password...". Below the fields is a "Sign In" button. At the bottom of the form, there is a link "No account? Sign up here.". The background of the main window is a light teal color with the word "Sign In" prominently displayed.

The second tab, titled "Remix Post App", is visible at the bottom and shows a navigation bar with "POSTS", "ADD POST", and "PROFILE" buttons. The "POSTS" button is highlighted.

On the right side of the screen, the developer tools are open, specifically the "Application" panel. This panel lists various storage components and their contents. Under "Manifest", there is one entry: "Name" \_ga\_DQJN..., "Value" GS1.1.1708240594.8.... Under "Service workers", there is one entry: "Name" \_gcl\_aw, "Value" GCL.1706861993.Cj.... Under "Storage", there are several entries under "LocalStorage": "Name" \_gcl\_au, "Value" 1.1.1319720516.170..., "Name" prism\_612..., "Value" f4f5597c-441e-43d7-..., "Name" \_ga\_KEPX..., "Value" GS1.1.1707398437.5..., "Name" \_ga, "Value" GA1.1.1385780176.1..., and "Name" \_omappvp, "Value" kTtW3LLswhMGoN2.... There is also an entry under "SessionStorage": "Name" \_ga, "Value" GA1.1.1882078899.1.... Under "IndexedDB", there is one entry: "Name" \_ga, "Value" 8a5b1d327ccd00064.... Under "Web SQL", there is one entry: "Name" \_ga, "Value" 8a5b1d327ccd00064.... Under "Cookies", there are several entries: "Name" ugld, "Value" 8a5b1d327ccd00064..., "Name" \_ga\_21SL..., "Value" GS1.1.1708011698.8..., "Name" \_fbp, "Value" fb.1.1704958669966..., "Name" CookieCon..., "Value" {stamp:%27KltgomT..., and "Name" \_sp\_id.0295, "Value" b8a98d47-3ad5-4aa.... Under "Private state tokens", "Interest groups", "Shared storage", and "Cache storage", there are no visible entries.

The screenshot shows a web application titled "Remix Post App" running at `localhost:3000/posts`. The main content area displays two posts:

- Rasmus Cederdorff** (Senior Lecturer) - A photo of a tall wooden clock tower.
- Dan Okkels Brendstrup** (Lecturer) - A photo of a cup of coffee on a saucer next to a plant, resting on a book titled "THE KINFOLK".

The developer tools are open, specifically the Application tab, which shows the following cookies:

Name	Value	D...	P...	E...	S...	H...	S...	S...	P...	P...
_ga_DQJN...	GS1.1.1708240594.8...	....	/	2...	53					M...
_gcl_aw	GCL.1706861993.Cj...	....	/	2...	1...					M...
_gcl_au	1.1.1319720516.170...	....	/	2...	32					M...
_ga	GA1.1.1385780176.1...	....	/	2...	30					M...
_omappvp	kTtW3LLswhMGoN2...	w...	/	2...	1...	✓	L...			M...
_ga	GA1.1.1882078899.1...	....	/	2...	30					M...
ugid	8a5b1d327ccd00064...	....	/	2...	43	✓	N...			M...
_ga_21SL...	GS1.1.1708011698.8...	....	/	2...	51					M...
_fbp	fb.1.1704958669966...	....	/	2...	31					M...
_ga_KEPX...	GS1.1.1707398437.5...	....	/	2...	52					M...
prism_612...	f4f5597c-441e-43d7...	....	/	2...	51					M...
CookieCon...	{stamp:%27KltgomT...	w...	/	2...	2...	✓				M...
_sp_id.0295	b8a98d47-3ad5-4aa...	....	/	2...	2...	✓	N...			M...
_session	eyJ1c2Vyljp7I9pZCI...	I...	/	S...	5...	✓	L...			M...

At the bottom of the Application tab, there is a large block of encoded cookie values:

```
eyJ1c2Vyljp7I9pZCI6ljY1Y2RINGNiMGQwOWNiNjE1YTlzzGlxNyIsImItYWdljoiaHR0cHM6Ly9zaGFyZS5jZWRIcmRvcMzmlmRrl2ItYWdlcy9yYWNILndlYnAiLCJtYWlsIjoicmFjZUBIWFhLmRrlwibmFtZSI6IlJhc211cyBDZWRIcmRvcMzmliwidGl0bGUIoijTZW5pb3IgTGVjdHVyZXliLCJIZHVjYXRpb25zlpblk11bHRpbWVkaWEgRGVzaWduliwiV2ViERldmVsb3BtZW50liwiRGIInaXRhbCBDb25jZXB0IERldmVsb3BtZW50Ii0sImNyZWFOZWRBdCI6ljlwMjQtMDItMThUMTI6MTY6MDEuNzY1Wilsl9fdil6MH0slnN0cmF0ZWd5ljoidXNlc1wYXNzln0%3D.IR0DPatsFVZp5L8%2Fd1gvi9P8fLe9XcdptRsZZRfOQw
```

# But Why?

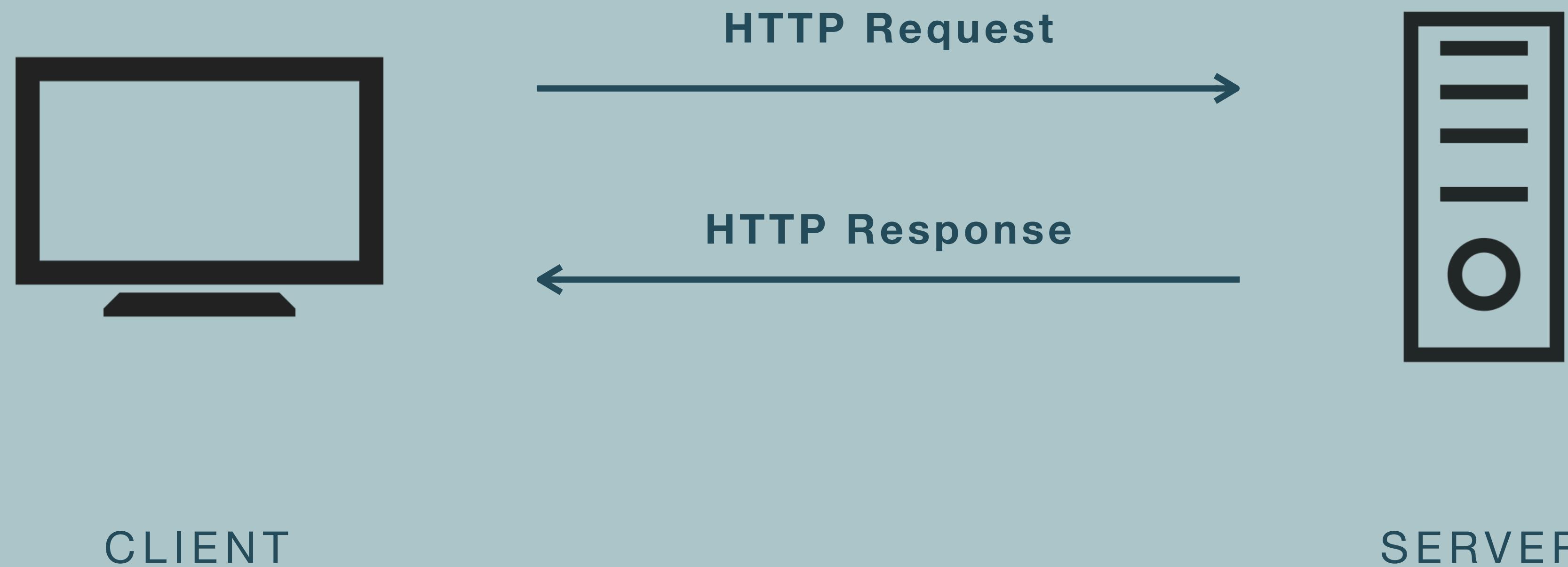
## HTTP is stateless

- 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.

**Remix**  
Auth

# Client-Server Model

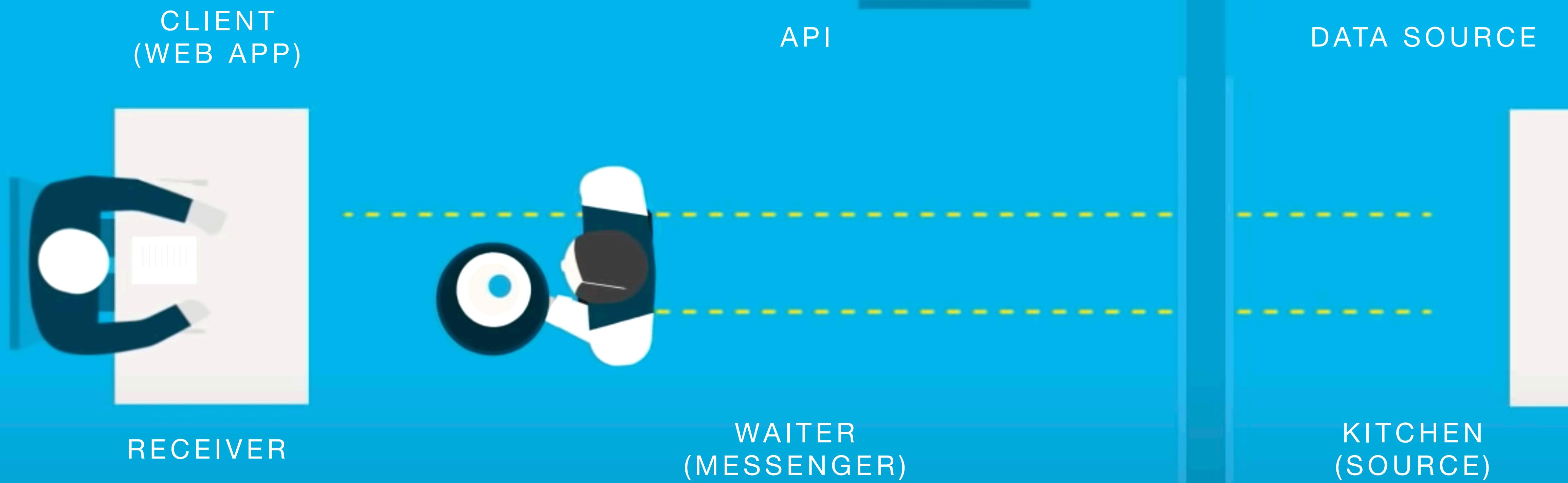
Communication between web **clients** and web **servers**.



# Client-Server Model

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





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

# 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.

Remix

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.

Remix

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.

**Remix**  
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

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

// 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 work across routes
    httpOnly: true, // for security reasons, make this cookie only accessible via HTTP
    secrets: ["s3cr3t"], // replace this with an actual secret
    secure: process.env.NODE_ENV === "production" // enables https only mode
  }
});

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

# 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.

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

// 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 work across routes
    httpOnly: true, // for security reasons, make this cookie only accessible via HTTP
    secrets: ["s3cr3t"], // replace this with an actual secret
    secure: process.env.NODE_ENV === "production" // enables https only mode
  }
});

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

# 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.

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

// 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 work across 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 if your app is served over https
  }
});

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

# 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.

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

// 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 work across routes
    httpOnly: true, // for security reasons, make this cookie only accessible via HTTP
    secrets: ["s3cr3t"], // replace this with an actual secret
    secure: process.env.NODE_ENV === "production" // enables the secure flag
  }
});

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

# 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.js

```
// Create an instance of the authenticator, pass a generic with
// strategies will return and will store in the session
export let authenticator = new Authenticator(sessionStorage, {
  sessionErrorKey: "sessionErrorKey" // keep in sync
});

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

route

export async function action({ request }) {
  // we call the method with the name of the strategy we want to
  // request object, optionally we pass an object with the URLs
  // to be redirected to after a success or a failure
  return await authenticator.authenticate("user-pass", request,
    successRedirect: "/posts",
    failureRedirect: "/signin"
  );
}
```

# Authentication Flows

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

auth.server.js

```
// Create an instance of the authenticator, pass a generic with
// strategies will return and will store in the session
export let authenticator = new Authenticator(sessionStorage, {
  sessionErrorKey: "sessionErrorKey" // keep in sync
});

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

route

```
export async function action({ request }) {
  await authenticator.logout(request, { redirectTo: "/signin" })
}
```

# Access Control

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

auth.server.js

```
// Create an instance of the authenticator, pass a generic with
// strategies will return and will store in the session
export let authenticator = new Authenticator(sessionStorage, {
  sessionErrorKey: "sessionErrorKey" // keep in sync
});

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

route

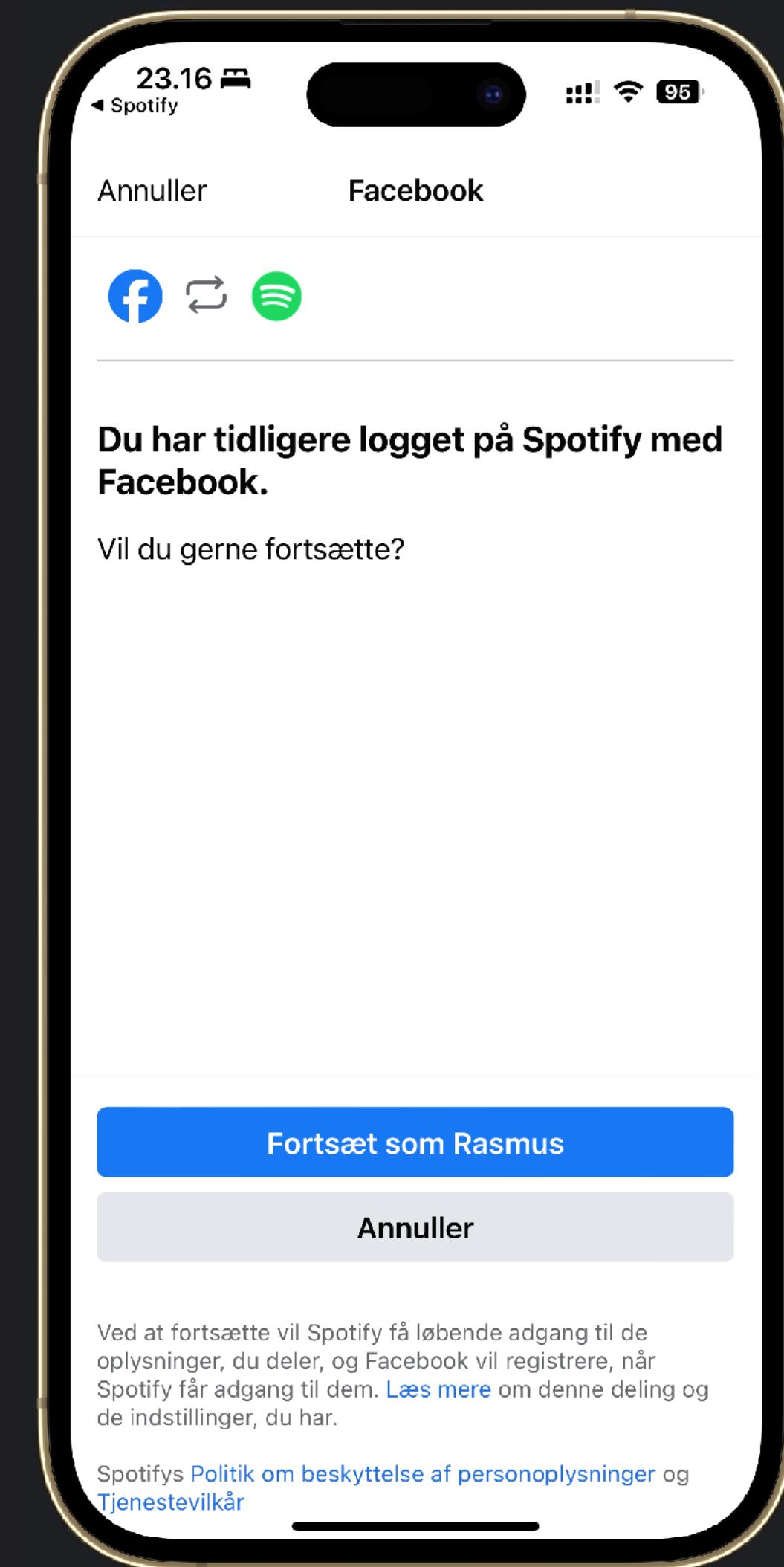
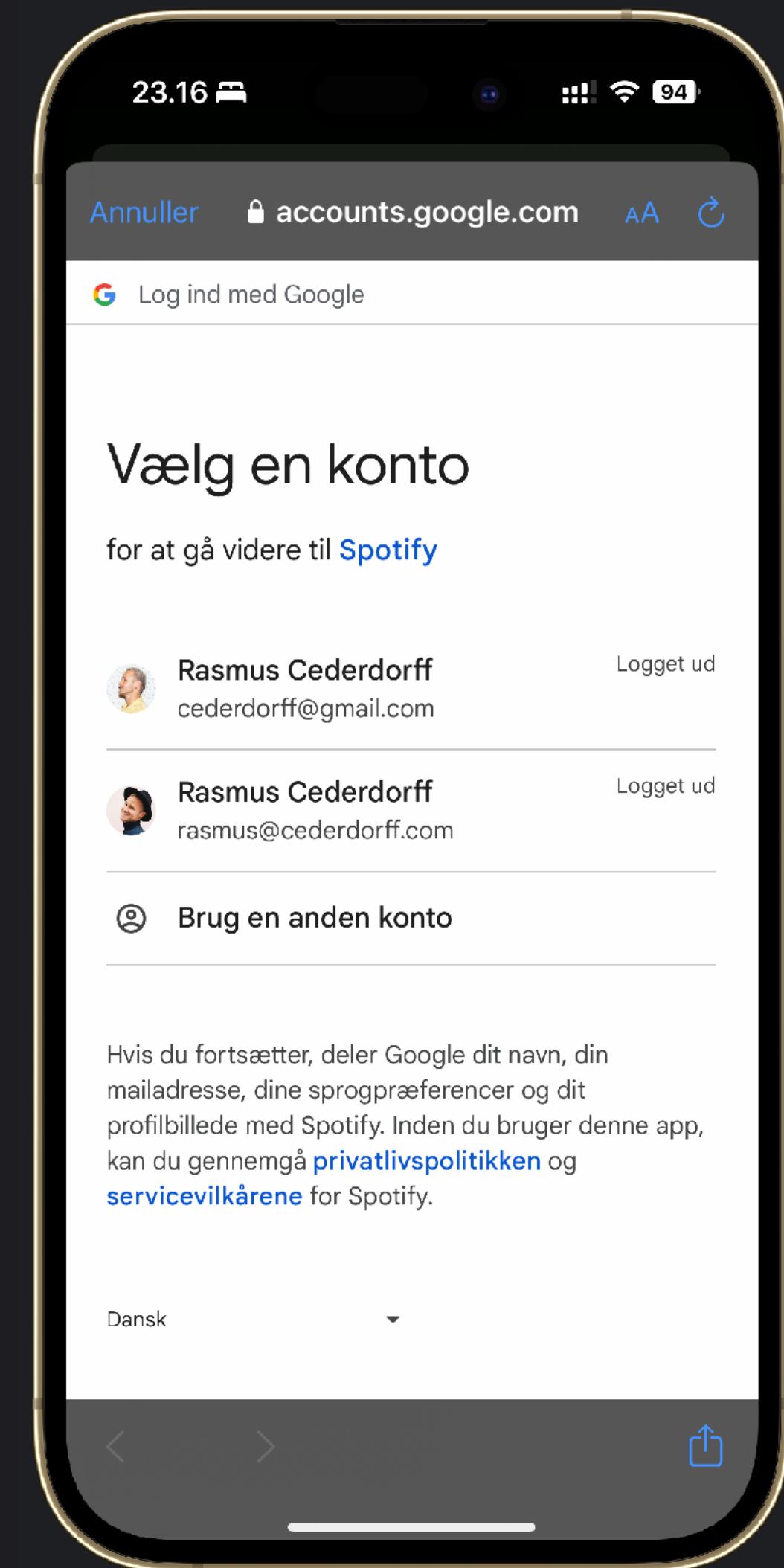
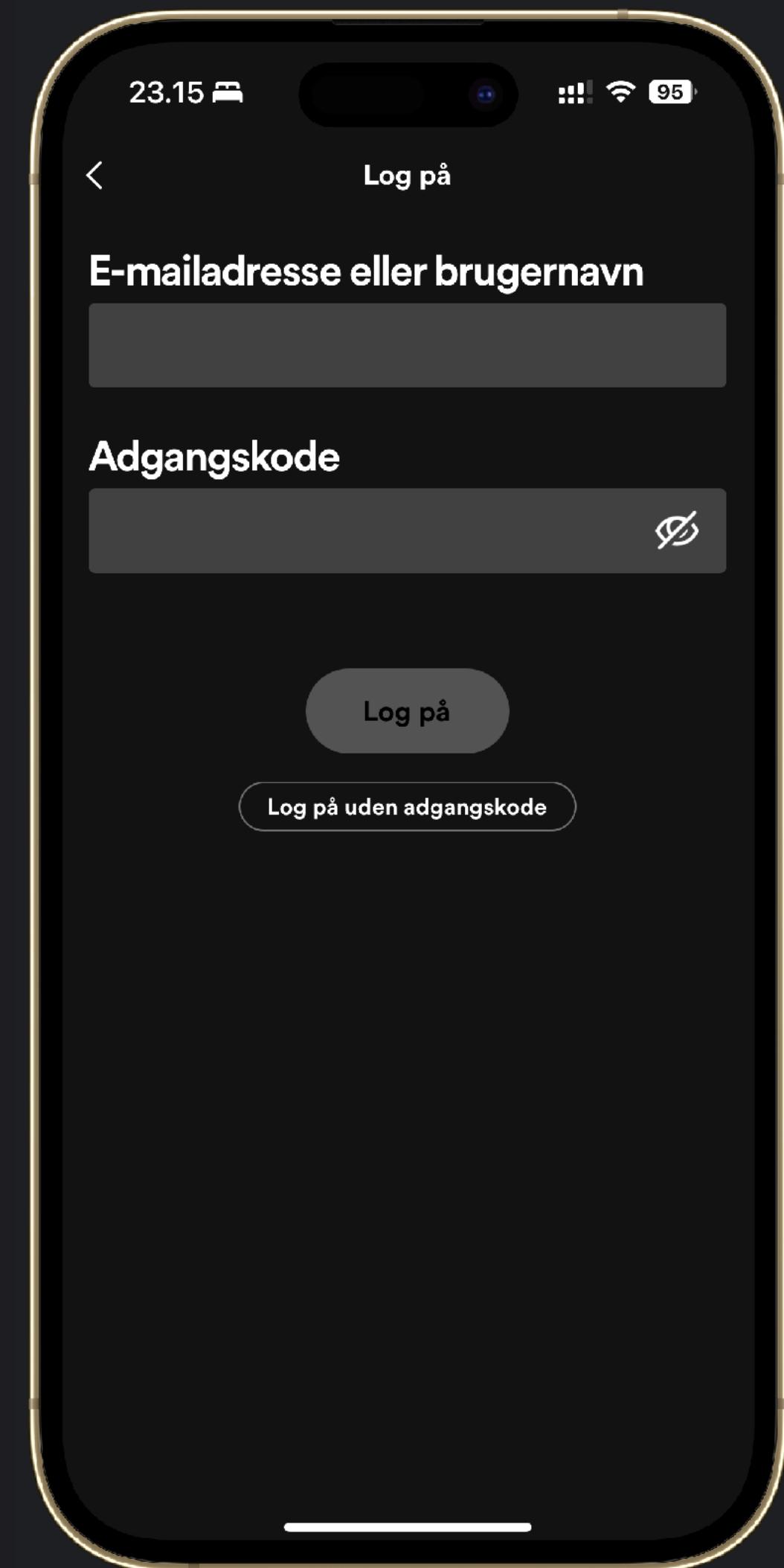
```
export async function loader({ request }) {
  return await authenticator.isAuthenticated(request, {
    failureRedirect: "/signin"
  });
}
```

# Remix

## Auth OAuth2 Strategy



# What is OAuth 2.0?

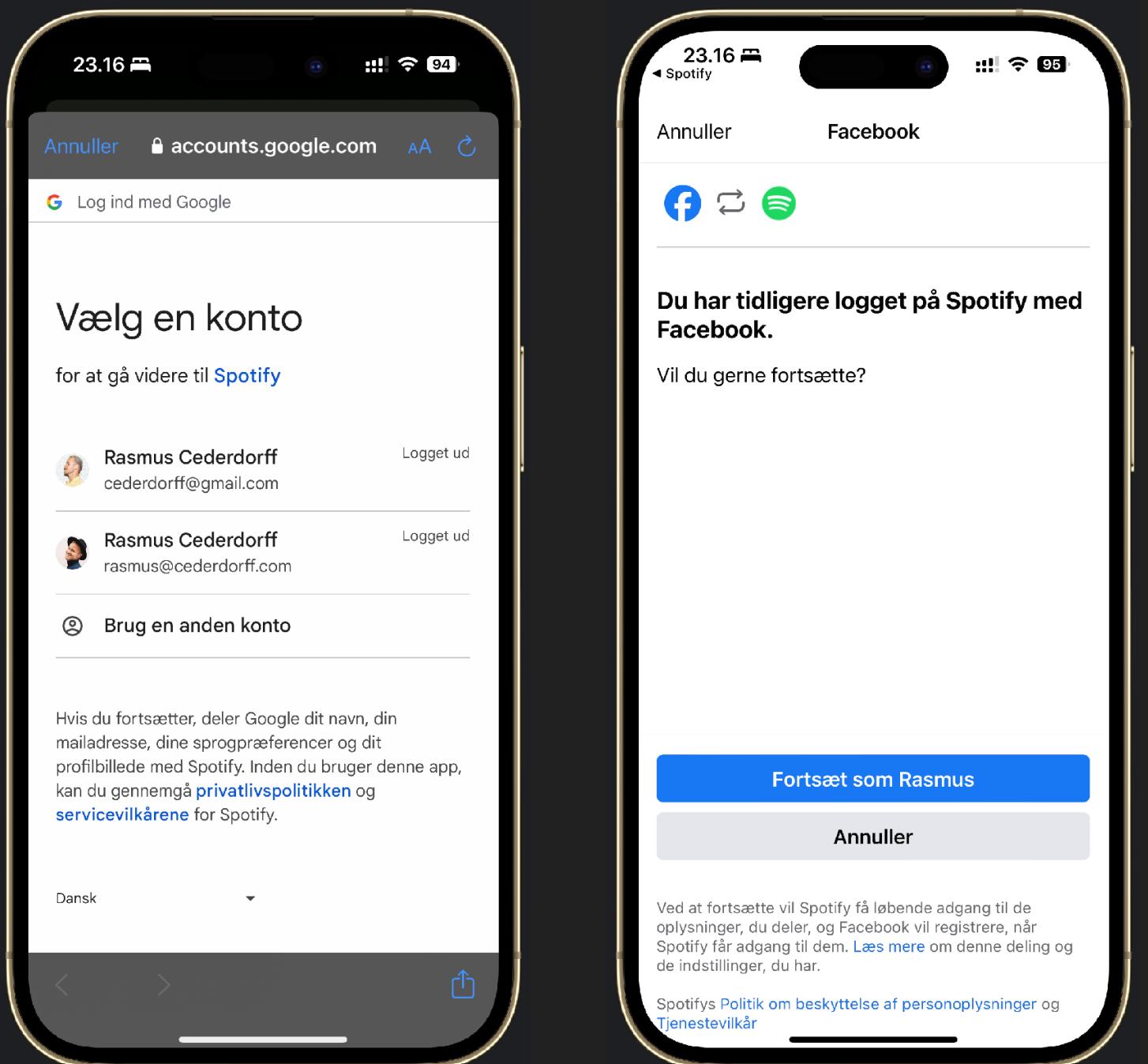


Form Strategy

OAuth 2.0 Strategy: Google & Facebook

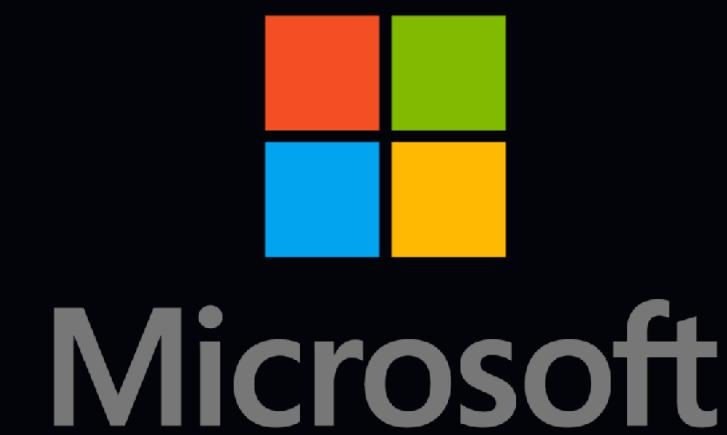
# What is OAuth 2.0?

*“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.”*



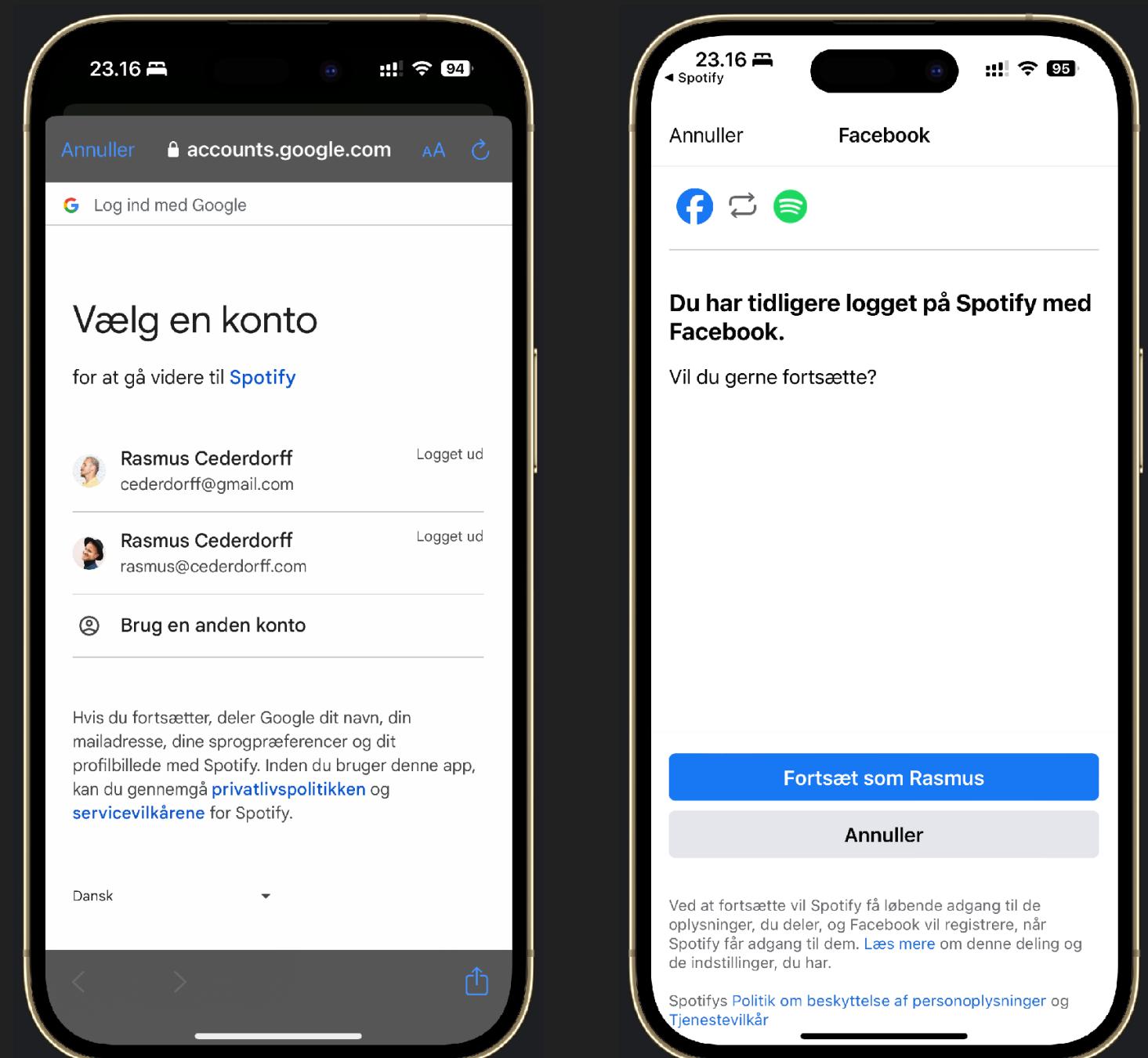
# What is OAuth 2.0?

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



- 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.

## What is OAuth 2.0?



# The Principles 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.



# 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.

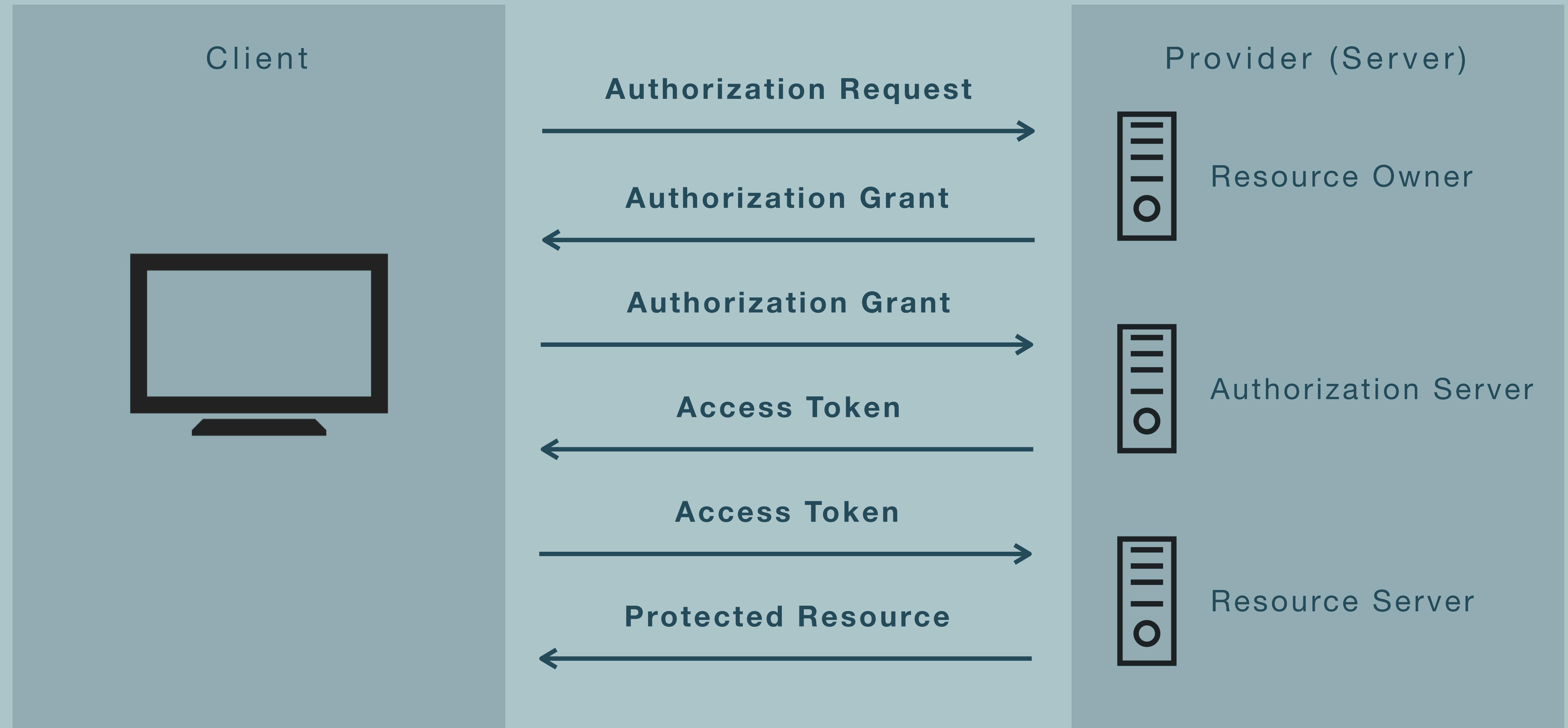
# How Does OAuth 2.0 Work?

“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.”

# OAuth 2



# 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**.

# 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.

# 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.



# 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 });
}
```

# Remix

## 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>