Auth

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

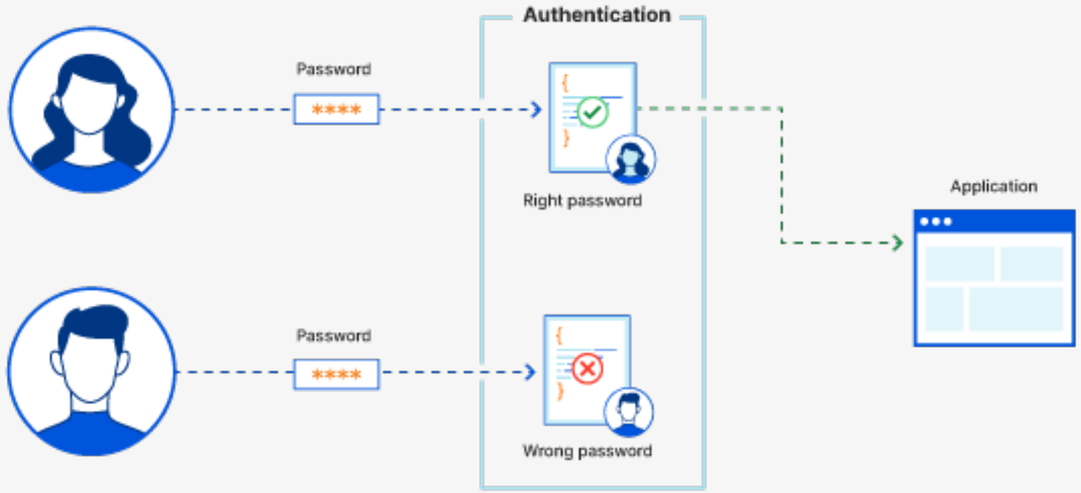
Logout

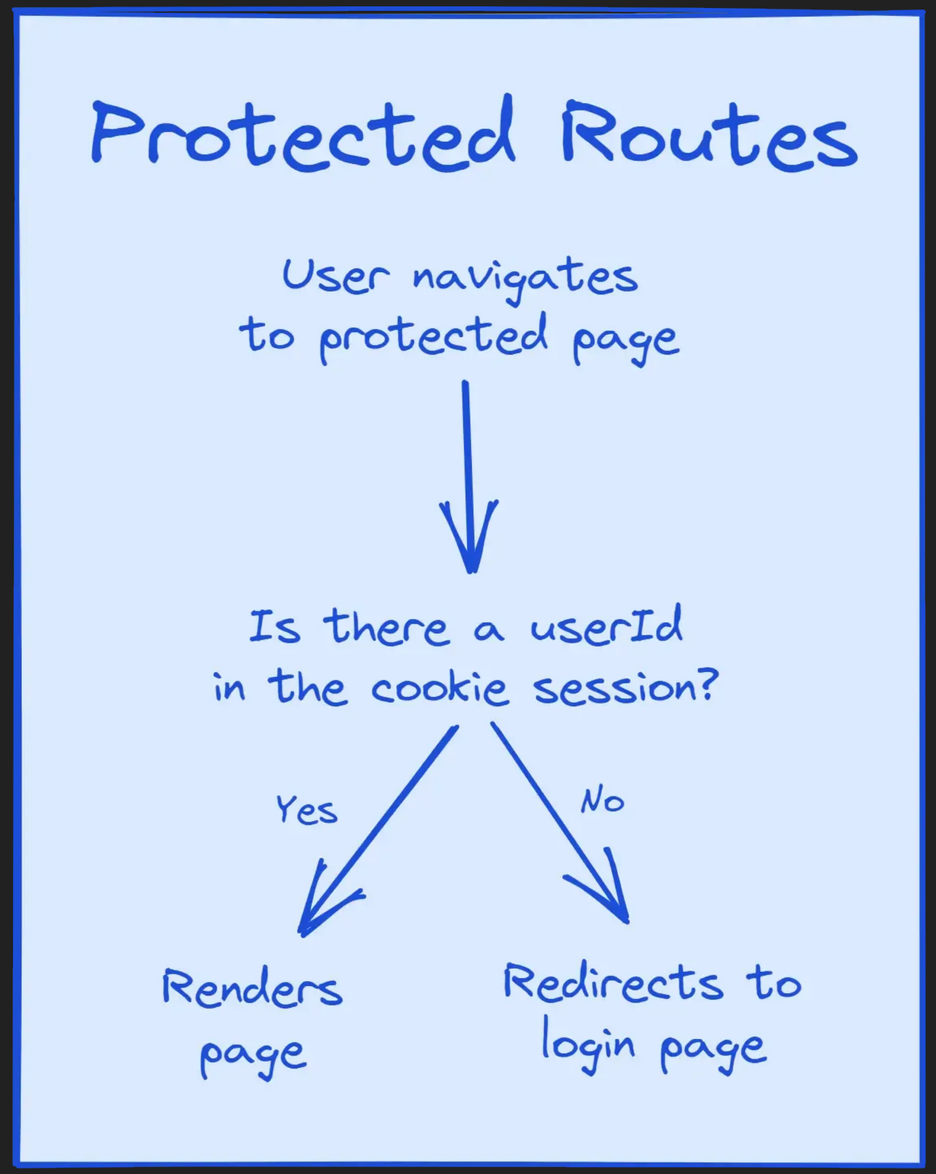
Auth

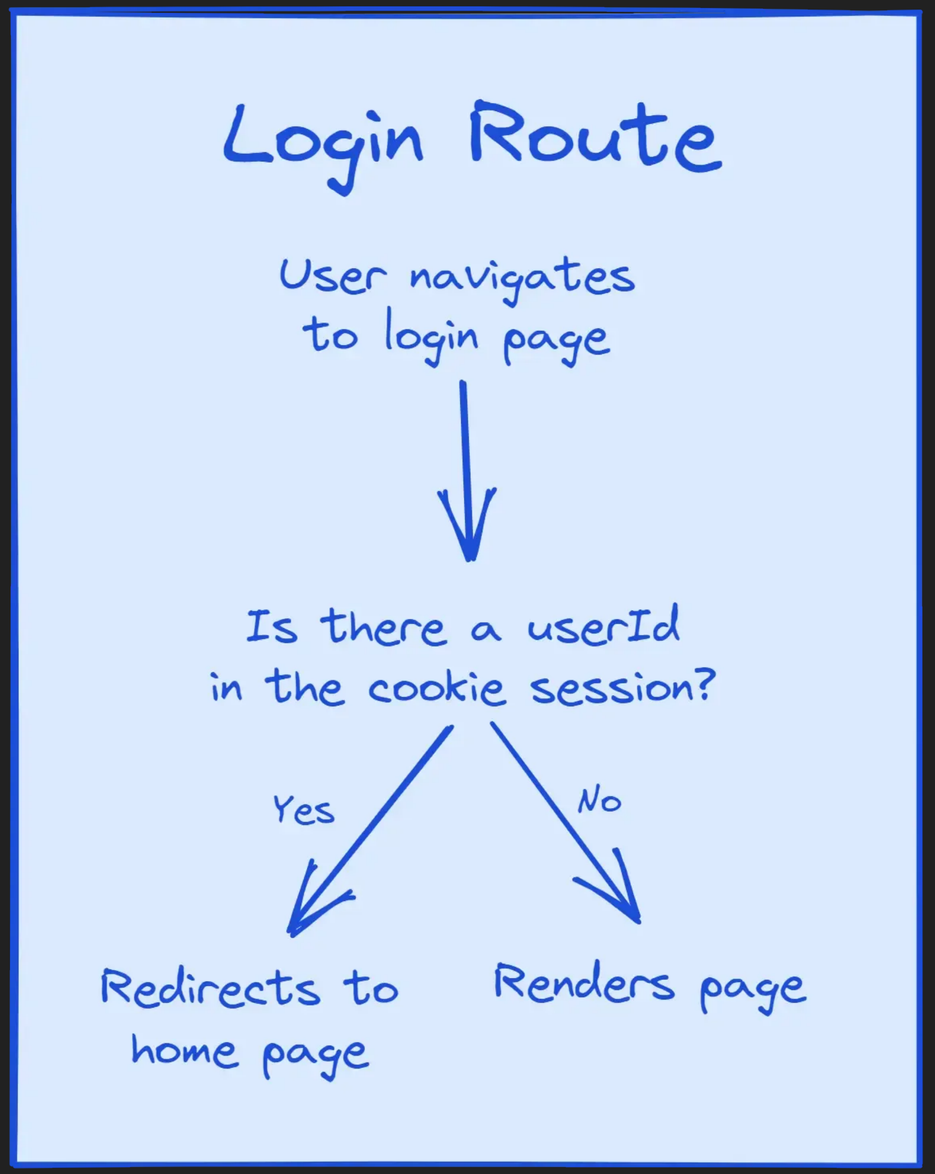
- Remix Auth is a library for implementing authentication in Remix Applications.
- Simplifies the process of adding authentication by providing a structured way to handle user authentication flows, including login, logout, and session management.
- Heavily inspired by Passport.js

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.







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), OAuth (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.

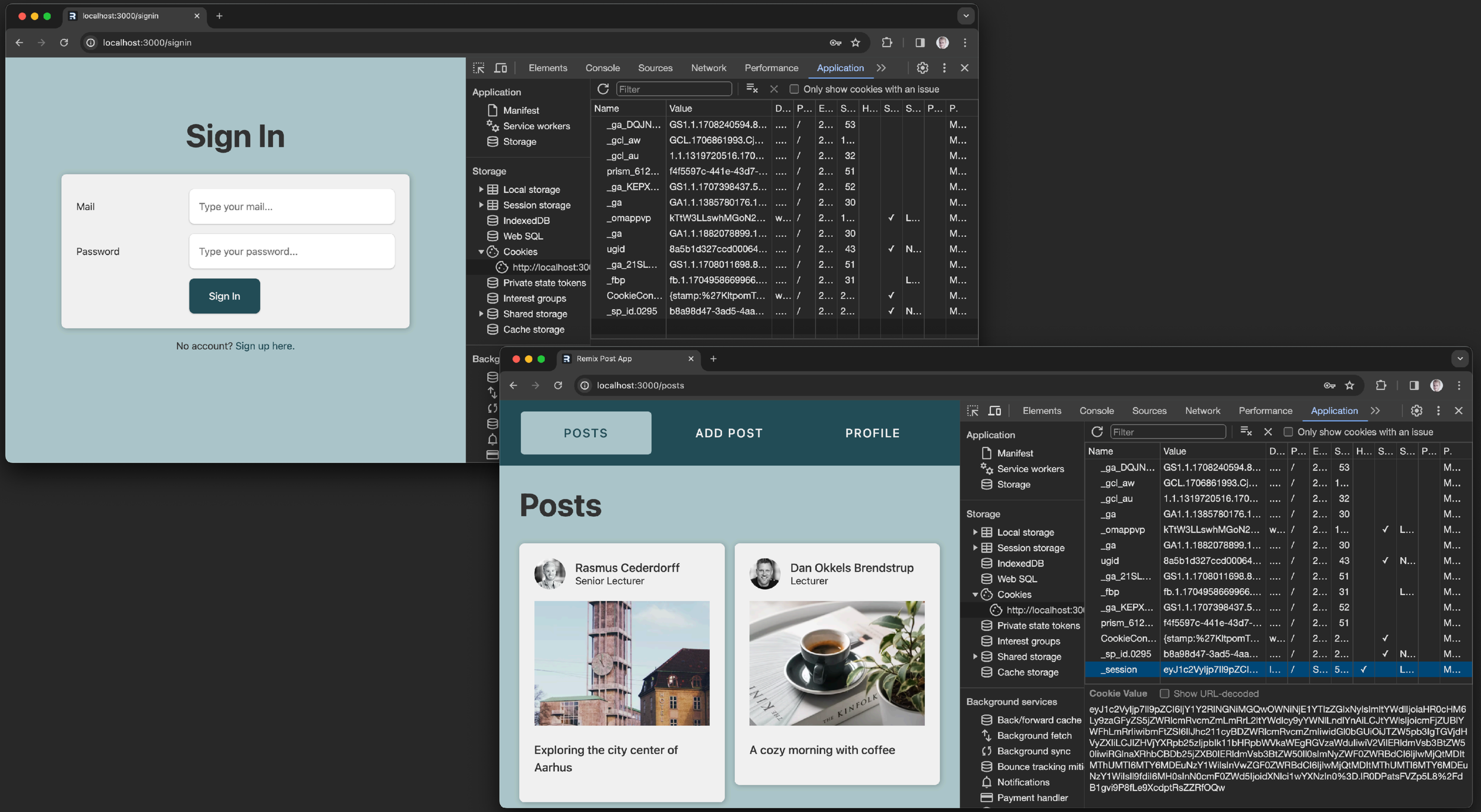
Cookies

- 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?

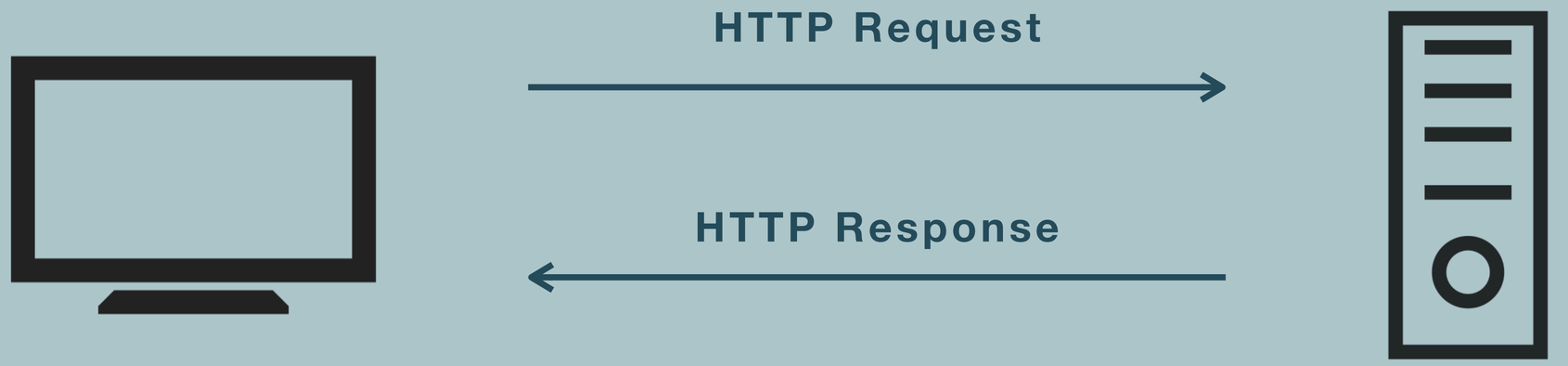
HTTP is stateless

- 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

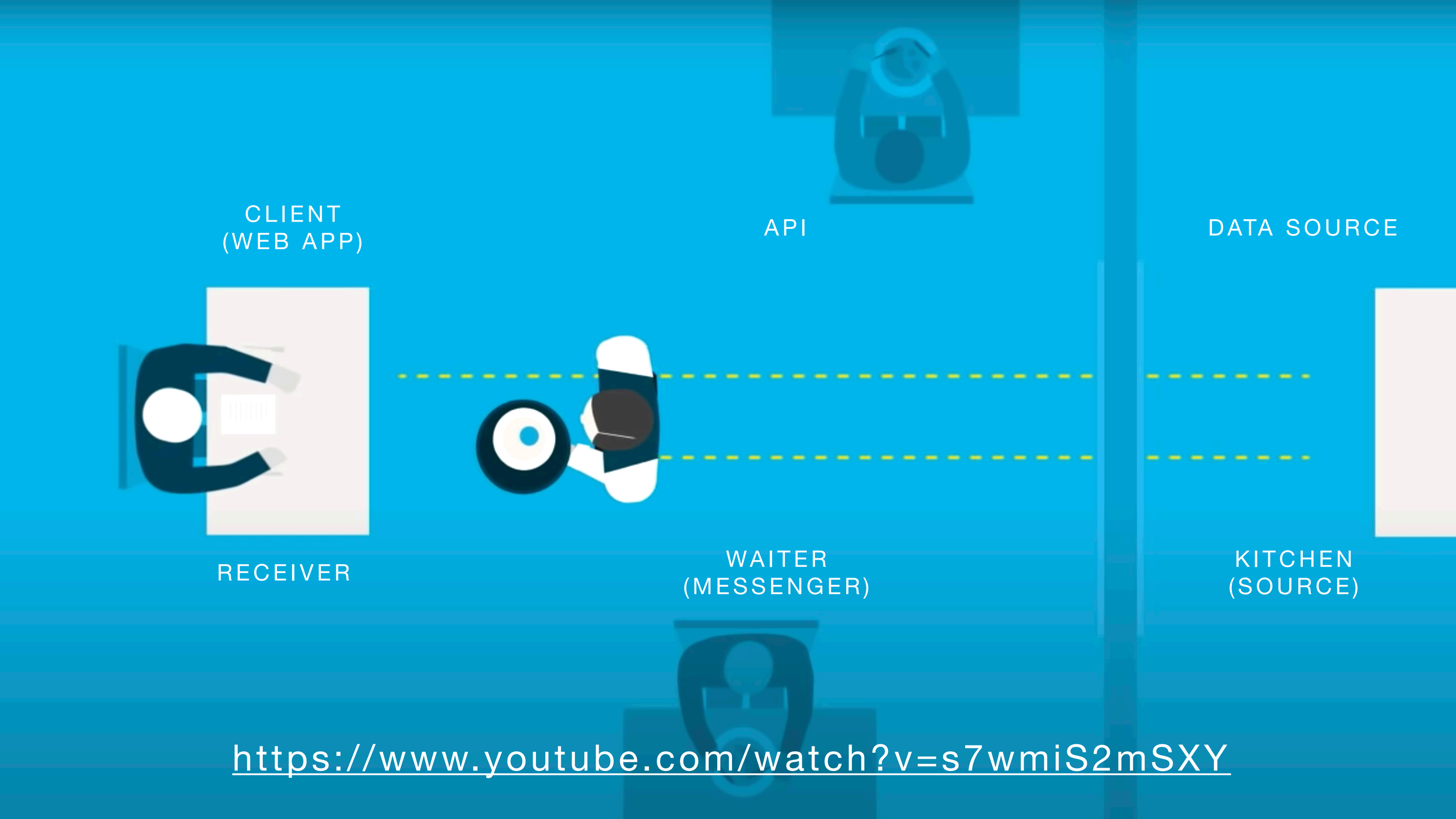
Client-Server Model

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



Client-Server Model

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



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

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

```
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 } =
```

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/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 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/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 } =
```

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/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 } =
```

Authentication Flows

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

```
// 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"
);
```

```
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.

```
// 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"
);
```

```
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.

```
// 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"
);
```

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