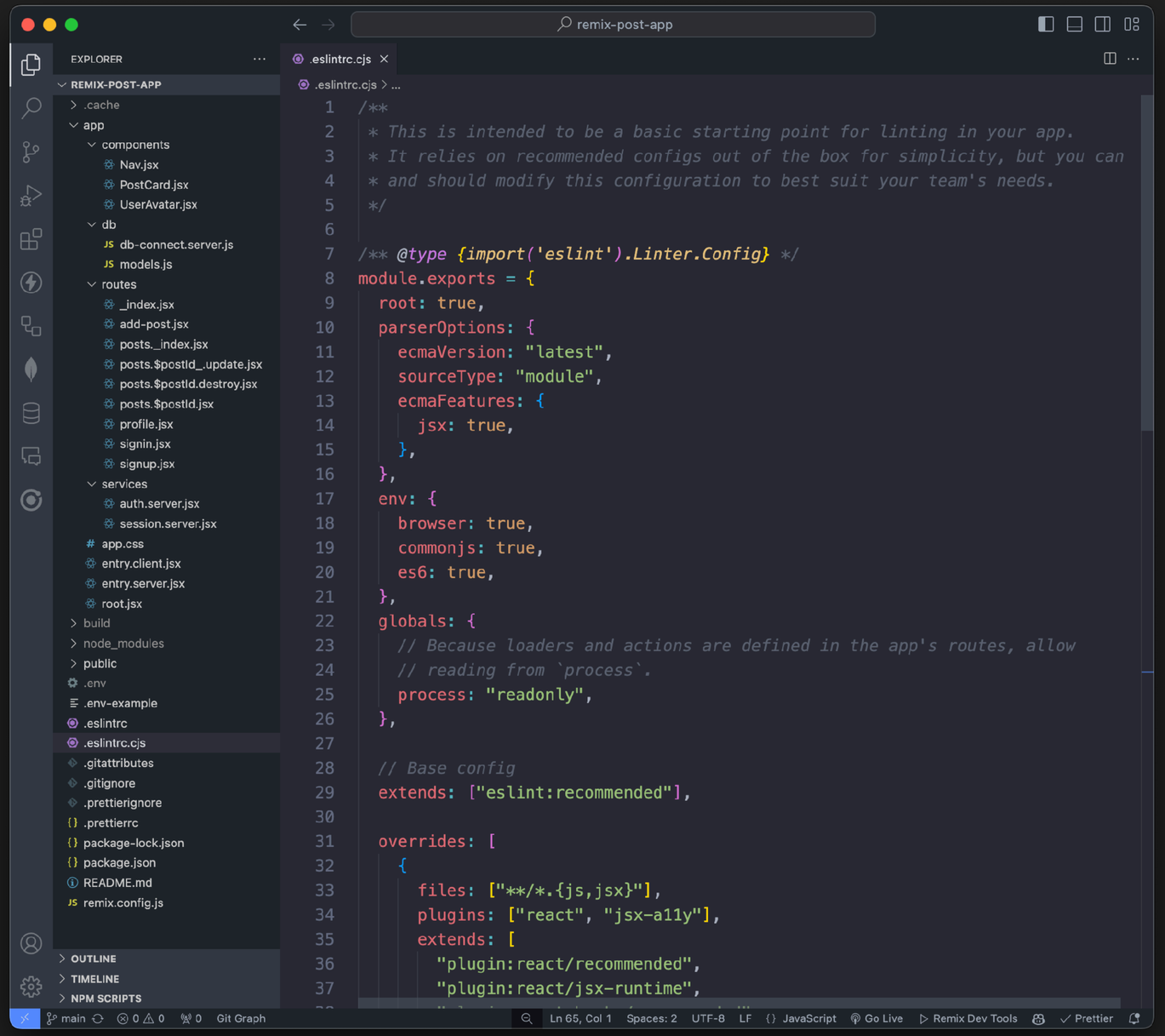
Recap - The RACE Remix Track

- ES Lint Config
- Remix Data Flow
- Remix Auth
  - Route Protection & Restriction
  - AuthorizationError
- Form validation & Error Messages
- Destructuring in Remix: loaders & actions
- Querying with Mongoose

ES Lint Config: `.eslintrc.cjs`

ES Lint Config

- Copy and paste into your `.eslintrc.cjs`: https://github.com/cederdorff/remix-post-app/blob/main/.eslintrc.cjs



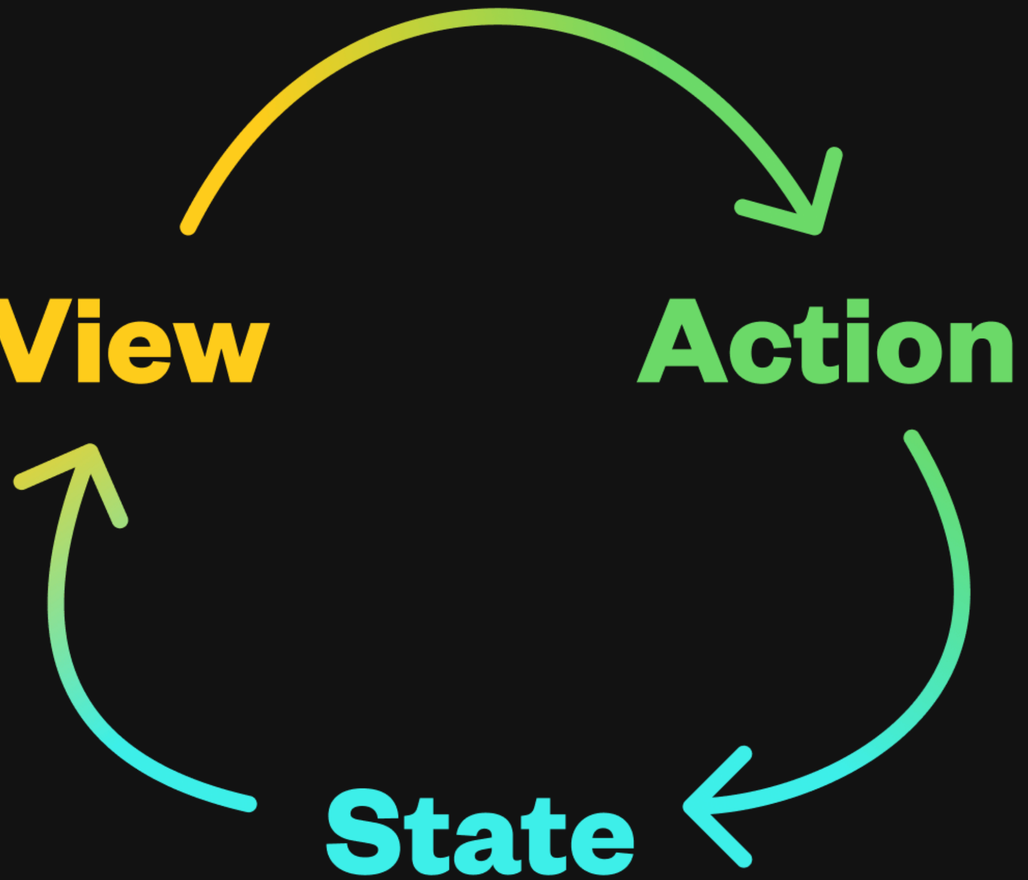
Remix

Data Flow

> ***Build Better Websites, Sometimes with Remix.***
>
> **If you get good at Remix, you will accidentally get good at web development in general.**
>
> **Get good at Remix, get good at the web.**

https://remix.run/docs/en/v1/pages/philosophy

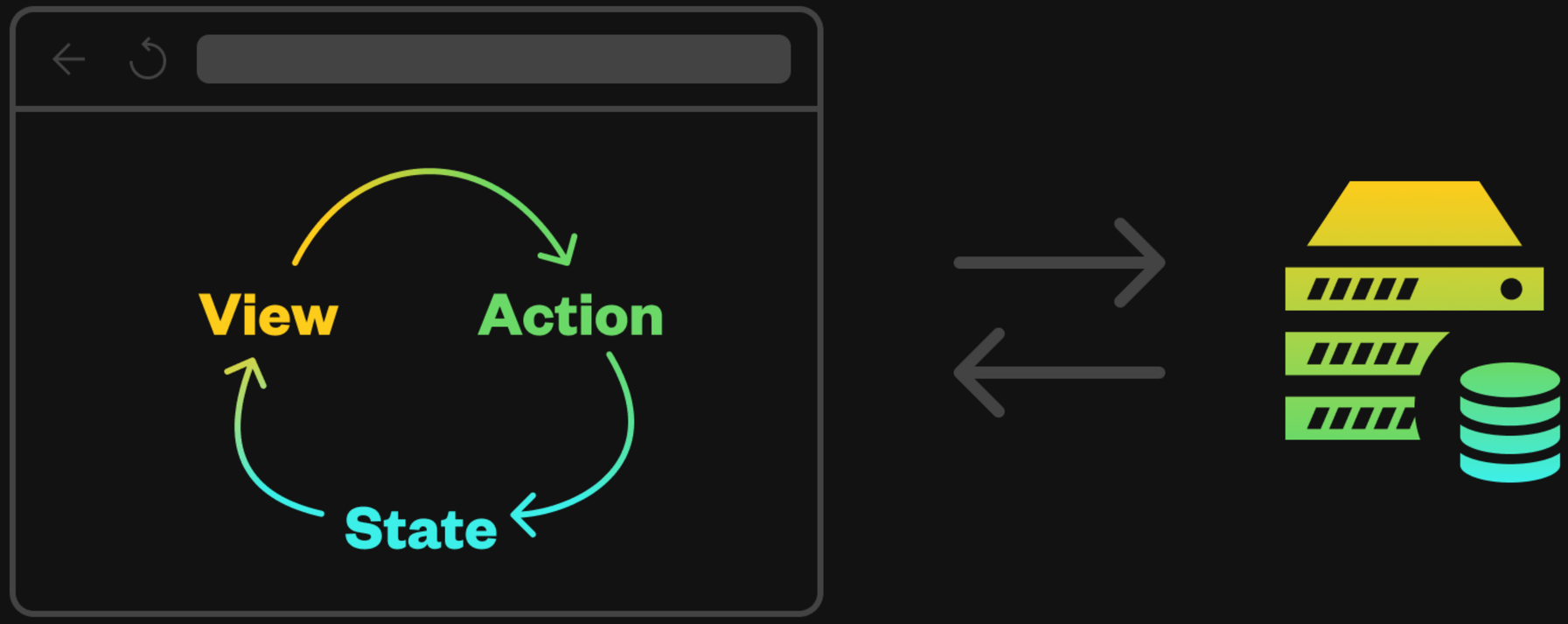
React: UI is a function of state



https://remix.run/blog/remix-data-flow

State management: Keep in sync

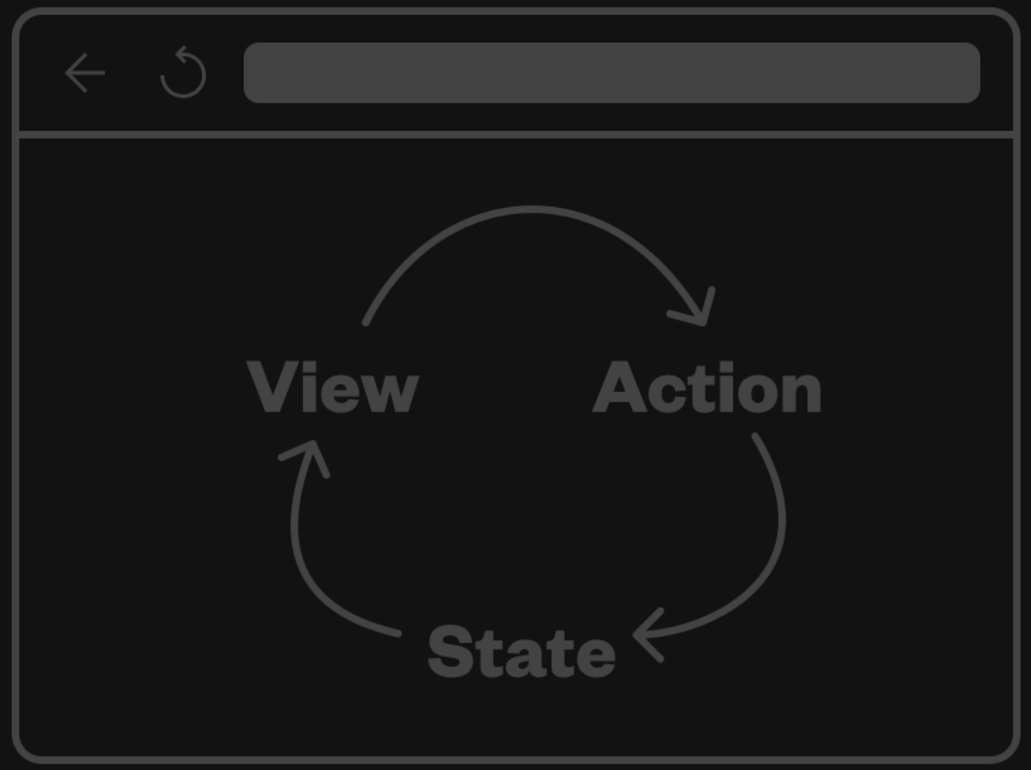
Across the network to the database.



https://remix.run/blog/remix-data-flow

State management: Keep in sync

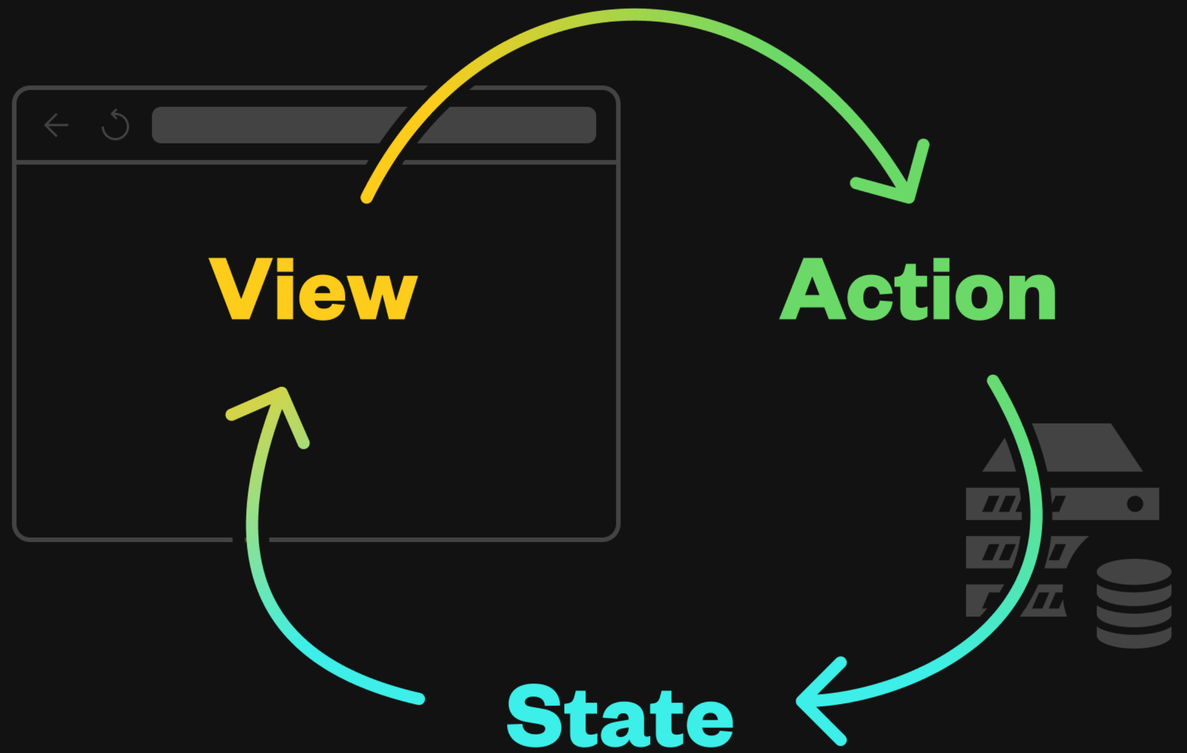
How do we keep the database in sync with UI?



https://remix.run/blog/remix-data-flow

Remix to the rescue

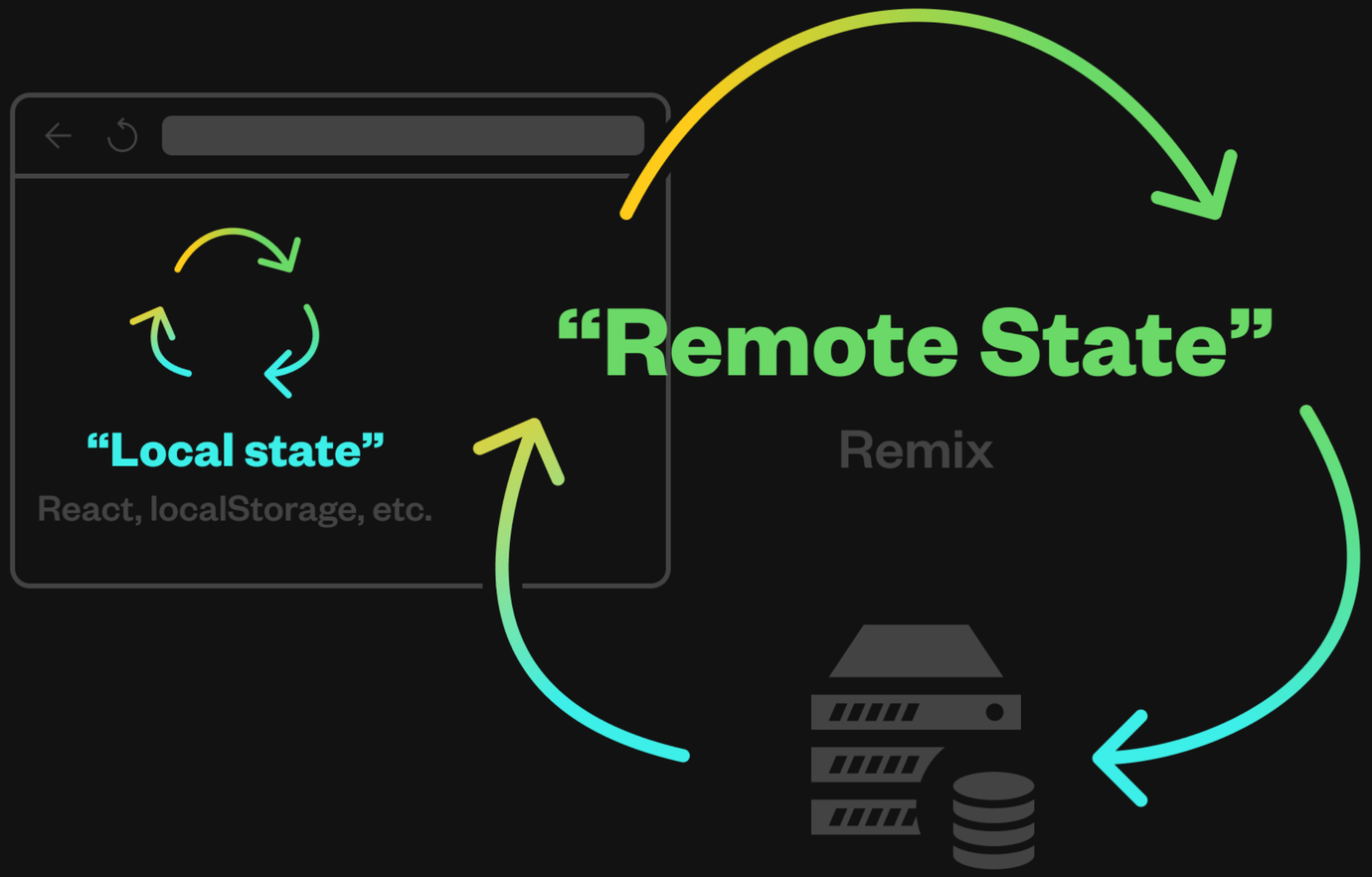
Remix simplifies the interactions with the server to get the data into the component.



https://remix.run/blog/remix-data-flow

Local state & Remote state

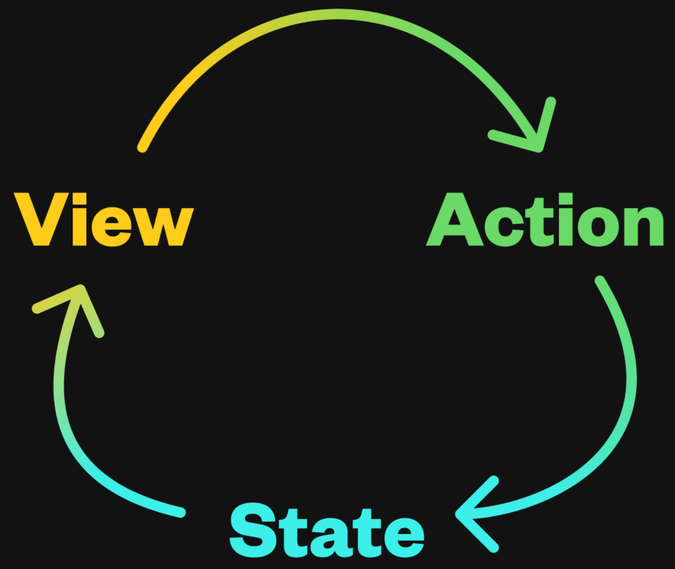
"UI is a function of your remote state and your local state."

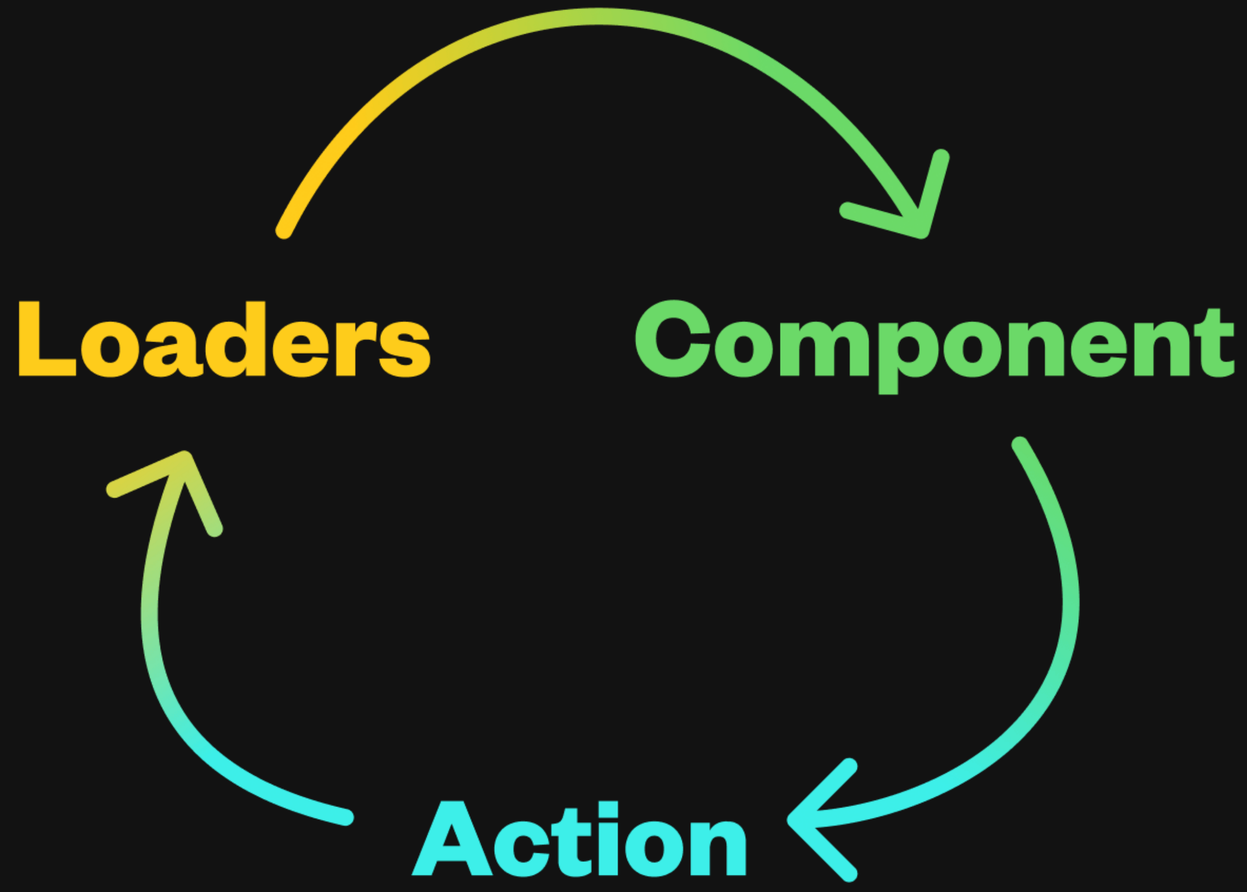


https://remix.run/blog/remix-data-flow

Loader -> Component -> Action

From View -> Action -> State





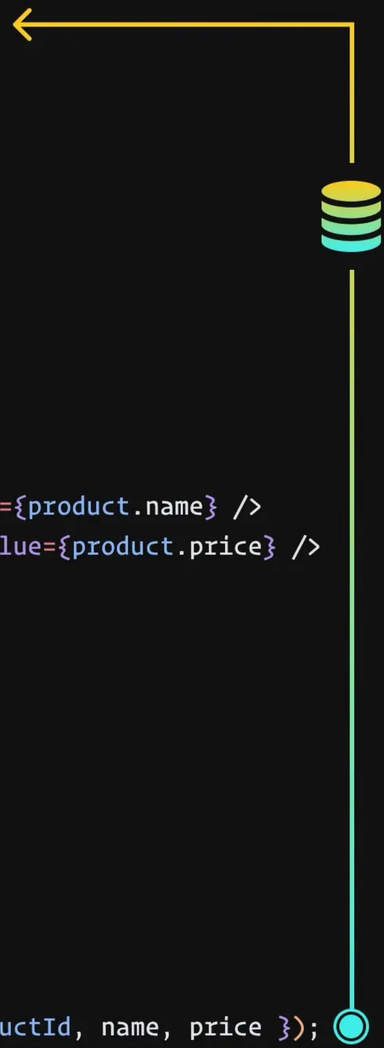
https://remix.run/blog/remix-data-flow

Loader

Component

Action

```
// File (route): /products/$productId/edit

export async function loader({ params }) {
  const { productId } = params;
  const product = await db.products.get(productId);
  return json(product)
};

export default function Route() {
  const product = useLoaderData();
  return (
    <>
      <h2>Edit Product #{product.id}</h2>
      <Form method="post">
        <input type="text" name="name" defaultValue={product.name} />
        <input type="number" name="price" defaultValue={product.price} />
        <button type="submit">Save</button>
      </Form>
    </>
  );
}

export async function action({ params, request }) {
  const { productId } = params;
  const form = await request.formData();
  const name = form.get("name");
  const price = form.get("price");
  const product = await db.products.edit({ id: productId, name, price });
  return redirect(`/products/${product.id}`);
}
```



https://remix.run/blog/remix-data-flow

loader

component

action

```
export async function loader({ params }) {
  const post = await mongoose.models.Post.findById(params.postId);
  return json({ post });
}

export default function UpdatePost() {
  const { post } = useLoaderData();

  return (
    <div className="page">
      <h1>Update Post</h1>
      <Form id="post-form" method="post">
        <input defaultValue={post.caption} name="caption" type="text" />
        <button>Save</button>
      </Form>
    </div>
  );
}

export async function action({ request, params }) {
  const formData = await request.formData();
  const post = Object.fromEntries(formData);

  await mongoose.models.Post.findByIdAndUpdate(params.postId, post);

  return redirect(`/posts/${params.postId}`);
}
```

loader

```
export async function loader({ params }) {
  const post = await mongoose.models.Post.findById(params.postId);
  return json({ post });
}
```

component

```
export default function UpdatePost() {
  const { post } = useLoaderData();

  return (
    <div className="page">
      <h1>Update Post</h1>
      <Form id="post-form" method="post">
        <input defaultValue={post.caption} name="caption" type="text" />
        <button>Save</button>
      </Form>
    </div>
  );
}
```

action

```
export async function action({ request, params }) {
  const formData = await request.formData();
  const post = Object.fromEntries(formData);

  await mongoose.models.Post.findByIdAndUpdate(params.postId, post);

  return redirect(`/posts/${params.postId}`);
}
```

loader

```
export async function loader({ params }) {
  const post = await mongoose.models.Post.findById(params.postId);
  return json({ post });
}
```

backend

component

```
export default function UpdatePost() {
  const { post } = useLoaderData();

  return (
    <div className="page">
      <h1>Update Post</h1>
      <Form id="post-form" method="post">
        <input defaultValue={post.caption} name="caption" type="text" />
        <button>Save</button>
      </Form>
    </div>
  );
}
```

frontend
(& backend)

action

```
export async function action({ request, params }) {
  const formData = await request.formData();
  const post = Object.fromEntries(formData);

  await mongoose.models.Post.findByIdAndUpdate(params.postId, post);

  return redirect(`/posts/${params.postId}`);
}
```

backend

loader

component

action

```tsx
export async function loader({ params }) {
  const post = await mongoose.models.Post.findById(params.postId);
  return json({ post });
}

export default function UpdatePost() {
  const { post } = useLoaderData();

  return (
    <div className="page">
      <h1>Update Post</h1>
      <Form id="post-form" method="post">
        <input defaultValue={post.caption} name="caption" type="text" />
        <button>Save</button>
      </Form>
    </div>
  );
}

export async function action({ request, params }) {
  const formData = await request.formData();
  const post = Object.fromEntries(formData);

  await mongoose.models.Post.findByIdAndUpdate(params.postId, post);

  return redirect(`/posts/${params.postId}`);
}
```

loader

component

action

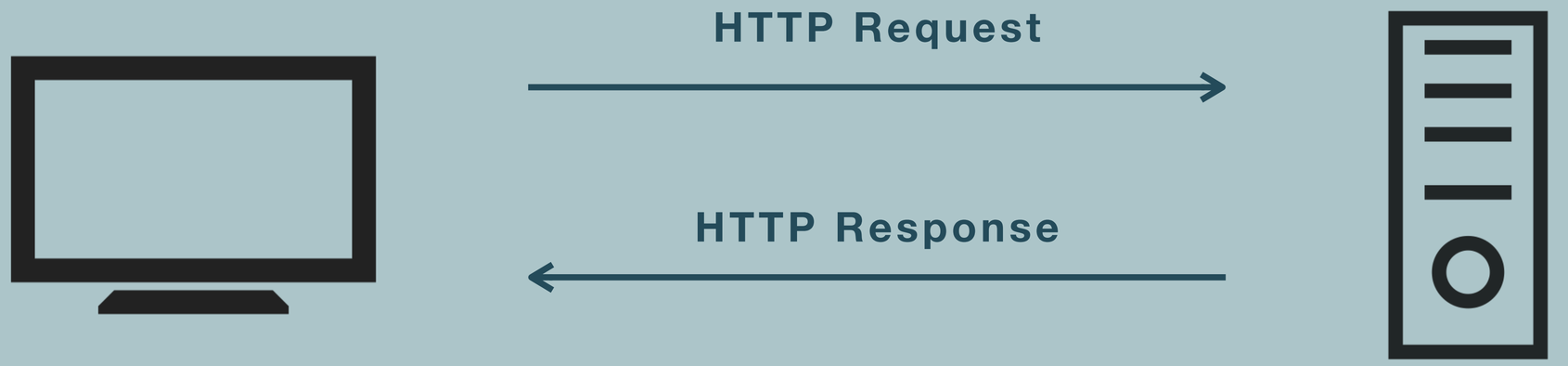
```
export async function loader({ params }) {
  const post = await mongoose.models.Post.findById(params.postId);
  return json({ post });
}

export default function UpdatePost() {
  const { post } = useLoaderData();

  return (
    <div className="page">
      <h1>Update Post</h1>
      <Form id="post-form" method="post">
        <input defaultValue={post.caption} name="caption" type="text" />
        <button>Save</button>
      </Form>
    </div>
  );
}

export async function action({ request, params }) {
  const formData = await request.formData();
  const post = Object.fromEntries(formData);

  await mongoose.models.Post.findByIdAndUpdate(params.postId, post);

  return redirect(`/posts/${params.postId}`);
}
```

response

request

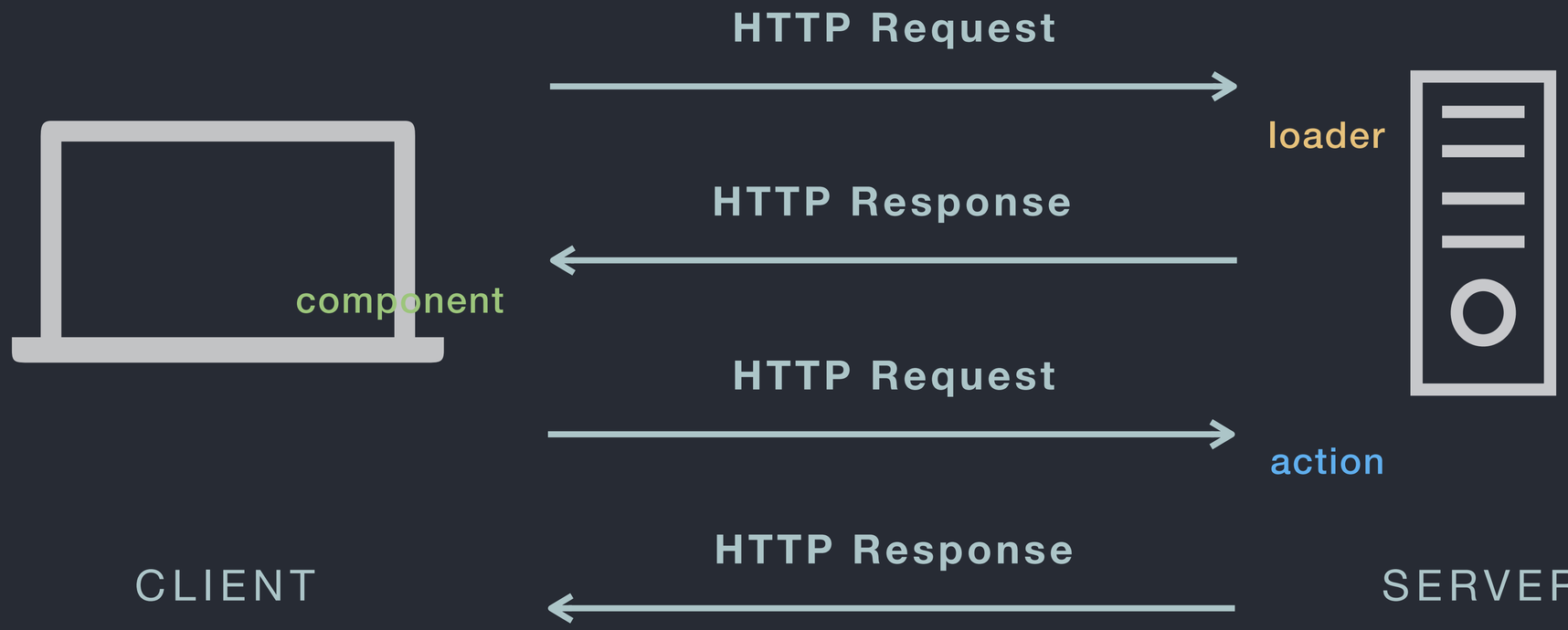
Client-Server Model

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



Client-Server Model

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



loader

component

action

```
export async function loader({ params }) {
  const post = await mongoose.models.Post.findById(params.postId);
  return json({ post });
}

export default function UpdatePost() {
  const { post } = useLoaderData();

  return (
    <div className="page">
      <h1>Update Post</h1>
      <Form id="post-form" method="post">
        <input defaultValue={post.caption} name="caption" type="text" />
        <button>Save</button>
      </Form>
    </div>
  );
}

export async function action({ request, params }) {
  const formData = await request.formData();
  const post = Object.fromEntries(formData);

  await mongoose.models.Post.findByIdAndUpdate(params.postId, post);

  return redirect(`/posts/${params.postId}`);
}
```

Review your routes

Loader -> Component -> Action

	Function name	Runs in browser	Runs on server
returns html	`export default function Component() {}`		
load data for route	`export function loader() {}`		
process form submit + save to db	`export function action()`		

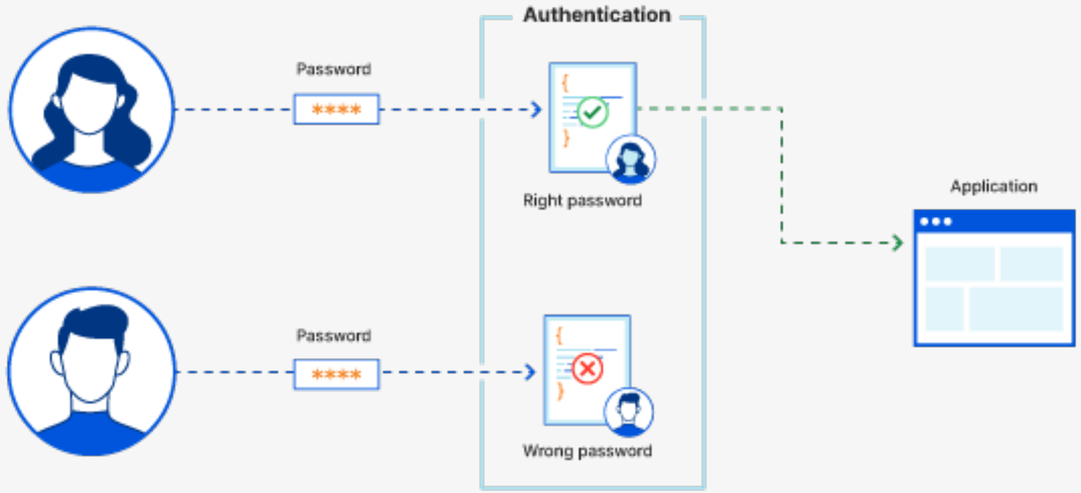
Auth

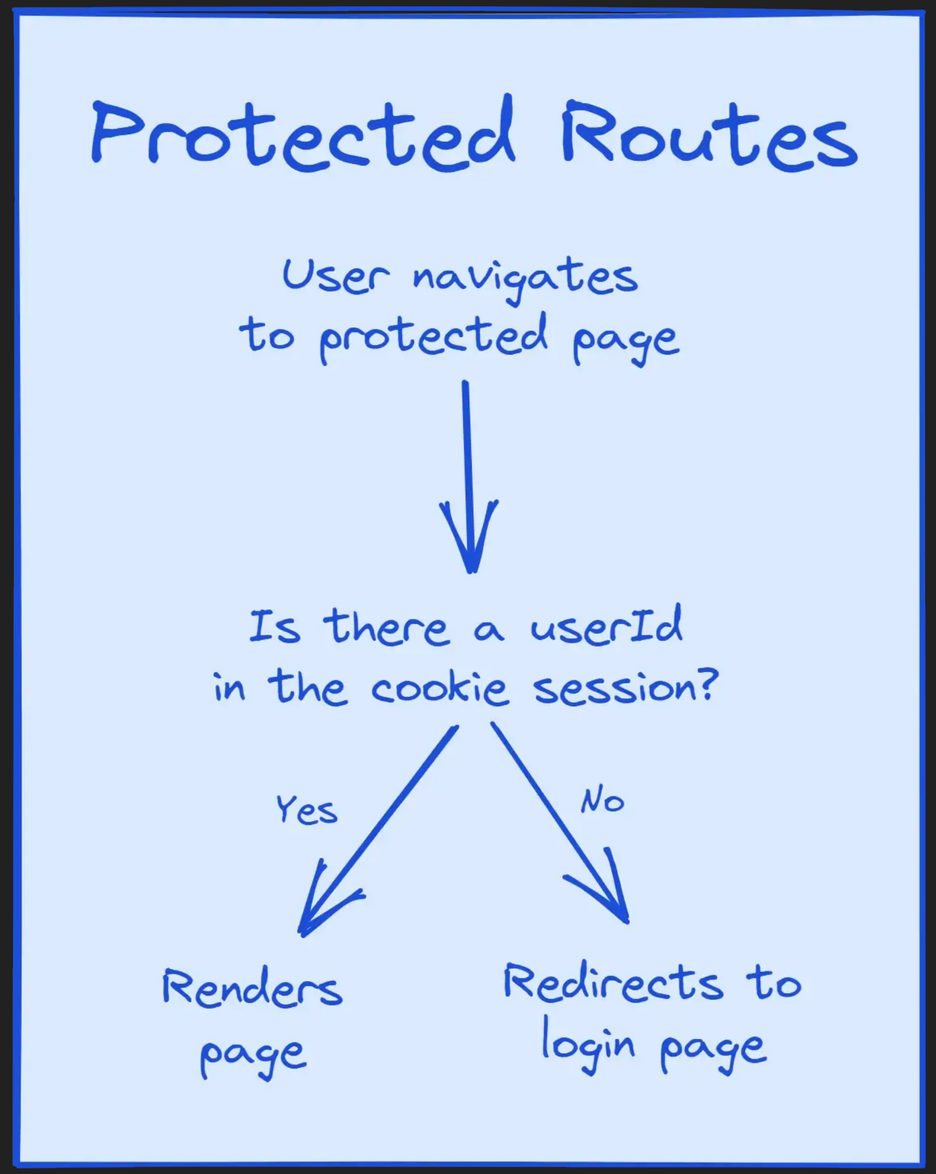
Auth

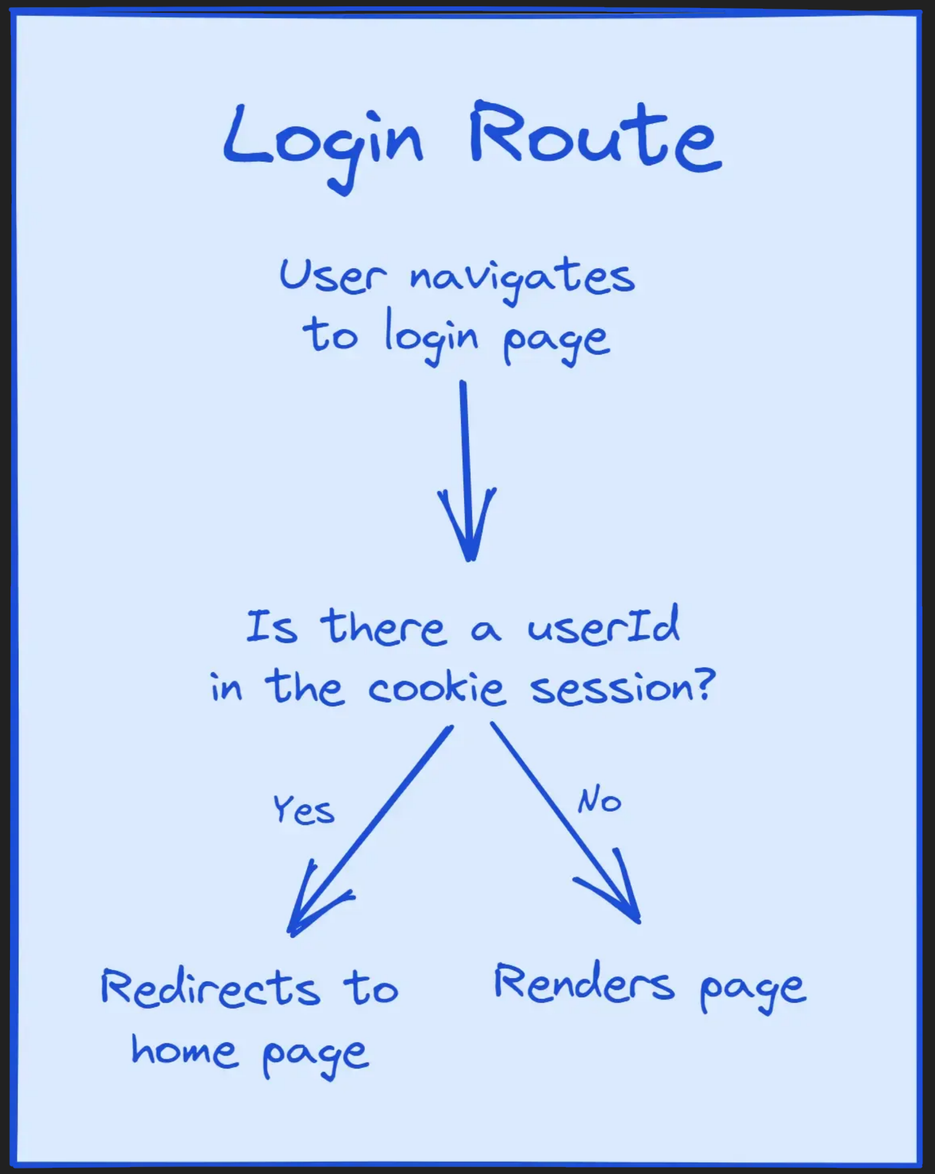
- Remix Auth is a library for implementing authentication in Remix Applications.
- Simplifies the process of adding authentication by providing a structured way to handle user authentication flows, including login, logout, and session management.
- Heavily inspired by `Passport.js`

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

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.

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"
  });
}
```

Auth: Route protection & restriction

Protect routes

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

```jsx
// app/routes/add-post.jsx

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

export default function AddPost() {…
}

export async function action({ request }) {
  // Get the authenticated user
  const user = await authenticator.isAuthenticated(request, {
    failureRedirect: "/signin",
  });

  console.log(user);
  const formData = await request.formData();
  const post = Object.fromEntries(formData);

  // Add the authenticated user's id to the post.user field
  post.user = user._id;
  // Save the post to the database
  await mongoose.models.Post.create(post);
```

Protect routes

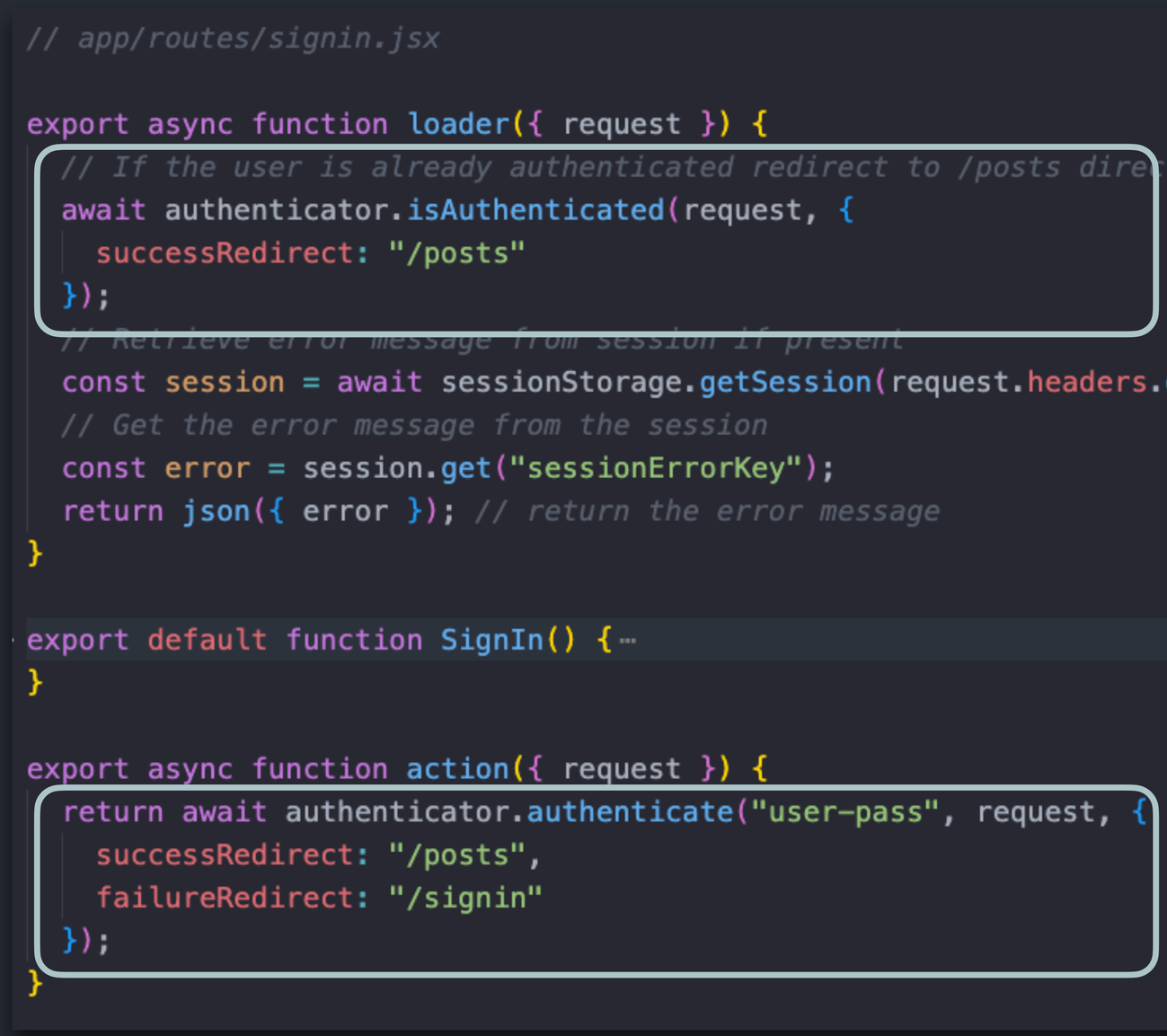
- Loaders: protect all your loaders!
- Actions: protect all your actions!

```jsx
// app/routes/posts.$postId.destroy.jsx

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

export async function action({ request, params }) {
  // Protect the route
  await authenticator.isAuthenticated(request, {
    failureRedirect: "/signin",
  });
  // Delete the post
  await mongoose.models.Post.findByIdAndDelete(params.pos
  return redirect("/posts");
}
```

Loaders and actions must return a response

Protect routes

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

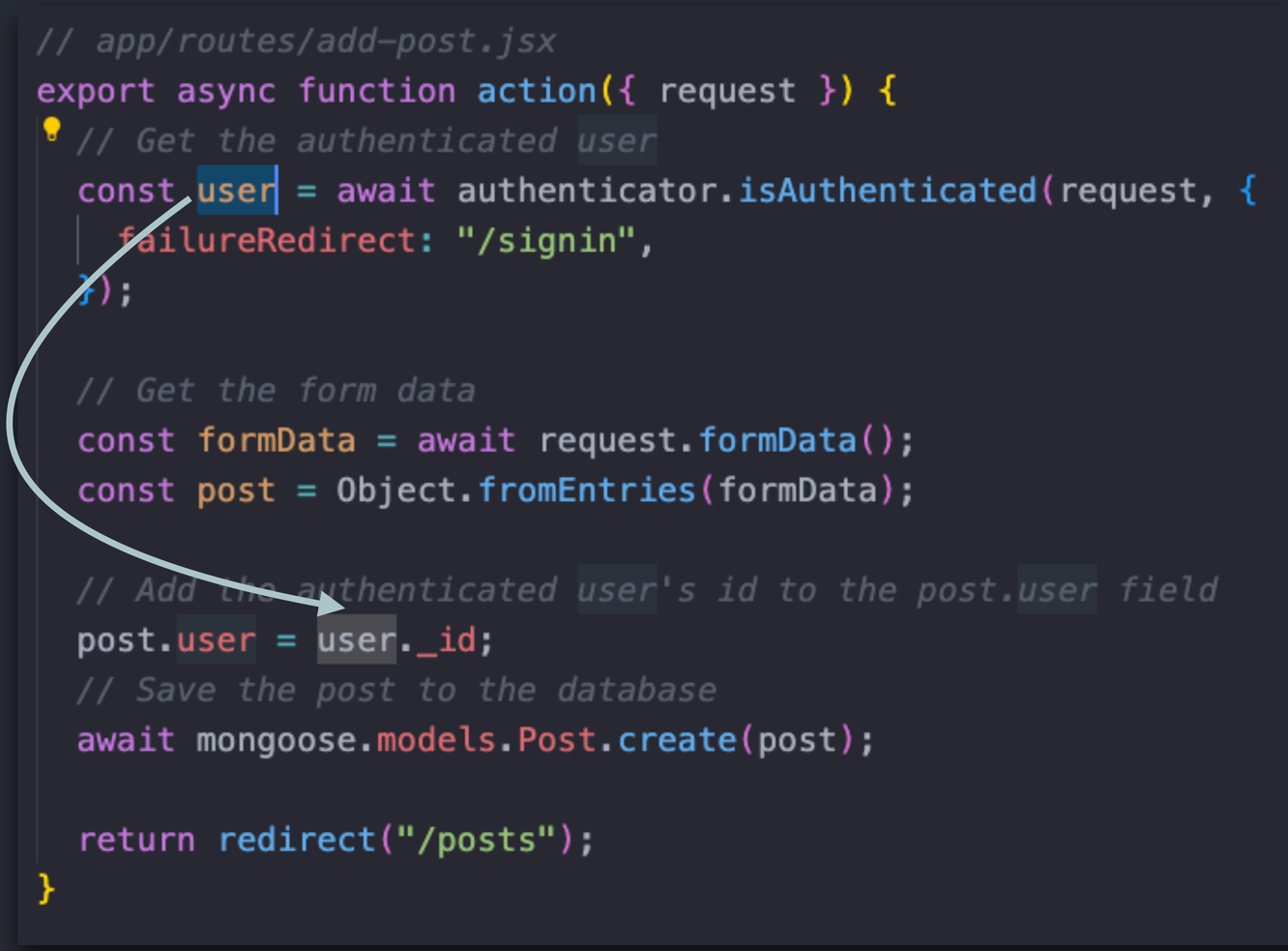


```jsx
// app/routes/signin.jsx

export async function loader({ request }) {
  // If the user is already authenticated redirect to /posts dire
  await authenticator.isAuthenticated(request, {
    successRedirect: "/posts"
  });
  // Retrieve error message from session if present
  const session = await sessionStorage.getSession(request.headers.
  // Get the error message from the session
  const error = session.get("sessionErrorKey");
  return json({ error }); // return the error message
}

export default function SignIn() {…
}

export async function action({ request }) {
  return await authenticator.authenticate("user-pass", request, {
    successRedirect: "/posts",
    failureRedirect: "/signin"
  });
}
```

Loaders and actions must return a response

Restrictions

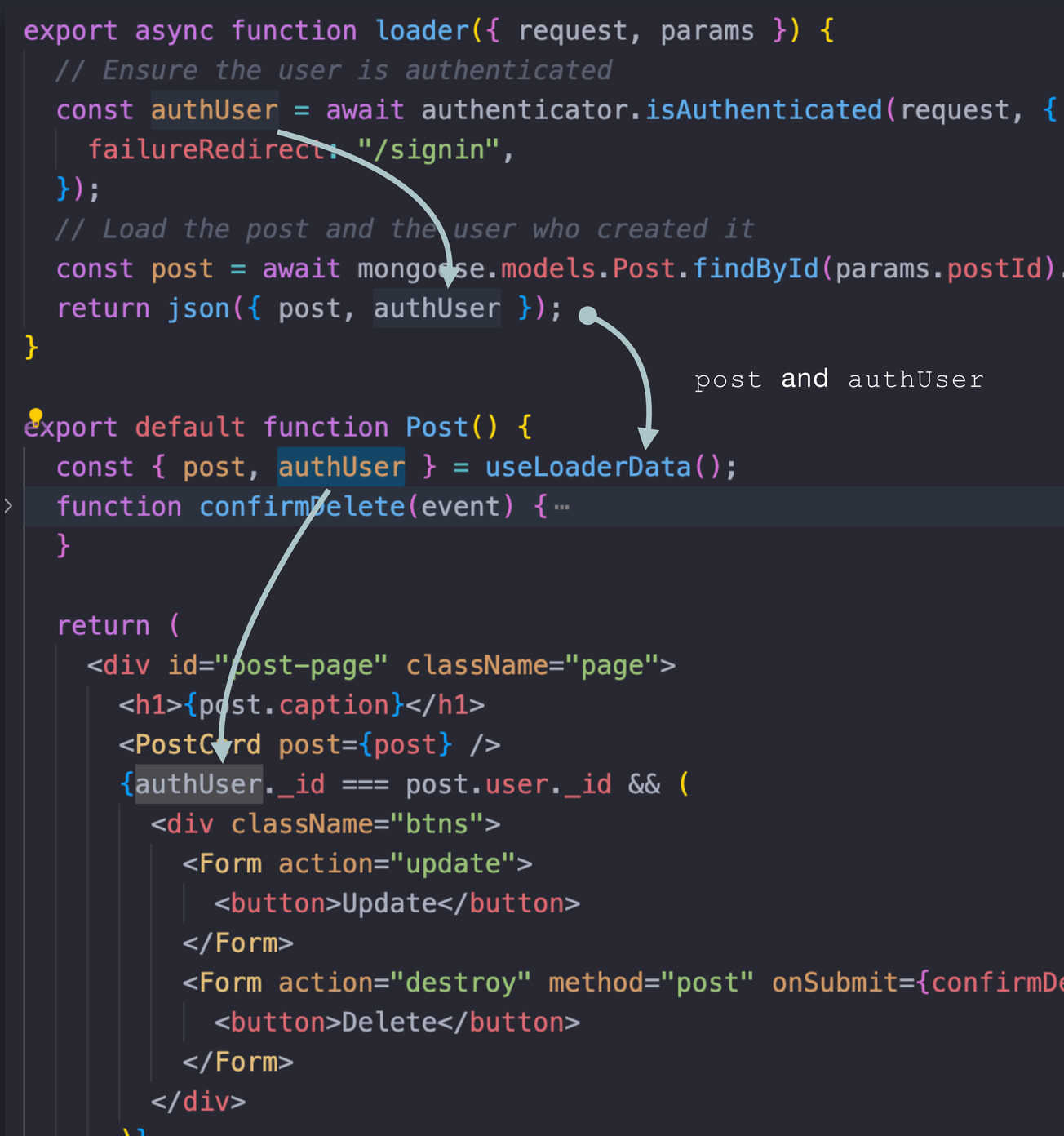
- Assigning Ownership: Assign the Authenticated User to a new post.
- Make sure the authenticated user becomes the owner/ creator of a new post.



```
// app/routes/add-post.jsx
export async function action({ request }) {
  // Get the authenticated user
  const user = await authenticator.isAuthenticated(request, {
    failureRedirect: "/signin",
  });

  // Get the form data
  const formData = await request.formData();
  const post = Object.fromEntries(formData);

  // Add the authenticated user's id to the post.user field
  post.user = user._id;
  // Save the post to the database
  await mongoose.models.Post.create(post);

  return redirect("/posts");
}
```

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 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.

```
export async function action({ request, params }) {
  // Protect the route
  const authUser = await authenticator.isAuthenticated(request, {
    failureRedirect: "/signin",
  });

  // Fetch the post to check if the current user is the creator
  const postToUpdate = await mongoose.models.Post.findById(params.postId);

  if (postToUpdate.user.toString() !== authUser._id.toString()) {
    // User is not the creator of the post, redirect
    return redirect(`/posts/${params.postId}`);
  }

  // User is authenticated and is the creator, proceed to update the post
  const formData = await request.formData();
  const post = Object.fromEntries(formData);

  // Since postToUpdate is already the document you want to update,
  // you can directly modify and save it, which can be more efficient than
  postToUpdate.caption = post.caption;
  postToUpdate.image = post.image;
  await postToUpdate.save();

  return redirect(`/posts/${params.postId}`);
}
```

Auth: AuthorizationError

AuthorizationError

- Set `AuthorizationError` in `authenticator`, `auth.server.jsx`

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

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

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

    // verify the user
    const user = await verifyUser({ mail, password });
    if (!user) {
      // if problem with user throw error AuthorizationError
      throw new AuthorizationError("User not found");
    }
    console.log(user);
    return user;
  }),
  "user-pass"
);
```

AuthorizationError

- And set `AuthorizationError` in `verifyUser`, `auth.server.jsx`

```
async function verifyUser({ mail, password }) {
  const user = await mongoose.models.User.findOne({ mail }).select("+p
  if (!user) {
    throw new AuthorizationError("No user found with this email.");
  }

  const passwordMatch = await bcrypt.compare(password, user.password);
  if (!passwordMatch) {
    throw new AuthorizationError("Invalid password.");
  }
  // Remove the password from the user object before returning it
  user.password = undefined;
  return user;
}
```

AuthorizationError

- In loaders, read errors from the session, `sessionErrorKey`.
- Save error in a variable and return to Component.
- Read error through `useLoaderData` in the Component.

```
export async function loader({ request }) {
  // If the user is already authenticated redirect to /posts directly
  await authenticator.isAuthenticated(request, {
    successRedirect: "/posts",
  });
  // Retrieve error message from session if present
  const session = await sessionStorage.getSession(request.headers.get("Cookie"));
  // Get the error message from the session
  const error = session.get("sessionErrorKey");
  // Remove the error message from the session after it's been retrieved
  session.unset("sessionErrorKey");
  // Commit the updated session that no longer contains the error message
  const headers = new Headers({
    "Set-Cookie": await sessionStorage.commitSession(session),
  });

  return json({ error }, { headers }); // return the error message
}

export default function SignUp() {
  // if i got an error it will come back with the loader dxata
  const loaderData = useLoaderData();
  console.log("error:", loaderData?.error);
```

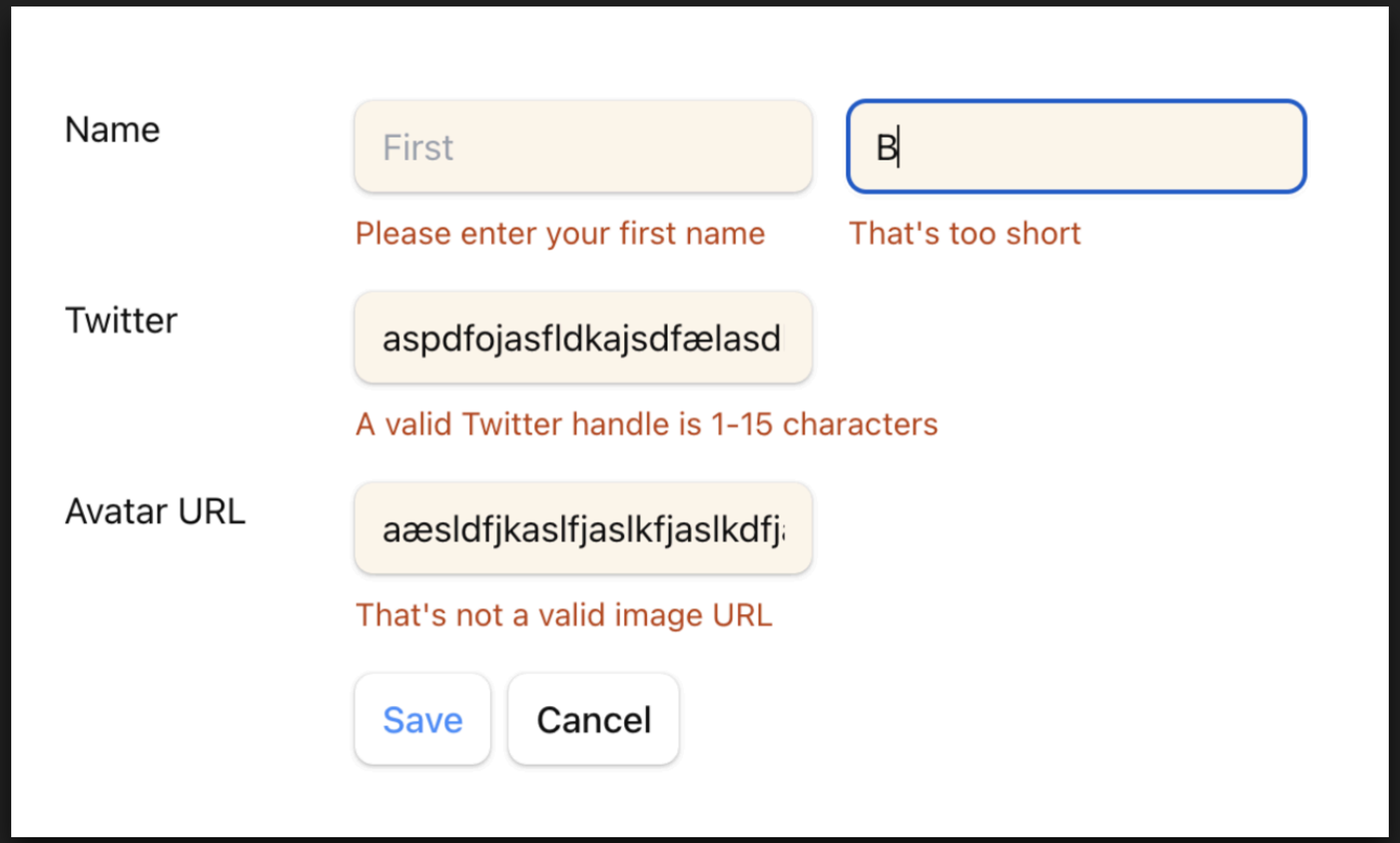
AuthorizationError

- Reset the `sessionErrorKey` in loader for signin and signup route.

```
export async function loader({ request }) {
  // If the user is already authenticated redirect to /posts direc
  await authenticator.isAuthenticated(request, {
    successRedirect: "/posts",
  });
  // Retrieve error message from session if present
  const session = await sessionStorage.getSession(request.headers.
  // Get the error message from the session
  const error = session.get("sessionErrorKey");
  // Remove the error message from the session after it's been ret
  session.unset("sessionErrorKey");
  // Commit the updated session that no longer contains the error
  const headers = new Headers({
    "Set-Cookie": await sessionStorage.commitSession(session),
  });

  return json({ error }, { headers }); // return the error message
}
```

Form Validation and Error Messages

Form Validation and Error Messages



Define and use “schema errors”

```
export async function action({ params, request }) {
  invariant(params.contactId, "Missing contactId param");
  const formData = await request.formData();
  const contact = await mongoose.models.Contact.findById(params.contactId);
  if (!contact) {
    throw new Response("Contact not found", { status: 404 });
  }
  contact.first = formData.get("first");
  contact.last = formData.get("last");
  contact.twitter = formData.get("twitter");
  contact.avatar = formData.get("avatar");
  try {
    await contact.save();
  } catch (error) {
    return json(
      {
        // Pass the validation errors back to the client to be displayed
        // alongside the relevant form fields. We get these errrors from the
        // Mongoose model's validation:
        // https://mongoosejs.com/docs/validation.html#validation-errors
        errors: error.errors,
        // Pass the form data back to the client as "values" so the user doesn't
        // lose their changes in case JavaScript is disabled and the page reloads
        // on submit.
        values: Object.fromEntries(formData),
      },
      { status: 400 },
    );
  }
  return redirect(`/contacts/${params.contactId}`);
}
```

```
const contactSchema = new mongoose.Schema(
  {
    first: {
      type: String,
      required: [true, "Please enter your first name"],
      minLength: [2, "That's too short"],
    },
    last: {
      type: String,
      required: [true, "Please enter your last name"],
      minLength: [2, "That's too short"],
    },
    avatar: {
      type: String,
      match: [
        /^(https?:\/\/.*\.(?:png|jpg|jpeg|gif|svg|webp))$/i,
        "That's not a valid image URL",
      ],
    },
    twitter: {
      type: String,
      match: [/^@?(\w){1,15}$/, "A valid Twitter handle is 1-15 characters"],
    },
    favorite: {
      type: Boolean,
      default: false,
    },
    notes: [
      {
        note: {
          type: String,
          required: true,
          minLength: [2, "That's too short"],
        },
      },
    ],
  },
  // Automatically add `createdAt` and `updatedAt` timestamps:
```

Read errors in Component

```tsx
export async function action({ params, request }) {
  invariant(params.contactId, "Missing contactId param");
  const formData = await request.formData();
  const contact = await mongoose.models.Contact.findById(params.contactId);
  if (!contact) {
    throw new Response("Contact not found", { status: 404 });
  }
  contact.first = formData.get("first");
  contact.last = formData.get("last");
  contact.twitter = formData.get("twitter");
  contact.avatar = formData.get("avatar");
  try {
    await contact.save();
  } catch (error) {
    return json(
      {
        // Pass the validation errors back to the client to be displayed
        // alongside the relevant form fields. We get these errrors from the
        // Mongoose model's validation:
        // https://mongoosejs.com/docs/validation.html#validation-errors
        errors: error.errors,
        // Pass the form data back to the client as "values" so the user doesn't
        // lose their changes in case JavaScript is disabled and the page reloads
        // on submit.
        values: Object.fromEntries(formData),
      },
      { status: 400 },
    );
  }
  return redirect(`/contacts/${params.contactId}`);
}
```

```tsx
export default function EditContact() {
  const { contact } = useLoaderData();
  const actionData = useActionData();
  const navigate = useNavigate();

  return (
    <Form id="contact-form" method="post">
      <p>
        <span>Name</span>
        <Input
          name="first"
          ariaLabel="First name"
          defaultValue={actionData?.values?.first ?? contact.first}
          placeholder="First"
          errorMessage={actionData?.errors?.first?.message}
        />
        <Input
          name="last"
          ariaLabel="Last name"
          defaultValue={actionData?.values?.last ?? contact.last}
          placeholder="Last"
          errorMessage={actionData?.errors?.last?.message}
        />
      </p>
      <label htmlFor="twitter">
        <span>Twitter</span>
        <Input
          name="twitter"
          defaultValue={actionData?.values?.twitter ?? contact.twitter}
          placeholder="Twitter"
          errorMessage={actionData?.errors?.twitter?.message}
        />
      </label>
      <label htmlFor="avatar">
        <span>Avatar URL</span>
        <Input
          name="avatar"
          ariaLabel="Avatar URL"
          defaultValue={actionData?.values?.avatar ?? contact.avatar}
          placeholder="https://example.com/avatar.jpg"
          errorMessage={actionData?.errors?.avatar?.message}
        />
      </label>
```

Destructuring in Remix: loaders & actions

Destructuring

- A JavaScript feature for unpacking or extracting *values from arrays* or *properties from objects* into distinct *variables*.
- Destructuring enhances code readability and efficiency by reducing the verbosity of accessing data in arrays and objects and simplifying parameter handling in functions.

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

  const posts = await mongoose.models.Post.find()
    .sort({ createdAt: -1 })
    .populate("user");

  return json({ posts });
}

export default function Index() {
  const { posts } = useLoaderData();
  return (
    <div className="page">
      <h1>Posts</h1>
      <section className="grid">
        {posts.map((post) => (
          <Link key={post._id} className="post-link" to={`${post._id}`}>
            <PostCard post={post} />
          </Link>
        ))}
      </section>
    </div>
  );
}
```

Destructuring

1.7. Destructuring

What: Extract what we need from an existing array or object.

Why: Makes it easy to extract only what we need from an existing array or object.

Objects

```
const teacher = {
  name: "Morten",
  email: "moab@eaaa.dk"
};

//choose name and email
const { name, email } = teacher;
//name: "Morten"
//email: "moab@eaaa.dk"
```

Arrays

```
const teachers = ["Rasmus", "Morten", "Dan"];

//choose two names
const [mrFrontend, mrWebComponents] = teachers;
//mrFrontend: "Rasmus"
```

Array Destructuring

- Allows unpacking elements from an array into separate variables based on their position.
- Can skip elements by leaving gaps in the variable list.
- Example:

```
const [first, second] = [1, 2];
```

```
const names = ["Anne", "Birthe", "Dan", "Kasper"];

// Without destructuring
const first = names[0]; // Anne
const second = names[1]; // Birthe
const third = names[2]; // Dan
const fourth = names[3]; // Kasper

// With destructuring
const [first, second, third, fourth] = names;
```

Object Destructuring

- Enables unpacking properties from objects into variables based on the property name.
- Allows for renaming variables to avoid naming conflicts or for clarity.
- Example:

```
const { name: userName, age } =
{ name: 'John', age: 30 };
```

```
const person = {
  name: "John Doe",
  age: 30,
};

// Without destructuring
const name = person.name;
const age = person.age;


// With destructuring
const { name, age } = person;
```

Function Parameter Destructuring

- Simplifies extracting values from an object or array passed as a function parameter.
- Reduces the need for manual extraction within the function body.
- Example: `function greet({ name }) { console.log(name); }`

```
function introduce({ name, age }) {
  console.log(`My name is ${name} and I am ${age} years old.`);
}

const person = {
  name: "John Doe",
  age: 30,
};

introduce(person); // Outputs: My name is John Doe and I am 30 ye
```

Destructuring in loaders and actions

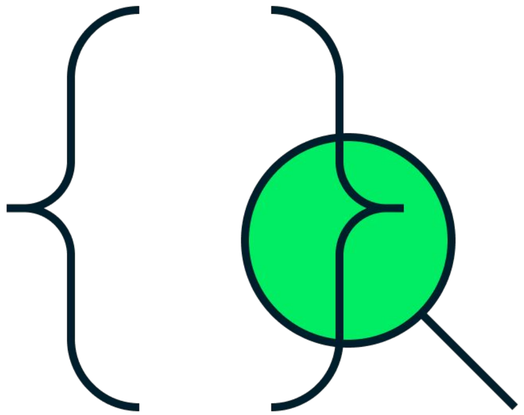
- **Function Parameter Destructuring**: Extracts request and params directly in the loader function, simplifying property access.
- **Response Object Destructuring**: Combines post and authUser into a single response in loader, enhancing code clarity.
- **Component Data Destructuring**: The Post component uses destructuring to access post and authUser from useLoaderData(), improving readability.
- **Conditional Rendering with Destructured Data**: Utilizes `authUser._id` and `post.user._id` to conditionally render update/delete buttons, streamlining authorization checks.
- **Readability and Maintenance**: Destructuring at the start of post clarifies data dependencies, aiding in readability and maintenance.

Querying with Mongoose

Simple syntax

Designed to query documents

Only queries a single collection



MongoDB Query Language

$lt Exists and less than

$lte: Exists and less than or equal to

$gt: Exists and greater than

$gte: Exists and greater than or equal to

$ne: Does not exist or does but not equal to

$eq: Exists and equal to

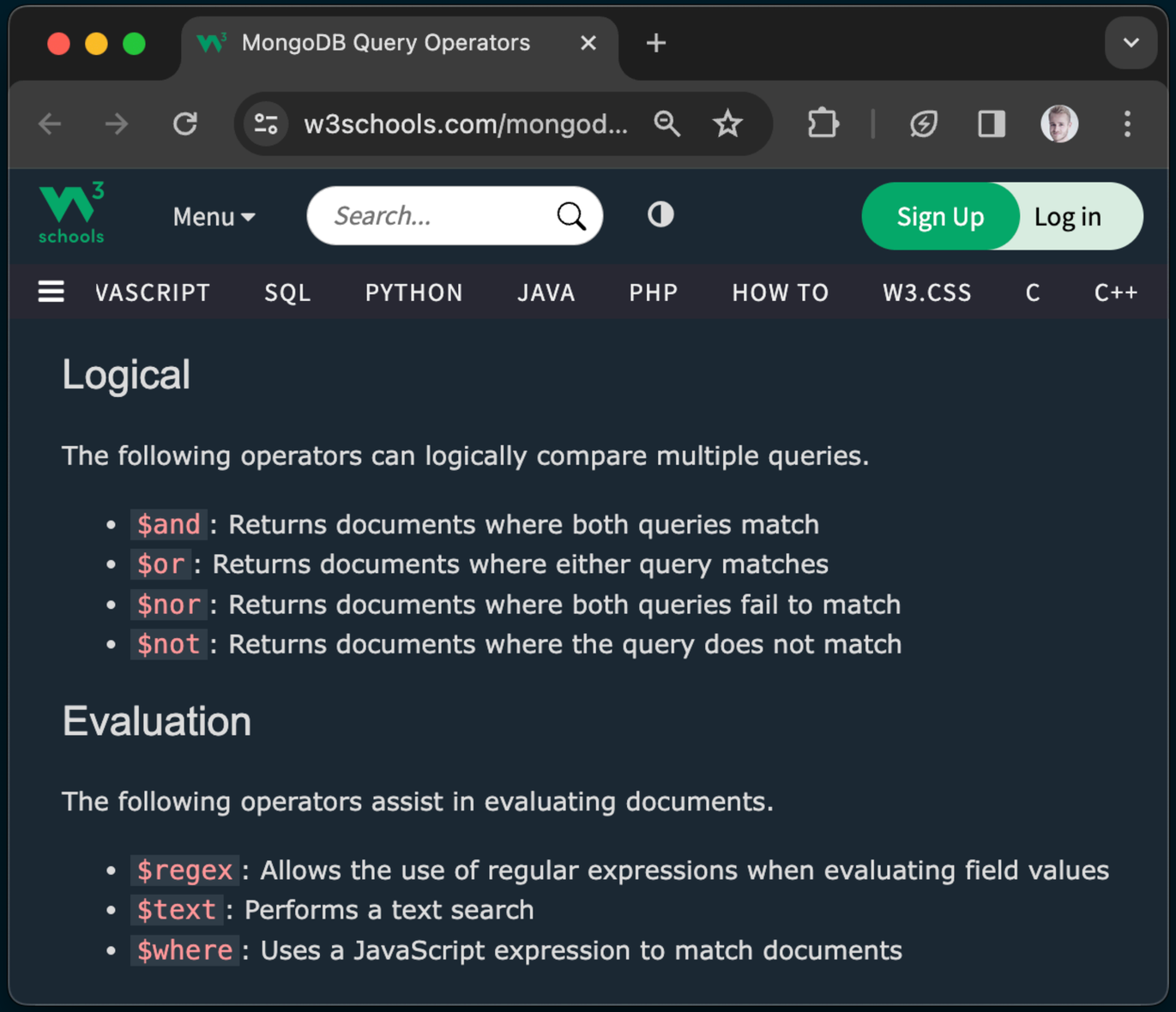
$in: Exists and in a set

$nin: Does not exist or not in a set

MQL Query Operators: Comparison

```
db.people.find(
    {
     “age”: { $lt: 30 }
    }
)
```

With the age field value being $lt (less than) 30



https://www.w3schools.com/mongodb/mongodb_query_operators.php

Query Users

1. **Find all users:**

javascript Copy code

```javascript
const allUsers = await User.find();
console.log(allUsers);
```

2. **Find a specific user by ID:**

javascript Copy code

```javascript
const specificUser = await User.findById("user_id_here");
console.log(specificUser);
```

3. **Find users with a specific title:**

javascript Copy code

```javascript
const usersWithTitle = await User.find({ title: "Senior Lecturer" });
console.log(usersWithTitle);
```

Query Posts

1. **Find all posts:**

javascript Copy code

```javascript
const allPosts = await Post.find();
console.log(allPosts);
```

2. **Find posts with a specific tag:**

javascript Copy code

```javascript
const postsWithTag = await Post.find({ tags: "tag_name_here" });
console.log(postsWithTag);
```

3. **Find posts with multiple tags:**

javascript Copy code

```javascript
const postsWithTags = await Post.find({ tags: { $in: ["tag1", "tag2"] } });
console.log(postsWithTags);
```

4. **Find posts by a specific user:**

javascript Copy code

```javascript
const userPosts = await Post.find({ user: "user_id_here" });
console.log(userPosts);
```

5. **Find posts with a minimum number of likes:**

javascript Copy code

```javascript
const popularPosts = await Post.find({ likes: { $gte: 100 } });
console.log(popularPosts);
```

Greater Than and Less Than Queries

1. **Find posts with more than 100 likes:**

javascript Copy code

```javascript
const postsWithMoreThan100Likes = await Post.find({ likes: { $gt: 100 } });
console.log(postsWithMoreThan100Likes);
```

2. **Find posts with less than 50 likes:**

javascript Copy code

```javascript
const postsWithLessThan50Likes = await Post.find({ likes: { $lt: 50 } });
console.log(postsWithLessThan50Likes);
```

3. **Find posts with likes between 50 and 100 (inclusive):**

javascript Copy code

```javascript
const postsWithLikesBetween50And100 = await Post.find({ likes: { $gte: 50, $lte: 100 } });
console.log(postsWithLikesBetween50And100);
```

One tag

```
export async function loader() {
  const posts = await mongoose.models.Post.find({ tags: "Aarhus" })
    .sort({ createdAt: -1 })
    .populate("user");

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

Multiple tags

```
export async function loader() {
  const posts = await mongoose.models.Post.find({ tags: { $in: ["tag1", "tag2"] } });

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

Multiple tags

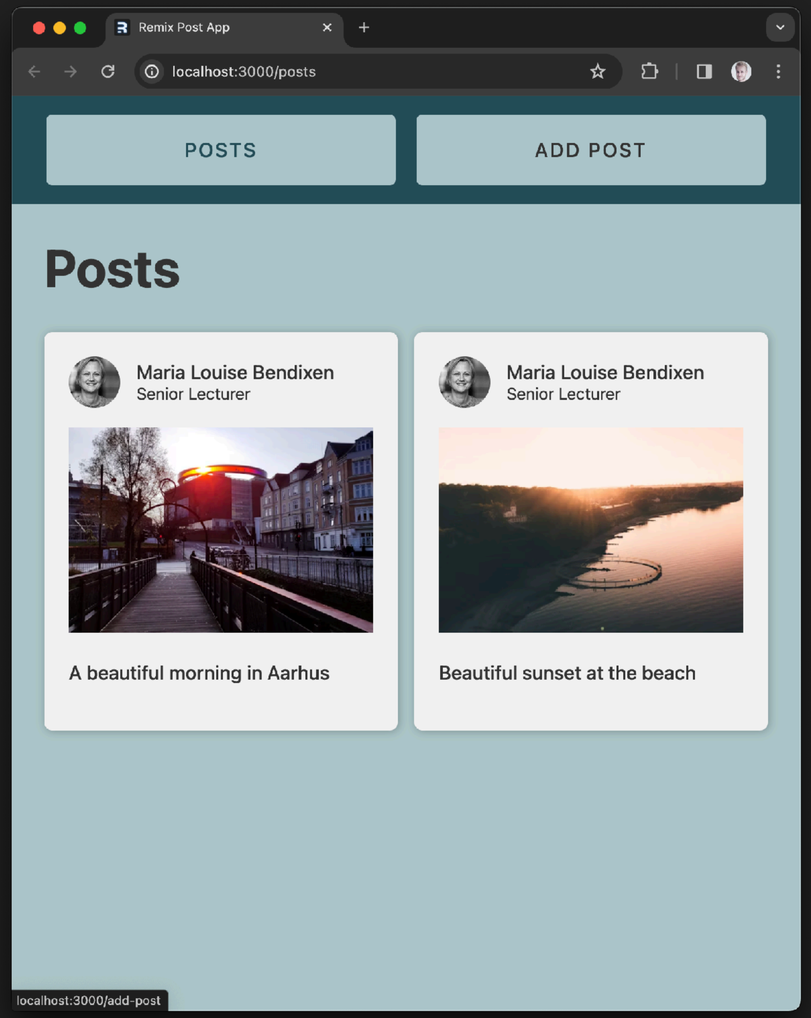
```
export async function loader() {
  const posts = await mongoose.models.Post.find({
    tags: { $in: ["Aarhus", "city"] },
  })
    .sort({ createdAt: -1 })
    .populate("user");

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

Post by specific user

```
export async function loader() {
  const posts = await mongoose.models.Post.find({ user: "65cde4cb0d09cb615a23db17" });

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

Search - caption: “beautiful”

```
export async function loader() {
  const posts = await mongoose.models.Post.find({
    caption: { $regex: "beautiful", $options: "i" }
  })
    .sort({ createdAt: -1 })
    .populate("user");

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



Search - caption: “aarh”

```
export async function loader() {
  const posts = await mongoose.models.Post.find({
    caption: { $regex: "aarh", $options: "i" }
  })
    .sort({ createdAt: -1 })
    .populate("user");

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