Evolve Full Stack Developer

Full Stack: React & React-Router

INCEPTIONU

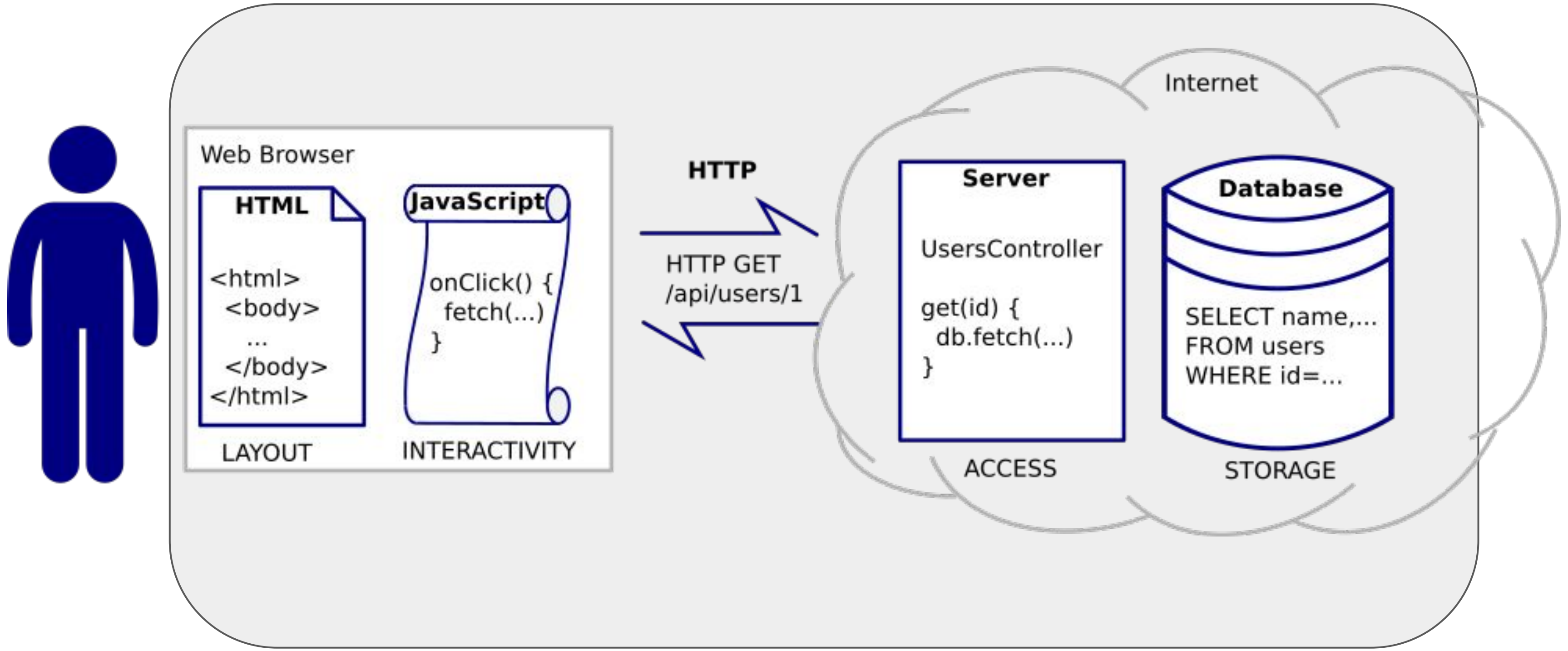
When you press the dog button, the whole thing transforms into the shape of a dog!

Agenda

Full stack development with REACT

- Our Project Idea
- Setup server (one simple ‘get’ endpoint)
- Setup client (useEffect to get our data)
- Detail view component (get one)
- Use “the router” to “change pages” in our app
- Add create page and post route

Focus Area



INCEPTIONU

Superhuman Registration Act

Our idea is to create the registration system to support the “Superhuman Registration Act”.

1. Phase One will support the registration of superhumans. The public should able to review the list.

2. Phase Two will add restrictions on who can add and edit superhuman profiles (Authorization/Authentication)

3. Phase Three will support reporting and notifications

The repos for this series of slides

Get them with Git

We will be working with this code. You can fetch the resources we will be using in class from here:

https://github.com/EvolveU-FSD/c11-superheroes

You can also grab them through VSCode

Our Backlog

To Do:

- Public: list view
- Agent: look at superhuman detail (and put it in it’s own page)
- If there is time, create superhero form

Decision Time

The first thing we will build?

- Get off the blank page as soon as we can
- Make a single thing that works ALL the way through the stack so the rest of the team can join in
- Choose “list superheroes”

Back to Front, or Front to Back?

- For every feature you build out, you have the decision of making the changes to the front end application first or build out the back end first?
- Some advantages and disadvantages to both
- Let’s choose together

Sample Data

Who is in our test data?

- What fields make up a “superhero” in our system?
- Let’s create a few records to start with… as .json

INCEPTIONU

Server Setup

Set up backend

```
cd backend

npm init

npm install express dotenv mongoose

create server.js file

add import to express, const express = require('express')
```

Server Setup

Set up .env file with environment variables and install dotenv

```
DEBUG=server:*

MONGODB_URI=mongodb+srv://admin:VZuvCR4ZZyN8HbM@cluster0.kknyaqy.mongodb.net/superhero?retryWrites=true&w=majority
```

Add dotenv configuration to server.js

```
require('dotenv').config();
```

Mongoose Setup

```
$ npm install mongoose debug dotenv
$ mkdir db
```

Create a new ***db*** folder at the root of the project to hold our database code

Create ***db/mongoose.js*** to connect to our superhero database

```
import mongoose from 'mongoose';
import dotenv from 'dotenv';
dotenv.config();

const connectionString = process.env.MONGODB_URI || 'mongodb://localhost:27017/superheroes';

mongoose.connect(connectionString, ()=>{
  debug(`connected to mongoose on ${connectionString}`);
});

export default mongoose;
```

INCEPTIONU

Schema and Model

$ mkdir models

Create a folder inside ***db*** called ***models***

Inside we will create a file called ***superheroModel.js***

We will structure it based on what we decided earlier, here's an example of what it might look like:

```
import mongoose from '../mongoose';
const Schema = mongoose.Schema;

const superheroSchema = new Schema({
  name: { type: String, required: true, unique: true },
  alterEgo: String,
  powers: [String],
  sidekicks: [{ name: String, alterEgo: String }],
});

const Superhero = mongoose.model("Superhero", superheroSchema);
```

Adding some superheroes to our database

Creating a new superhero

In ***superheroModel.js***, let’s create a function for *adding a new superhero* to the database:

```
export const createSuperhero = async (superhero) => {
  const newSuperhero = await Superhero.create(superhero);
  return newSuperhero;
};
```

Adding some superheroes to our database

Load a few superheroes to start

in our db folder let’s create a file to initialize the database with some starting data:

```
import initialData from './superheroes.json';
import { createSuperhero } from '../db/models/superheroModel.js';

const loadInitalData = async () => {
  for (let i = 0; i < initialData.length; i++) {
    const superhero = initialData[i];
    try {
      console.log(`creating superhero ${superhero.name}`);
      const newSuperhero = await createSuperhero(superhero);
      console.log(`created superhero ${newSuperhero.name} with id ${newSuperhero._id}`);
    } catch (err) {
      console.log(`error creating superhero ${superhero.name}`);
      console.log(err.message);
    }
  }
  console.log('finished loading initial data');
};
loadInitalData();
```

INCEPTIONU

Front End (Create vite project)

npm create vite@latest

- Choose the name of the project, we'll use frontend
- Choose React
- Choose Javascript
- Navigate to the frontend folder (cd frontend)
- Install packages (npm install)
- Start dev server (npm run dev)

The basic vite React app should now have started and be showing in the browser

INCEPTIONU

Front End (Create a component)

Work up to using state to hold our list of superheroes

- Layout a **`<div>`** with our information in it (single entry with static content)
- Extract that **`<div>`** to be a component with props for the variable content
- Push that component to another file
- Use **`Array.map()`** to make this component repeat for every element in a list (static content)
- Use state to hold the list

This is a no-fail way to VERY QUICKLY make components that get the job done. Add complexity later

Connect to the server

useState and useEffect

First, let's adjust the useState variable in ***App.js*** to better represent what the data will hold

```
const [superheroes, setSuperheroes] = useState();
```

Then we'll adjust the fetch call in ***App.js*** to fetch our superheroes list and set our state to the response

```
useEffect(() => {
    const getSuperheroes = async () => {
      let response = await fetch('/api/superhero');
      let superheroesData = await response.json();
      console.log(`superheroes is:`, superheroes);
      setSuperheroes(superheroesData);
    };
    getSuperheroes();
  }, []);
```

Listing the superheroes in our database (backend)

Creating a new superhero

Let’s create a function in our superheroModel.js to get a *list of superheroes*:

```
export const getAllSuperheroes = async () => {
  const superheroes = await Superhero.find();
  return superheroes;
};
```

Routes

Endpoint for listing all superheroes

In our routes file, let’s import our ***getAllSuperheroes*** function and create an endpoint for listing all superheroes:

```
import { getAllSuperheroes } from '../db/models/superheroModel.js';

router.get('/', async (req, res, next) => {
  try {
    const superheroes = await getAllSuperheroes();
    res.send(superheros);
  } catch (err) {
    debug(err.message);
  }
});
```

Start up the server (npm run start) and try to use this route in postman to list all superheroes.

Link our DetailComponent to the List View

Clicking elements

- onClick is usually supported... lets try it

Wire something up

- useState to hold onto the “selected” id
- Change the onClick to update the state
- Add our component to the page and see that it shows the selected hero!

INCEPTIONU

Success!

Next todo is reorganizing our code and work to make the list and detail views onto their own “pages”

- First some organization
- Making pages with the router
- Create the superhero list “page”
- Create the detail “page”

Create our DetailComponent

Server

- `/superhero/:id`
- Create the model method to look up the superhero

Component

- Accepts an **`id`** as a property
- **`useEffect`** to fetch the superhero
- Add **`<div>`**s to show the properties
- Use the no-fail way to build this component in three steps

Test the component

- Add it to the page, with a hard-coded id

INCEPTIONU

Creating separate “pages” in your front end

Say hello to React Router

- The router interacts with the “url bar”
- It helps you choose which components render based on what the path of the currently selected URL is.
- The main components:
  - Router - an outer level component that is usually at the “App” level
  - Routes - introduces a part of the component hierarchy that depends on the selected path. It contains Route components
  - Route - an entry that gets rendered if the path matches
  - Link - a “hyperlink” that when it gets clicked on, will change the current route

Put our DetailComponent in its own “page”

Add a route to the Routes that matches “superhero/:id”

- ***TECHNIQUE:*** Using route parameters
- Notice this is similar to express and the “route parameters”
- The “useParams” hook can get that id
- Create a component (DetailPage) that gets the id, and then contains our DetailComponent

Success!

Next todo is editing a superhero

- We already have a registration component
- Can we reuse it?

Front End (Create a component)

Creating a form to create a superhero

- Add a form tag <form>
- Inside our form tag, we will add labels and input elements for each field
    - <label><input /></label>
- Create a state variable for each field (useState)
- Add an onChange listener to each input
    - <input onChange={(e)=>setName(e.target.value)} />
- Add a value={name} to our input
- Add an onSubmit handler to the form tag <form onSubmit={handleSubmit}>
- Add a button with type “submit” to the bottom
    - <button type=”submit”>Add Superhero</button>
- Create our handle submit function

Routes

Endpoint for creating superhero

In our routes file, let’s import our ***createSuperhero*** function and create an endpoint for creating a new superhero:

```
import { createSuperhero } from '../db/models/superheroModel.js';

router.post('/', async (req, res, next) => {
  try {
    const superhero = req.body
    const newSuperhero = await createSuperhero(superhero);
    res.send(newSuperhero);
  } catch (err) {
    debug(err.message);
  }
});
```

Start up the server (npm run start) and try to use this route in postman to create a new superhero

Refactor EditComponent from SuperheroForm

EditSuperheroComponent

- Accepts initial values for the four fields (or none)
- Accepts a caption for the button
- Accepts a callback to call when the button is clicked

Test the component

- Use EditSuperheroComponent
- Callback calls the “post” on the superhero endpoint to create
- Does registration form still work?

INCEPTIONU

Create EditPage

EditPage (can start as copy of DetailPage)

- Accepts id
- useEffect to load the superhero
- REUSES the EditSuperheroComponent
- Callback calls “put” or “patch” or whatever

Hook it in

- Add a route “superhero/edit/:id” to allow editing client side
- Add a link on the DetailComponent to go there
- After update - go back to the superhero detail

Test the component

- Does registration form still work?
- Does the edit form also work?

Success!

Almost ready to release!