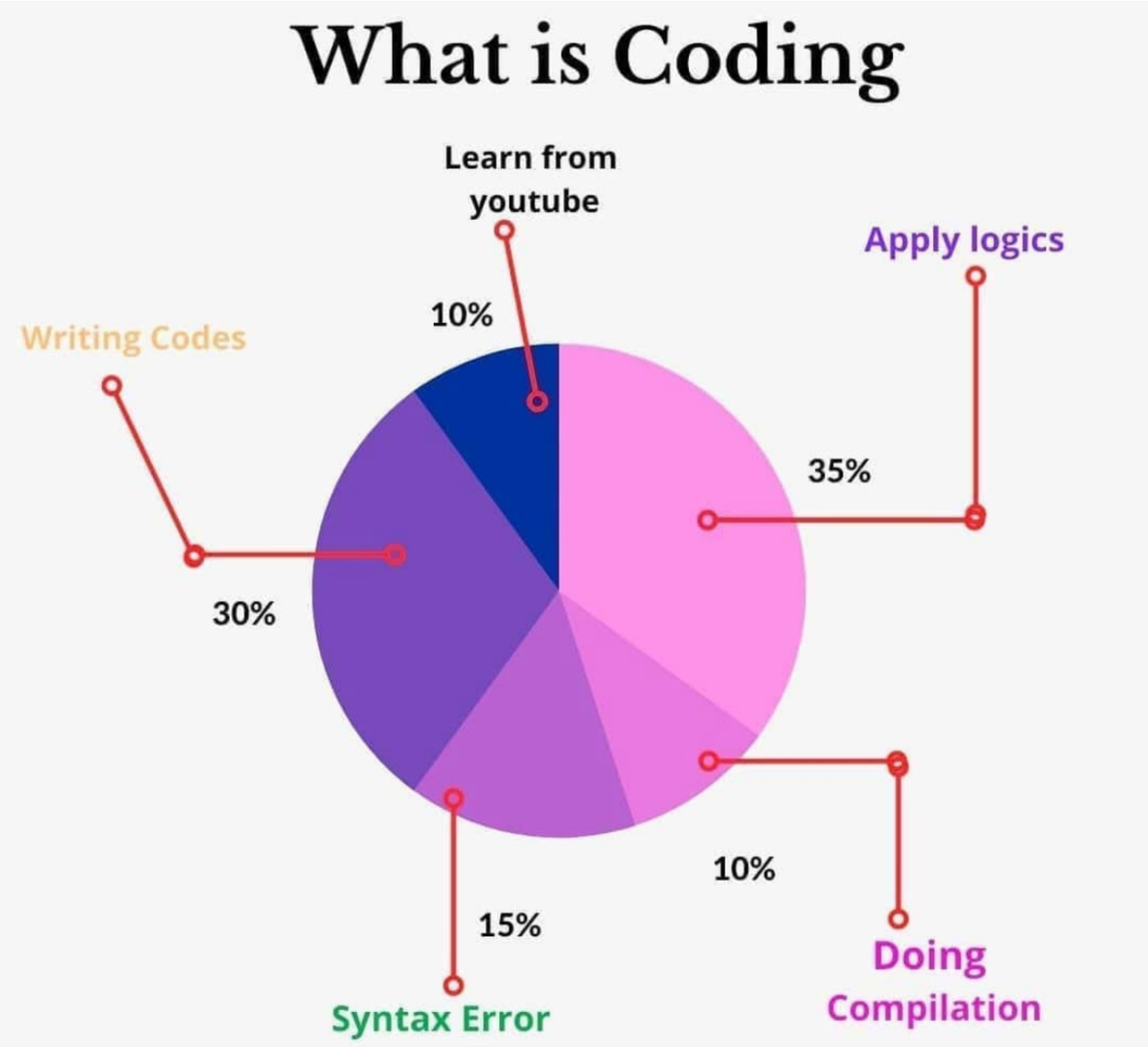


https://www.instagram.com/p/CWAFtsrFCTd/

ADVANCED JAVASCRIPT

FRONTEND DEVELOPMENT

BUSINESS ACADEMY AARHUS
UNIVERSITY OF APPLIED SCIENCES

PURPOSE

... become familiar with frontend setups, build tools & package managers.
Develop an SPA with an object & component-oriented approach.

AGENDA

Frontend build tools & CLI

SPA, Fetch, HTTP methods & client/server

Inspirational talk by Lasse

Working with Object-oriented JS, Modules & Components

EXERCISES

Getting started with Parcel JS

Parcel JS Exercise

Dating SPA using Classes & Modules

BUILDS TOOLS:
PACKAGE MANGERS & MODULE BUNDLERS
&
COMMAND LINE INTERFACE (CLI)

IT IS SIMPLY JUST TOOLS!

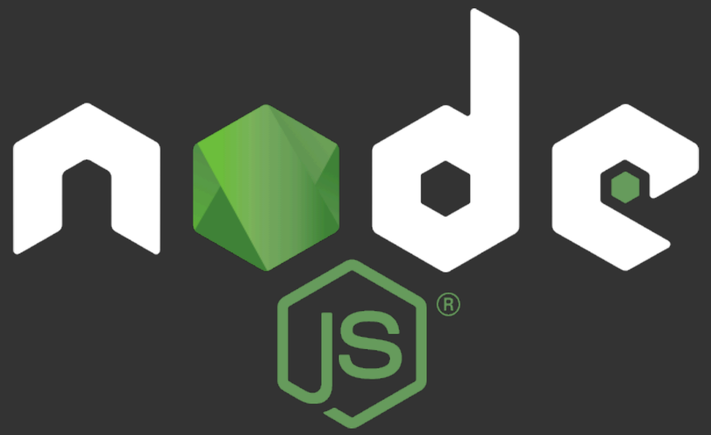
WHY BUILD TOOLS?

ONE OF MANY PACKAGE MANAGERS

Build amazing things

PACKAGE MANAGER & INSTALLER

THE WORLD'S LARGEST SOFTWARE LIBRARY (CODE PACKAGES)



JAVASCRIPT RUNTIME ENVIRONMENT

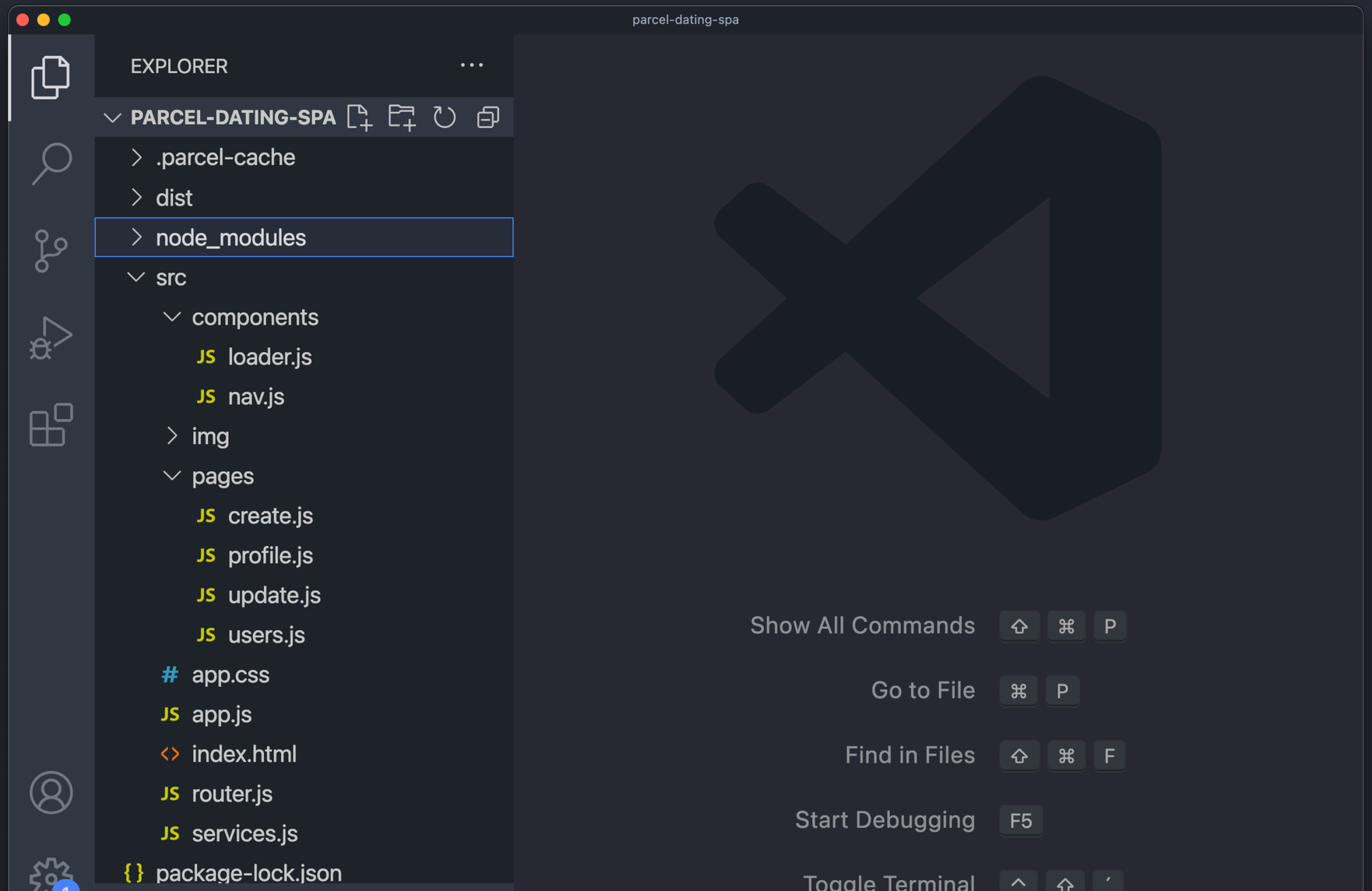
EXECUTES JAVASCRIPT OUTSIDE OF THE BROWSER

NPM IS A PACKAGE MANAGER FOR NODE.JS PACKAGES - OR MODULES 🤷🏼‍♂️

Node.js packages contains all the files you need for a module.

Modules are JavaScript libraries you can include in your project.

node_modules



NPM CONSISTS OF

1. WEBSITE TO DISCOVER PACKAGES
2. COMMAND LINE INTERFACE (CLI)
3. A REGISTRY - DATABASE WITH JS SOFTWARE / NODE MODULES

Use npm to . . .

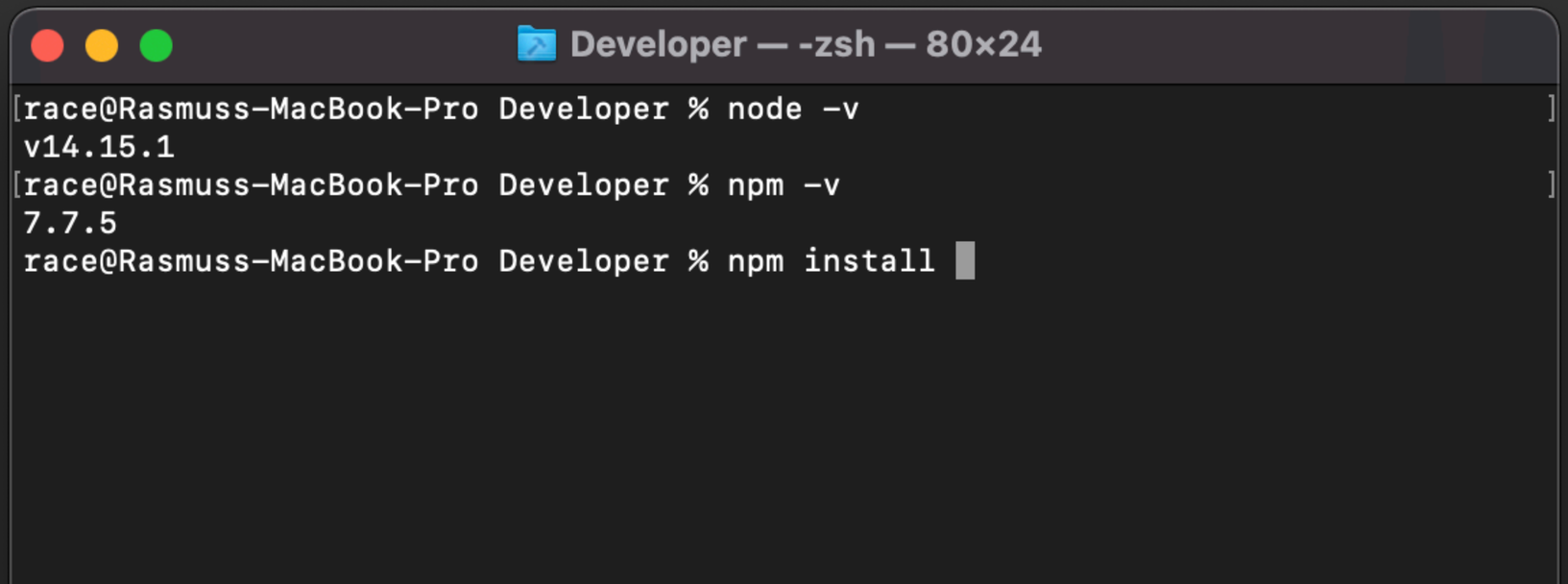
- Adapt packages of code for your apps, or incorporate packages as they are.
- Download standalone tools you can use right away.
- Run packages without downloading using npx.
- Share code with any npm user, anywhere.
- Restrict code to specific developers.
- Create Orgs (organizations) to coordinate package maintenance, coding, and developers.
- Form virtual teams by using Orgs.
- Manage multiple versions of code and code dependencies.
- Update applications easily when underlying code is updated.
- Discover multiple ways to solve the same puzzle.
- Find other developers who are working on similar problems and projects.

https://docs.npmjs.com/about-npm/

COMMAND LINE INTERFACE

... a program that allows users to type text commands to execute functions.

npm uses CLI to install software & packages.



GETTING STARTED

DOWNLOAD & INSTALL NODE.JS & NPM:

NODEJS.ORG/EN/

CHECK YOUR VERSION OF NPM & NODE.JS

race — -zsh — 80×24

```
Last login: Sun Nov 14 19:08:40 on ttys000
race@Rasmuss-MacBook-Pro ~ % node -v
v14.15.1
race@Rasmuss-MacBook-Pro ~ % npm -v
7.7.5
race@Rasmuss-MacBook-Pro ~ %
```

Code File Edit Selection View Go Run Terminal Window Help

Wed 10 Nov 09.28

web-frontend

New Terminal
Split Terminal ⌃⌥⇧⌘7

Run Task...
Run Build Task... ⇧⌘B
Run Active File
Run Selected Text

Show Running Tasks...
Restart Running Task...
Terminate Task...

Configure Tasks...
Configure Default Build Task...

EXPLORE...

- css-grid-layout
- css-selectors
- dating-spa-jsonbin
- dating-spa-modules
- dating-spa-php-backend
- dating-spa-template
- fetch-await-async
- file-upload
- input-suggestions
- json-parse-stringify
- object-page
- object-person
- object-teachers
- parcel-dating-spa
- parcel-spa
- parcel-starter
- project-template
- promises
- spa-canvas-users
- spa-canvas-users-template
- spa-like-carousel
- spa-modules-pages
- spa-modules-portfolio
- spa-movies-fav

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

zsh

```
race@Rasmuss-MacBook-Pro web-frontend % npm -v
7.7.5
race@Rasmuss-MacBook-Pro web-frontend %
```

main ⊗ 0 ⚠ 0

Go Live

... A NODE.JS MODULE

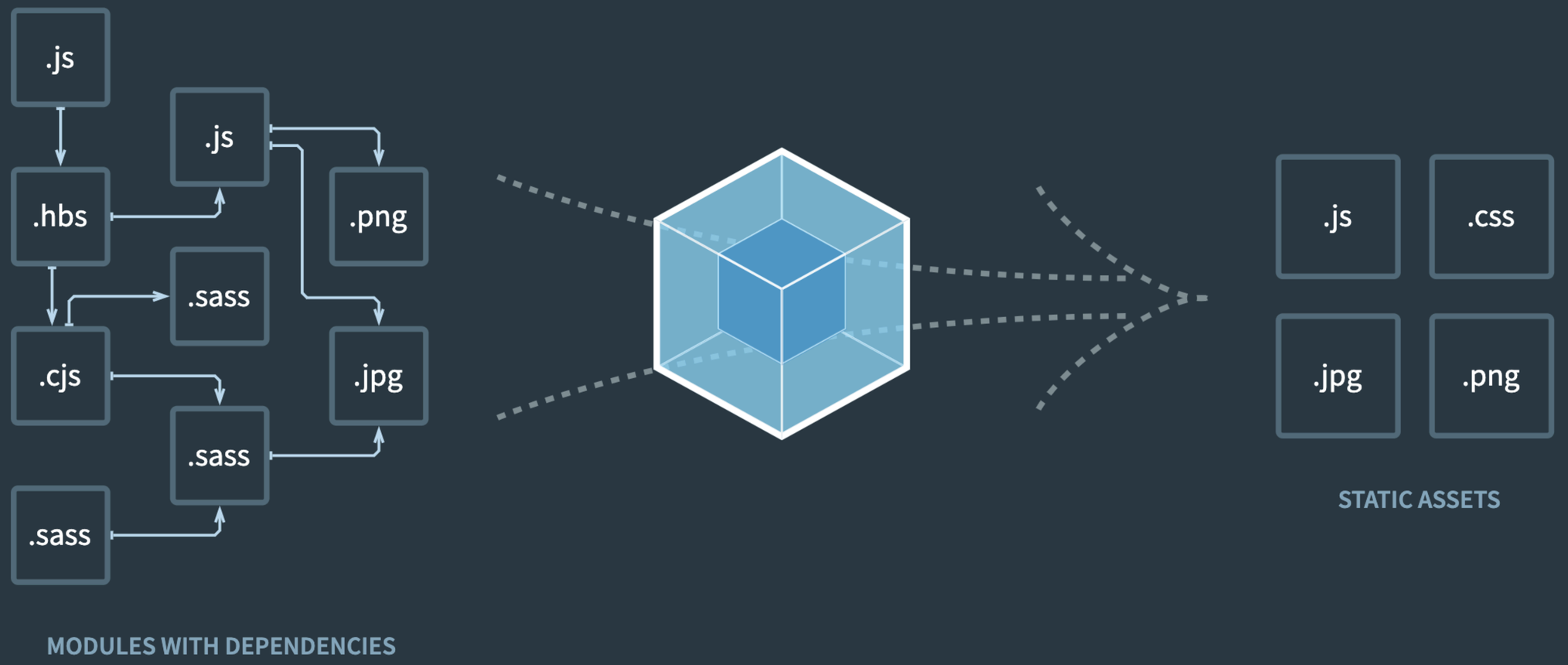
... A MODULE BUNDLER / BUILD TOOL

ONE OF MANY NODE PACKAGES

FROM DEVELOPMENT TO PRODUCTION

MODULE BUNDLER: FROM DEV TO PROD

bundle your scripts



PARCEL

PARCEL

The zero configuration build tool for the web.

ZERO CONFIG: GO EASY FROM DEVELOPMENT TO PRODUCTION

Zero config

Start with an HTML file. Add a **`<script>`** tag. Maybe some CSS. How about TypeScript? SASS? Images? No problem. Parcel works out of the box just as you'd expect.

Parcel supports many languages and file types out of the box, from web technologies like HTML, CSS, and JavaScript, to assets like images, fonts, videos, and more. And when you use a file type that isn't included by default, Parcel will **automatically install** all of the necessary plugins and dev dependencies for you!

Get started →

index.html

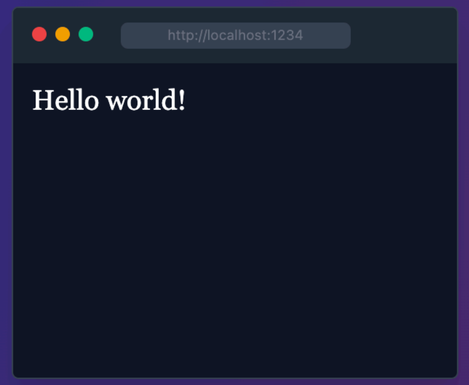
```html
<html>
  <head>
    <title>My First Parcel App</title>
    <link rel="stylesheet" href="styles.css">
  </head>
  <body>
    <h1>Hello, World!</h1>
    <script type="module" src="app.tsx"></script>
  </body>
</html>
```

https://parceljs.org/

INCLUDED TOOLS

Dev server

Parcel includes a development server out of the box. Just run **`parcel index.html`** to get started.



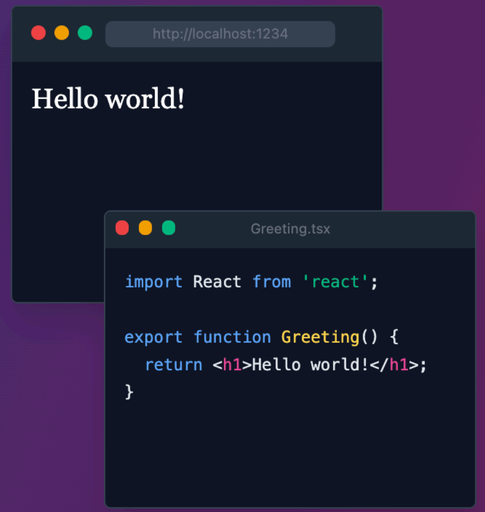
Need HTTPS? Run Parcel with the **`--https`** flag, and it will automatically generate a certificate for you! Or, if you like, you can provide your own.

Parcel also has a built-in API proxy, which can help simulate your production environment.

Learn more →

Hot reloading

When you make a change, Parcel automatically updates your code in the browser – no page reload necessary!



Parcel also integrates with **React Fast Refresh** and the **Vue Hot Reloading API** to automatically preserve your application state between updates. This gives you instant feedback as you make changes, without taking you out of context.

Learn more →

https://parceljs.org/

INCLUDED TOOLS

Diagnostics

If you make an error in your code or configuration, Parcel displays **beautiful diagnostics** in your terminal and in the browser.

Every error includes a syntax highlighted code frame pointing to the exact location where the error occurred, along with hints about how to fix the issue.

Many diagnostics even include a documentation link where you can learn more.

```
$ parcel index.html
Server running at http://localhost:1234
🚨 Build failed.

@parcel/core: Cannot resolve 'ract' from './index.js'
/dev/app/index.js:1:19
> 1 | import React from 'ract';
>   |                   ^^^^^^
  2 |
  3 | function Test() {

@parcel/resolver-default: Cannot find module 'ract'
Did you mean 'react'?
```

https://parceljs.org/

GETTING STARTED

Building a web app with Parcel

parceljs.org/getting-started/webapp/

Incognito

PARCEL

Docs Blog

Search...

GETTING STARTED

- Web app
- Library
- Migration

FEATURES

- Development
- Code Splitting
- Dependency resolution
- Bundle inlining
- Targets
- Production
- Scope hoisting
- Node Emulation
- CLI
- Parcel API
- Plugins

RECIPES

Building a web app with Parcel

Installation

Before we get started, you'll need to install Node and Yarn or npm, and create a directory for your project. Then, install Parcel into your app using Yarn:

```
yarn add --dev parcel
```

Or when using npm run:

```
npm install --save-dev parcel
```

Project setup

Now that Parcel is installed, let's create some source files for our app. Parcel accepts any type of file as an entry point, but an HTML file is a good place to start. Parcel will follow all of your dependencies from there to build your app.

src/index.html:

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8"/>
```

ON THIS PAGE

- Installation
- Project setup
- Package scripts
- Declaring browser targets
- Next steps

GETTING STARTED WITH PARCEL

BACK

1. Install node.js and test versions (`node -v` & `npm -v`).
2. Create a new project folder and open the folder in VS Code.
3. Use terminal inside VS Code and follow the docs: https://parceljs.org/getting-started/webapp/

COMMANDS

Install Parcel in app folder:

```
npm install --save-dev parcel
```

Run/ serve Parcel project:

```
npx parcel src/index.html
```

Or use npm script section in VS Code (see slide)

Open project in browser

Server running at http://localhost:1234

index.html — parcel-dating-spa

EXPLORER

PARCEL-DATING-SPA

- components
- img
- pages
  - create.js
  - update.js
  - user.js
  - users.js
- app.css
- app.js
- index.html
- router.js
- services.js
- package-lock.json
- package.json

NPM SCRIPTS

- package.json
  - start
  - run
  - build

index.html

```html
<!doctype html>
<html lang="en">

<head>
  <meta charset="utf-8">
  <title>User CRUD SPA</title>
  <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
  <meta name="description" content="Get started with HTML, CSS and JavaScript">
  <meta name="author" content="Rasmus Cederdorff">
  <link rel="stylesheet" href="./app.css">
  <link rel="shortcut icon" type="image/png" href="./img/favicon.png" />
</head>

<body>
  <main id="root"></main>
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
> Executing task: npm run start <


> start
> parcel

Server running at http://localhost:1234
✨ Built in 598ms
```

zsh

start Task

main 0 0 1 Ln 15, Col 26 Spaces: 2 UTF-8 LF HTML Go Live Prettier

PARCEL EXERCISE

BACK

Work in your newly created Parcel project and use your knowledge about classes and modules.

1. Inspect the structure and make sure you have the following files inside of src: `index.html`, `styles.css` & `app.js`
2. Run the build command and inspect the list folder. What's inside of the folder and what's the purpose?
3. In the `src` folder create a new module (js file) called `UserService.js`.
4. In `UserService.js` define a class called `UserService` with a `constructor` declaring and empty `users` array (`this.users = [];`).
5. Inside `UserService`, declare a function called `getUsers()` fetching users from https://web-frontend.cederdorff.com/dating-spa-modules/backend/userService.php?action=getUsers, saving users in `this.users` and returning `this.users` in the end of the function.
6. Export the class and import it in your `app.js` module.
7. In `app.js` declare a new instance (new object) of `UserService` and call `getUsers()` on the instance. Test your solution by logging or appending users to the DOM.
8. Remember to test and build your solution by using the commands and/or npm scripts in VS Code.

SPA, FETCH, HTTP METHODS & CLIENT/SERVER

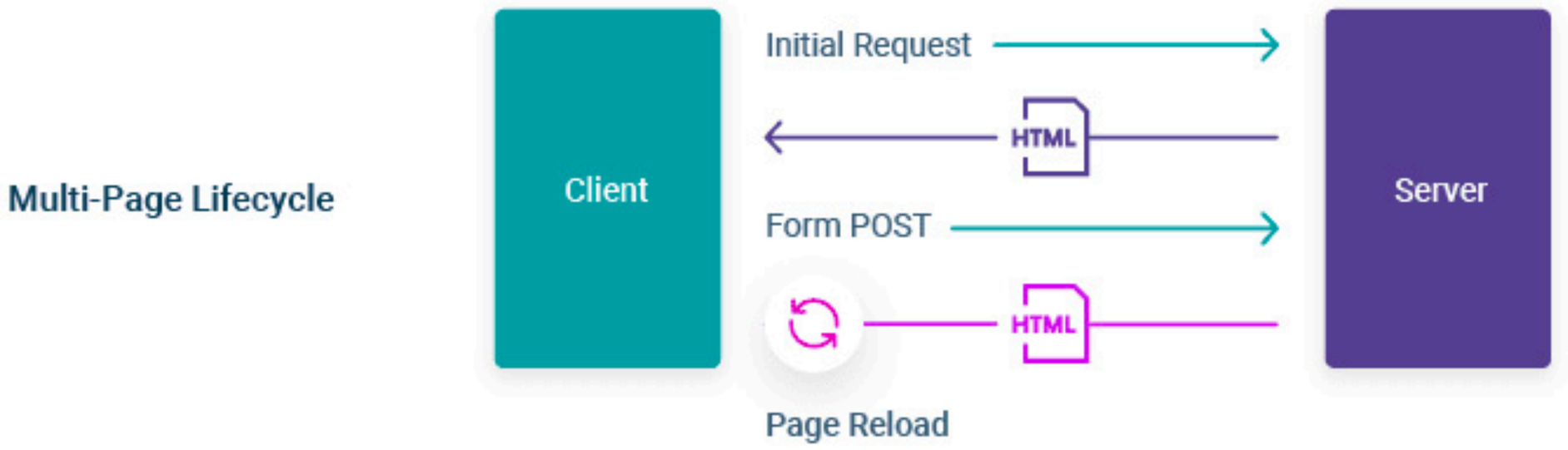
SINGLE PAGE APPLICATION

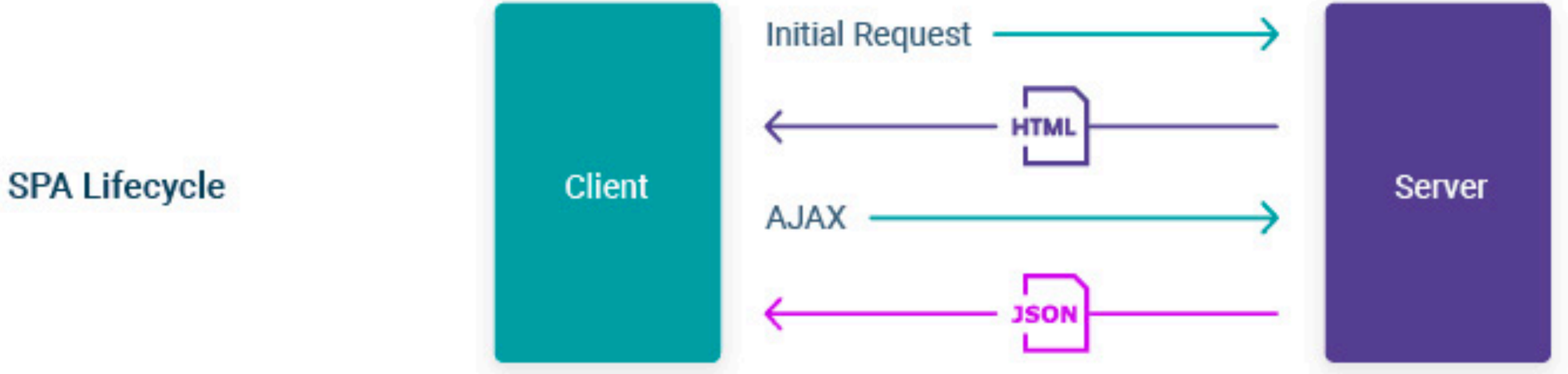
“A single-page application is an application that loads a single HTML page and all the necessary assets (such as JavaScript and CSS) required for the application to run. Any interactions with the page or subsequent pages do not require a round trip to the server which means the page is not reloaded.”

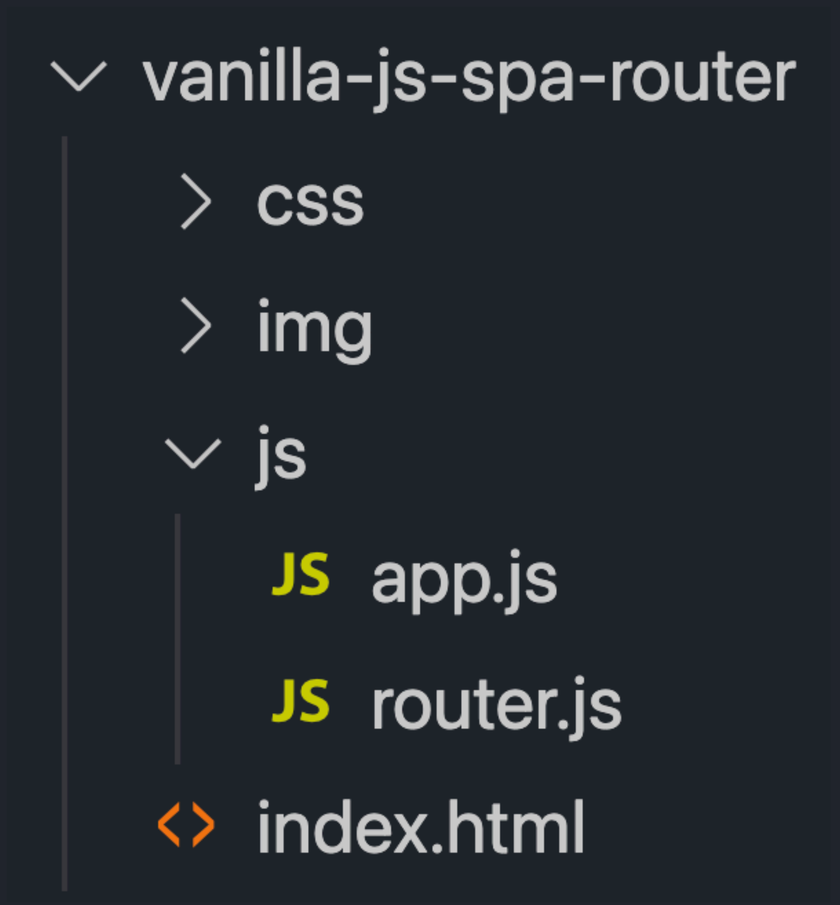
https://reactjs.org/docs/glossary.html#single-page-application

How does Single Page Application Works?

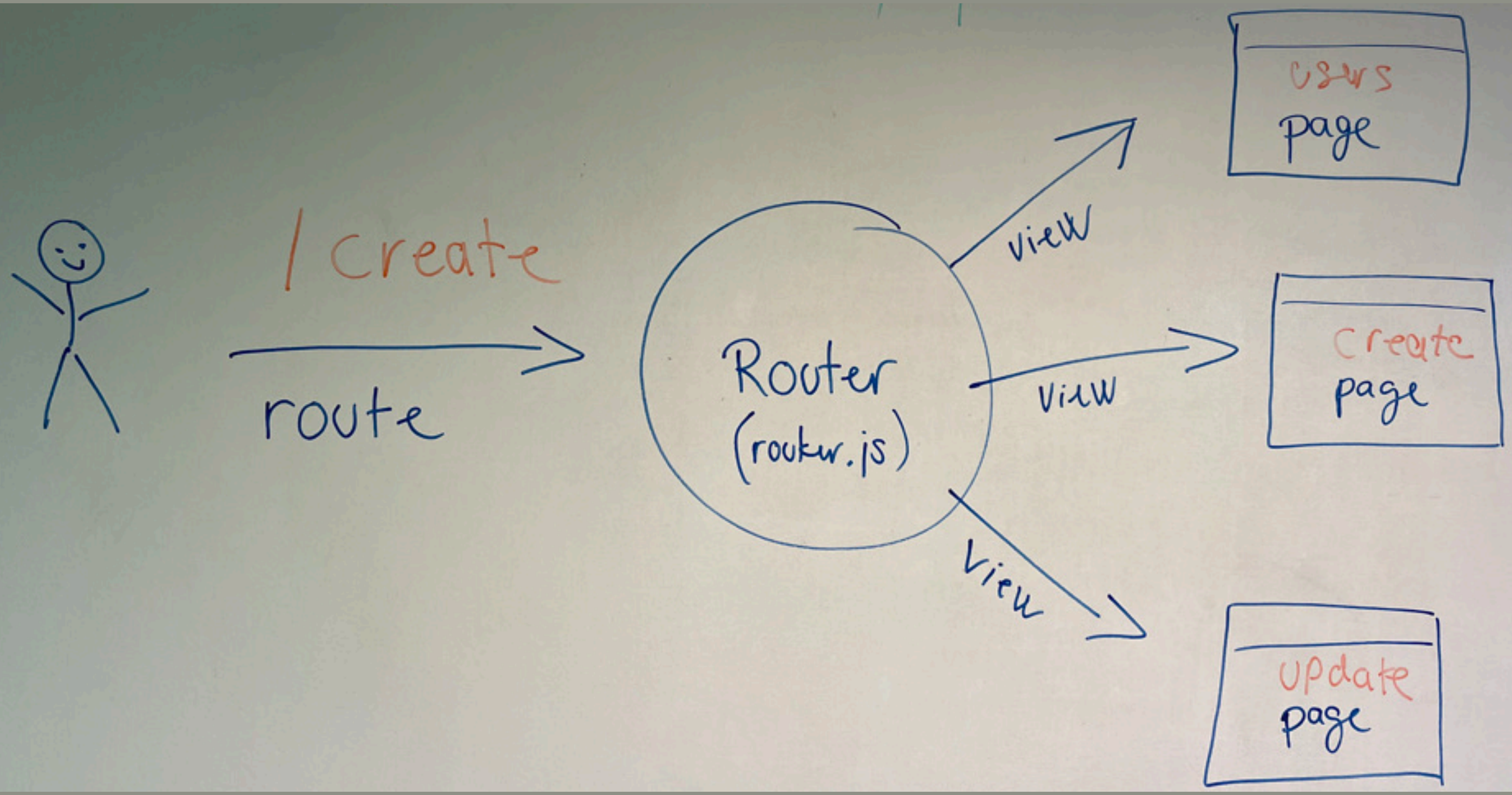
Monocubed


Multi-Page Lifecycle
Client
Initial Request
HTML
Form POST
HTML
Page Reload
Server


SPA Lifecycle
Client
Initial Request
HTML
AJAX
JSON
Server



SIMPLE SPA PROJECT
WITH ROUTER


/create
route
Router
(router.js)
view
view
view
users
page
create
page
update
page

R O U T E R

“It is the piece of software in charge to organize the states of the application, switching between different views.”

It’s a key component in Single Page Applications.

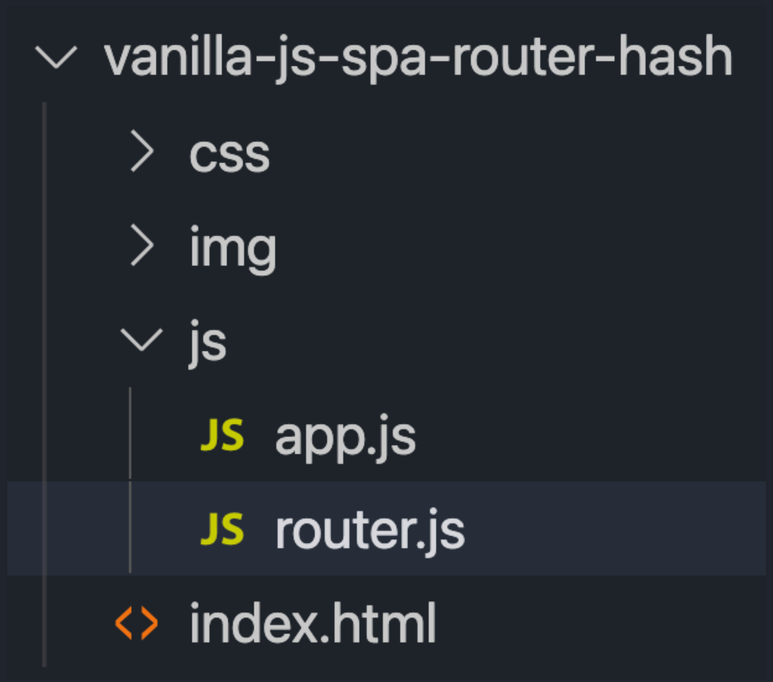
https://medium.com/@fro_g/routing-in-javascript-d552ff4d2921

SPA & ROUTER

In an SPA the router is in charge of displaying the right view by watching changes on the URL.

The router makes it looks like the web app has multiple pages and HTML files. But it's just changing the view, what is displayed, in side of the single `index.html` file. The page transition and change of content, is done dynamically by JavaScript (`router.js`).

https://medium.com/@fro_g/routing-in-javascript-d552ff4d2921




```javascript
 * All routes of the SPA
 * "path": "id of page in DOM"
 */
const _routes = {
    "#/": "home",
    "#/about": "about",
    "#/clients": "clients",
    "#/contact": "contact"
};
const _pages = document.querySelectorAll(".page");
const _basePath = location.pathname.replace("index.html", ""); // remove
const _navLinks = document.querySelectorAll("nav a");

/**
 * Changing display to none for all pages
 */
function hideAllPages() {
    for (const page of _pages) {
        page.style.display = "none";
    }
}

/**
 * Navigating SPA to specific page by given path
 */
function navigateTo(path) {
    window.history.pushState({}, path, _basePath + path);
    showPage(path);
}

/**
 * Displaying page by given path
 */
```

vanilla-js-spa-router-hash

route

```
const _routes = {
    "#/": "home",
    "#/about": "about",
    "#/clients": "clients",
    "#/contact": "contact"
};
```

id of view

```
<body>
  <!-------------- Navigation -------------->
  <nav>
    <a class="nav-link" href="#/">Home</a>
    <a class="nav-link" href="#/about">About</a>
    <a class="nav-link" href="#/clients">Clients</a>
    <a class="nav-link" href="#/contact">Contact</a>
  </nav>

  <main id="appRoot">
    <!-------------- Pages -------------->
    <!-- home page -->
    <section id="home" class="page">…
    </section>

    <!-- about page -->
    <section id="about" class="page">…
    </section>

    <!-- clients page -->
    <section id="clients" class="page">…
    </section>

    <!-- clients page -->
    <section id="contact" class="page">…
    </section>

  </main>
```

change default event for all `.nav-link` items

```html
<!-------------- Navigation -------------->
<nav>
  <a class="nav-link" href="#/">Home</a>
  <a class="nav-link" href="#/about">About</a>
  <a class="nav-link" href="#/clients">Clients</a>
  <a class="nav-link" href="#/contact">Contact</a>
</nav>
```

```javascript
/**
 * Attaching event to nav links and preventing default anchor
 */
function attachNavLinkEvents() {
    const navLinks = document.querySelectorAll(".nav-link");
    for (const link of navLinks) {
        link.addEventListener("click", function (event) {
            const path = link.getAttribute("href");
            navigateTo(path);
            event.preventDefault();
        });
    }
}
```

when a `.nav-link` item is clicked,
navigateTo(...) is called with the value of `href`

```html
<!-------------- Navigation -------------->
<nav>
  <a class="nav-link" href="#/">Home</a>
  <a class="nav-link" href="#/about">About</a>
  <a class="nav-link" href="#/clients">Clients</a>
  <a class="nav-link" href="#/contact">Contact</a>
</nav>
```

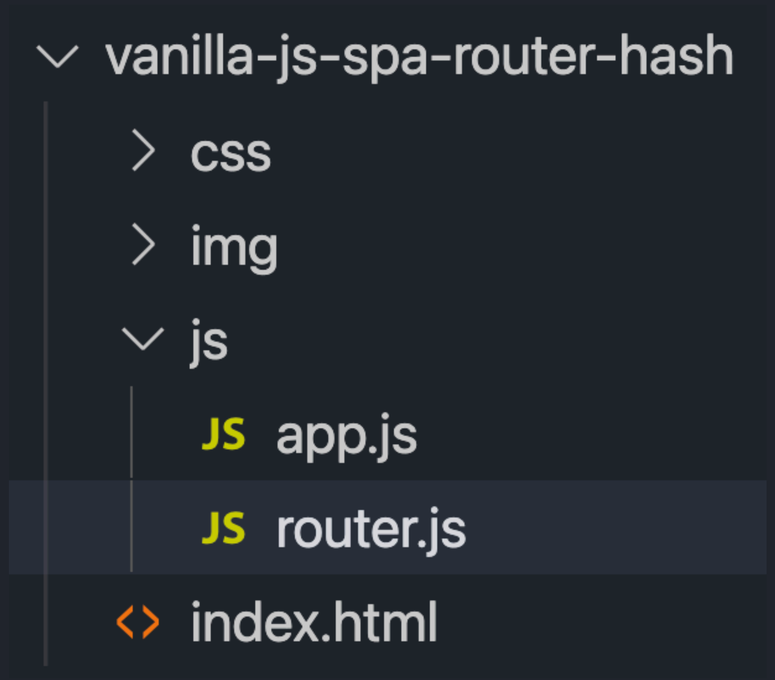
```javascript
/**
 * Navigating SPA to specific page by given path
 */
function navigateTo(path) {
    window.history.pushState({}, path, _basePath + path);
    showPage(path);
}

/**
 * Displaying page by given path
 */
function showPage(path) {
    hideAllPages(); // hide all pages
    document.querySelector(`#${_routes[path]}`).style.display = "block"; /
    setActiveTab(path);
}
```

SPA & ROUTER EXERCISE

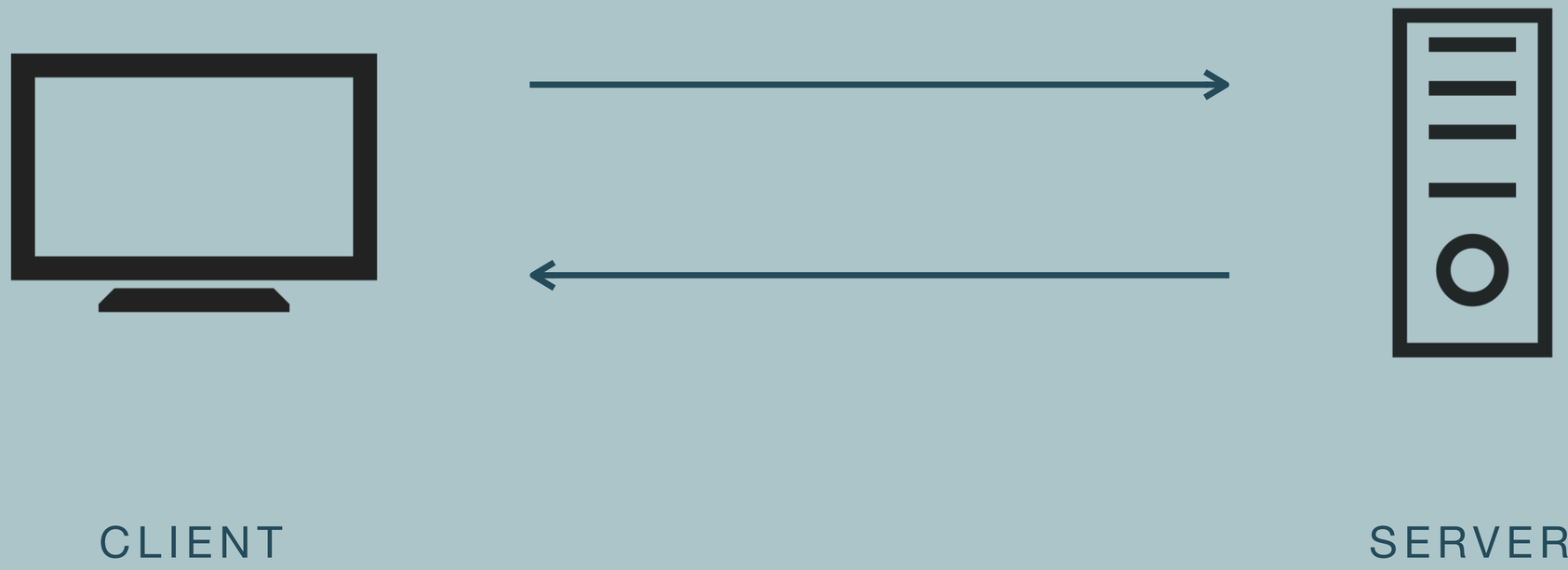
BACK

- Inspect the `index.html` & `router.js` in `vanilla-js-spa-router-hash`
- Add another page called users.
- Remember to add a new section for users page, a nav-link and the route in `router.js`.
- Test your are able to navigate to the users page.
- Change the clients page to a posts page. Remember to update id's, href & routes.
- Test your solution.
- Explain `navigateTo()`, `showPage()` & `initRouter()`



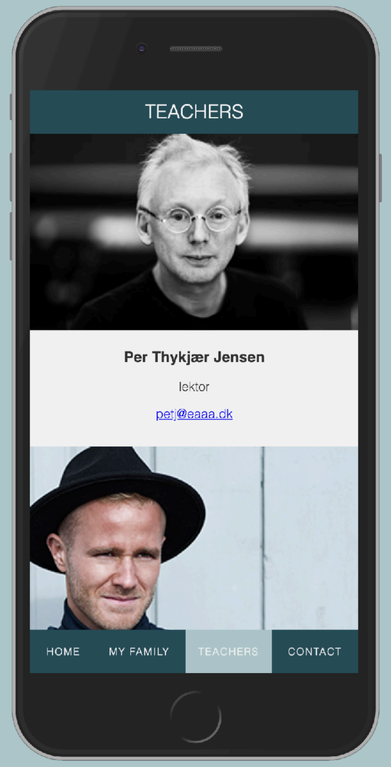
CLIENT SERVER MODEL

... how to communicate with backend (server)



WEB DEVELOPMENT

FRONTEND



API

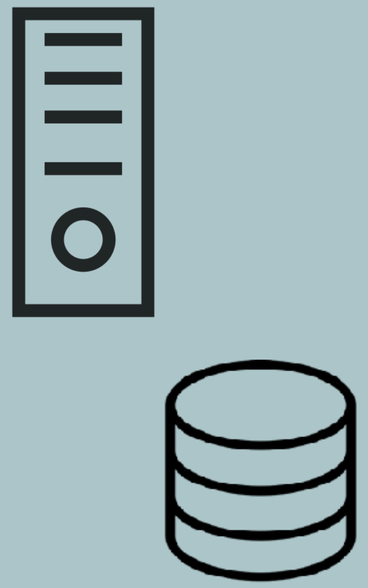
FETCH(...)

REQUEST

DATA

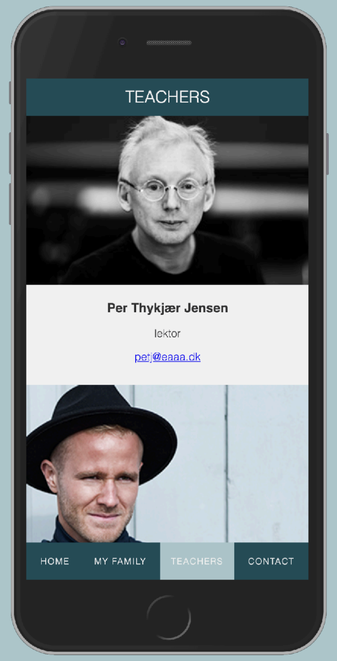
RESPONSE

BACKEND



INTERDISCIPLINARY PROJECT

FRONTEND



HTML JS CSS

(API)

FETCH(...)

REQUEST

JSON DATA

RESPONSE

JSON

BACKEND

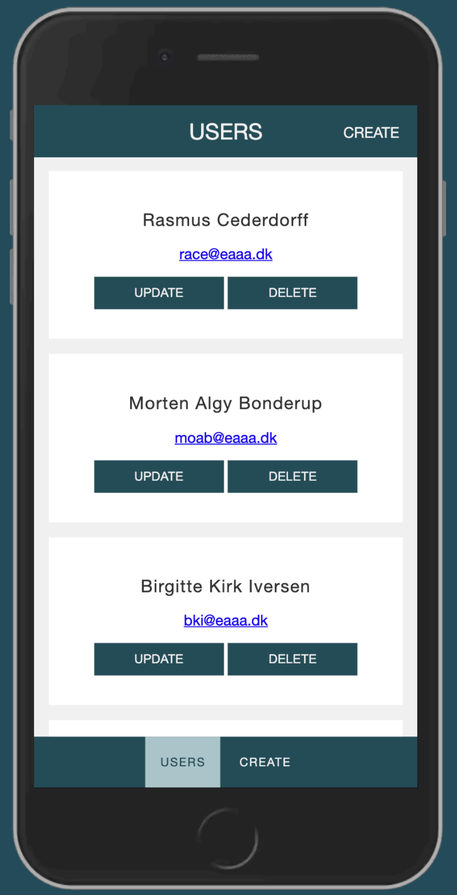
php

SQL

MySQL

FETCH, HTTP REQUEST & RESPONSE

WEB APP



API

FETCH(... GET OR PUT ...)

REQUEST

JSON DATA

RESPONSE

DATA SOURCE

FETCH: HEADERS, BODY & HTTP METHODS

HTTP REQUEST METHODS

GET - POST - PUT - DELETE

HTTP (Hypertext Transfer Protocol) is the standard way to communicate between clients and servers (request-response protocol).

"HTTP defines a set of **request methods** to indicate the desired action to be performed for a given resource.”

https://www.w3schools.com/tags/ref_httpmethods.asp
https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods

REQUEST headers

“[…] contain more information about the resource to be fetched, or about the client requesting the resource.”

"A request header is an HTTP header that can be used in an HTTP request to provide information about the request context, so that the server can tailor the response. For example, the Accept-* headers indicate the allowed and preferred formats of the response. Other headers can be used to supply authentication credentials (e.g. Authorization), to control caching, or to get information about the user agent or referrer, etc.”

https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers
https://developer.mozilla.org/en-US/docs/Glossary/Request_header

REQUEST `body`

When making HTTP request we sometimes need to send data. The data is wrapped inside of the request `body`.

The request `body` is one of the following:

a string (often JSON encoded string with data, object, arrays, etc.)

form data (`form/multipart`)

blob/ buffer source - binary data

URL search params (`x-www-form-urlencoded`)

https://javascript.info/fetch#post-requests

FETCH WITHOUT HEADERS & BODY

THE DEFAULT HTTP METHODS IS “GET”

```
async getMatches(userId) {
    const url = `${this.baseUrl}?action=getMatches&userId=${userId}`;
    const response = await fetch(url);
    const data = await response.json();
    return data;
}
```

FETCH WITH BODY

... AND HTTP METHOD "POST"

```
const response = await fetch(this.baseUrl + "?action=createUser", {
    method: "POST",
    body: JSON.stringify(newUser), // parsing js object to json object
});
// waiting for the result
const result = await response.json();
```

FETCH WITH HEADERS & BODY

... AND HTTP METHOD "POST"

```
const user = {
  name: "John",
  age: 31
};

const response = await fetch('/article/fetch/post/user', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json;charset=utf-8'
  },
  body: JSON.stringify(user)
});

const result = await response.json();
```

FETCH WITH BODY

... AND HTTP METHOD "PUT"

```
const response = await fetch(this.baseUrl + "?action=updateUser", {
    method: "PUT",
    body: JSON.stringify(userToUpdate), // parsing js object to json object
});
// waiting for the result
const result = await response.json();
```

FETCH

... AND HTTP METHOD "DELETE"

```
async deleteUser(userId) {
    const response = await fetch(`${this.baseUrl}?action=deleteUser&userId=${userId}`, {
        method: "DELETE",
    });
    // waiting for the result
    const result = await response.json();
    // the result is the new updated users array
    this.users = result;
    return this.users;
}
```

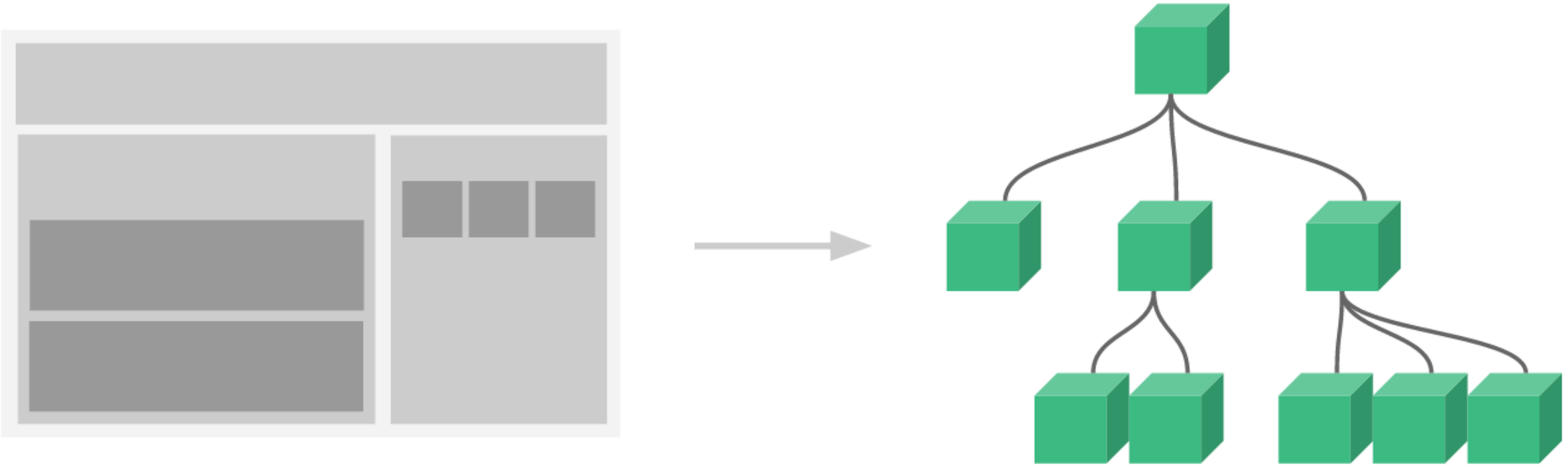
OBJECT-ORIENTED JS, MODULES & COMPONENTS

OUR CODE IS STRUCTURED IN OBJECTS & CLASSES

MODULES & COMPONENTS

BUILD REUSABLE MODULES & COMPONENTS

COMPONENTS



FRAMEWORKS & LIBRARIES

Object-oriented approach

(or component-based approach)

MODULES

A MODULE IS JUST A FILE (OR A SCRIPT).
ONE SCRIPT IS ONE MODULE.

As our application grows bigger, we want to split it into multiple files, called “modules”. A module may contain a class or a library of functions for a specific purpose.

Modules can load each other and use special directives export and import to interchange functionality, call functions of one module from another one ...

EXPORT => IMPORT

```
// 📂 user.js
export default class User { // just add "default"
  constructor(name) {
    this.name = name;
  }
}
```

A MODULE (SCRIPT)

```
// 📂 main.js
import User from './user.js'; // not {User}, just User

new User('John');
```

ANOTHER MODULE (SCRIPT)

user.js — mdu-frontend

main.js

```javascript
import User from "./user.js";

const users = [
  new User("Birgitte", "bki@mail.dk", "1966-01-14", "https://www.eaa
  new User("Martin", "mnor@mail.dk", "1989-05-02", "https://media-ex
  new User("Rasmus", "race@mail.dk", "1990-09-15", "https://www.eaaa
];

console.log(users);

for (const user of users) {
  document.querySelector("#content").innerHTML += user.getHtmlTempla
}
```

user.js

```javascript
export default class User {
  constructor(name, mail, birthDate, img) {
    this.name = name;
    this.mail = mail;
    this.birthDate = birthDate;
    this.img = img;
  }

  log() {
    console.log(`
      Name: ${this.name},
      Mail: ${this.mail},
      Birth date: ${this.birthDate},
      Image Url: ${this.img}
    `);
  }

  getAge() {
    const birthDate = new Date(this.birthDate);
    const today = new Date();
    const diff = new Date(today - birthDate);
    return diff.getFullYear() - 1970;
  }

  getHtmlTemplate() {
    const template = /*html*/`
      <article>
        <img src="${this.img}">
        <h2>${this.name}</h2>
        <a href="mailto:${this.mail}">${this.mail}</a>
        <p>Birth date: ${this.birthDate}</p>
        <p>Age: ${this.getAge()} years old</p>
      </article>
    `;
```

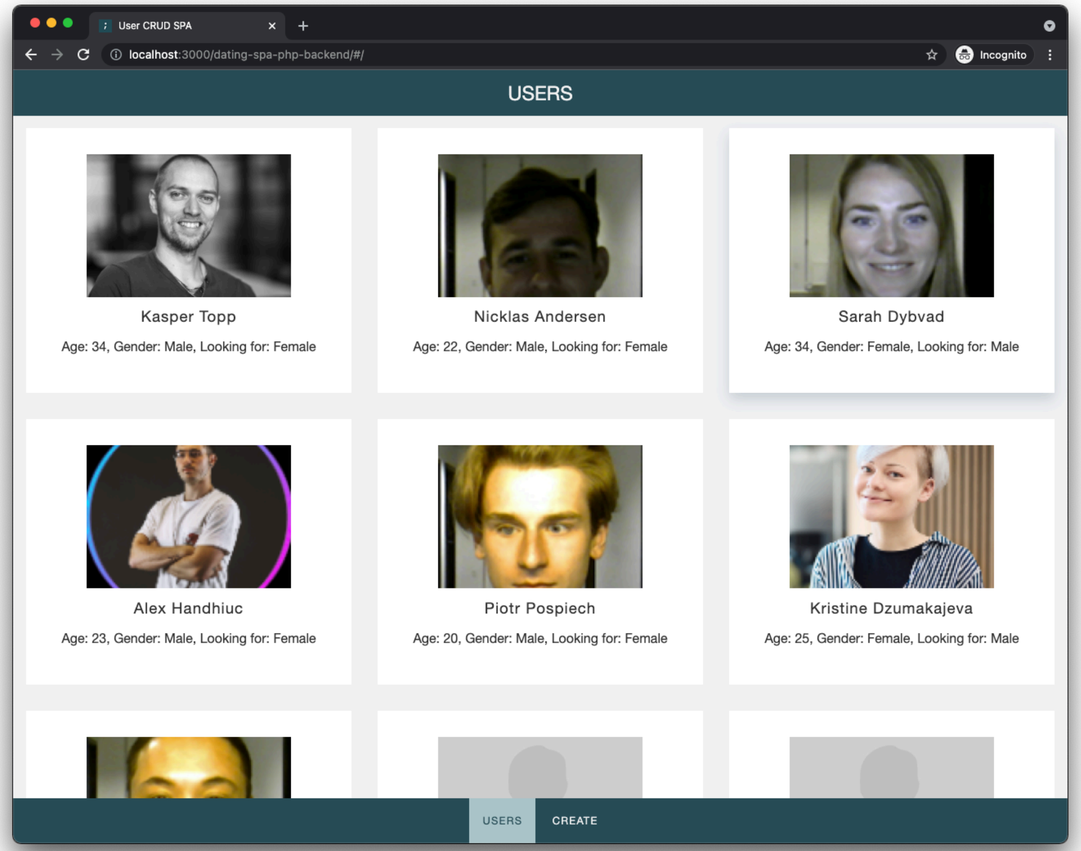
main ⊗ 0 ⚠ 0 Ln 40, Col 1 Spaces: 2 UTF-8 LF JavaScript Port : 5502 Prettier

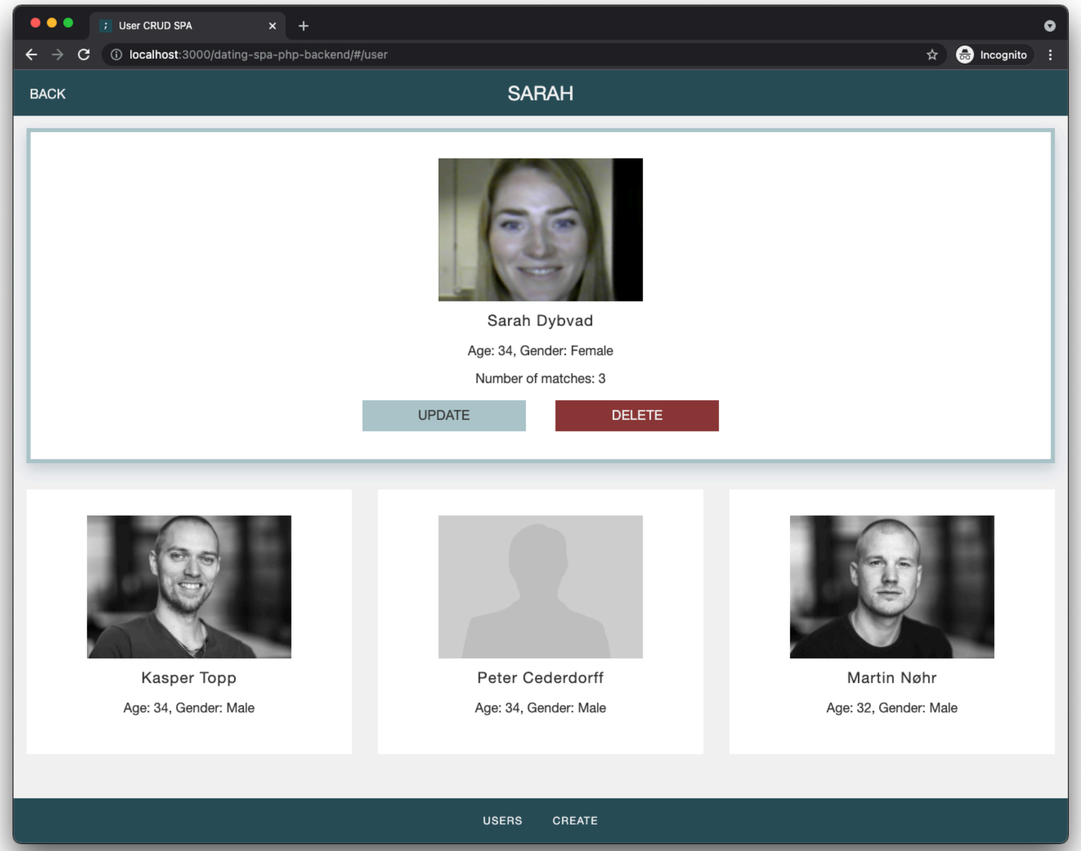
TELL THE BROWSER IT'S A MODULE

```
<script src="js/main.js" type="module"></script>
```

DATING SPA PHP BACKEND

BACK





Solution: `dating-spa-php-backend`

dating-spa-jsonbin
css
img
js
main.js
router.js
index.html
src
components
loader.js
nav.js
img
pages
create.js
profile.js
update.js
users.js
app.css
app.js
index.html
router.js
services.js

router.js — parcel-dating-spa

EXPLORER

PARCEL-DATING-SPA

- .parcel-cache
- dist
- node_modules
- src
  - components
    - loader.js
    - nav.js
  - img
  - pages
    - create.js
    - profile.js
    - update.js
    - users.js
  - app.css
  - app.js
  - index.html
  - router.js
  - services.js
- package-lock.json
- package.json

NPM SCRIPTS

router.js

```javascript
import CreatePage from "./pages/create.js";
import UpdatePage from "./pages/update.js";
import UserProfilePage from "./pages/profile.js";
import UsersPage from "./pages/users.js";

class Router {
    constructor() {
        this.routes = [
            {
                path: "/",
                view: new UsersPage("users"),
            },
            {
                path: "/create",
                view: new CreatePage("create"),
            },
            {
                path: "/update/:id",
                view: new UpdatePage("update"),
            },
            {
                path: "/user/:id",
                view: new UserProfilePage("profile"),
            },
        ];

        this.pages;
        this.navLinks;
    }

    /**
     * Changing display to none for all pages
     */
    hideAllPages() {
        for (const page of this.pages) {
```

main | 0 0 | 1 | Ln 24, Col 15 | Tab Size: 4 | UTF-8 | LF | JavaScript | Go Live | Prettier

IT’S ALL ABOUT

STRUCTURE

PHP BACKEND SERVICE

Instead of using JSONBIN, implement a PHP Backend Service reading and writing to a JSON file or a MySQL Database. The PHP Backend Service must include the following functionality:

- Get and read all data as JSON (`GET`)
- Create new objects, saved and stored in JSON file or database. The new objects must be posted from the frontend as JSON (`POST`). Remember to generate `id` for new objects.
- Update existing object by the `id`. Use `PUT` to put the updates to the PHP Backend service and save updates in JSON file or database.
- Delete object by the `id`. Use `DELETE` to delete objects and save changes in JSON file or database.
- Extra: Implement upload of images as file instead of `URL`.

Inspiration: `users-match-frontend`, `users-match-backend` and `file-upload`

Solution: `dating-spa-php-backend`

PHP BACKEND SERVICE

Instead of using JSONBIN, implement a PHP Backend Service reading and writing to a JSON file or a MySQL Database. The PHP Backend Service must include the following functionality:

- Get and read all data as JSON (`GET`)
- Create new objects, saved and stored in JSON file or database. The new objects must be posted from the frontend as JSON (`POST`). Remember to generate `id` for new objects.
- Update existing objects by the `id`. Use `PUT` to put the updates to the PHP Backend service and save updates in JSON file or database.
- Delete object by the `id`. Use `DELETE` to delete objects and save changes in JSON file or database.
- Extra: Implement upload of images as file instead of `URL`.

Inspiration: `users-match-frontend`, `users-match-backend` and `file-upload`

Solution: `dating-spa-php-backend`

AS FRONTEND DEVELOPER

WE DON'T GIVE A F***

when you ask Rasmus
for help and he says
"Read documentation"

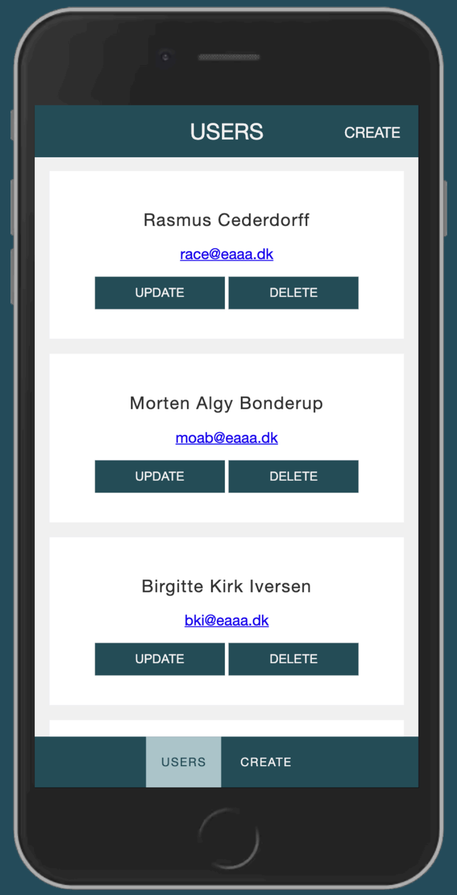
ALL WE NEED IS A DOCUMENTATION

HOW TO INTERACT (COMMUNICATE) WITH THE BACKEND SERVICE

PHP User Service: https://race.notion.site/The-Docs-PHP-User-Service-55835841105f4ed79aa36c2f052a65d5

FETCH FROM PHP BACKEND SERVICE

DATING SPA



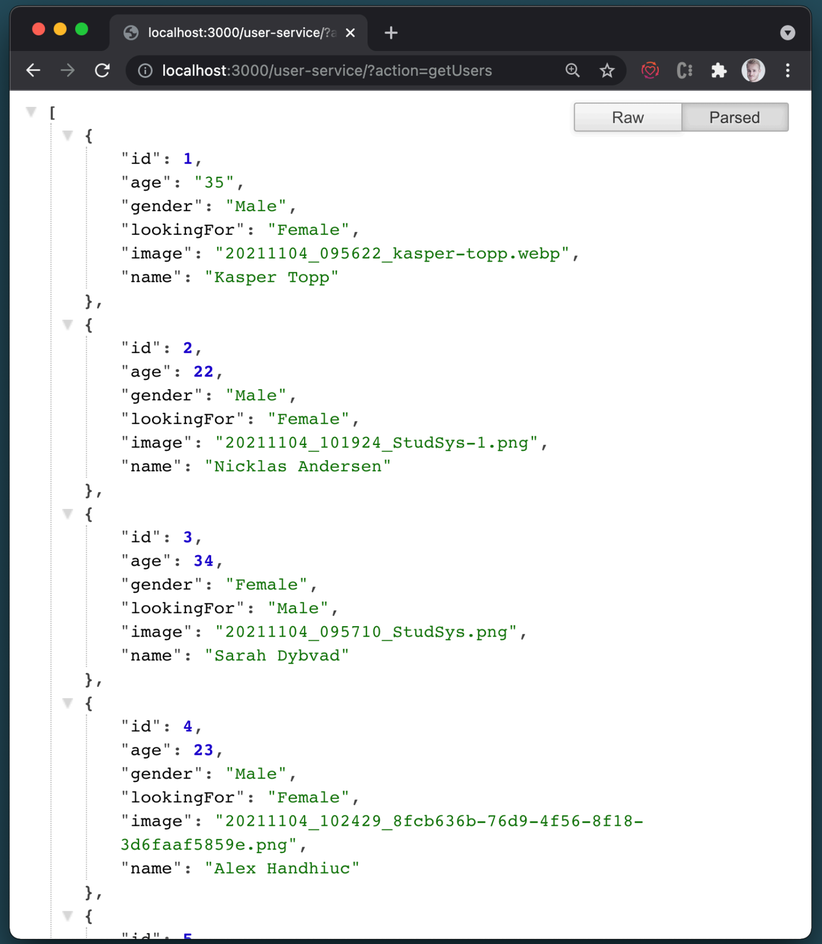
FETCH

REQUEST

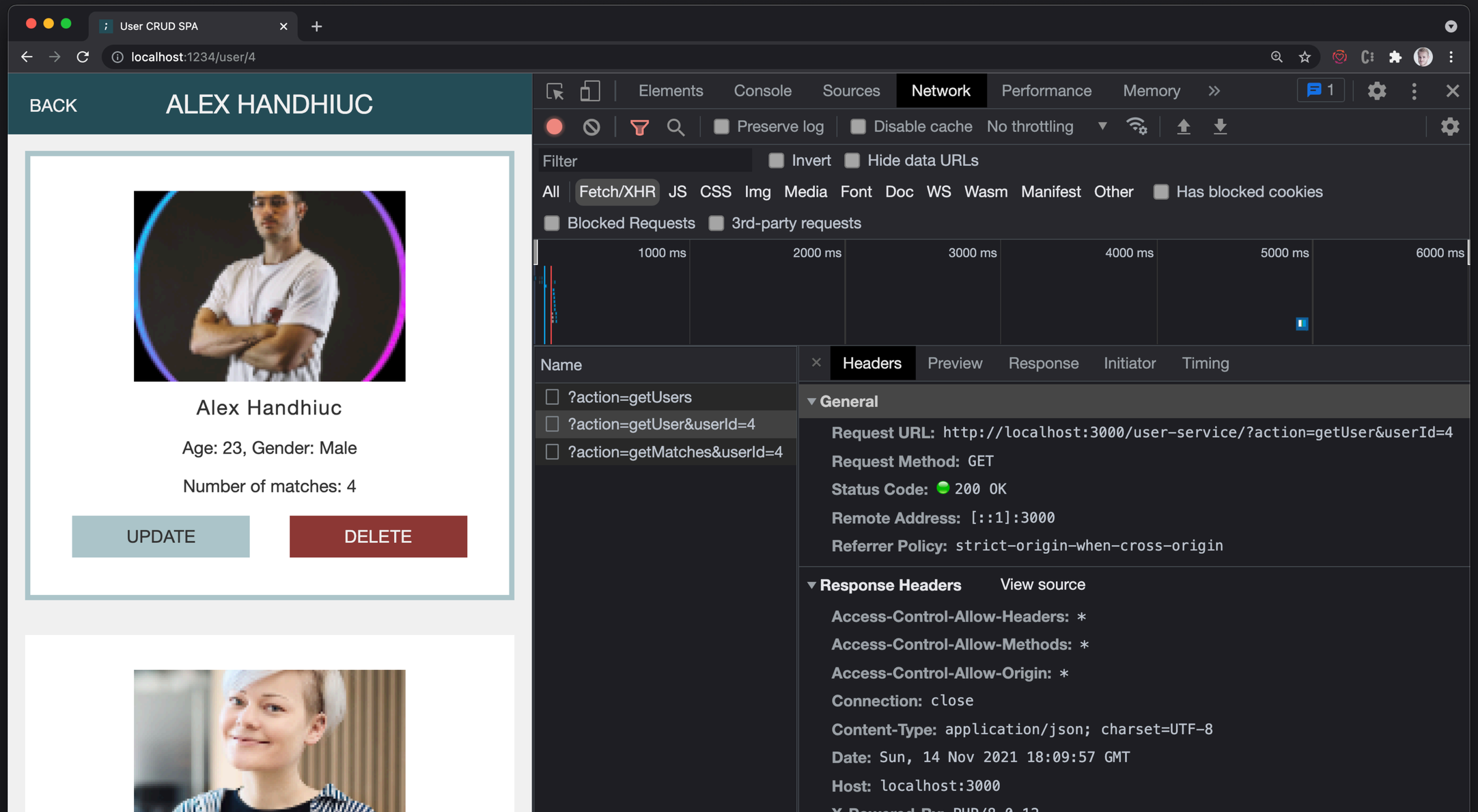
JSON DATA

RESPONSE

USER SERVICE



INSPECT FETCH CALLS



DATING SPA USING CLASSES & MODULES #1

BACK

1. Create a new project folder, initialize and setup with Parcel.
2. Create a src folder and defined an initial project structure (inspiration here & `parcel-dating-spa-template`). See page structure here.
3. Import nav.js and router.js in app.js - render the nav and init the router.
4. Run the backend `user-service` locally with PHP server (make sure to pull latest from GitHub). Test the service: http://localhost:3000/user-service/?action=getUsers. Your PHP Server might run on another port/url.
5. Implement service functions in services.js. The service is going to be imported and used across pages.
6. Implement users.js (Users Page): read & display all users.

DATING SPA USING CLASSES & MODULES #2

BACK

7. Implement create.js (Create Page): create a new user.
8. Implement profile.js (User Profile Page): display user info & all matches.
9. Implement update.js: update user details using the service.
10. Implement delete functionality using the service.
11. Implement the loader.js component and make sure to display loader when needed. The loader should be imported and used by the page modules.
12. Implement improvements and extra functionality like search, filters, etc.

PAGES

```
export default class Page {
    constructor(id) {…
    }

    render() {…
    }

    attachEvents() {…
    }

    beforeShow(props) {…
    }
}
```

Initialising page with need properties. Required: `id`

Render the initial page template

Attach events need for the page

Execute functions we need before the page is displayed

IMPORT NAV.JS AND ROUTER.JS IN APP.JS

BACK

- nav.js and router.js are already defined.
- You need to import the modules in app.js, render the nav and init the router.
- In the router.js you must define your routes with a path and a view property.
- The view property is a new instance of the page connected to a given path.
- As you define your Page modules and implement the structure of your pages, go back to the router an implement the route for each page.
- The value you add when declaring a new page instance will be the id of the page.
- NB: please don't overthink the router.js. I don't expect you to understand every single line. All you need to know is how to add new routes.
- navigateTo and props: when calling navigateTo you will be able to pass properties from one page (component) to another. navigateTo is executing beforeShow inside the page, right before the page is displayed.

```
//app.js
import nav from "./components/nav.js";
import router from "./router.js";

nav.render();
router.init();
```

```
//router.js
import CreatePage from "./pages/create.js";
import UpdatePage from "./pages/update.js";
import UserProfilePage from "./pages/profile.js";
import UsersPage from "./pages/users.js";

class Router {
    constructor() {
        this.routes = [
            {
                path: "/",
                view: new UsersPage("users"),
            },
            {
                path: "/create",
                view: new CreatePage("create"),
            },
            {
                path: "/update/:id",
                view: new UpdatePage("update"),
            },
            {
                path: "/user/:id",
                view: new UserProfilePage("profile")
```

IMPLEMENT SERVICE FUNCTIONS IN SERVICES.JS

BACK

- Implement the `Service Class` with needed functions.
- Make sure to point to the backend service (`this.baseUrl = "http://localhost:3000/user-service/"`).
- Start with `getUsers()` and implement the Users Page.
- Go back and implement service functions when needed (when implementing other pages).
- Declare an instance of the `Service Class` and `export` the instance (object). By that we will be able to import the service object in other modules and use the functionality across all pages.

```
class Service {
    constructor() {
        this.users = [];
        this.baseUrl = "http://localhost:3000/user-service/";
        this.selectedUserId;
    }

    async getUsers() {…
    }

    async getMatches(userId) {…
    }

    async getUser(userId) {…
    }

    async uploadImage(imageFile) {…
    }

    async deleteUser(userId) {…
    }

    async createUser(name, age, gender, lookingFor, image) {…
    }

    async updateUser(id, name, age, gender, lookingFor, image) {…
    }
}

const service = new Service();
export default service;
```

IMPLEMENT USERS.JS

BACK

- All pages must have the functions `render()`, `attachEvents()` & `beforeShow()` and the property `id`.
- Pages can have other helper functions like `init()`, `append(…)`, etc. A page can import needed functionality from other modules like `service`, `router` and `loader`.
- `render()` is rendering the initial template of the page and appending to the root element in the DOM.
- `appendUsers(users)` appends all users in to the `#grid-users` section.
- `appendUsers(users)` is called by `init()`.
- Before `init()` is executing `appendUsers(users)` you must make sure to call the service to get all users.
- `attachEvents()` is attaching events like `onclick`. Add an `onclick` for every user navigating to user profile page.
- Extra: implement loader showing a spinner when loading and appending users. The loader functionality must be imported from `loader` module. `loader` module is describe here.

```
export default class UsersPage {
    constructor(id) {
        this.id = id;
        this.render();
        this.init();
    }

    render() {
        document.querySelector("#root").insertAdjacentHTML(
            "beforeend",
            /*html*/ `
            <section id="${this.id}" class="page">
                <header class="topbar">
                    <h2>Users</h2>
                </header>
                <section class="users-grid grid-container"></section>
            </section>
        `
        );
    }

    appendUsers(users) {…
    }

    async init() {
        loader.show();
        const users = await service.getUsers();
        this.appendUsers(users);
        loader.hide();
    }

    attachEvents() {…
    }

    beforeShow(props) {
        if (props.users) {
            this.appendUsers(props.users);
        }
    }
}
```

IMPLEMENT CREATE.JS

BACK

- To left there's a suggested structure of `create.js`.
- In the `constructor` you are able to declare properties with reference to each input field. The input fields must be rendered by `render()` before adding references.
- Attach need events in `attachEvents()`: on change for image preview and `onclick` for save button.
- The save button executes `create()`. `create()` is using the service to upload the new image and create the new user. It is interacting with the backend service.
- The response from `service.createUser` is a new updated array of users. The array must be passed along with `router.navigateTo("/", {users: newUsersArray});`
  - By that the updated users array is passed to `users.js`
  - Make sure `beforeShow(…)` in `users.js` takes care of `props.users` in order to update the DOM.
- Extra: implement form validation, checking the all inout fields before upload image and create.
- Extra: reset all inputs after successfully creating a user.

```
import loader from "../components/loader.js";
import router from "../router.js";
import services from "../services.js";

export default class CreatePage {
    constructor(id) {
        this.id = id;
        this.render();

        this.nameInput = document.querySelector(`#${this.id} [name="name"]`);
        this.ageInput = document.querySelector(`#${this.id} [name="age"]`);
        this.genderInput = document.querySelector(`#${this.id} [name="gender"]`);
        this.lookingForInput = document.querySelector(`#${this.id} [name="lookingFor"]`);
        this.imagePreview = document.querySelector(`#${this.id} [name="imagePreview"]`);
        this.imageInput = document.querySelector(`#${this.id} [name="profileImage"]`);

        this.attachEvents();
    }

    render() {…
    }

    attachEvents() {
        this.imageInput.onchange = () => this.previewImage();
        document.querySelector(`#${this.id} .save`).onclick = () => this.create();
    }

    previewImage() {…
    }

    async create() {…
    }

    validate() {…
    }
```

IMPLEMENT PROFILE.JS

BACK

- Start by implementing `render()`
- Make sure you have the needed functions in `service`: functions to get user data and matches. Then use the `service` to get the data in user profile page.
- Append user data and matches.
- User data, matches and append should be called by `beforeShow(…)`.
- `beforeShow(…)` is executed right before the page is displayed.
- In `beforeShow(…)` we are able to access `props` containing properties from the router (ex user id).
  - With `router.navigateTo(…)`, `props` and `page.beforeShow(…)` we are able to pass properties from one component to another.
- `showDeleteDialog()` is described here.

```
export default class UserProfilePage {
    constructor(id) {
        this.id = id;
        this.selectedUser;
        this.matches;
        this.render();
    }

    render() {…
    }

    appendUserData() {…
    }

    attachEvents() {…
    }

    async showDeleteDialog() {…
    }

    async beforeShow(props) {
        loader.show();
        this.selectedUser = await service.getUser(props.id);
        this.matches = await service.getMatches(props.id);
        this.appendUserData();
        loader.hide();
    }
}
```

IMPLEMENT UPDATE.JS

BACK

- Required functions: `render()`, `attachEvents()` & `beforeShow(props)`.
- Suggested functions: `previewImage()` to update and display image preview, `save()` saving the user using `service` & `validate()` validating input fields before save.
- In the `constructor` it's convenient to declare inputs as properties (reference to inputs, defined in `render()`).
- `attachEvents()` is attaching `onclick` events to cancel and save button. The buttons are defined as html inside of `render()`.
- In `beforeShow(…)` we are able to access `props.id` (the `id` of the user we want to update). By the `id` we are able to get user data from our `service` and then set input field values.

```
export default class UpdatePage {
    constructor(id) {
        this.id = id;
        this.render();
        this.selectedUser;

        this.nameInput = document.querySelector(`#${this.id} [name="name"]`);
        this.ageInput = document.querySelector(`#${this.id} [name="age"]`);
        this.genderInput = document.querySelector(`#${this.id} [name="gender"]`);
        this.lookingForInput = document.querySelector(`#${this.id} [name="lookingFor"]`);
        this.imagePreview = document.querySelector(`#${this.id} [name="imagePreview"]`);
        this.imageInput = document.querySelector(`#${this.id} [name="profileImage"]`);

        this.attachEvents();
    }

    render() {…
    }

    attachEvents() {
        document.querySelector(`#${this.id} .cancel`).onclick = () => router.goBack();
        document.querySelector(`#${this.id} .save`).onclick = () => this.save();
        this.imageInput.onchange = () => this.previewImage();
    }

    previewImage() {…
    }

    async save() {…
    }

    validate() {…
    }

    async beforeShow(props) {
        loader.show();
        const user = await service.getUser(props.id);
        this.selectedUser = user;

        this.nameInput.value = this.selectedUser.name;
        this.ageInput.value = this.selectedUser.age;
        this.genderInput.value = this.selectedUser.gender;
        this.lookingForInput.value = this.selectedUser.lookingFor;
        this.imagePreview.src = `${service.baseUrl}/files/medium/${
            this.selectedUser.image || "placeholder.jpg"
        }`;
        this.imageInput.value = ""; // reset value
        loader.hide();
    }
```

IMPLEMENT DELETE

BACK

- Delete could be implemented as a part of the user profile page.
- `attachEvents()` attach an `onclick` to the delete button: `showDeleteDialog()`.
- `showDeleteDialog()` show an confirm dialog and uses `service` to delete user by given user `id`.
- `service.deleteUser()` returns an updated `users` array and could be passed back to users page through `router.navigateTo(…)`.
  - Make sure `beforeShow(…)` in `users.js` takes care of `props.users` in order to update the DOM.

```
export default class UserProfilePage {
    constructor(id) {
        this.id = id;
        this.selectedUser;
        this.matches;
        this.render();
    }

    render() {…
    }

    appendUserData() {…
    }

    attachEvents() {
        document.querySelector(`#${this.id} .back`).onclick = () => router.navigateTo("/");

        document.querySelectorAll(`#${this.id} [data-user-id]`).forEach(element => {…
        });

        document.querySelector(`#${this.id} .update`).onclick = () =>
            router.navigateTo(`/update/${this.selectedUser.id}`);

        document.querySelector(`#${this.id} .delete`).onclick = () => this.showDeleteDialog();
    }

    async showDeleteDialog() {
        const deleteUser = confirm("Do you want to delete user?");

        if (deleteUser) {
            loader.show();
            const users = await service.deleteUser(this.selectedUser.id);
            router.navigateTo("/", {
                users: users,
            });
            loader.hide();
        }
    }

    async beforeShow(props) {…
    }
}
```

LOADER.JS

BACK

- The `loader` is yet another component and can be imported by other components (pages).
- In the end of the `loader` module declare a new instance of the `Loader` class and `export` the instance.

```
class Loader {
    constructor() {
        this.render();
        this.loader = document.querySelector(".loader");
    }

    render() {
        document.querySelector("#root").insertAdjacentHTML(
            "beforeend",
            /*html*/ `
            <section class="loader">
                <section class="spinner"></section>
            </section>
            `
        );
    }

    show() {
        this.loader.classList.remove("hide");
    }

    hide() {
        this.loader.classList.add("hide");
    }
}

const loader = new Loader();
export default loader;
```

NEXT UP

RECAP ON (eaaa.padlet.org/race/frontend_programming)?

AUTHENTICATION, SIGN UP & LOGIN

INTERDISCIPLINARY PROJECT & EXAM

FULLSTACK JS TEASER?