

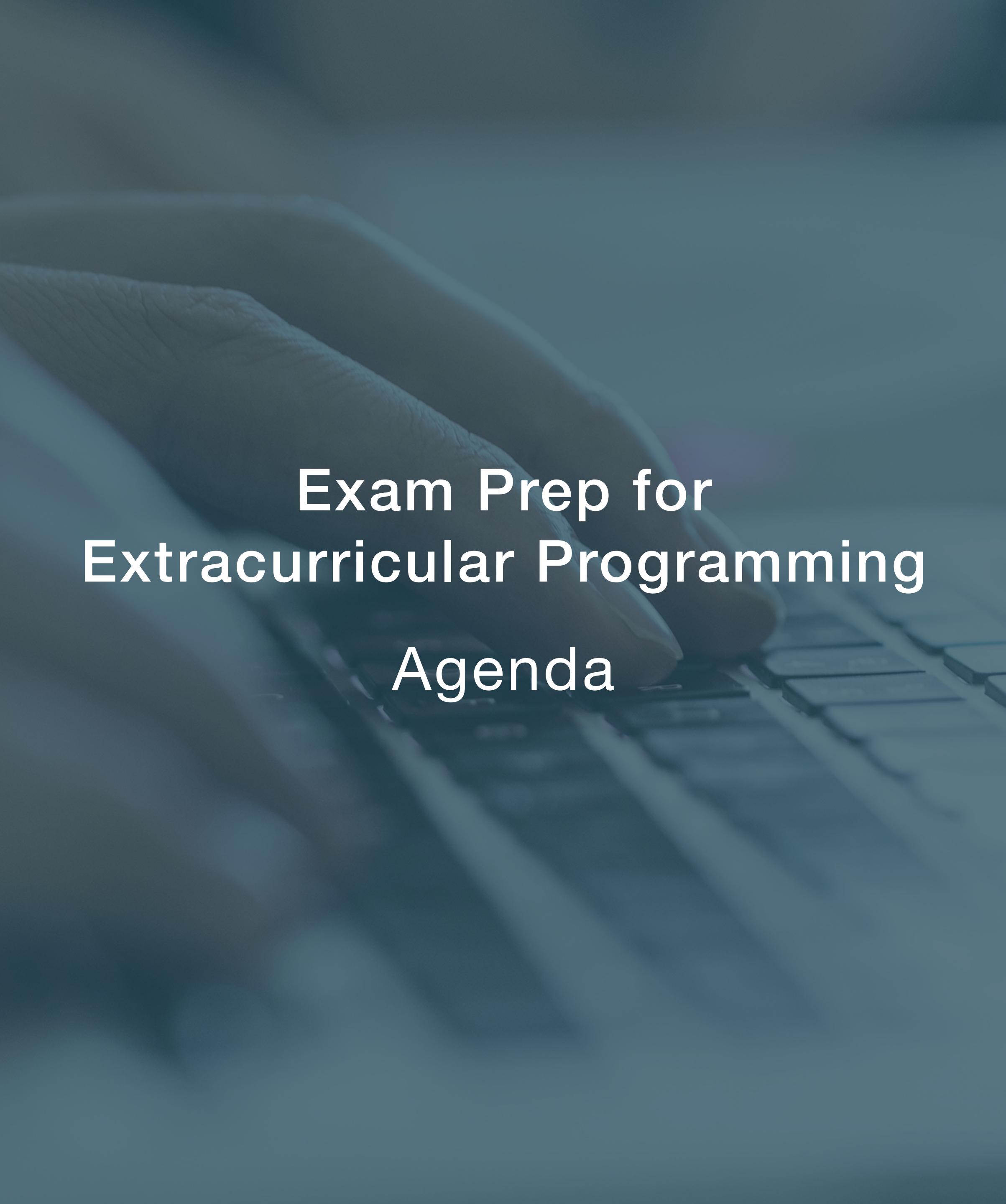


vevo

ERHVERVSAKADEMI AARHUS
BUSINESS ACADEMY AARHUS

Exam Prep for Extracurricular Programming

Welcome Back Programming 

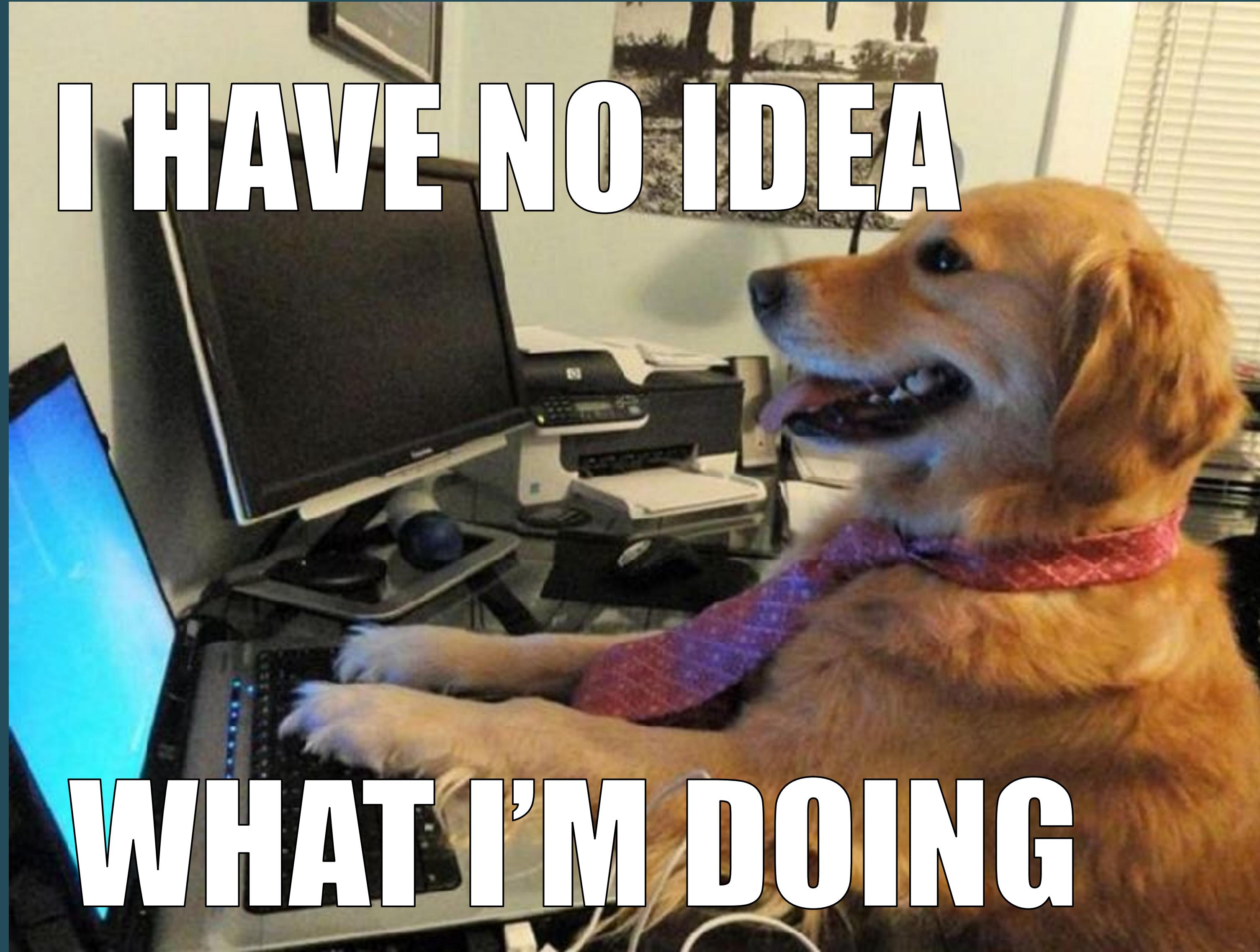


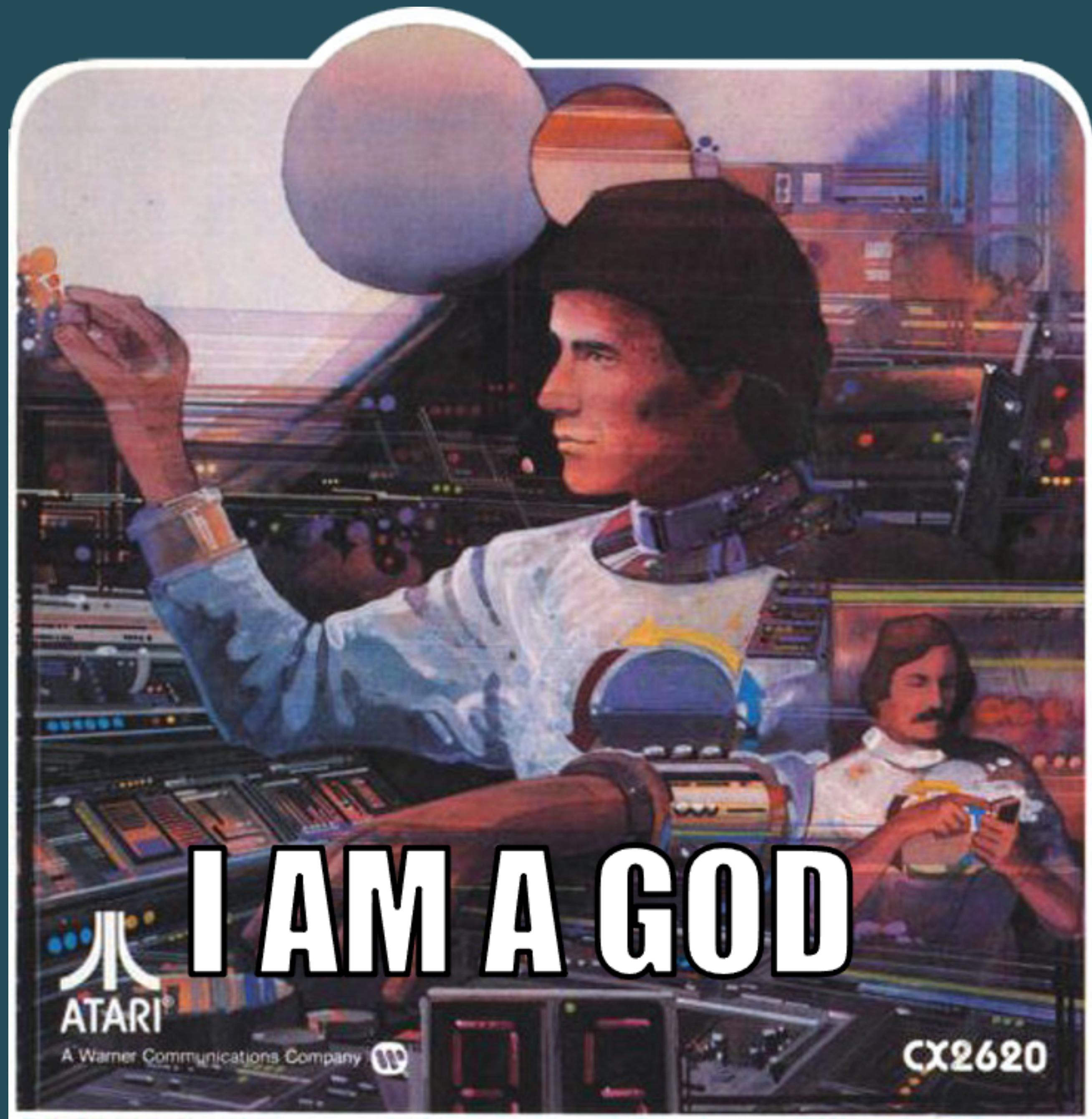
Exam Prep for Extracurricular Programming Agenda

1. Client Server
2. Fetch, JSON & DOM Manipulation
3. Headless WordPress
4. Recap on JS terms & concepts
5. Exam Prep

I HAVE NO IDEA

WHAT I'M DOING

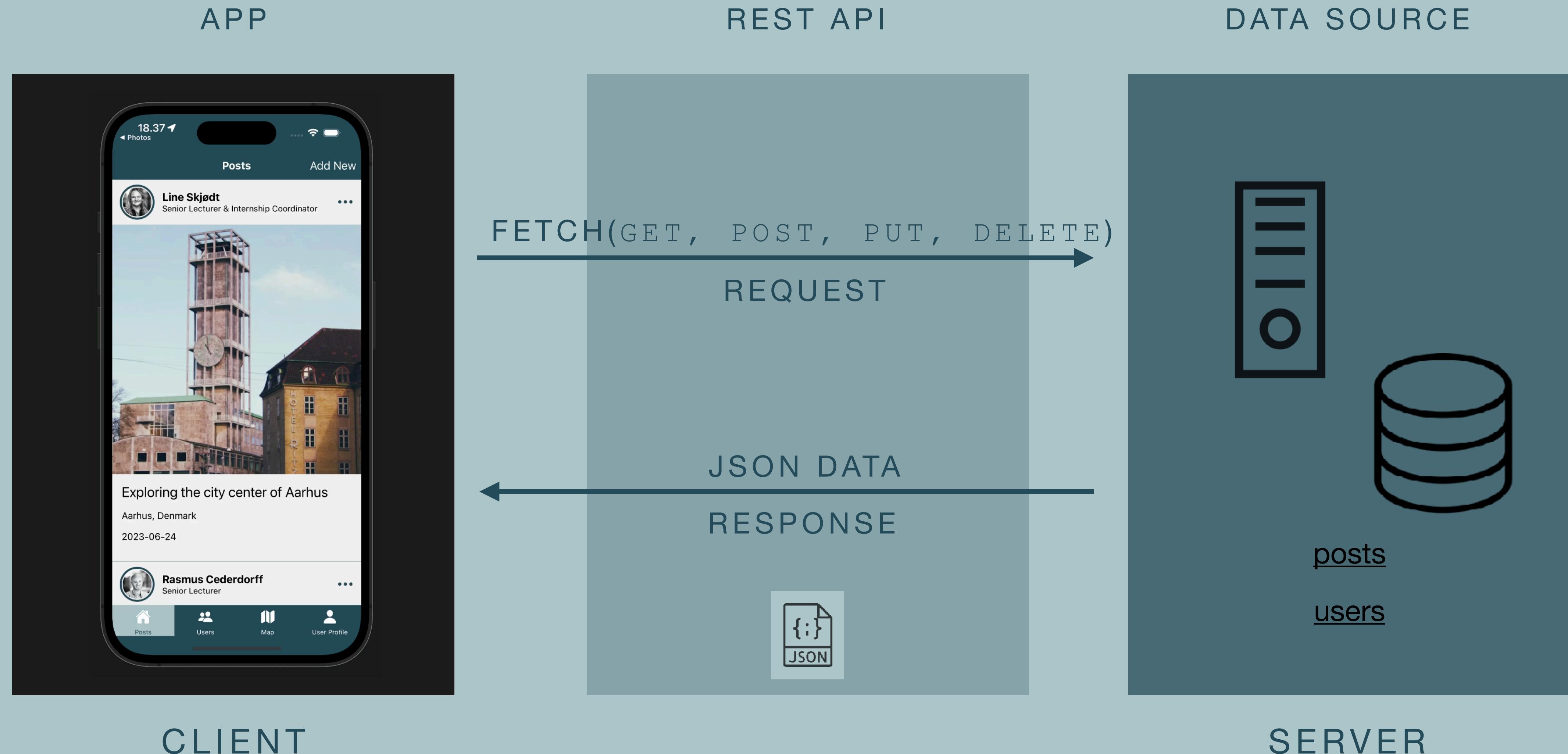




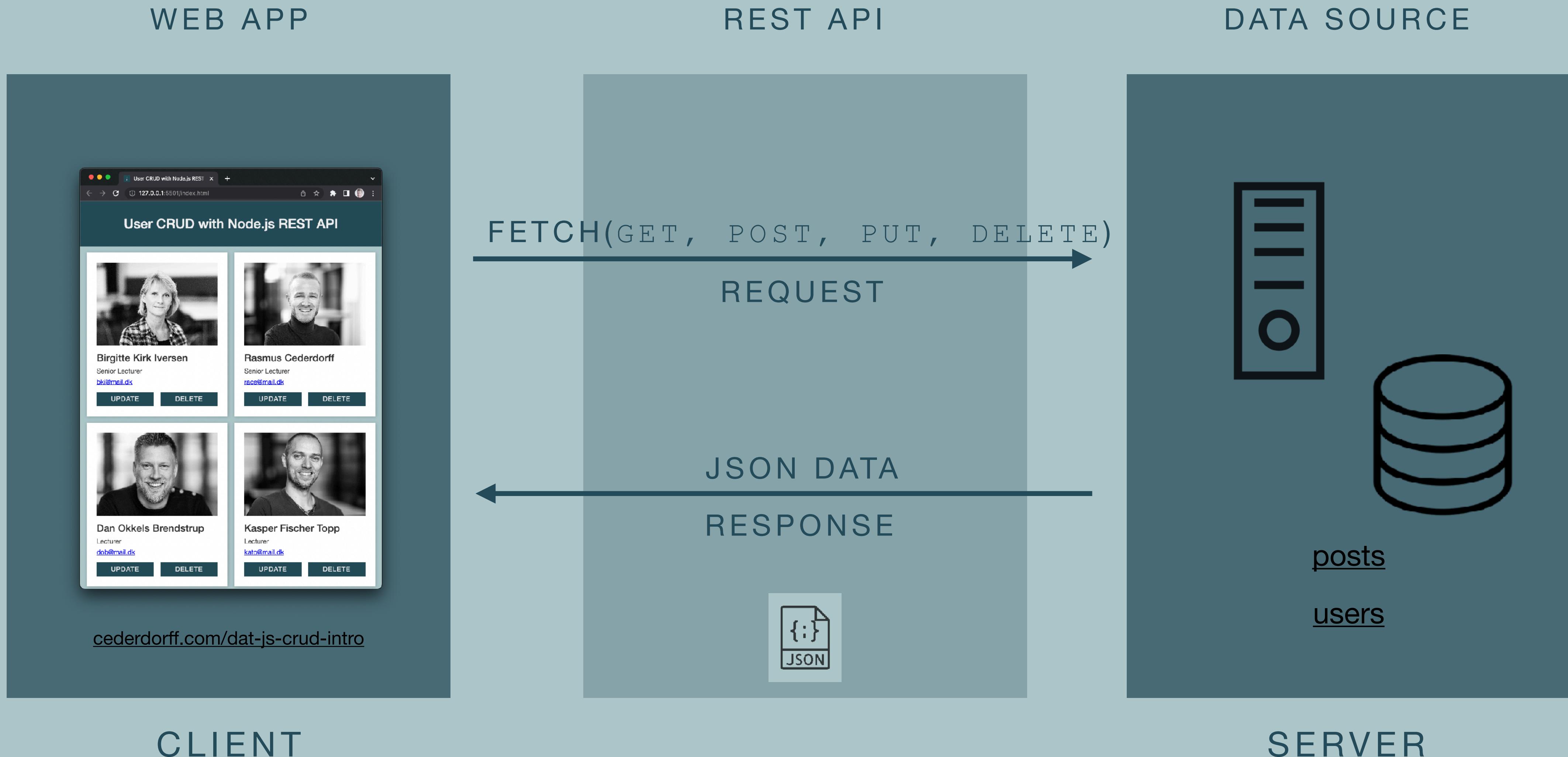
Client Server

The Basic Architecture of the Web

Client Server Architecture

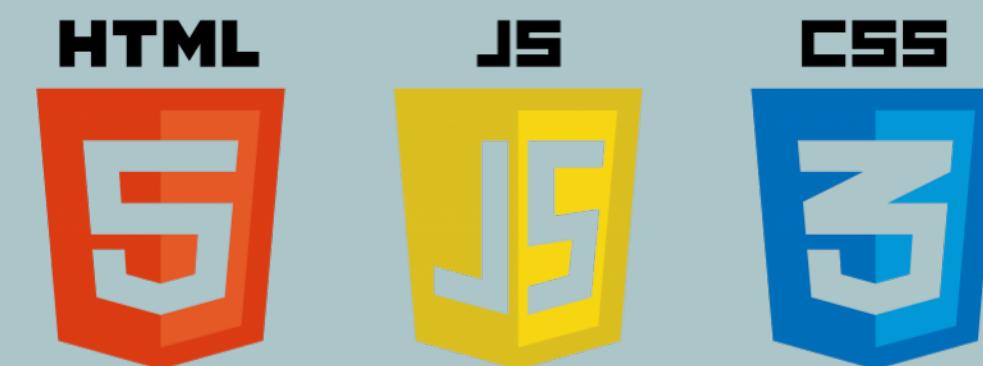


Web Development



Webudvikling

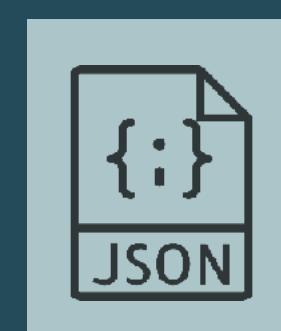
FRONTEND



REST API

FETCH(GET, POST, PUT, DELETE)
REQUEST

JSON DATA
RESPONSE

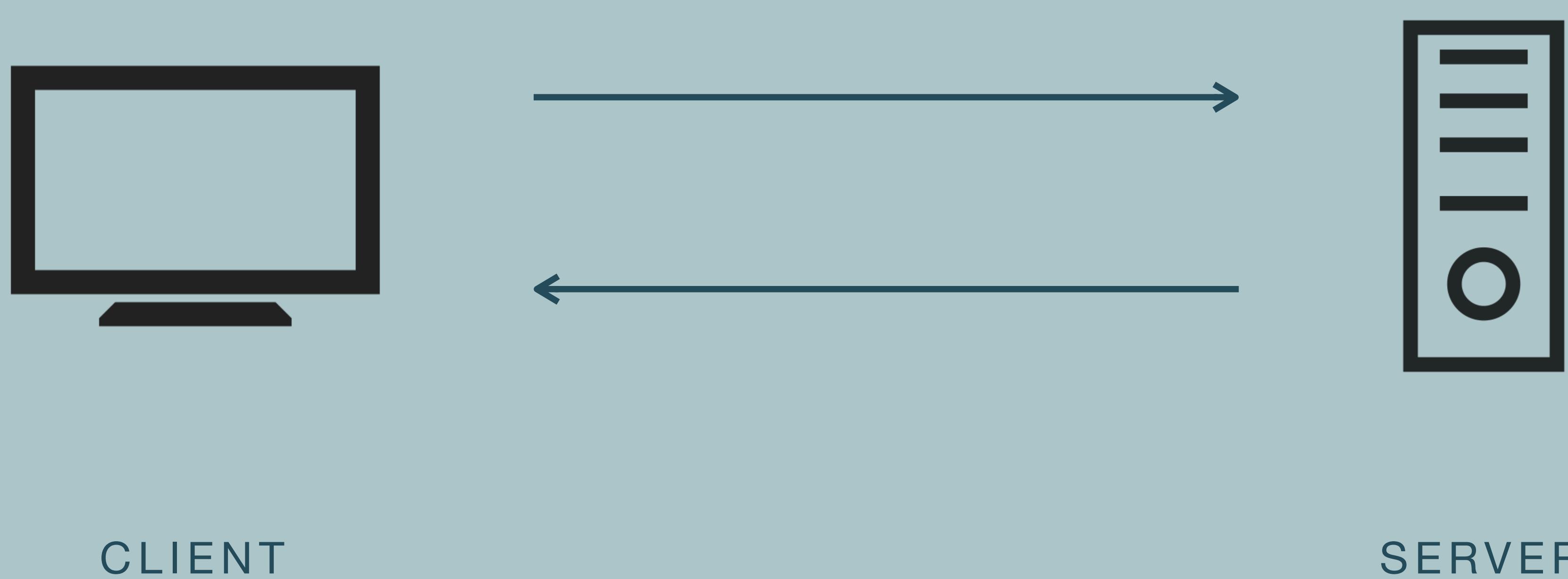


BACKEND



Client-Server Model

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



Client-Server Model

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

Browsers
Or any type of
program or device

CLIENT

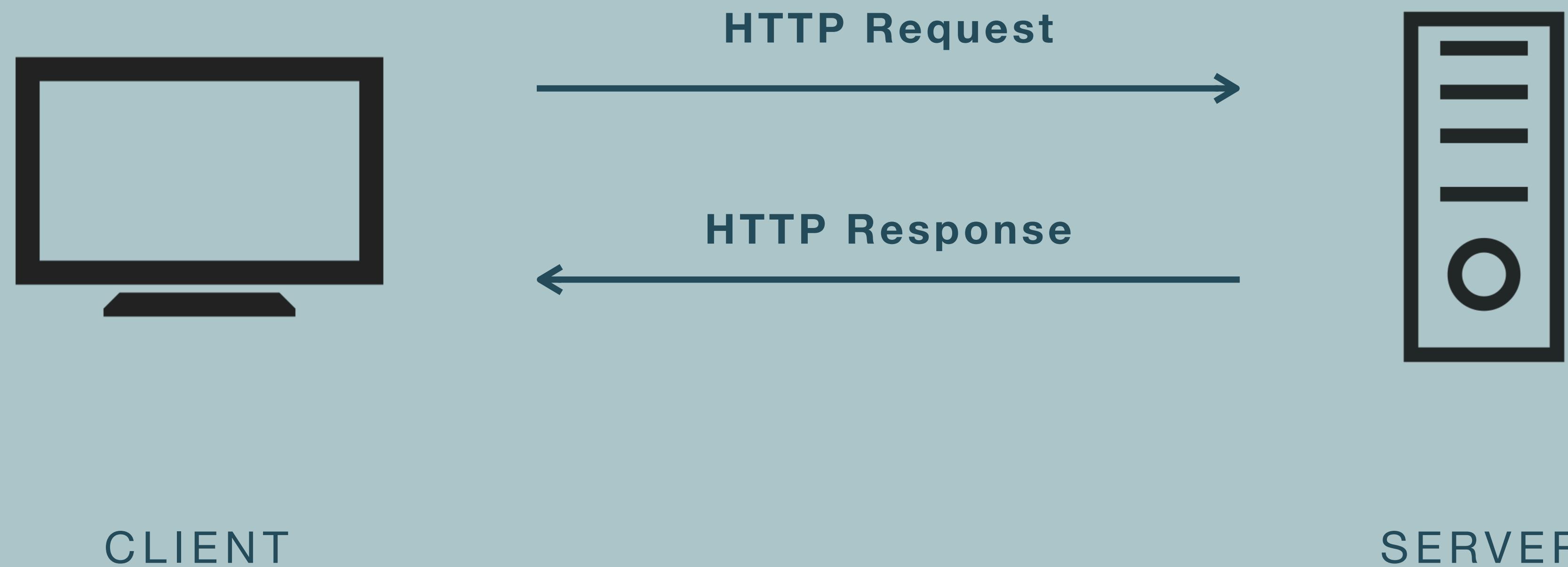


Cloud Computers
Often computers
in the cloud

SERVER

Client-Server Model

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



Hyper Text Transfer Protocol

- A protocol and standard for fetching data, HTML and other resources (text, images, videos, scripts, JSON).
- The foundation of the web.



What is HTTP

Not Secure | w3schools.com/whatis/whatis_http.asp

HTML CSS JAVASCRIPT SQL PYTHON

HTTP Request / Response

Communication between clients and servers is done by **requests** and **responses**:

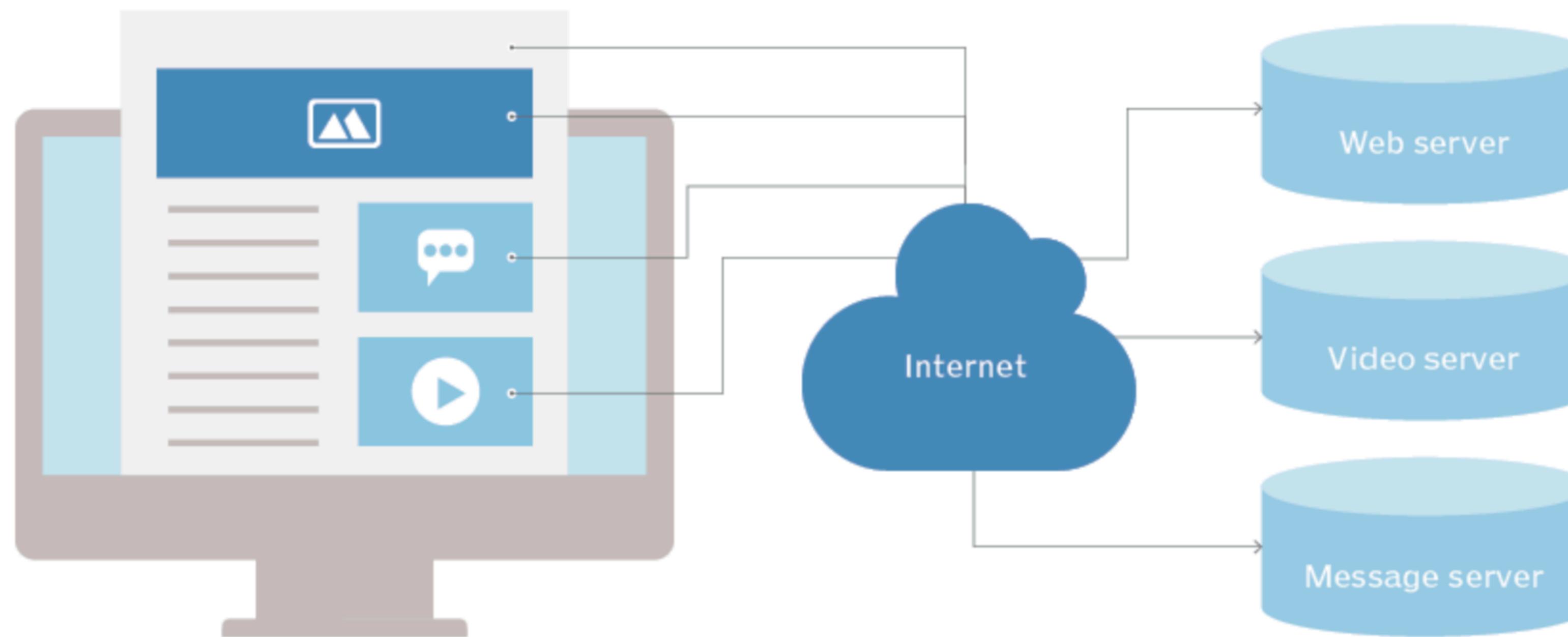
1. A client (a browser) sends an **HTTP request** to the web
2. A web server receives the request
3. The server runs an application to process the request
4. The server returns an **HTTP response** (output) to the browser
5. The client (the browser) receives the response

The HTTP Request Circle

A typical HTTP request / response circle:

1. The browser requests an HTML page. The server returns an HTML file.
2. The browser requests a style sheet. The server returns a CSS file.
3. The browser requests an JPG image. The server returns a JPG file.
4. The browser requests JavaScript code. The server returns a JS file
5. The browser requests data. The server returns data (in XML or JSON).

How HTTP Works



<https://www.techtarget.com/whatis/definition/HTTP-Hypertext-Transfer-Protocol>

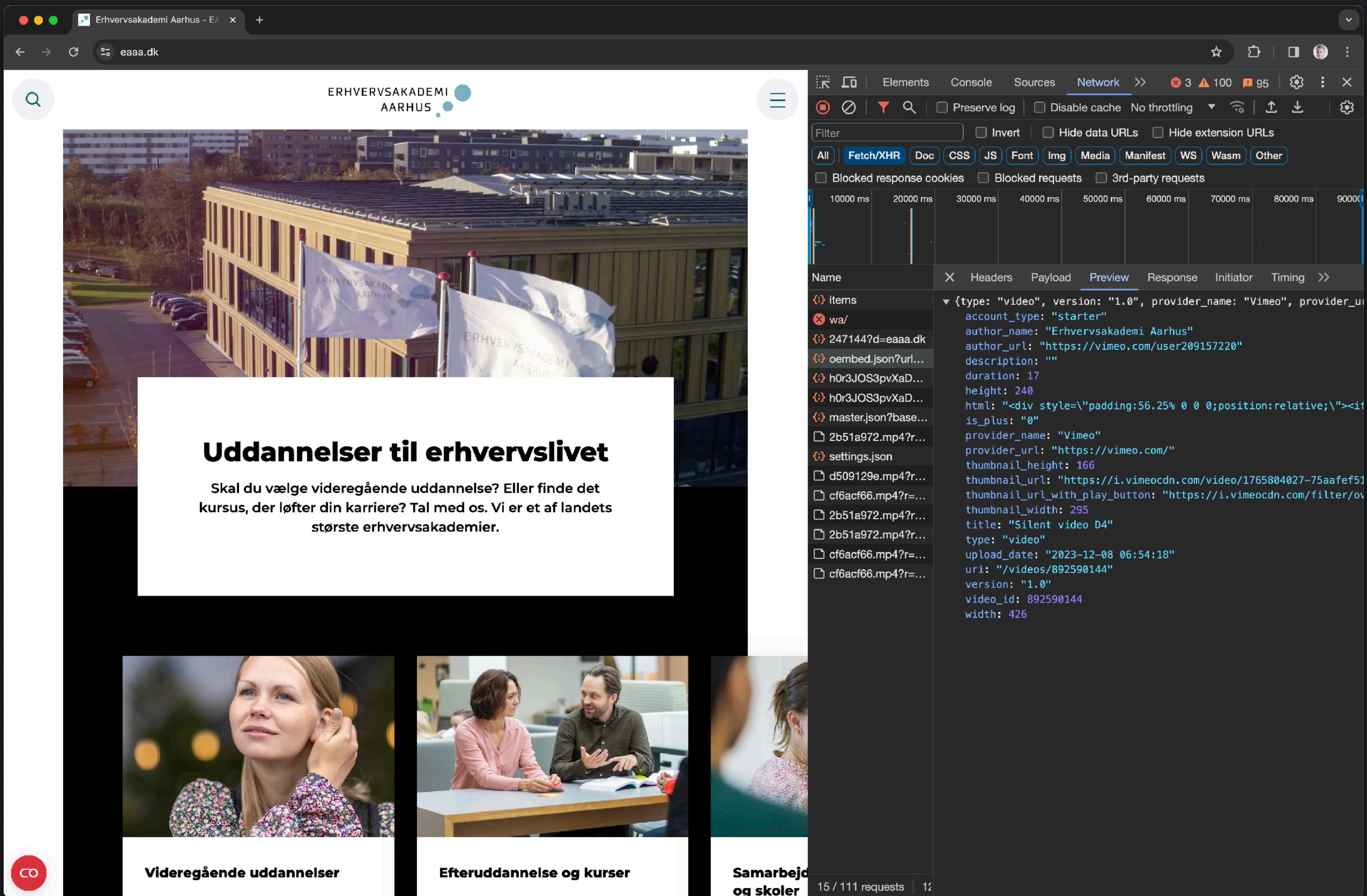
Network Tab

The screenshot illustrates the Network tab of a browser's developer tools, overlaid on a live website. The website's content includes a header for 'ERHVERVSAKADEMI AARHUS', a main image of two people working, and three cards at the bottom: 'Uddannelser til erhvervslivet', 'Videregående uddannelser', 'Efteruddannelse og kurser', and 'Samarbejde og s...

Name	Method	Status	Type	Initiator	Size	T..	Waterfall
1.gif?dgi=70fb8fd...	GET	200	gif	uc.js?cbid...	(me... 0...		
heatmaps.js	GET	200	script	monsido-s...	(disk... 2...		
page-correct.js	GET	200	script	monsido-s...	(disk... 2...		
?a=h0r3JOS3pvXaDa...	GET	200	gif	monsido-s...	57 B 1...		
favicon.ico	GET	200	x-icon	Other	(disk... 1...		
h0r3JOS3pvXaDabSjt...	GET	200	xhr	heatmaps....	(disk... 0...		
h0r3JOS3pvXaDabSjt...	GET	200	xhr	page-corr...	(disk... 0...		
cast_sender.js?loadC...	GET	200	script	vendor.mo...	(disk... 0...		
1765804027-75aafef...	GET	200	avif	Other	(disk... 0...		
master.json?base64_i...	GET	200	xhr	vendor.mo...	3.0 kB 1...		
1765804027-75aafef...	GET	200	avif	vendor.mo...	(me... 0...		
cast_framework.js	GET	200	script	cast_send...	14 B 1...		
cast_sender.js	GET	200	script	cast_send...	(disk... 1...		
settings.json	GET	200	xhr	uc.js?cbid...	(disk... 0...		
widgetIcon.min.js	GET	200	script	uc.js?cbid...	(disk... 0...		
2b51a972.mp4?r=dX...	GET	200	xhr	vendor.mo...	5.4 kB 2...		
d509129e.mp4?r=dX...	GET	200	xhr	vendor.mo...	264 kB 3...		
cf6acf66.mp4?r=dXM...	GET	200	xhr	vendor.mo...	4.3 ... 2...		
2b51a972.mp4?r=dX...	GET	200	xhr	vendor.mo...	5.4 kB 1...		
2b51a972.mp4?r=dX...	GET	200	xhr	vendor.mo...	5.1 kB 2...		
cf6acf66.mp4?r=dXM...	GET	200	xhr	vendor.mo...	4.6 ... 2...		
cf6acf66.mp4?r=dXM...	GET	200	xhr	vendor.mo...	3.5 ... 3...		
collect?v=2&tid=G-D...	POST	204	ping	js?id=G-D...	17 B 6...		

91 requests | 12.8 MB transferred | 16.2 MB resources | Finish: 5.47 s | DOMContentLoaded: 290 ms

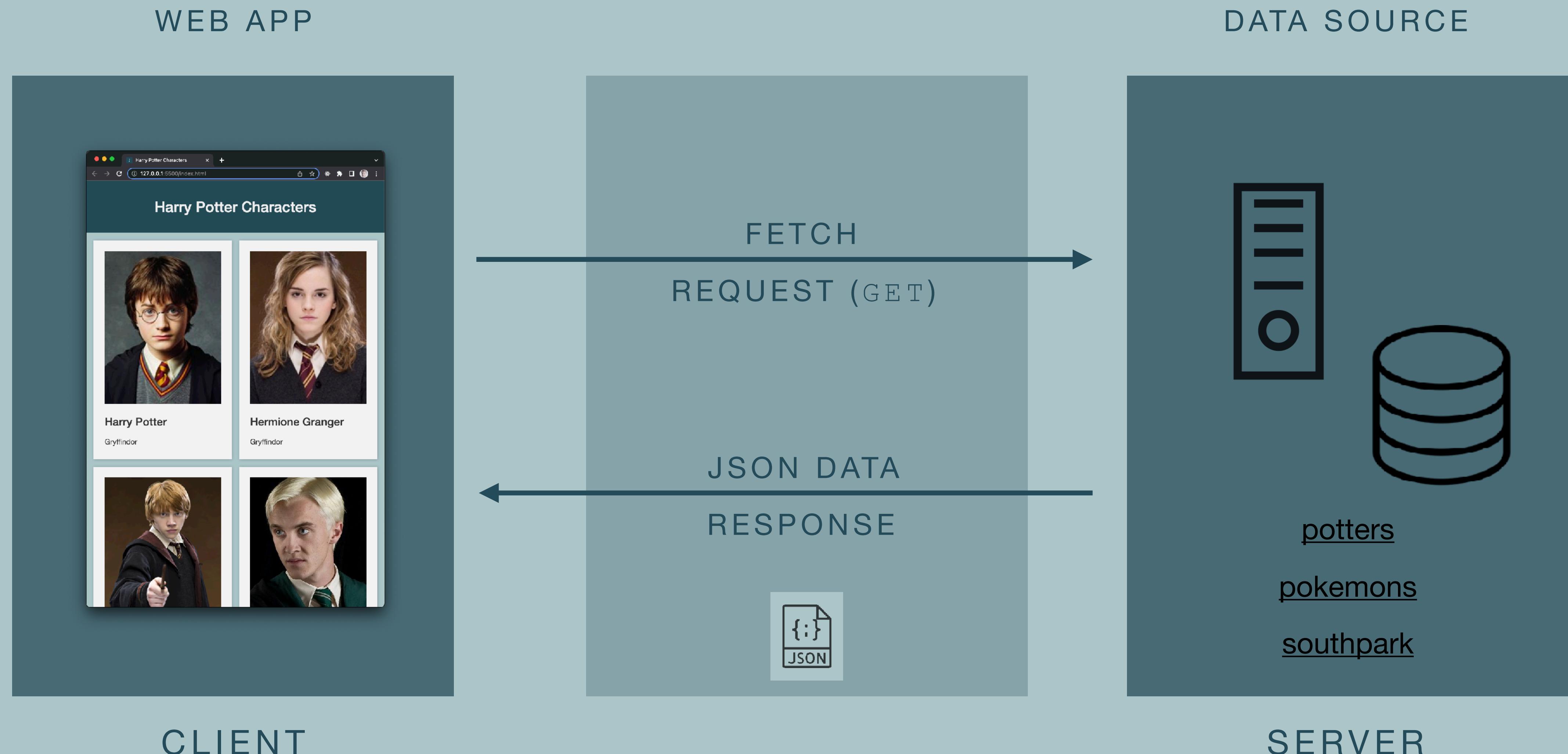
Network Tab



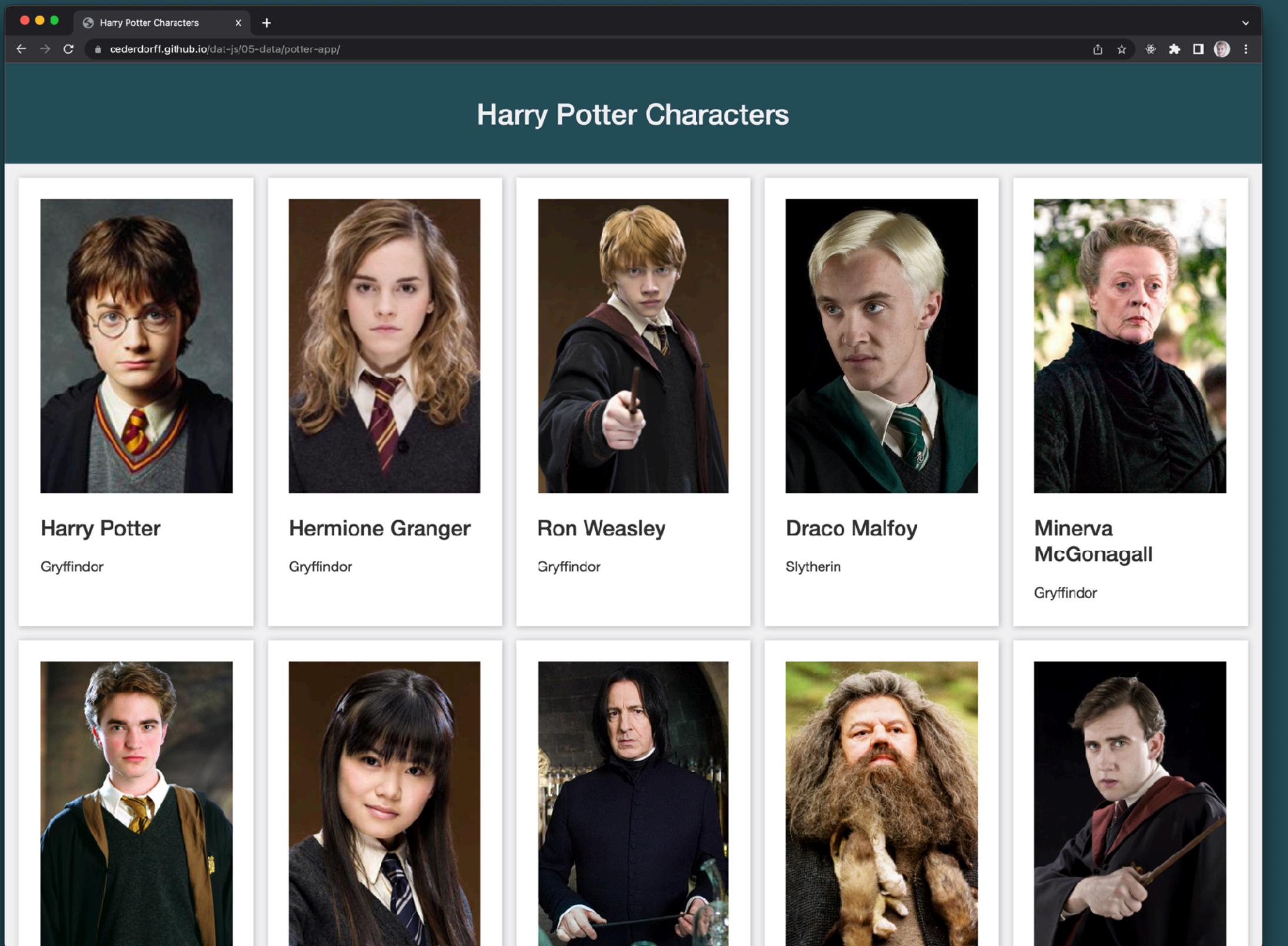
- Use the Network tab to investigate a website (e.g. [kea.dk](#), [dr.dk](#), [google.dk](#), [react-rest-and-auth.web.app](#) or any other).
- Open the website in Google Chrome (or another browser).
- Open the Developer Tool (see: [How to open the dev tool in your browser](#)) and go to Netværk/Network.
- Reload the page while you are in the Network tab and check which resources are being downloaded.
- What type(s) of resources are they?
- Consider how the resources are being downloaded and why I tell you to do all this.

Network Tab

Fetch, HTTP Request & Response



Frontend (client)



<https://cederdorff.com/potter-app/>

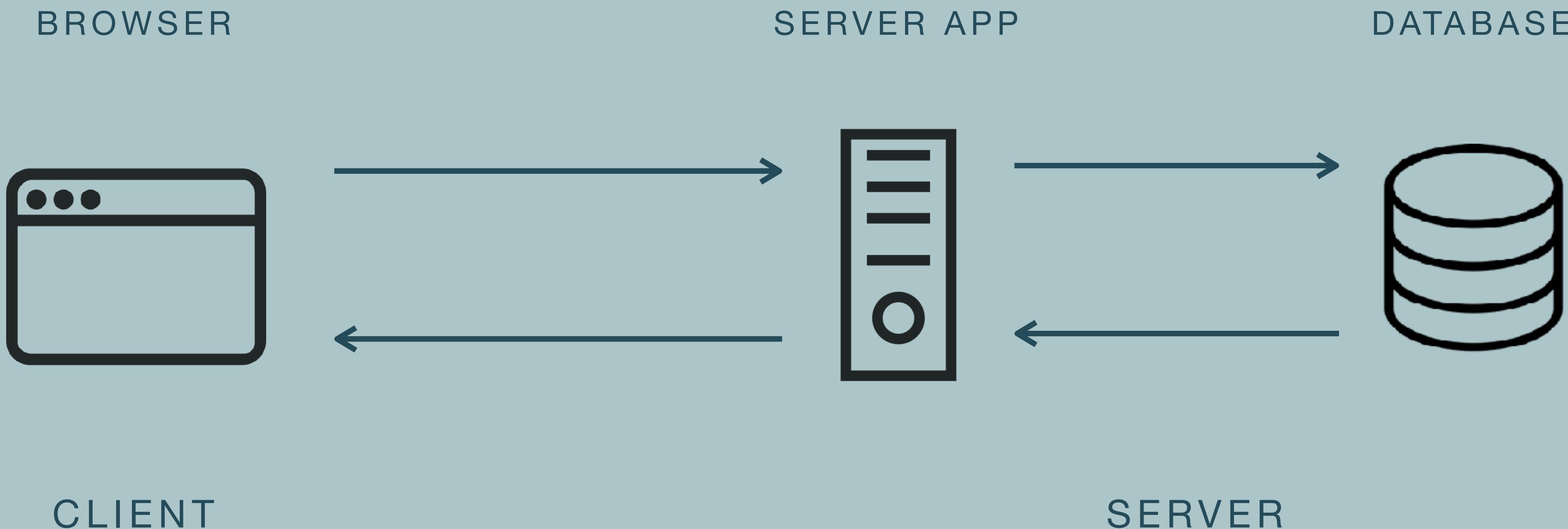
JSON Data Source (Server)

The screenshot shows a browser window displaying a JSON object. The JSON represents the Harry Potter characters with their names, species, gender, house, date of birth, year of birth, ancestry, eye color, hair color, and wand details. The "Raw" tab is selected, showing the raw JSON code, and the "Parsed" tab is also visible below it.

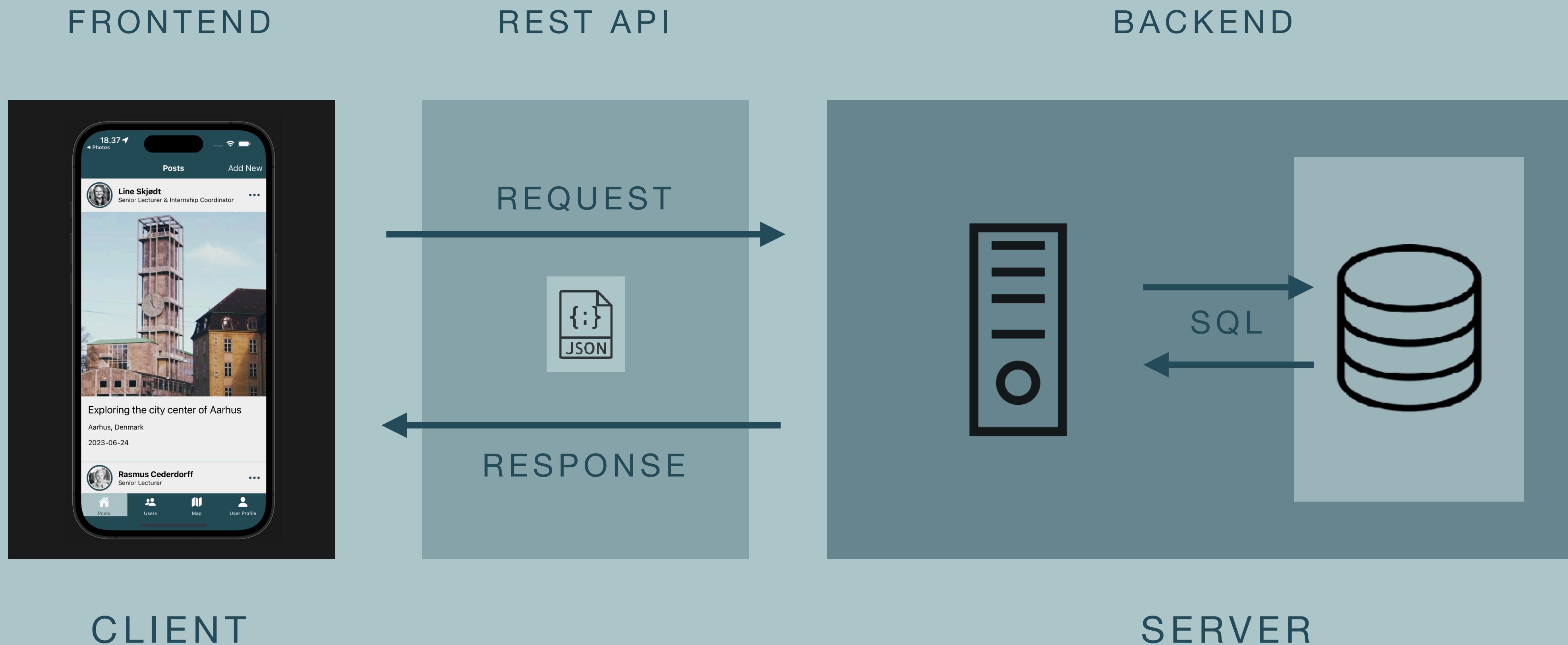
```
name: "Harry Potter",
species: "human",
gender: "male",
house: "Gryffindor",
dateOfBirth: "31-07-1980",
yearOfBirth: 1980,
ancestry: "half-blood",
eyeColour: "green",
hairColour: "black",
wand: {
  wood: "holly",
  core: "phoenix feather",
  length: 11
},
patronus: "stag",
hogwartsStudent: true,
hogwartsStaff: false,
actor: "Daniel Radcliffe",
alive: true,
image: "http://hp-api.herokuapp.com/images/harry.jpg"
},
{
name: "Hermione Granger",
species: "human",
gender: "female",
house: "Gryffindor",
dateOfBirth: "19-09-1979",
yearOfBirth: 1979,
ancestry: "muggleborn",
eyeColour: "brown",
hairColour: "brown",
wand: {
  wood: "vine",
  core: "dragon heartstring",
  length: ""
},
patronus: "otter",
hogwartsStudent: true,
hogwartsStaff: false,
actor: "Emma Watson",
alive: true,
image: "http://hp-api.herokuapp.com/images/hermione.jpeg"
},
{
name: "Ron Weasley",
species: "human",
gender: "male",
house: "Gryffindor",
dateOfBirth: "01-03-1980",
yearOfBirth: 1980,
ancestry: "pure-blood",
eyeColour: "blue",
hairColour: "red",
wand: {
  wood: "willow",
  core: "unicorn hair",
  length: 12
},
patronus: "raven",
hogwartsStudent: true,
hogwartsStaff: false,
actor: "Rupert Grint",
alive: true,
image: "http://hp-api.herokuapp.com/images/ron.jpg"
}
```

<https://raw.githubusercontent.com/cederdorff/dat-js/main/data/potter.json>

Web Dev Architecture



Web Dev Architecture



Payment Gateways

Social Media Integration

Healthcare Integration

Maps and Location Services

IoT (Internet of Things)

API

Weather Data

News and Content Aggregation

Financial Data

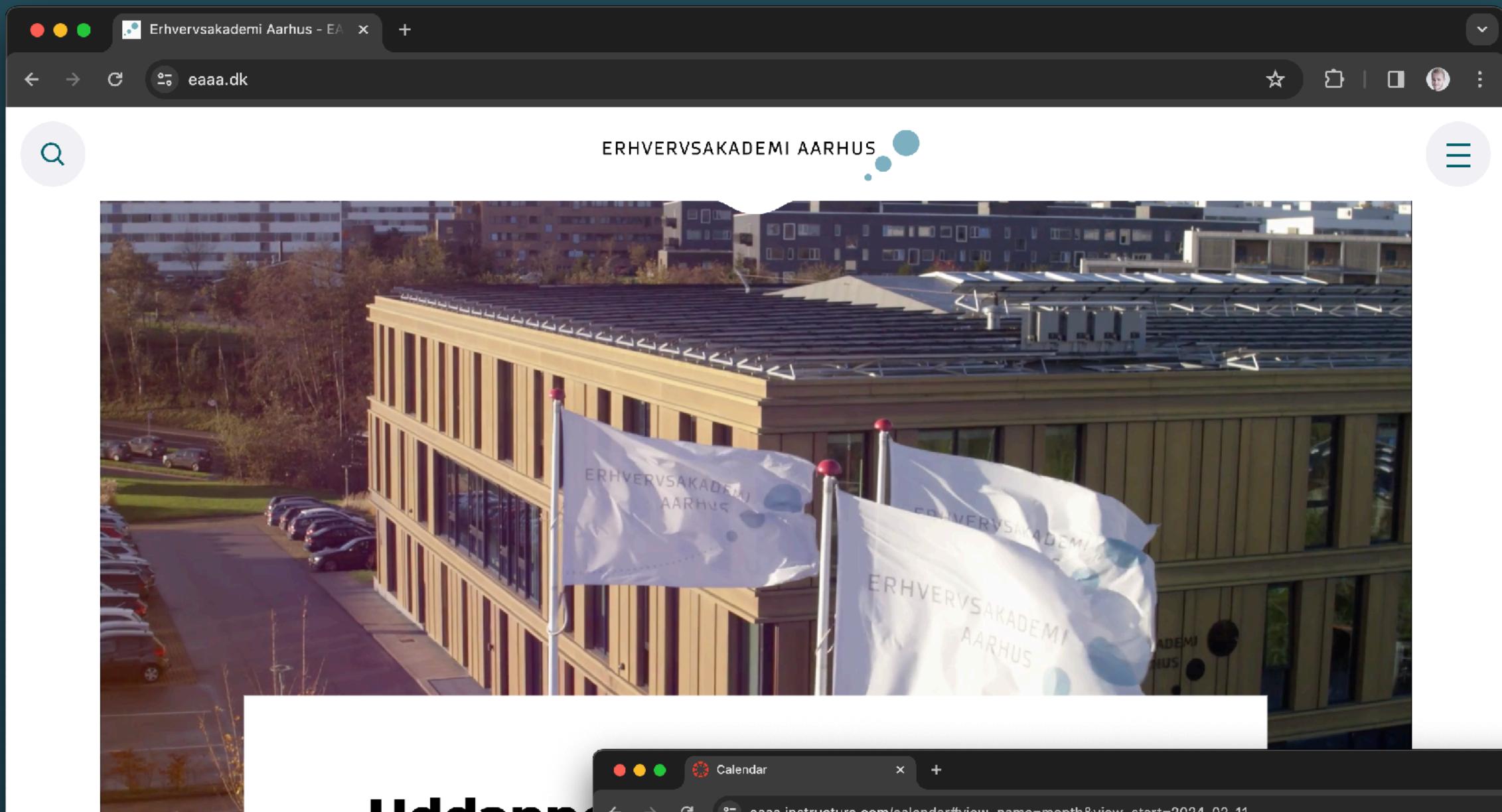
Cloud Services

E-commerce Platforms

The screenshot shows a web browser window titled "City Info App" with the URL "127.0.0.1:5500/index.html". The browser has a dark mode theme. On the left, there is a mobile application interface with a rounded rectangular background. It displays the title "Zip Code City App" and two input fields: "Zip Code:" with the value "8000" and "City:" with the value "Aarhus C". On the right, the browser's developer tools are open, showing the "Console" tab. The console output is a JSON object from "app.js:10" with the following structure:

```
app.js:10
{
  href: 'https://api.dataforsyningen.dk/postnumre/8000',
  nr: '8000',
  navn: 'Aarhus C',
  stormodtageradresser: null,
  bbox: Array(4),
  ...
}

bbox: [10.17165345, 56.11626813, 10.33170603, ...]
dagi_id: "198000"
geo_version: 3
geo_endret: "2023-03-24T22:28:38.837Z"
href: "https://api.dataforsyningen.dk/postnumre/8000"
kommuner: [...]
navn: "Aarhus C"
nr: "8000"
stormodtageradresser: null
visueltcenter: [10.27802097, 56.15051496]
endret: "2023-03-24T22:28:38.837Z"
[[Prototype]]: Object
```



A screenshot of the Instructure calendar interface for February 2024. The calendar shows a grid of days from Monday, February 29, to Sunday, March 3. Most days have multiple events listed, such as "08:30 AWU undervisning" at various times. There are also several "Hackathon" events scheduled throughout the month. The sidebar on the left includes links for Account, Admin, Dashboard, Courses, Calendar (with a red notification dot), Inbox, History, Commons, and Student links.

A screenshot of a Google search results page for the query "house of vincent ring". The results are sponsored ads for various gold rings from the brand House of Vincent. The first few ads show rings like the "GUL CITRIN FORGYLDT. 699 kr. SUN GAZER RING GUL CITRIN FORGYLDT." and the "CANDY ROCK RING". The right side of the page features a map of Europe with Denmark highlighted, and a section for "HOUSE OF VINCENT" with links to their website and a "Gem" section.

Twitter API

Stripe API

Google Maps API

Microsoft HealthVault API

Fitbit API

API

OpenWeatherMap API

The New York Times API

Amazon S3 API

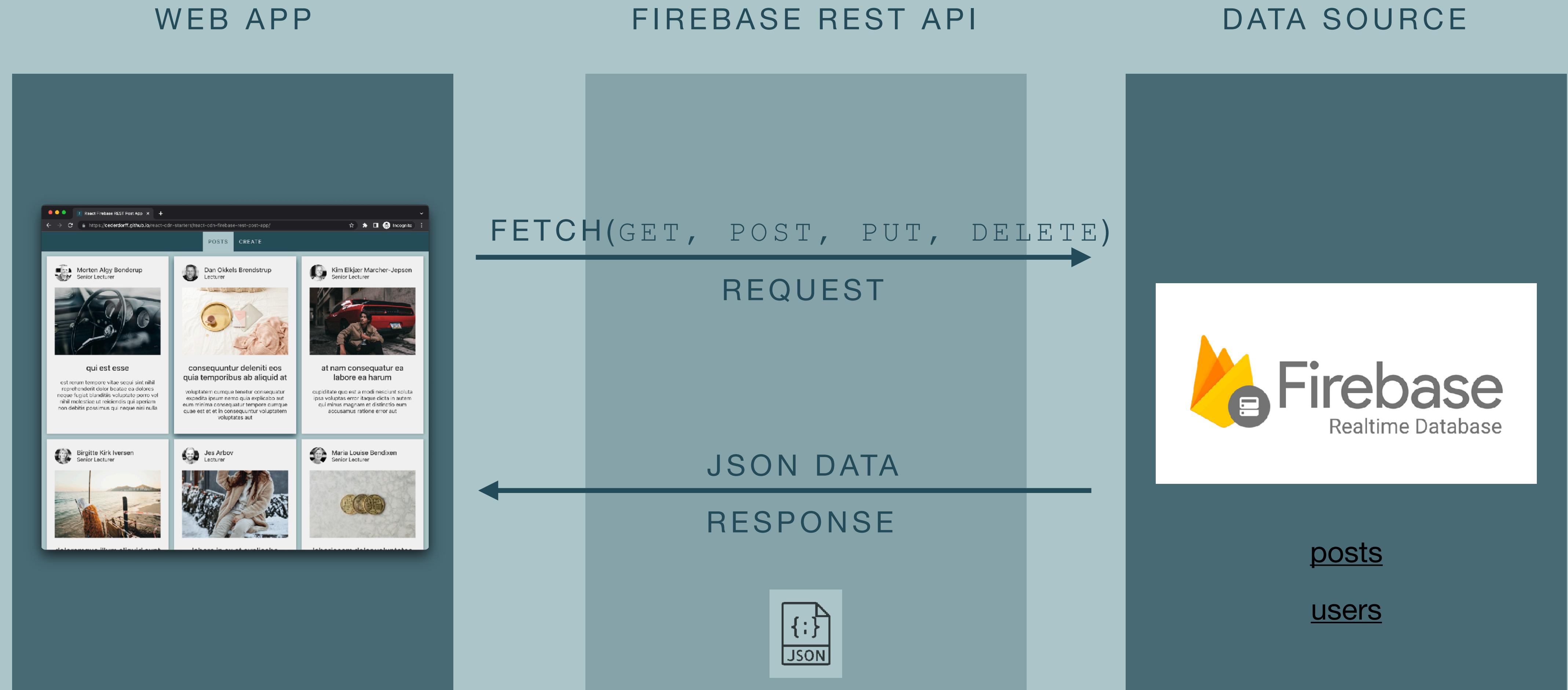
Alpha Vantage API

Shopify API

Fetch

HTTP Requests in JavaScript

Fetch, HTTP Request & Response



fetch(...)

HTTP Requests in
JavaScript.

A way to get and post data
from and to data sources.

```
// fetch with callbacks
fetch("https://cederdorff.github.io/web-frontend/callback.js")
  .then(function (response) {
    return response.json();
  })
  .then(function (data) {
    console.log(data);
  });
// or with promises
const response = fetch("https://cederdorff.github.io/web-frontend/promise.js");
const data = response.json();
console.log(data);
```

fetch(...)

HTTP Requests in
JavaScript.

A way to get and post data
from and to data sources.

getCharacters fetches a list of characters
from a JSON data source, parses the JSON
to JS and returns the data.

```
async function getCharacters() {  
  const response = await fetch(url);  
  const data = await response.json();  
  console.log(data);  
  return data;  
}
```

fetch(...)

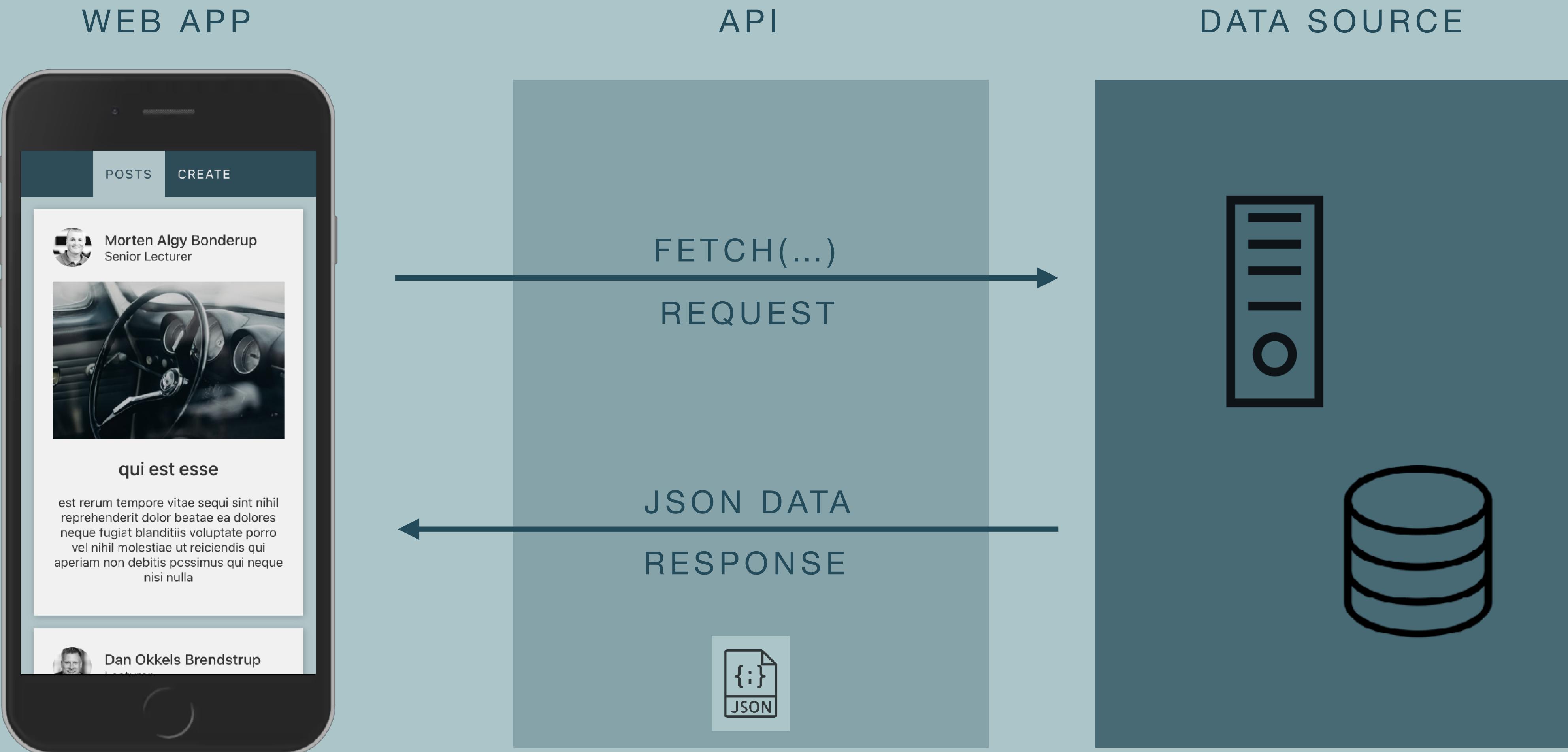
... get & post data from and to a data source
... can perform network requests to a server

```
1 let promise = fetch(url, [options])
```

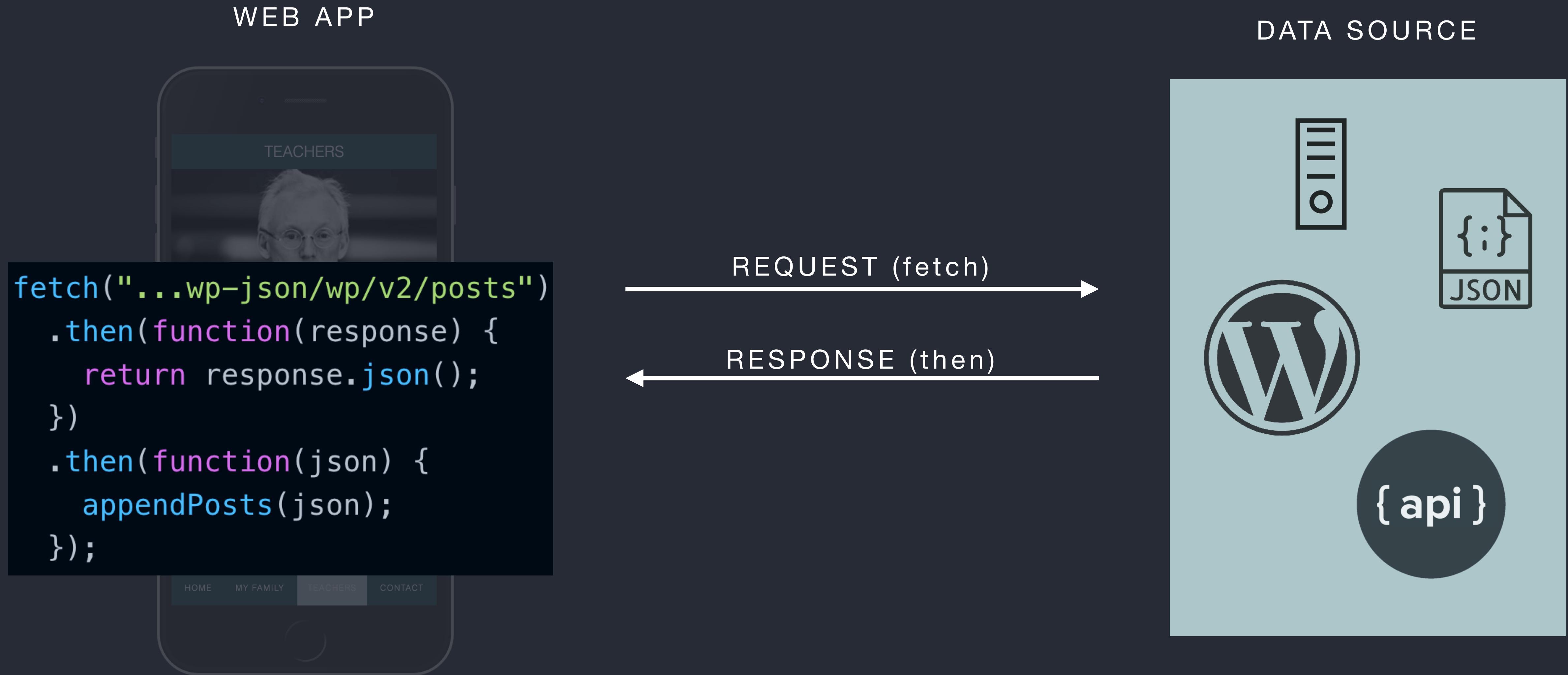
- `url` – the URL to access.
- `options` – optional parameters: method, headers etc.

Without `options`, this is a simple GET request, downloading the contents of the `url`.

Fetch, fetch, fetch



Fetch



```

/*
Fetches json data from the file persons.json
*/
fetch('json/persons.json')
  .then(function (response) {
    return response.json();
  })
  .then(function (jsonData) {
    console.log(jsonData);
    appendPersons(jsonData)
  });

/*
Appends json data to the DOM
*/
function appendPersons(persons) {
  let htmlTemplate = "";
  for (let person of persons) {
    htmlTemplate += /*html*/
      `


        <h4>${person.name}</h4>
        <p>${person.age} years old</p>
        <p>Hair color: ${person.hairColor}</p>
        <p>Relation: ${person.relation}</p>

`;
  }
  document.querySelector("#persons").innerHTML = htmlTemplate;
}

```

```

[
  {
    "name": "Peter Madsen",
    "age": 52,
    "hairColor": "blonde",
    "relation": "dad",
    "img": "img/dad.jpg"
  },
  {
    "name": "Ane Madsen",
    "age": 51,
    "hairColor": "brown",
    "relation": "mom",
    "img": "img/ane.jpg"
  },
  {
    "name": "Rasmus Madsen",
    "age": 28,
    "hairColor": "blonde",
    "relation": "brother",
    "img": "img/IMG_0526_kvadrat.jpg"
  },
  {
    "name": "Mie Madsen",
    "age": 25,
    "hairColor": "brown",
    "relation": "blonde",
    "img": "img/mie.jpg"
  },
  {
    "name": "Mads Madsen",
    "age": 18,
    "hairColor": "dark",
    "relation": "blonde",
    "img": "img/mads.jpg"
  },
  {
    "name": "Jens Madsen",
    "age": 14,
    "hairColor": "blonde",
    "relation": "uncle",
    "img": "img/jenspeter.jpg"
  }
]

```

```

/*
Fetches json data from the file persons.json
*/
fetch('json/persons.json')
  .then(function (response) {
    return response.json();
  })
  .then(function (jsonData) {
    console.log(jsonData);
    appendPersons(jsonData);
  });

/*
Appends json data to the DOM
*/
function appendPersons(persons) {
  let htmlTemplate = "";
  for (let person of persons) {
    htmlTemplate += /*html*/
      <article>
        
        <h4>${person.name}</h4>
        <p>${person.age} years old</p>
        <p>Hair color: ${person.hairColor}</p>
        <p>Relation: ${person.relation}</p>
      </article>
  }
  document.querySelector("#persons").innerHTML = htmlTemplate;
}

```

REQUEST (fetch)

RESPONSE (then)

```
[
  {
    "name": "Peter Madsen",
    "age": 52,
    "hairColor": "blonde",
    "relation": "dad",
    "img": "img/dad.jpg"
  },
  {
    "name": "Ane Madsen",
    "age": 51,
    "hairColor": "brown",
    "relation": "mom",
    "img": "img/ane.jpg"
  },
  {
    "name": "Rasmus Madsen",
    "age": 28,
    "hairColor": "blonde",
    "relation": "brother",
    "img": "img/IMG_0526_kvadrat.jpg"
  },
  {
    "name": "Mie Madsen",
    "age": 25,
    "hairColor": "brown",
    "relation": "blonde",
    "img": "img/mie.jpg"
  },
  {
    "name": "Mads Madsen",
    "age": 18,
    "hairColor": "dark",
    "relation": "blonde",
    "img": "img/mads.jpg"
  },
  {
    "name": "Jens Madsen",
    "age": 14,
    "hairColor": "blonde",
    "relation": "uncle",
    "img": "img/jenspeter.jpg"
  }
]
```

fetch-family-members

```
/*
Fetches json data from the file persons.json
*/
fetch('json/persons.json')
  .then(function (response) {
    return response.json();
  })
  .then(function (jsonData) {
    console.log(jsonData);
    appendPersons(jsonData)
  });

/*
Appends json data to the DOM
*/
function appendPersons(persons) {
  let htmlTemplate = "";
  for (let person of persons) {
    htmlTemplate += /*html*/
      `


        <h4>${person.name}</h4>
        <p>${person.age} years old</p>
        <p>Hair color: ${person.hairColor}</p>
        <p>Relation: ${person.relation}</p>

`;
  }
  document.querySelector("#persons").innerHTML = htmlTemplate;
}
```

```
},
{
  "name": "Rasmus Madsen",
  "age": 28,
  "hairColor": "blonde",
  "relation": "brother",
  "img": "img/IMG_0526_kvadrat.jpg"
},
{
  "name": "Mie Madsen",
  "age": 25,
  "hairColor": "brown",
  "relation": "blonde",
  "img": "img/mie.jpg"
},
{
  "name": "Mads Madsen",
  "age": 18,
  "hairColor": "dark",
  "relation": "blonde",
  "img": "img/mads.jpg"
},
{
```

request (fetch)

app.js — web-diplom-frontend

```
JS app.js x
fetch-persons-grid > JS app.js > ...
1  let persons = [] // global variable
2
3  async function getPerson() {
4      const response = await fetch(
5          "https://raw.githubusercontent.com/cederdorff/web-diplom-frontend/main/_data/persons.json"
6      ); // fetch request - fetch data from a given url
7      persons = await response.json(); // setting global variable with fetched data
8      displayPersons(persons); // calling displayPersons with persons as parameter
9  }
10
11 function displayPersons(listOfPersons) {
12     let html = ""; // variable to store html
13     //loop through all persons and create an article with content for each
14     for (const person of listOfPersons) {
15         html += /*html*/
16             `

17                 
18                 <h2>${person.name}</h2>
19                 <p>${person.title}</p>
20                 <a href="mailto:${person.mail}">${person.mail}</a>
21             </article>
22         `; // generate and save html for every person in html variable
23     }
24     // set grid container content with person <article> elements
25     // saved in html
26     document.querySelector("#content").innerHTML = html;
27 }
28
29 getPerson(); // execute get persons to start the fun
30


```

Ln 1, Col 1 Spaces: 4 UTF-8 LF {} JavaScript Go Live ✓ Prettier

The diagram illustrates the flow of data. A curved arrow originates from the URL in the fetch statement at line 5 of the app.js code and points to the JSON response shown in the browser window. Another curved arrow originates from the JSON response in the browser and points back to the app.js code, specifically to the line where the response is converted to a global variable.

https://raw.githubusercontent.com/cederdorff/web-diplom-frontend/main/_data/persons.json

Raw Parsed

```
[{"name": "Birgitte Kirk Iversen", "mail": "bkj@mail.dk", "title": "Senior Lecturer", "img": "https://www.eaaa.dk/media/u4gorzsd/birgitte-kirk-iversen2.jpg?width=800&height=450"}, {"name": "Martin Aagaard N\u00f8hr", "mail": "mnor@mail.dk", "title": "Lecturer", "img": "https://www.eaaa.dk/media/oayjq02h/martin-n%C3%B8hr.jpg?width=800&height=450"}, {"name": "Rasmus Cederdorff", "mail": "race@mail.dk", "title": "Senior Lecturer", "img": "https://www.eaaa.dk/media/devlvgj/rasmus-cederdorff.jpg?width=800&height=450"}, {"name": "Dan Okkels Brendstrup", "mail": "dob@mail.dk", "title": "Lecturer", "img": "https://www.eaaa.dk/media/bdojel41/dan-okkels-brendstrup.jpg?width=800&height=450"}, {"name": "Line Skj\u00f8dt", "mail": "lskj@mail.dk", "title": "Senior Lecturer & Internship Coordinator", "img": "https://www.eaaa.dk/media/14qpfq4/line-skj%C3%B8dt.jpg?width=800&height=450"}, {"name": "Kasper Fischer Topp", "mail": "kato@mail.dk", "title": "Lecturer", "img": "https://www.eaaa.dk/media/lxzcybme/kasper-topp.jpg?width=800&height=450"}, {"name": "Anne Kirketerp", "mail": "anki@mail.dk", "title": "", "img": ""}]
```

fetch-persons-grid

request (fetch)

The diagram illustrates the flow of data from the `app.js` file to a browser displaying the JSON response.

app.js — web-diplom-frontend

```
JS app.js ×
fetch-persons-grid > JS app.js > ...
1 let persons = [] // global variable
2
3 async function getPerson() {
4     const response = await fetch(
5         "https://raw.githubusercontent.com/cederdorff/web-diplom-frontend/main/_data/persons.json"
6     ); // fetch request - fetch data from a given url
7     persons = await response.json(); // setting global variable with fetched data
8     displayPersons(persons); // calling displayPersons with persons as parameter
9 }
10
11 function displayPersons(listOfPersons) {
12     let html = ""; // variable to store html
13     //loop through all persons and create an article with content for each
14     for (const person of listOfPersons) {
15         html += /*html*/
16             `

17                 
18                 <h2>${person.name}</h2>
19                 <p>${person.title}</p>
20                 <a href="mailto:${person.mail}">${person.mail}</a>
21             </article>
22         `; // generate and save html for every person in html variable
23     }
24     // set grid container content with person <article> elements
25     // saved in html
26     document.querySelector("#content").innerHTML = html;
27 }
28
29 getPerson(); // execute get persons to start the fun
30


```

response (JSON)

The `getPerson()` function in `app.js` makes a `fetch` request to the URL `"https://raw.githubusercontent.com/cederdorff/web-diplom-frontend/main/_data/persons.json"`. The browser screenshot shows the JSON response containing data for six individuals:

```
[{"name": "Birgitte Kirk Iversen", "mail": "bkj@mail.dk", "title": "Senior Lecturer", "img": "https://www.eaaa.dk/media/u4gorzsd/birgitte-kirk-iversen2.jpg?width=800&height=450"}, {"name": "Martin Aagaard N\u00f8hr", "mail": "mnor@mail.dk", "title": "Lecturer", "img": "https://www.eaaa.dk/media/oayjq02h/martin-n%C3%B8hr.jpg?width=800&height=450"}, {"name": "Rasmus Cederdorff", "mail": "race@mail.dk", "title": "Senior Lecturer", "img": "https://www.eaaa.dk/media/devlvgj/rasmus-cederdorff.jpg?width=800&height=450"}, {"name": "Dan Okkels Brendstrup", "mail": "dob@mail.dk", "title": "Lecturer", "img": "https://www.eaaa.dk/media/bdojel41/dan-okkels-brendstrup.jpg?width=800&height=450"}, {"name": "Line Skj\u00f8dt", "mail": "lskj@mail.dk", "title": "Senior Lecturer & Internship Coordinator", "img": "https://www.eaaa.dk/media/14qpfeq4/line-skj%C3%B8dt.jpg?width=800&height=450"}, {"name": "Kasper Fischer Topp", "mail": "kato@mail.dk", "title": "Lecturer", "img": "https://www.eaaa.dk/media/lxzcybme/kasper-topp.jpg?width=800&height=450"}, {"name": "Anne Kirketerp", "mail": "anki@mail.dk", "title": "", "img": ""}]
```

fetch-persons-grid

The diagram illustrates the data flow in a web application. On the left, a screenshot of a code editor shows the file `app.js` with the following code:

```
1 let persons = [] // global variable
2
3 async function getPerson() {
4     const response = await fetch(
5         "https://raw.githubusercontent.com/cederdorff/web-diplom-frontend/main/_data/persons.json"
6     ); // fetch request - fetch data from a given url
7     persons = await response.json(); // setting global variable with fetched data
8     displayPersons(persons); // calling displayPersons with persons as parameter
9 }
10
11 function displayPersons(listOfPersons) {
12     let html = ""; // variable to store html
13     //loop through all persons and create an article with content for each
14     for (const person of listOfPersons) {
15         html += /*html*/
16             `

17                 
18                 <h2>${person.name}</h2>
19                 <p>${person.title}</p>
20                 <a href="mailto:${person.mail}">${person.mail}</a>
21             </article>
22         `; // generate and save html for every person in html variable
23     }
24     // set grid container content with person <article> elements
25     // saved in html
26     document.querySelector("#content").innerHTML = html;
27 }
28
29 getPerson(); // execute get persons to start the fun
30


```

On the right, a screenshot of a browser window shows the JSON data source at `https://raw.githubusercontent.com/cederdorff/web-diplom-frontend/main/_data/persons.json`. The JSON array contains the following objects:

```
[{"name": "Birgitte Kirk Iversen", "mail": "bki@mail.dk", "title": "Senior Lecturer", "img": "https://www.eaaa.dk/media/u4gorzsd/birgitte-kirk-iversen2.jpg?width=800&height=450"}, {"name": "Martin Aagaard N\u00f8hr", "mail": "mnor@mail.dk", "title": "Lecturer", "img": "https://www.eaaa.dk/media/oayjq02h/martin-n%C3%B8hr.jpg?width=800&height=450"}, {"name": "Rasmus Cederdorff", "mail": "race@mail.dk", "title": "Senior Lecturer", "img": "https://www.eaaa.dk/media/devlvgj/rasmus-cederdorff.jpg?width=800&height=450"}, {"name": "Dan Okkels Brendstrup", "mail": "dob@mail.dk", "title": "Lecturer", "img": "https://www.eaaa.dk/media/bdojel41/dan-okkels-brendstrup.jpg?width=800&height=450"}, {"name": "Line Skj\u00f8dt", "mail": "lskj@mail.dk", "title": "Senior Lecturer & Internship Coordinator", "img": "https://www.eaaa.dk/media/14qpfq4/line-skj%C3%B8dt.jpg?width=800&height=450"}, {"name": "Kasper Fischer Topp", "mail": "kato@mail.dk", "title": "Lecturer", "img": "https://www.eaaa.dk/media/lxzcybme/kasper-topp.jpg?width=800&height=450"}, {"name": "Anne Kirketerp", "mail": "anki@mail.dk", "title": "", "img": ""}]
```

fetch-persons-grid

index.html — web-diplom-frontend

JS app.js

```
fetch-persons-grid > JS app.js > ...
1 let persons = [] // global variable
2
3 async function getPerson() {
4     const response = await fetch(
5         "https://raw.githubusercontent.com/cederdorff/web-diplom-frontend/main/_data/persons.json"
6     ); // fetch request - fetch data from a given url
7     persons = await response.json(); // setting global variable with fetched data
8     displayPersons(persons); // calling displayPersons with persons as parameter
9 }
10
11 function displayPersons(listOfPersons) {
12     let html = ""; // variable to store html
13     //loop through all persons and create an article with content for each
14     for (const person of listOfPersons) {
15         html += /*html*/ `
16             <article>
17                 
18                 <h2>${person.name}</h2>
19                 <p>${person.title}</p>
20                 <a href="mailto:${person.mail}">${person.mail}</a>
21             </article>
22         `; // generate and save html for every person in html variable
23     }
24     // set grid container content with person <article> elements
25     // saved in html
26     document.querySelector("#content").innerHTML = html;
27 }
28
29 getPerson(); // execute get persons to start the fun
30
```

index.html

```
fetch-persons-grid > index.html > ...
1 <!DOCTYPE html>
2 <html lang="en">
3
4 <head>
5     <meta charset="UTF-8">
6     <meta http-equiv="X-UA-Compatible" content="IE=edge">
7     <meta name="viewport" content="width=device-width, initial-scale=1.0">
8     <link rel="stylesheet" href="app.css">
9     <title>Fetch Persons</title>
10
11 </head>
12 <body>
13     <header>
14         <h1>Fetch Persons</h1>
15     </header>
16     <main>
17         <section id="content" class="grid-container"></section>
18     </main>
19     <script src="app.js"></script>
20
21 </body>
22 </html>
```

DOM Manipulation

Ln 17, Col 9 Spaces: 4 UTF-8 LF HTML ⚡ Go Live ✅ Prettier

fetch-persons-grid

Fetch

... get & post data from and to a data source

```
// Simple javascript 😒  
  
//Synchronous fetch using async/await.  
  
// Usual way  
✓ const jsonData = fetch('URL')  
    .then(response => response.json())  
    .then(json => console.log(json));  
  
// Using await  
✓ const jsonData = await fetch('URL').then(res => res.json())  
  
// Shorter syntax 😊  
✓ const jsonData = await (await fetch('URL')).json();
```

<https://www.instagram.com/p/B0nxQjXj9Zi/>

Async JS

JavaScript reads and runs the script from top to bottom.

JavaScript functions are executed in the sequence they are called. Not in the sequence they are defined.

... by default JavaScript is synchronous.

JS is Synchronous & Single-Threaded

```
function myFirst() {  
  console.log("Hello");  
}
```

```
function mySecond() {  
  console.log("Goodbye");  
}
```

```
mySecond();  
myFirst();
```

```
/* ----- Global Variables ----- */
let _users = [];
let _selectedUserId;

/* ----- */

async function fetchUsers() { ... }
function appendUsers(usersArray) { ... }
}

// ===== INIT APP =====

async function initApp() {
    await fetchUsers();
    appendUsers(_users);
}

initApp();
```

With callbacks, we can make JS Asynchronous

```
setTimeout(() => {
  console.log("Hey, I'm async!");
}, 3000);

btn.addEventListener('click', () => {
  alert("Hey, you clicked me!");
});
```

```
// fetch with callbacks
fetch("https://cederdorff.github.io/web-frontend/canvas")
  .then(function (response) {
    return response.json();
  })
  .then(function (data) {
    console.log(data);
  });
});
```

Fetch is Asynchronous

And it's about making HTTP requests in JavaScript.
... and a way to get & post data from and to a data source.

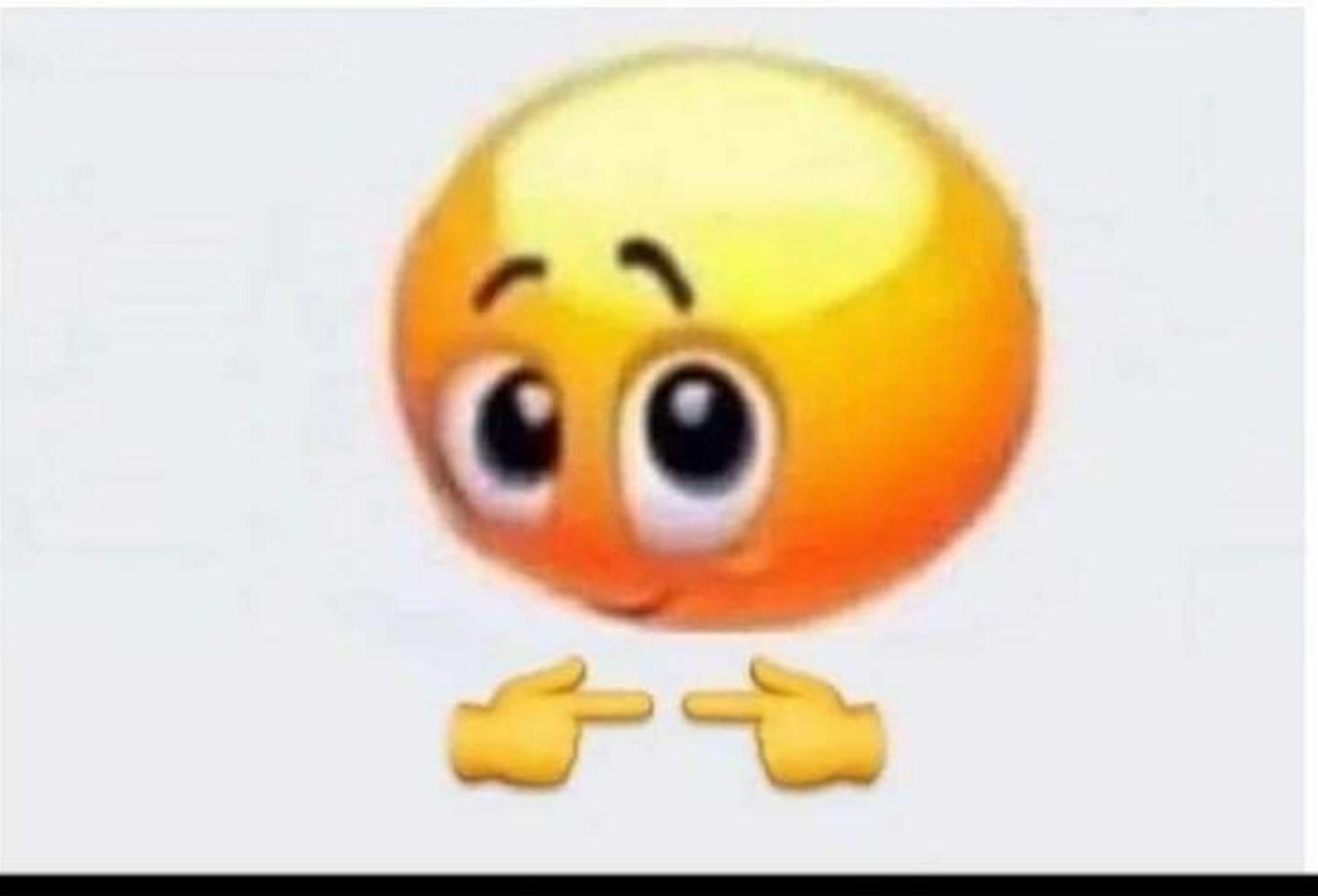
```
// fetch with callbacks
fetch("https://cederdorff.github.io/web-frontend/canvas-users/data.json")
  .then(function (response) {
    return response.json();
  })
  .then(function (data) {
    console.log(data);
  });

```

Fetch: callback (then) vs async/await

```
// fetch with callbacks
fetch("https://cederdorff.github.io/web-frontend/canvas-users/data.json")
  .then(function (response) {
    return response.json();
  })
  .then(function (data) {
    console.log(data);
  });
// 
// 
// or with async/await
const response = await fetch("https://cederdorff.github.io/web-frontend/canvas-users/data.json");
const data = await response.json();
console.log(data);
```

will you be async to my await?



async & await

- Use await to tell JS to wait for a fetch call to finish and to wait for JSON to parse.
- When using await you must tell JS that inside of the function goes some asynchronous code by wrapping it in an async function.

```
async function getPosts() {  
  const url = "https://raw.githubusercontent.com/.../data.json";  
  const response = await fetch(url);  
  const data = await response.json();  
  setPosts(data);  
}  
  
getPosts();
```

**"async and await makes promises
easier to write"**

async makes a function return a Promise

await makes a function wait for a Promise

https://www.w3schools.com/js/js_async.asp

Fetch returns a promise

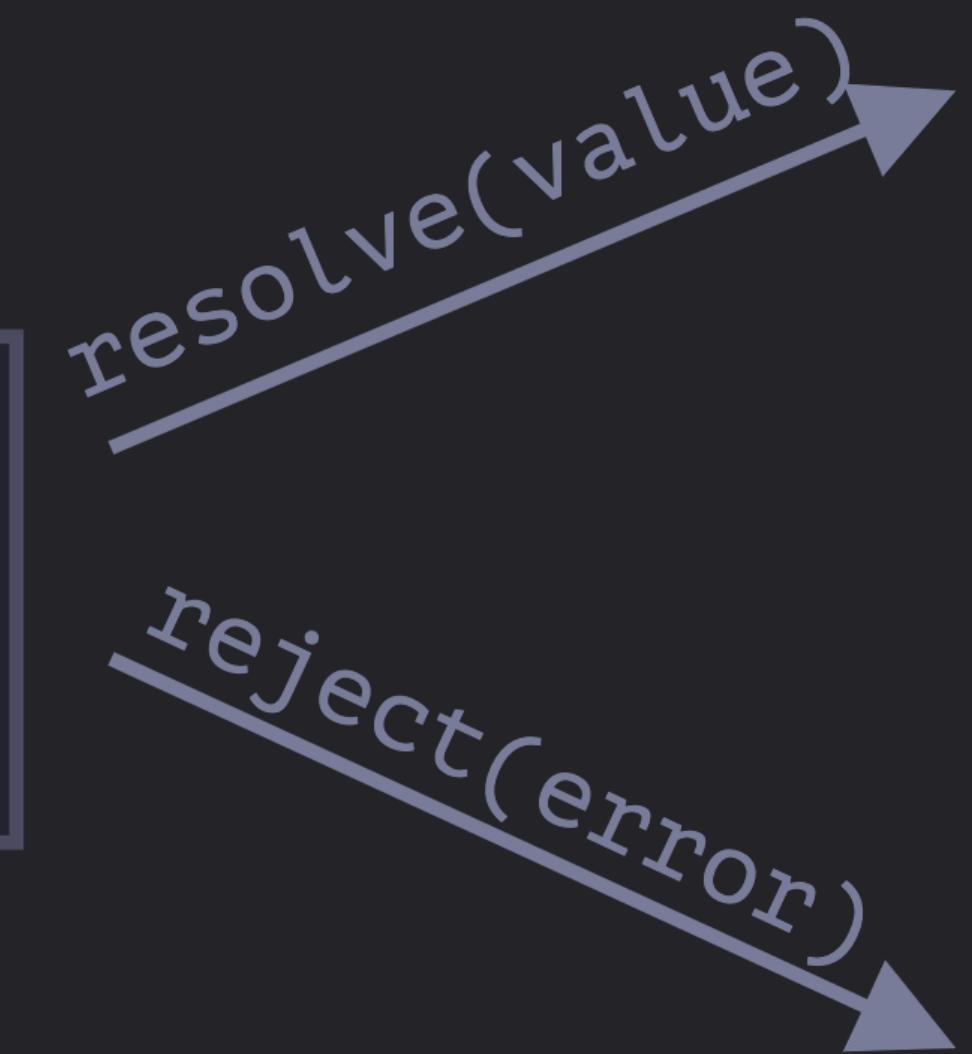
We can use `await` to wait for `fetch` to finish



https://www.w3schools.com/js/js_async.asp

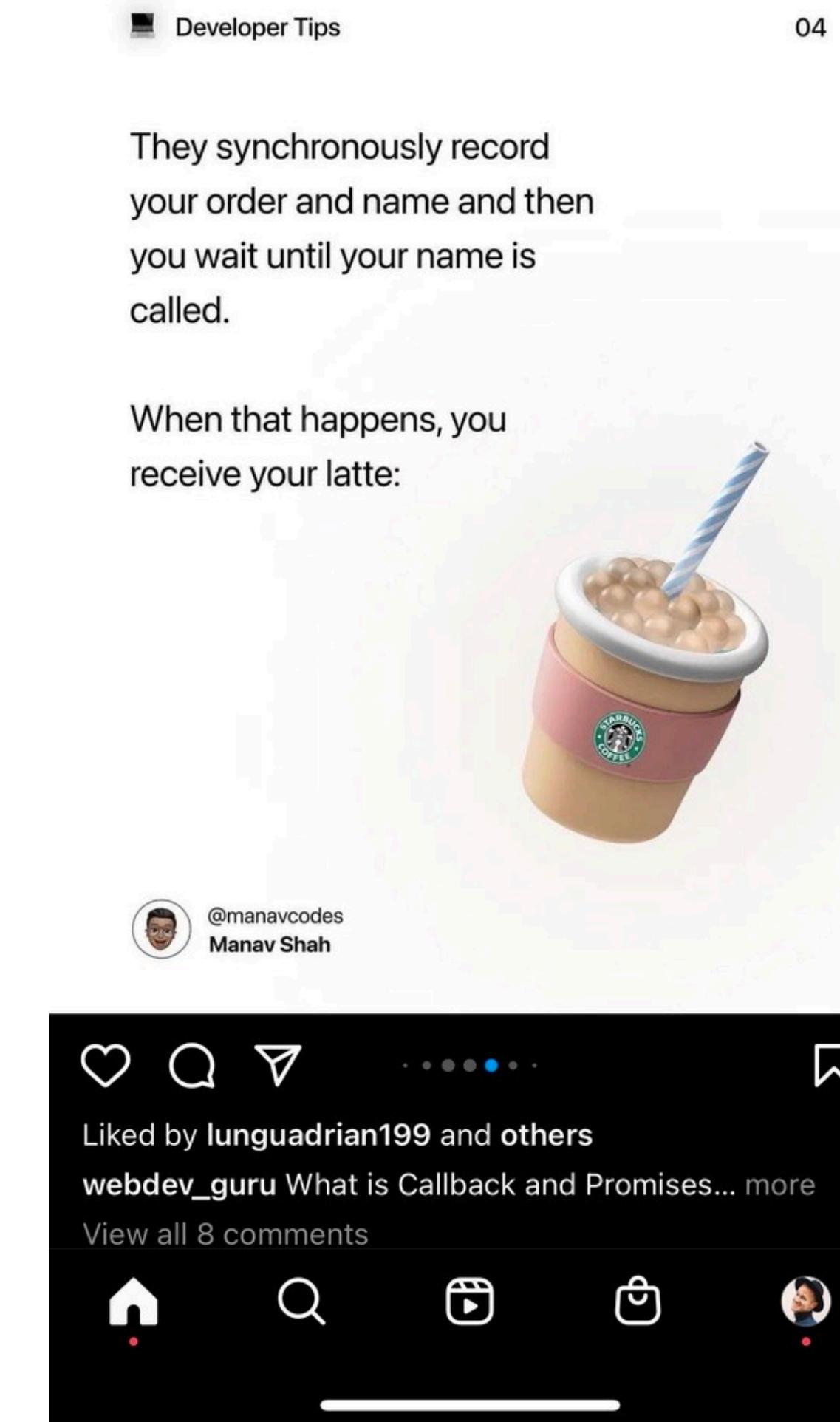
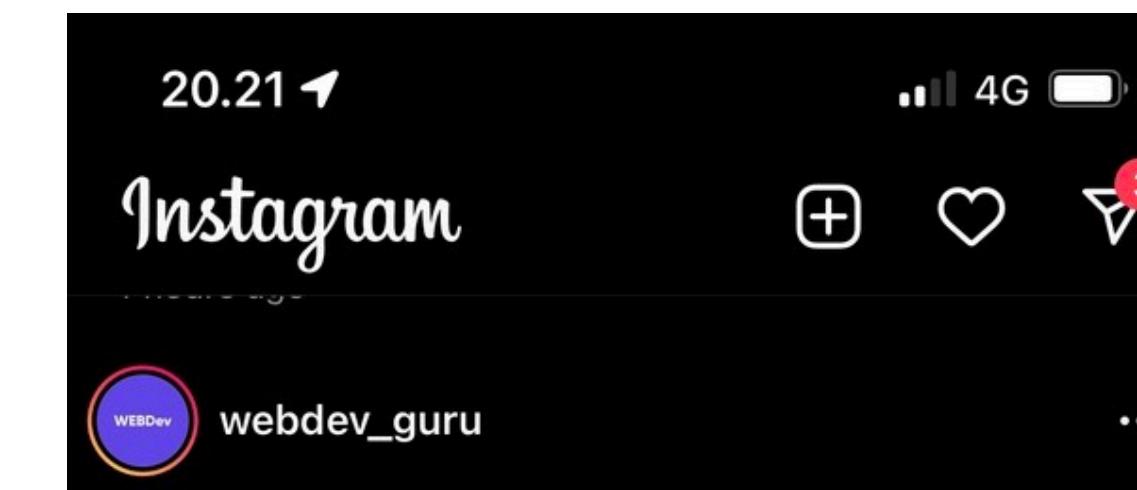
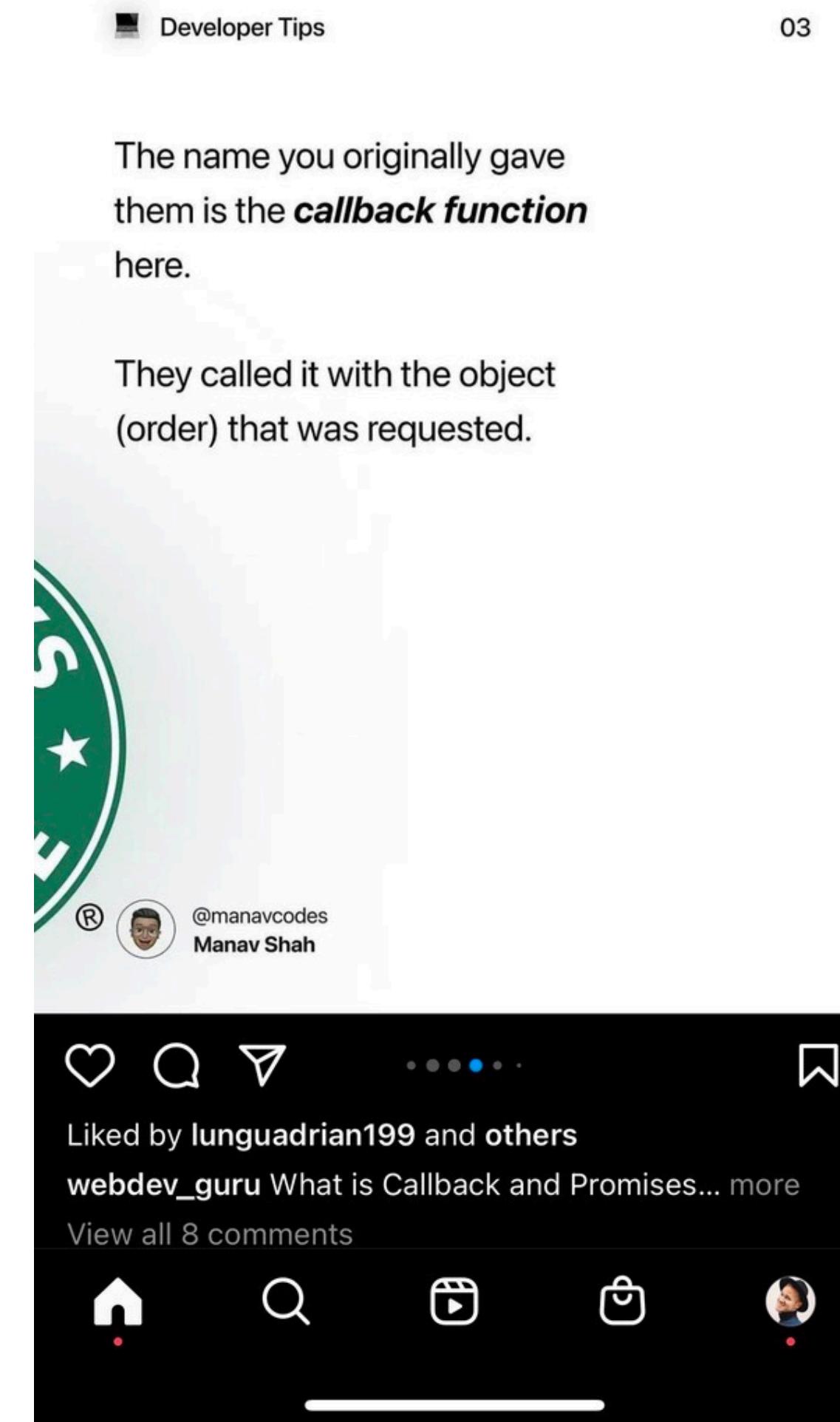
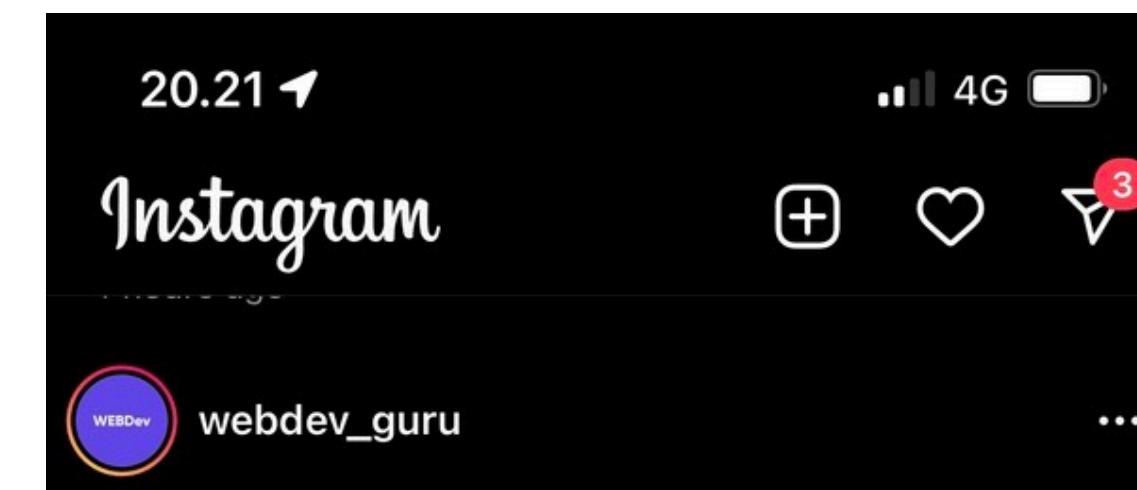
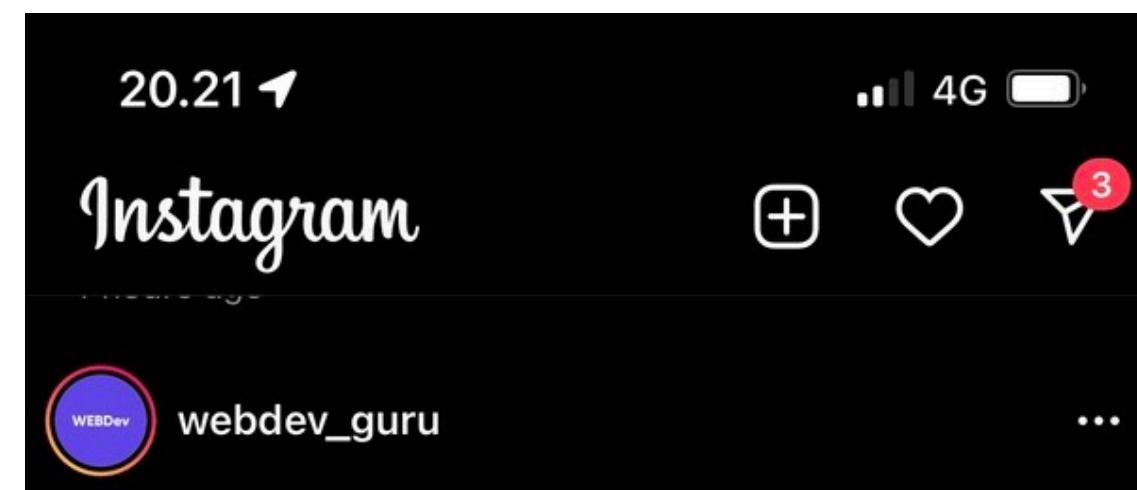
```
new Promise(executor)
```

state: "pending"
result: undefined



state: "fulfilled"
result: value

state: "rejected"
result: error



```
let myPromise = new Promise(function (resolve, reject) {
    // "Producing Code" (May take some time)
    // Making coffee ...
    resolve("Yaaay, here's your coffee"); // when successful
    reject("Ohh, f***. Something went wrong"); // when error
});

// "Consuming Code" (Must wait for a fulfilled Promise)
myPromise.then(
    function (value) {
        /* code if successful */
        console.log(value); // Yaaay, here's your coffee
    },
    function (error) {
        /* code if some error */
        console.log(error); // Ohh, f***. Something went wrong
    }
);
```

All you need to know

- Use await to tell JS to wait for a fetch call to finish.
- When using await you must tell JS that here goes some asynchronous code by wrapping it in an async function.

```
function PostsPage() {
  const [posts, setPosts] = useState([]);

  useEffect(() => {
    async function getPosts() {
      const url = "https://raw.githubusercontent.com";
      const response = await fetch(url);
      const data = await response.json();
      setPosts(data);
    }
    getPosts();
  }, []);

  return (
    <section className="page">
      <h1>Posts</h1>
      <section className="grid-container">
        {posts.map(post => (
          <PostItem post={post} key={post.id} />
        ))}
    </section>
  );
}
```

JSON

JavaScript Object Notation

JSON

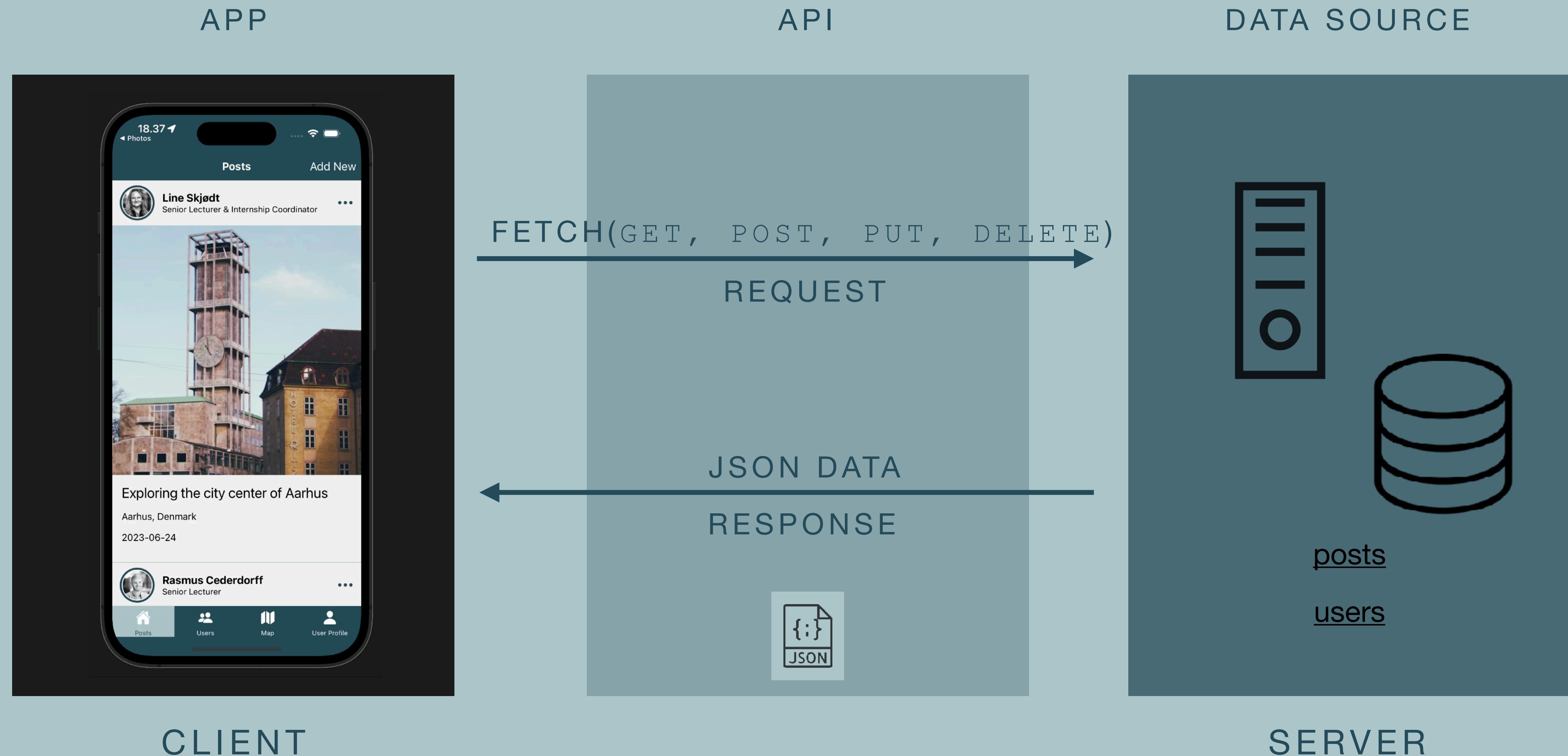
JavaScript Object Notation

... a syntax for storing & exchanging data
over the web



<https://www.instagram.com/p/CVqbCzgsZUF/>

JSON (& API) is the glue



JSON

... a syntax for storing and exchanging data over the web

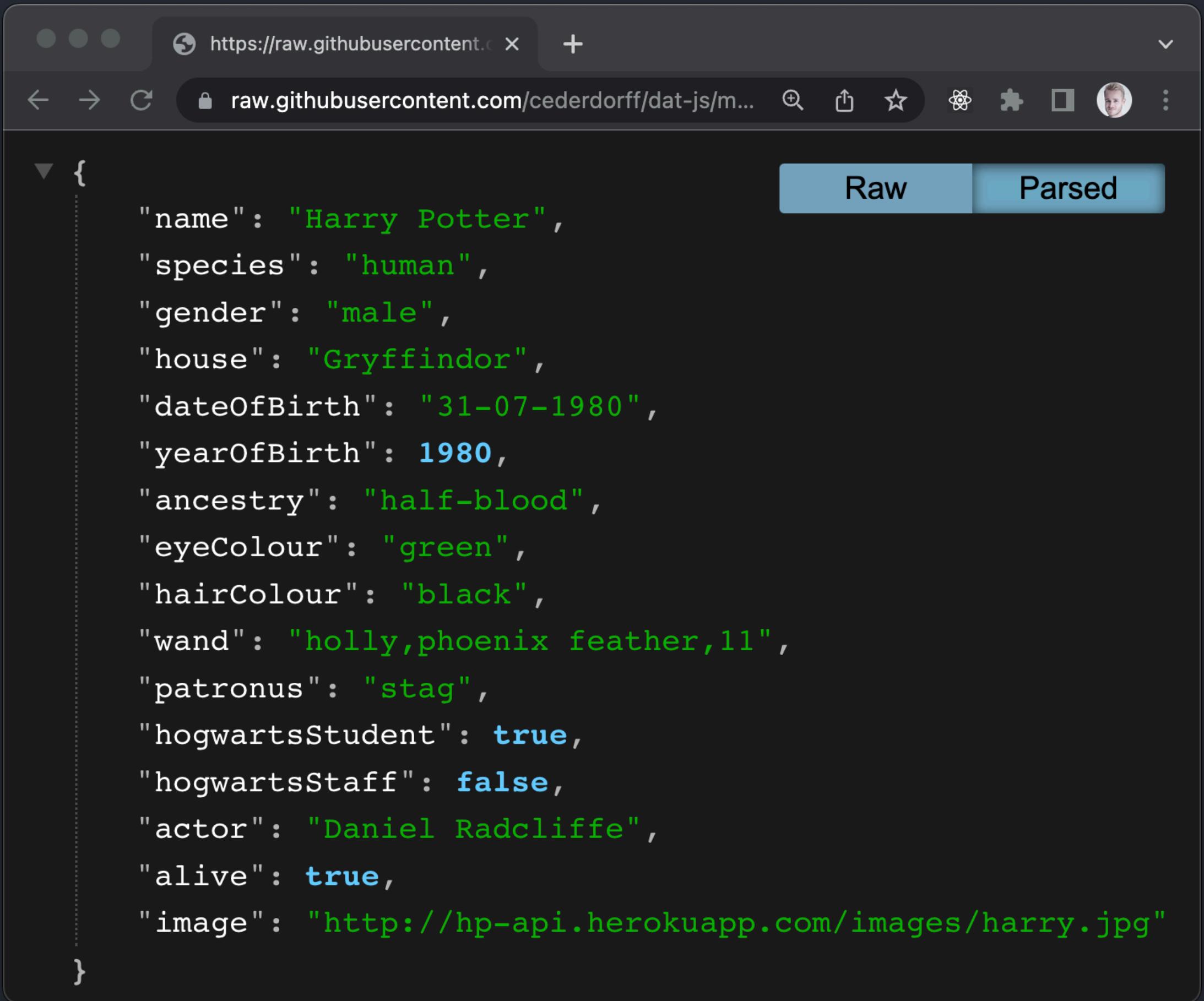
```
{  
  "name": "Alicia",  
  "age": 6  
}
```

JSON OBJECT

```
[{  
  "name": "Alicia",  
  "age": 6  
, {  
  "name": "Peter",  
  "age": 22  
}]
```

LIST OF JSON OBJECTS

JSON

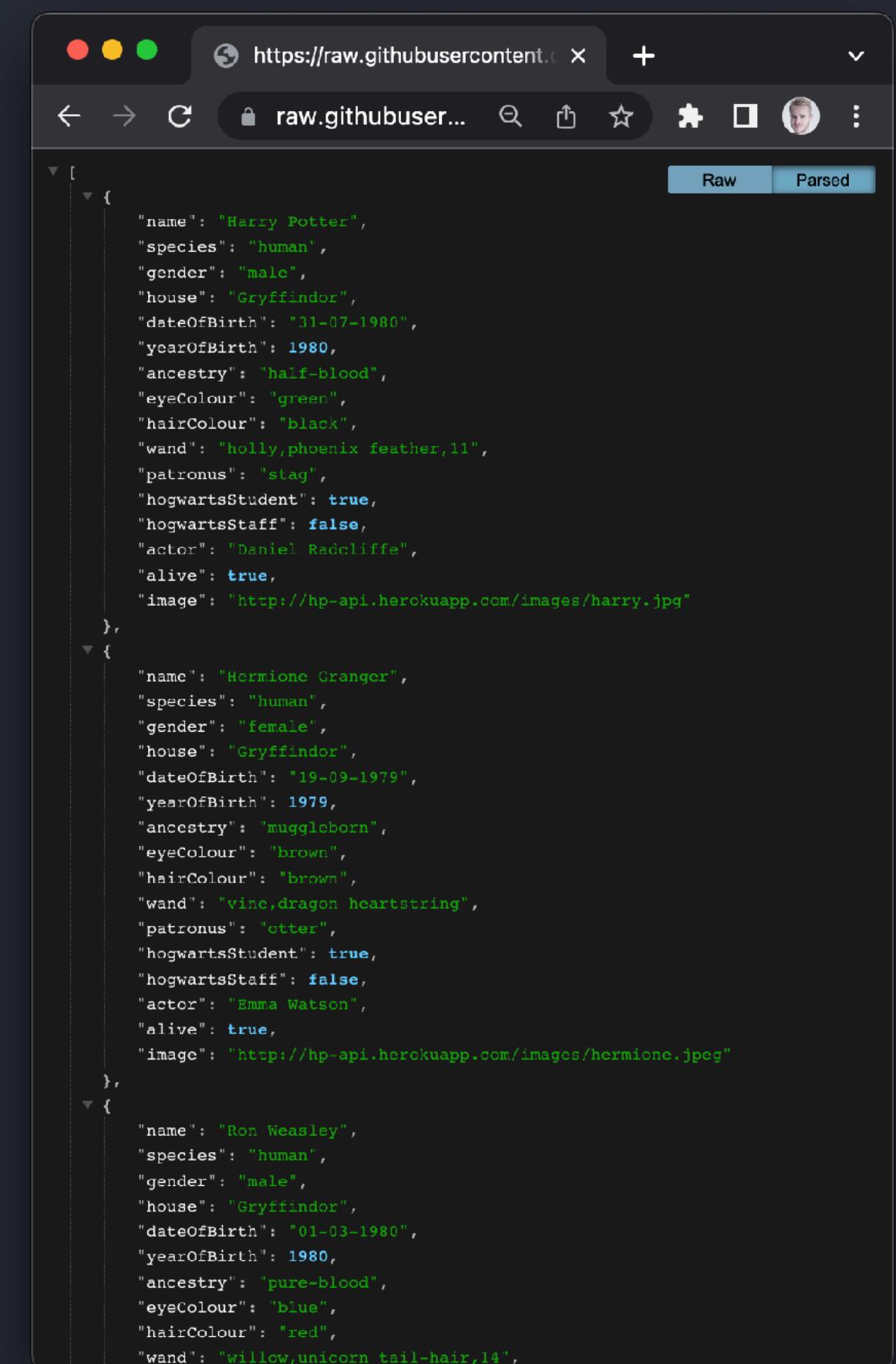


A screenshot of a web browser window displaying a single JSON object. The URL is <https://raw.githubusercontent.com/cederdorff/dat-js/master/data/harry.json>. The JSON structure is as follows:

```
{  
  "name": "Harry Potter",  
  "species": "human",  
  "gender": "male",  
  "house": "Gryffindor",  
  "dateOfBirth": "31-07-1980",  
  "yearOfBirth": 1980,  
  "ancestry": "half-blood",  
  "eyeColour": "green",  
  "hairColour": "black",  
  "wand": "holly,phoenix feather,11",  
  "patronus": "stag",  
  "hogwartsStudent": true,  
  "hogwartsStaff": false,  
  "actor": "Daniel Radcliffe",  
  "alive": true,  
  "image": "http://hp-api.herokuapp.com/images/harry.jpg"  
}
```

The browser has two tabs: "Raw" and "Parsed". The "Raw" tab is selected.

JSON OBJECT



A screenshot of a web browser window displaying a list of three JSON objects. The URL is <https://raw.githubusercontent.com/cederdorff/dat-js/master/data/characters.json>. The JSON structure is as follows:

```
[  
  {  
    "name": "Harry Potter",  
    "species": "human",  
    "gender": "male",  
    "house": "Gryffindor",  
    "dateOfBirth": "31-07-1980",  
    "yearOfBirth": 1980,  
    "ancestry": "half-blood",  
    "eyeColour": "green",  
    "hairColour": "black",  
    "wand": "holly,phoenix feather,11",  
    "patronus": "stag",  
    "hogwartsStudent": true,  
    "hogwartsStaff": false,  
    "actor": "Daniel Radcliffe",  
    "alive": true,  
    "image": "http://hp-api.herokuapp.com/images/harry.jpg"  
  },  
  {  
    "name": "Hermione Granger",  
    "species": "human",  
    "gender": "female",  
    "house": "Gryffindor",  
    "dateOfBirth": "19-09-1979",  
    "yearOfBirth": 1979,  
    "ancestry": "muggleborn",  
    "eyeColour": "brown",  
    "hairColour": "brown",  
    "wand": "vine,dragon heartstring",  
    "patronus": "otter",  
    "hogwartsStudent": true,  
    "hogwartsStaff": false,  
    "actor": "Emma Watson",  
    "alive": true,  
    "image": "http://hp-api.herokuapp.com/images/hermione.jpeg"  
  },  
  {  
    "name": "Ron Weasley",  
    "species": "human",  
    "gender": "male",  
    "house": "Gryffindor",  
    "dateOfBirth": "01-03-1980",  
    "yearOfBirth": 1980,  
    "ancestry": "pure-blood",  
    "eyeColour": "blue",  
    "hairColour": "red",  
    "wand": "willow,unicorn tail-hair,14",  
  }  
]
```

The browser has two tabs: "Raw" and "Parsed". The "Parsed" tab is selected.

LIST OF JSON OBJECTS

JSON Object

The diagram illustrates the connection between a JSON object and a web application displaying Harry Potter characters.

Left Side: Web Application

A screenshot of a web browser window titled "Harry Potter Characters". The page displays three cards, each representing a character:

- Harry Potter**: Gryffindor. Shows a portrait of a young boy with glasses and a yellow and red striped tie.
- Hermione Granger**: Gryffindor. Shows a portrait of a girl with long brown hair and a yellow and red striped tie.
- Ron Weasley**: Gryffindor. Shows a portrait of a boy with red hair and a yellow and red striped tie, holding a wand.

Right Side: JSON Object

A screenshot of a web browser window showing the JSON representation of the Harry Potter character data. The JSON is displayed in a collapsible tree view with the "Parsed" tab selected. The data for Harry Potter is as follows:

```
{  
  "name": "Harry Potter",  
  "species": "human",  
  "gender": "male",  
  "house": "Gryffindor",  
  "dateOfBirth": "31-07-1980",  
  "yearOfBirth": 1980,  
  "ancestry": "half-blood",  
  "eyeColour": "green",  
  "hairColour": "black",  
  "wand": "holly,phoenix feather,11",  
  "patronus": "stag",  
  "hogwartsStudent": true,  
  "hogwartsStaff": false,  
  "actor": "Daniel Radcliffe",  
  "alive": true,  
  "image": "http://hp-api.herokuapp.com/images/harry.jpg"  
}
```

Red arrows point from the character cards in the application to their corresponding entries in the JSON object, indicating the mapping between the visual representation and the data structure.

<https://raw.githubusercontent.com/cederdorff/dat-js/main/data/harry.json>

JSON Object

The diagram illustrates the flow of data between a JavaScript application and a browser's developer tools.

JavaScript Application (Left): A screenshot of a code editor showing a file named `app.js`. The code defines a function `showCharacter` that inserts an HTML article element into a container with the ID `#characters`. The article contains an image and two pieces of text: the character's name and house. The variable `character` is passed to this function.

```
28
29  function showCharacter(character) {
30      document.querySelector("#characters").insertAdjacentHTML(
31          "beforeend",
32          /*html*/
33          `

34              
35              <h2>${character.name}</h2>
36              <p>${character.house}</p>
37          </article>
38      `);
39  }
40 }


```

Browser Console (Right): A screenshot of a browser window displaying a JSON object. The object represents a character named Harry Potter, detailing his species, gender, house, birth dates, ancestry, eye and hair colors, wand, patronus, and current status. An arrow points from the `character` variable in the `showCharacter` function to this JSON object, indicating it is the data being passed.

```
{
  "name": "Harry Potter",
  "species": "human",
  "gender": "male",
  "house": "Gryffindor",
  "dateOfBirth": "31-07-1980",
  "yearOfBirth": 1980,
  "ancestry": "half-blood",
  "eyeColour": "green",
  "hairColour": "black",
  "wand": "holly,phoenix feather,11",
  "patronus": "stag",
  "hogwartsStudent": true,
  "hogwartsStaff": false,
  "actor": "Daniel Radcliffe",
  "alive": true,
  "image": "http://hp-api.herokuapp.com/images/harry.jpg"
}
```

<https://raw.githubusercontent.com/cederdorff/dat-js/main/data/harry.json>

JAVASCRIPT OBJECT NOTATION

- Collection of key-value pair: “key” : “value”
- List of values, collections or objects
- Lightweight data-interchange format
- Syntax / text format for storing and exchanging data over the web
- Human and machine readable **text**: small, fast and simple
- Language independent
- Can be parsed directly to JavaScript Object
- JavaScript Objects can be converted directly to JSON
- The glue between programs (interface between frontend and backend)

```
[  
 {  
   "id": "1",  
   "firstname": "Kasper",  
   "lastname": "Topp",  
   "age": "34",  
   "haircolor": "Dark Blonde",  
   "countryName": "Denmark",  
   "gender": "Male",  
   "lookingFor": "Female"  
 },  
 {  
   "id": "2",  
   "firstname": "Nicklas",  
   "lastname": "Andersen",  
   "age": "22",  
   "haircolor": "Brown",  
   "countryName": "Denmark",  
   "gender": "Male",  
   "lookingFor": "Female"  
 },  
 {  
   "id": "3",  
   "firstname": " Sarah",  
   "lastname": "Dybvad ",  
   "age": "34",  
   "haircolor": "Blonde",  
   "countryName": "Denmark",  
   "gender": "Female",  
   "lookingFor": "Male"  
 },  
 {  
   "id": "4",  
   "firstname": "Alex",  
   "lastname": "Hansen",  
   "age": "21",  
   "haircolor": "Blonde",  
   "countryName": "Denmark",  
   "gender": "Male",  
   "lookingFor": "Female"  
 }]
```

JSON METHODS

```
const user = {  
    name: "John",  
    age: 30,  
    gender: "male",  
    lookingFor: "female"  
};  
  
// === JSON.stringify === //  
const jsonUser = JSON.stringify(user);  
console.log(jsonUser); // {"name":"John","age":30,"gender":"male","lookingFor":"female"}  
  
// === JSON.parse === //  
const jsonString = '{"name":"John","age":30,"gender":"male","lookingFor":"female"}';  
const userObject = JSON.parse(jsonString);  
console.log(userObject); // logging userObject
```

CRUD App



Rasmus Cederdorff

Senior Lecturer
race@dev.dk

[UPDATE](#) [DELETE](#)



Lars Bogetoft

Head of Education
larb@eaaa.dk

[UPDATE](#) [DELETE](#)



Edith Terte

Lecturer
edan@kea.dk

[UPDATE](#) [DELETE](#)



Frederikke Bender

Head of Education
fbe@kea.dk

[UPDATE](#) [DELETE](#)



Murat Kilic

Senior Lecturer
mki@eaaa.dk

[UPDATE](#) [DELETE](#)



Anne Andersen

Head of Education
anki@mail.dk

[UPDATE](#) [DELETE](#)

Network

Preserve log Disable cache No throttling Invert Hide data URLs Hide extension URLs

All Doc JS Fetch/XHR CSS Font Img Media Manifest WS Wasm Other Blocked response cookies

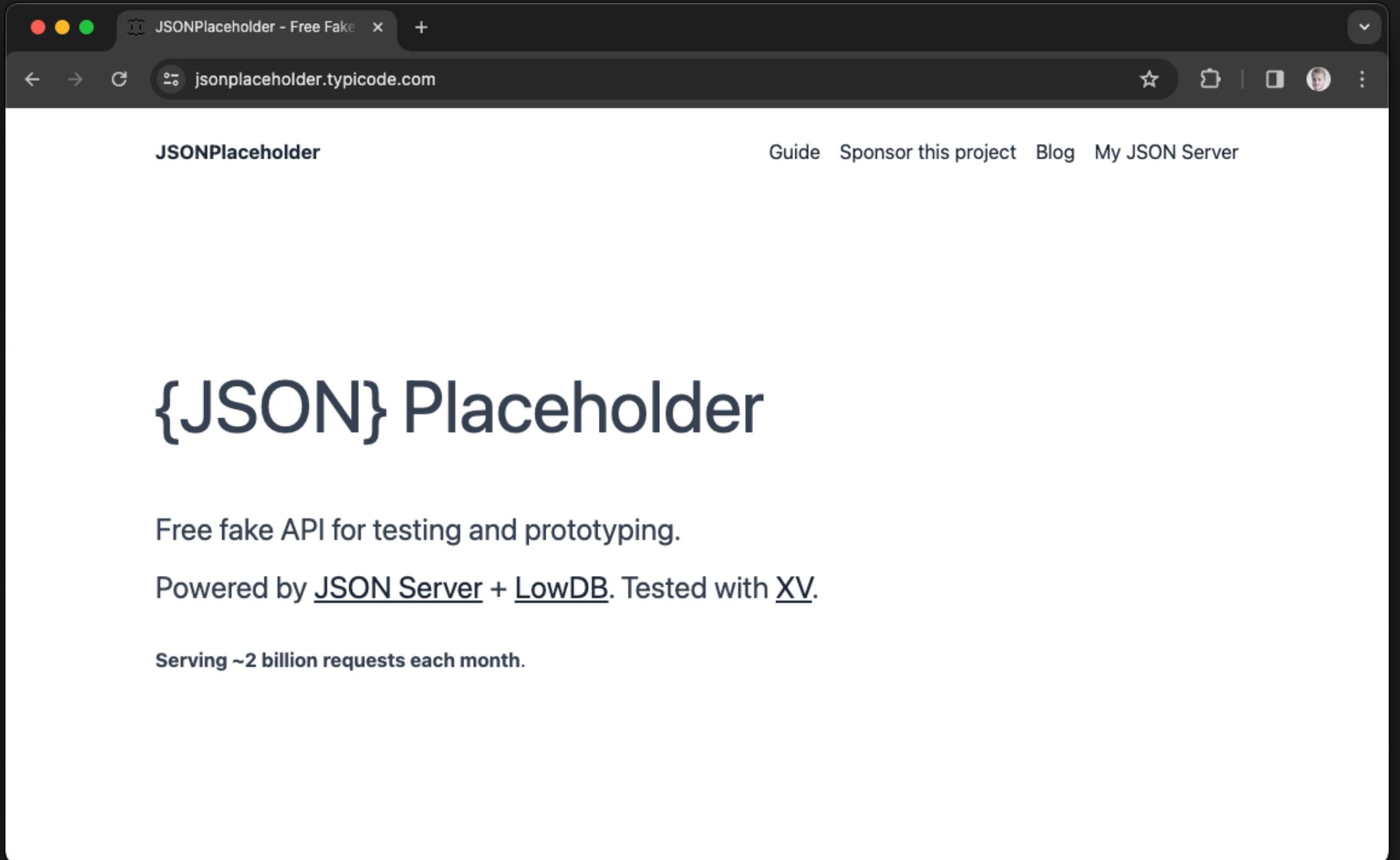
Blocked requests 3rd-party requests

10 ms 20 ms 30 ms 40 ms 50 ms 60 ms 70 ms 80 ms 90 ms 100 ms 110 ms

Name	X	Headers	Preview	Response	Initiator	Timing
users	1	[{ "id": 2, "name": "Rasmus Cederdorff", "mail": "race@dev.dk", "title": "Senior Lecturer", "image": "https://share.cederdorff.com/images/race.jpg" }, {id": 3, "name": "Lars Bogetoft", "mail": "larb@eaaa.dk", "title": "Head of Education", "image": "https://kea.dk/slir/w200-c1x1/images/user-profile/chefer/larb.jpg" }, {id": 4, "name": "Edith Terte", "mail": "edan@kea.dk", "title": "Lecturer", "image": "https://kea.dk/slir/w180-c1x1/images/user-profile/synced/EDAN.jpg" }, {id": 5, "name": "Frederikke Bender", "mail": "fbe@kea.dk", "title": "Head of Education", "image": "https://kea.dk/slir/w200-c1x1/images/user-profile/chefer/fbe.jpg" }, {id": 6, "name": "Murat Kilic", "mail": "mki@eaaa.dk", "title": "Senior Lecturer", "image": "https://www.eaaa.dk/media/llyavasj/murat-kilic.jpg?width=800&height=800" }, {id": 7, "name": "Anne Andersen", "mail": "anki@mail.dk", "title": "Head of Education", "image": "https://www.eaaa.dk/media/5buh1xeo/anne-kirketerp.jpg?width=800&height=800" }]			

1 / 11 requests | 21 { }

Test endpoints



<https://jsonplaceholder.typicode.com/>

- Install [JSON Formatter](#)
- Test the following endpoints:
 - <https://jsonplaceholder.typicode.com/posts/>
 - <https://jsonplaceholder.typicode.com/posts/5>
 - <https://jsonplaceholder.typicode.com/users/>
 - <https://jsonplaceholder.typicode.com/users/1>
 - <https://jsonplaceholder.typicode.com/todos>
 - <https://jsonplaceholder.typicode.com/todos/5>

```
{  
  "name": "Rasmus Cederdorff",  
  "birthday": "1990-03-12",  
  "title": "Experienced JavaScript Developer",  
  "experienceYears": 10,  
  "education": {  
    "degree": "Master of Science in IT - Web Communication",  
    "specialization": "Web Architecture"  
  },  
  "skills": ["JavaScript", "React", "Node.js", "UI/UX Design"],  
  "currentPosition": "Senior Lecturer at EAAA",  
  "teachingSubjects": ["Web Development", "JavaScript", "React", "BaaS & Node.js"],  
  "lovesJavaScript": true  
}
```

Define yourself with JSON

- <https://race.notion.site/Define-yourself-with-JSON-391944b6b1a041bfa341a0d46f5c0e4a?pvs=4>

REST

A set of architectural **rules**, **principles** or **constraints** of implementing an API.

REST

It is an architectural style for designing networked applications.

Here's a concise overview:

- **Key Principles:**
 - Stateless communication between client and server
 - Resources identified by URIs (Uniform Resource Identifiers)
 - CRUD operations (Create, Read, Update, Delete) performed using standard HTTP methods (GET, POST, PUT, DELETE)
 - Stateless nature simplifies scalability and enhances reliability
- **Data Formats:** Commonly uses JSON or XML for data representation
- **RESTful APIs:** APIs adhering to REST principles are often referred to as RESTful APIs.
- **Flexibility:** Supports a wide range of client types and is commonly used for web services.

Key principles of REST

These principles collectively contribute to the effectiveness, simplicity, and scalability of RESTful architecture in designing networked applications.

- **Statelessness:** Each request from a client to a server contains all the information needed for the server to understand and fulfill it. The server does not store any information about the client's state between requests.
- **Client-Server Architecture:** Separates the client and server responsibilities, enabling them to evolve independently. Clients are not concerned with data storage, while servers are not concerned with user interfaces.
- **Uniform Interface:** Defines a standardized way for clients to interact with resources. This includes resource identification through URLs, manipulation through representations, and self-descriptive messages.
- **Resource-Based:** Resources, identified by URLs, are the key abstractions. Each resource can be manipulated using standard HTTP methods (GET, POST, PUT, DELETE).
- **Representation:** Resources can have different representations (e.g., JSON or XML). Clients interact with these representations, and servers send data in the requested format.
- **Stateless Communication:** Stateless communication between the client and server enhances reliability and simplifies scalability. Each request from a client contains all the necessary information.
- **Cacheability:** Responses from the server can be explicitly marked as cacheable or non-cacheable, optimizing performance by reducing unnecessary data transfers.
- **Layered System:** Allows for the use of intermediaries, such as proxies and gateways, to enhance scalability, security, and performance without affecting the overall system.

What's REST?

GET

POST

PUT

DELETE

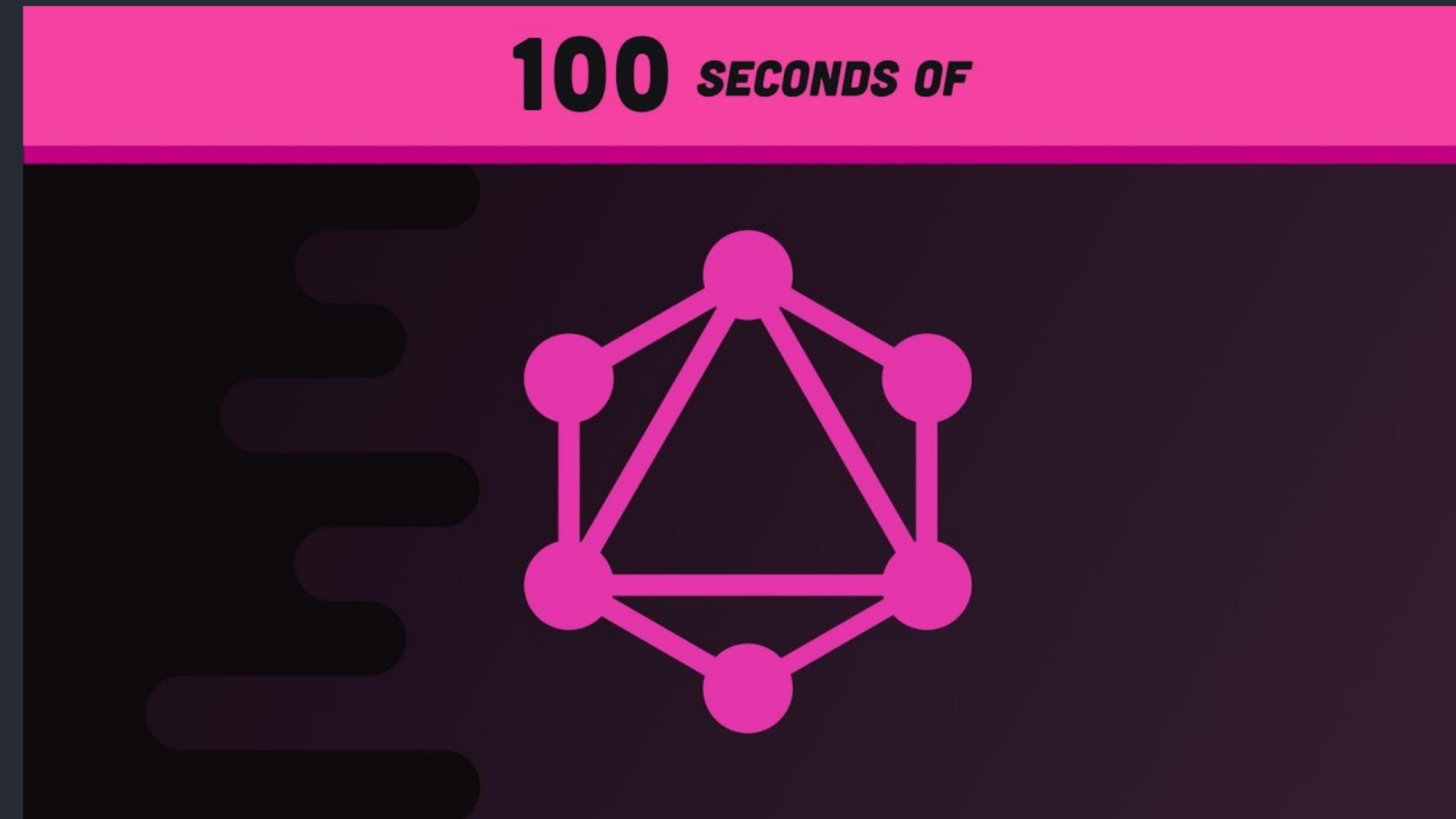
- REpresentational State Transfer
- A standard for systems (client & server) to communicate over HTTP to retrieve or modify (data) resources.
- Stateless, meaning the two systems don't need to know anything about the state.
- The client makes the requests using the 4 basic HTTP verbs to define the operation.

100 *SECONDS OF* node



https://www.youtube.com/watch?v=-MTSQjw5DrM&ab_channel=Fireship

GraphQL - REST Alternative



https://www.youtube.com/watch?v=elQh02xuVw4&ab_channel=Fireship

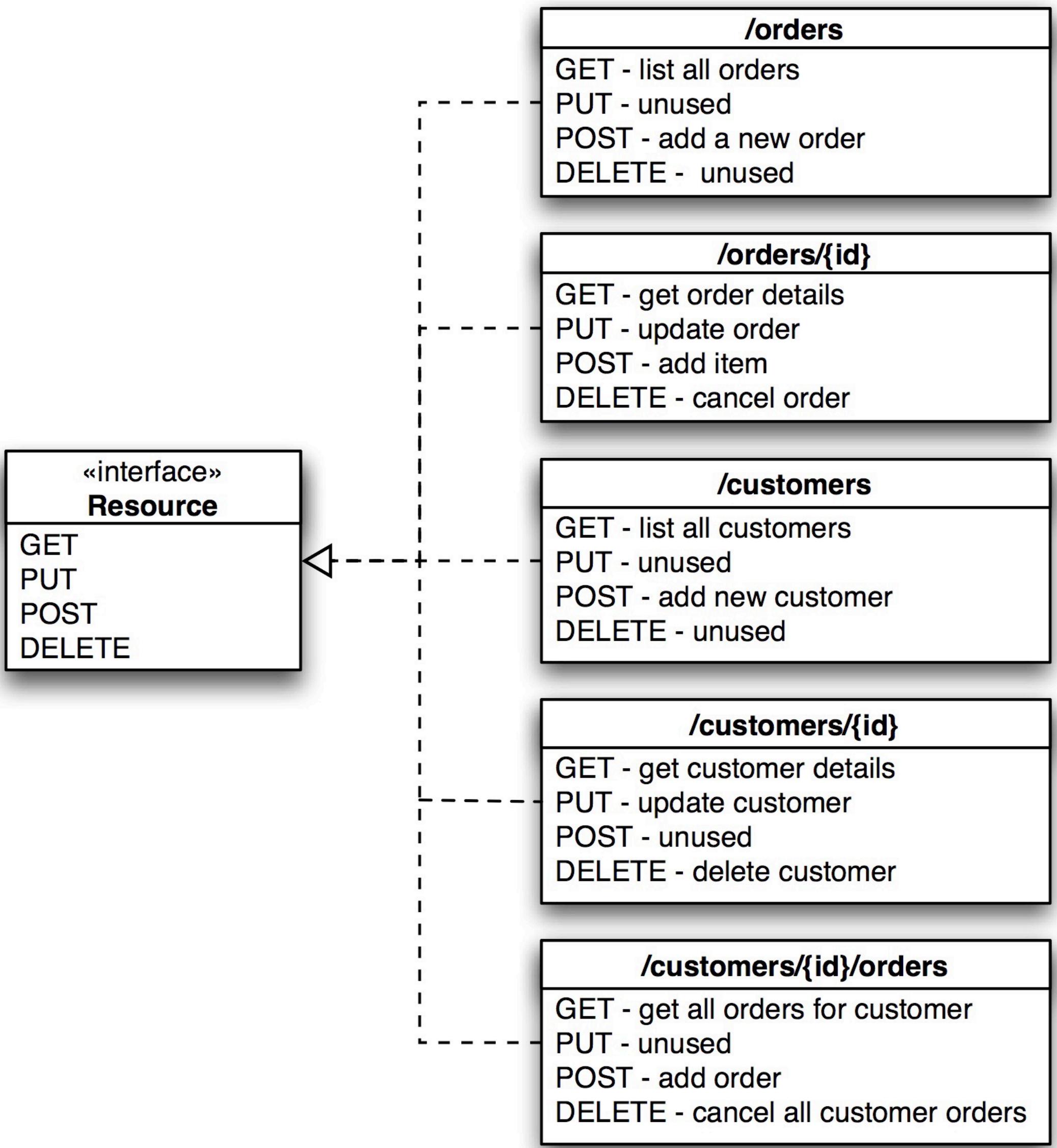
What are REST APIs?

<https://www.youtube.com/watch?v=SLwpqD8n3d0>

REpresentational State Transfer

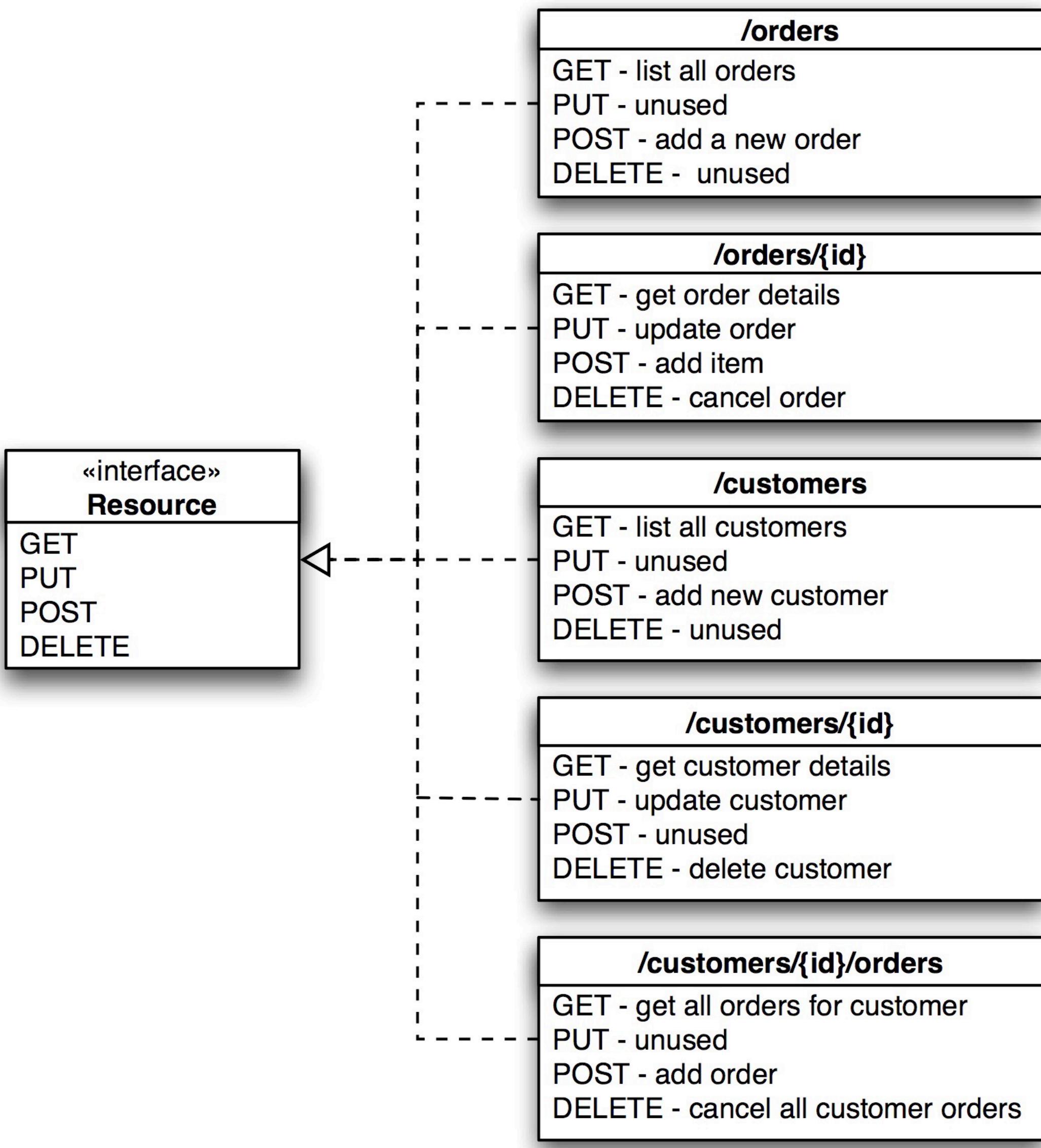
- Clean, semantic URL:
 - <http://example.com/products/2/25> instead of:
 - [http://example.com/products?
category=2&id=25](http://example.com/products?category=2&id=25)
- Basic HTTP requests to perform create, read, update and delete with the HTTP methods GET, POST, PUT and DELETE

Principles of REST



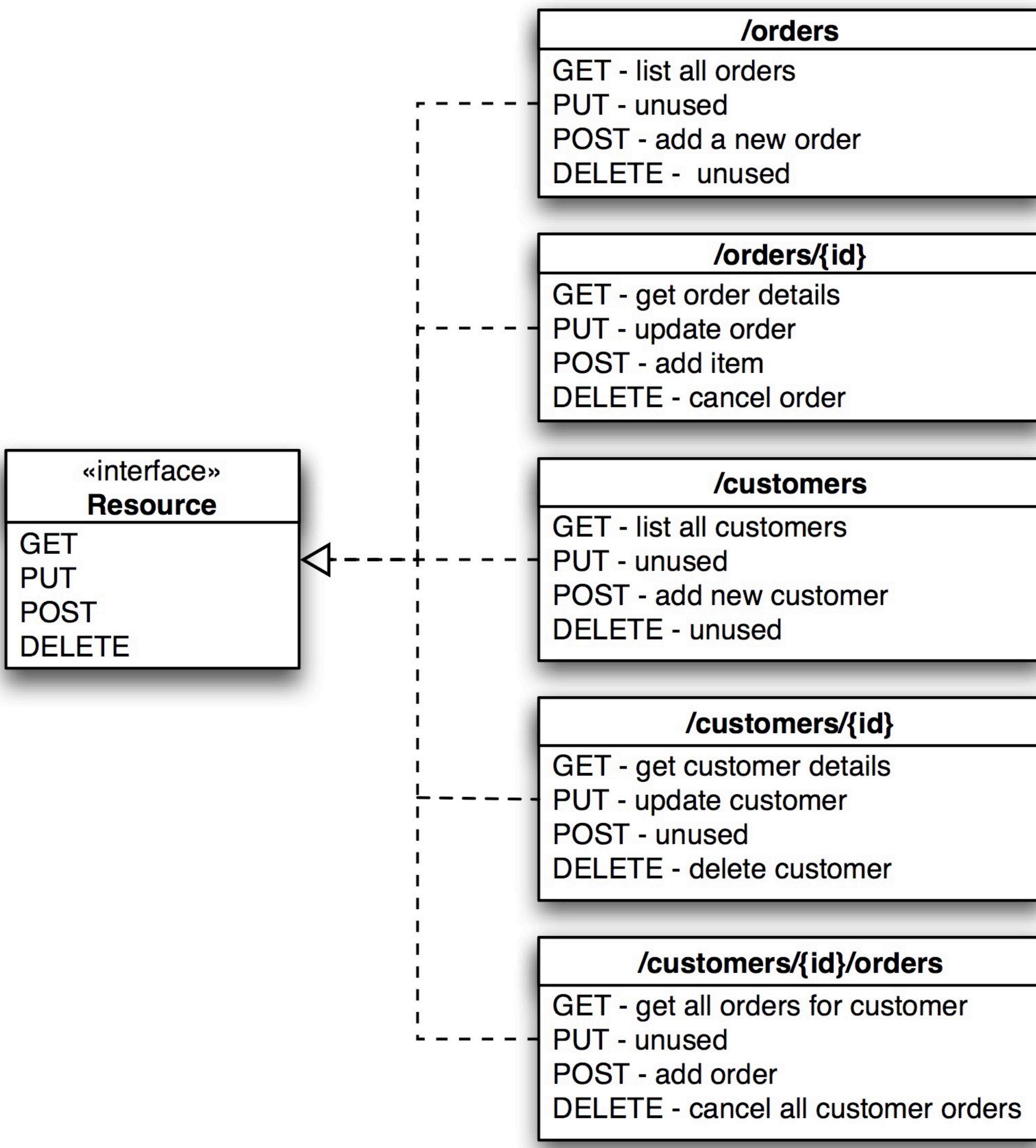
- REST defines a set of rules and guidelines on how you can interact with an API:
 - With REST, each piece of data in the database is treated as a resource. It has a unique id and URL. The URL structure represents the hierarchy of the data, allowing you to access specific data nodes.
 - You can use standard HTTP methods like GET, PUT, POST, PATCH, and DELETE to read, write, update, or delete that data.
 - REST promotes stateless communication, meaning that each request contains everything the server needs to process it.
 - The data exchanged with the API is typically in JSON format.

Principles of REST



- 1. Resource identification:** Each piece of data is treated as a resource and is identified by a unique URL (Uniform Resource Locator). The URL structure represents the hierarchy of the data, allowing you to access specific data nodes.
- 2. HTTP methods:** RESTful APIs utilize standard HTTP methods to perform operations on resources. The REST API supports the following methods:
 1. GET: Retrieves data from the specified endpoint.
 2. PUT: Replaces or updates data at the specified endpoint.
 3. POST: Appends data to a specified endpoint, generating a unique key.
 4. PATCH: Updates specific fields in the data at the specified endpoint.
 5. DELETE: Removes data at the specified endpoint.
- 3. Stateless communication:** Each request sent to the REST API contains all the necessary information for the server to process it. The API does not maintain any session or state information between requests. This statelessness allows for scalability and simplicity.
- 4. Data format:** The data exchanged is typically in JSON format. JSON provides a lightweight and flexible way to represent structured data.

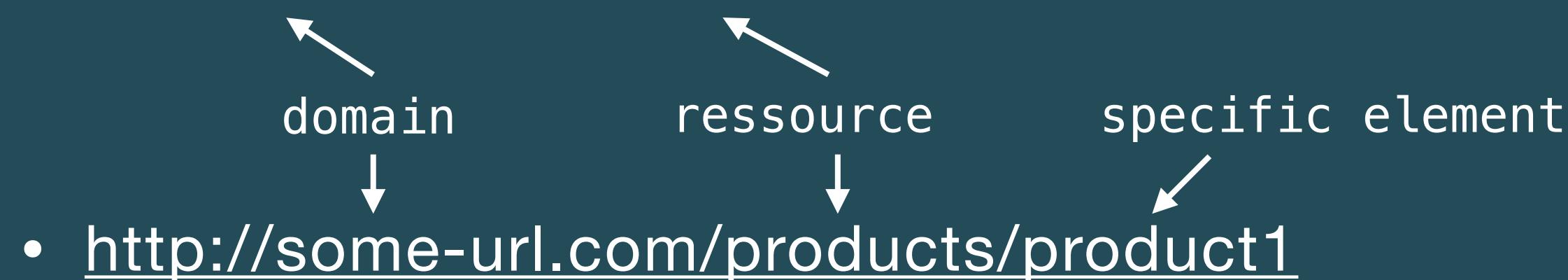
Principles of REST



- Ressources and endpoints:
 - <http://example.com/products>
 - <http://example.com/users>
 - <http://domain.com/orders>
- IDs:
 - <http://example.com/products/2/25>
 - <http://example.com/users/7503>
 - <http://domain.com/orders/2014/06/4022>
- Connect data
 - <http://example.com/users/7503/orders/3/item/1>
- Standard HTTP request methods (GET, POST, PUT, DELETE)
- Exchanges data, often JSON (oldschool: XML)
- Stateless Communication

RESTful API

- Base URL: http://some-url.com/products



- Data type → JSON

Ressource	GET	POST	PUT	DELETE
Collection: <u>http://some-url.com/products</u>	Returns a list with all products	Creates new product, added to the collections	Replaces a collection with a another	Deletes all products
Element: <u>http://some-url.com/products/product1</u>	Returns a specific product	÷	Replaces product with new (updated) data	Deletes product

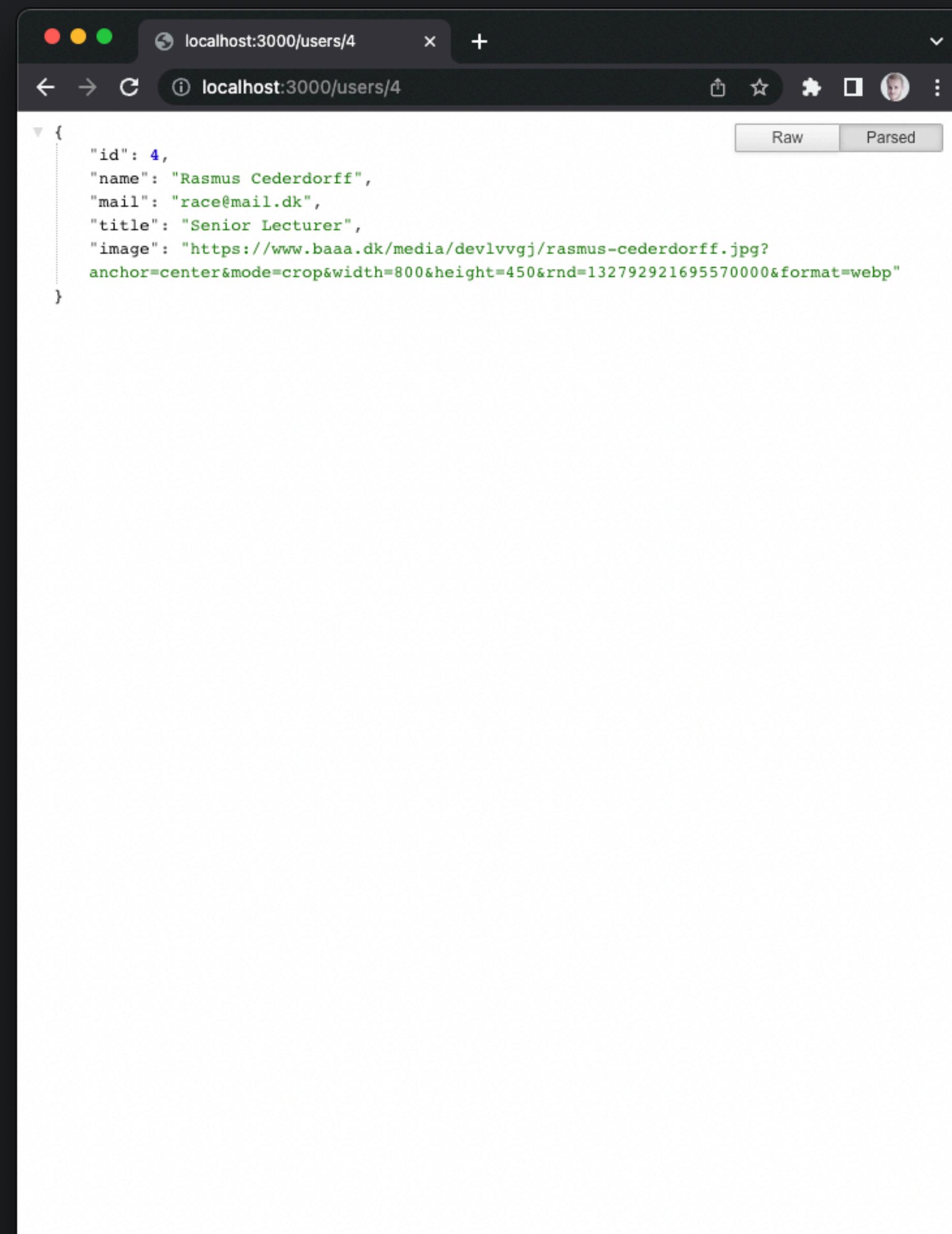
Collections (JSON Array)



A screenshot of a web browser window titled "localhost:3000/users". The page displays a JSON array of user objects. Each object contains fields: id, name, mail, title, and image. The image field includes a URL and a query string for image processing. The browser interface shows "Raw" and "Parsed" buttons.

```
[{"id": 1, "name": "Birgitte Kirk Iversen", "mail": "bki@mail.dk", "title": "Senior Lecturer", "image": "https://www.baaa.dk/media/u4gorzsd/birgitte-kirk-iversen2.jpg?anchor=center&mode=crop&width=800&height=450&rnd=132792921541630000&format=webp"}, {"id": 4, "name": "Rasmus Cederdorff", "mail": "race@mail.dk", "title": "Senior Lecturer", "image": "https://www.baaa.dk/media/devlvgj/rasmus-cederdorff.jpg?anchor=center&mode=crop&width=800&height=450&rnd=132792921695570000&format=webp"}, {"id": 5, "name": "Dan Okkels Brendstrup", "mail": "dob@mail.dk", "title": "Lecturer", "image": "https://www.eaaa.dk/media/bdojel41/dan-okkels-brendstrup.jpg?anchor=center&mode=crop&width=800&height=450&rnd=132792921559630000&format=webp"}, {"id": 6, "name": "Kasper Fischer Topp", "mail": "kato@mail.dk", "title": "Lecturer", "image": "https://www.eaaa.dk/media/lxzcybme/kasper-topp.jpg?anchor=center&mode=crop&width=800&height=450&rnd=132792921618200000&format=webp"}, {"id": 7, "name": "Line Skjødt", "mail": "lskj@mail.dk", "title": "Senior Lecturer & Internship Coordinator", "image": "https://www.eaaa.dk/media/14qpfeq4/line-skj%C3%B8dt.jpg?anchor=center&mode=crop&width=800&height=450&rnd=132792921638700000&format=webp"}, {"id": 8, "name": "Martin Aagaard Nøhr", "mail": "mnor@mail.dk", "title": "Lecturer", "image": "https://www.eaaa.dk/media/oayjq02h/martin-n%C3%B8hr.jpg?anchor=center&mode=crop&width=800&height=450&rnd=132792921658800000&format=webp"}]
```

Element (JSON Object)



A screenshot of a web browser window titled "localhost:3000/users/4". The page displays a single JSON object representing a user with id 4. The object includes fields: id, name, mail, title, and image. The image field includes a URL and a query string for image processing. The browser interface shows "Raw" and "Parsed" buttons.

```
{ "id": 4, "name": "Rasmus Cederdorff", "mail": "race@mail.dk", "title": "Senior Lecturer", "image": "https://www.baaa.dk/media/devlvgj/rasmus-cederdorff.jpg?anchor=center&mode=crop&width=800&height=450&rnd=132792921695570000&format=webp"}
```

Advantages of REST

- By following these REST principles, a REST API provides a simple and consistent way to interact with a database or other services. It allows developers to perform common operations using familiar HTTP methods, making it easier to integrate and interact with the database from different programming languages or platforms.
- It enables developers to perform CRUD (Create, Read, Update, Delete) operations on data resources using standard HTTP methods and benefit from the flexibility and interoperability that REST offers.

Advantages of REST

- Independent of platform and programming language
- Based on existing standards (on top of HTTP)
- Semantic URL → Nice and clean URLs → SEO
- Restful API
- Scalable
- Performance
- Exchange formats like JSON, XML, or both

Dataforsyningen - REST API

Why is it a REST API?



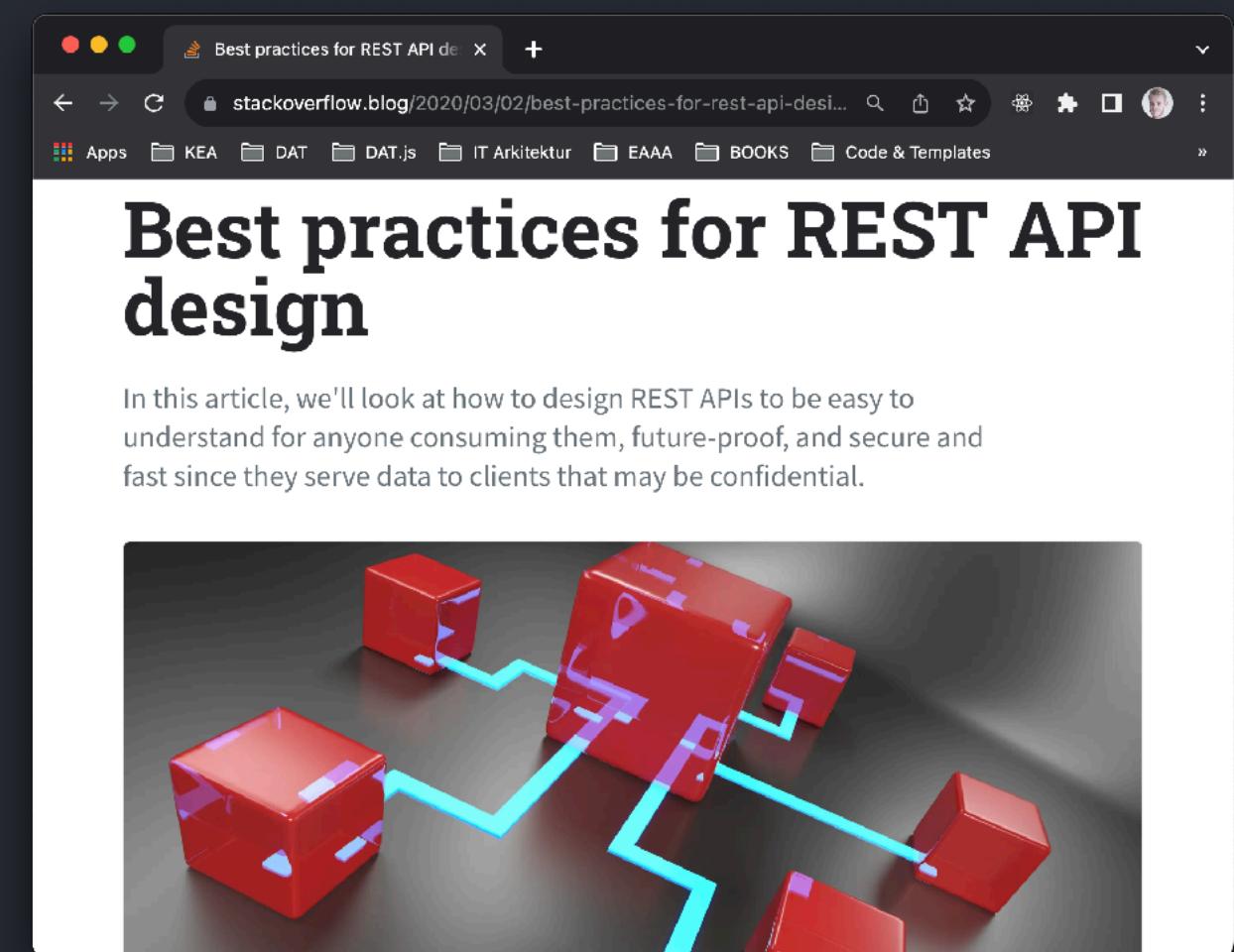
<https://dataforsyningen.dk/>

The screenshot shows the REST API documentation page. The title is "Dataforsyningen REST API". Below it, there are links to the API ([API](https://api.dataforsyningen.dk/)), documentation ([DOCS](https://dataforsyningen.dk/)), and a data overview ([Dataoversigt](https://dataforsyningen.dk/data)). A section titled "Eksempel på kald" (Example call) shows a link to "Danmarks adresser og vejnavne" (<https://dataforsyningen.dk/data/4729>) and a "Docs (REST Services)" link (<https://dawadocs.dataforsyningen.dk/dok/api>). Another section titled "Regioner" (Regions) shows a "alle regioner" (all regions) link with a GET method (<https://api.dataforsyningen.dk/regioner>) and an "en region" (one region) link with a GET method (<https://api.dataforsyningen.dk/regioner/1082>).

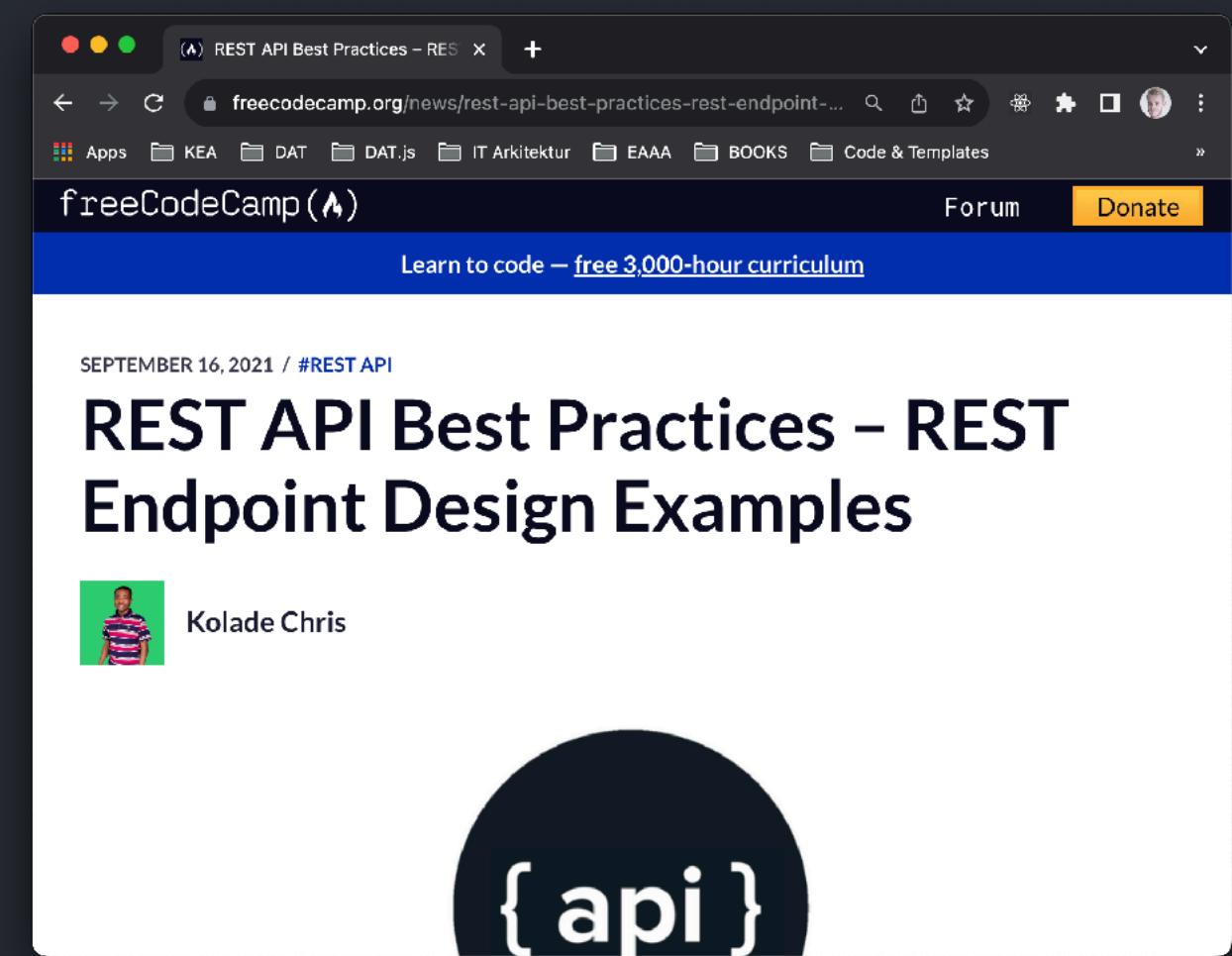
[Dataforsyningen REST API](#)

10 Best Practices for REST API Design

1. Data Format: Use JSON for Data Exchange
2. Endpoint Design: Use nouns, not verbs in endpoint paths
3. Resource Naming: Pluralize Collection Names
4. Error Handling: Utilize HTTP Status Codes & JSON-based error responses
5. Resource Relationships: Employ Nested Endpoints
6. Data Retrieval: Implement Filtering, Sorting, and Pagination
7. Security: Enforce SSL/TLS Encryption
8. Versioning: Clearly Define API Versions
9. Caching: Provide data caching to improve performance
10. Documentation: Provide Comprehensive API Docs



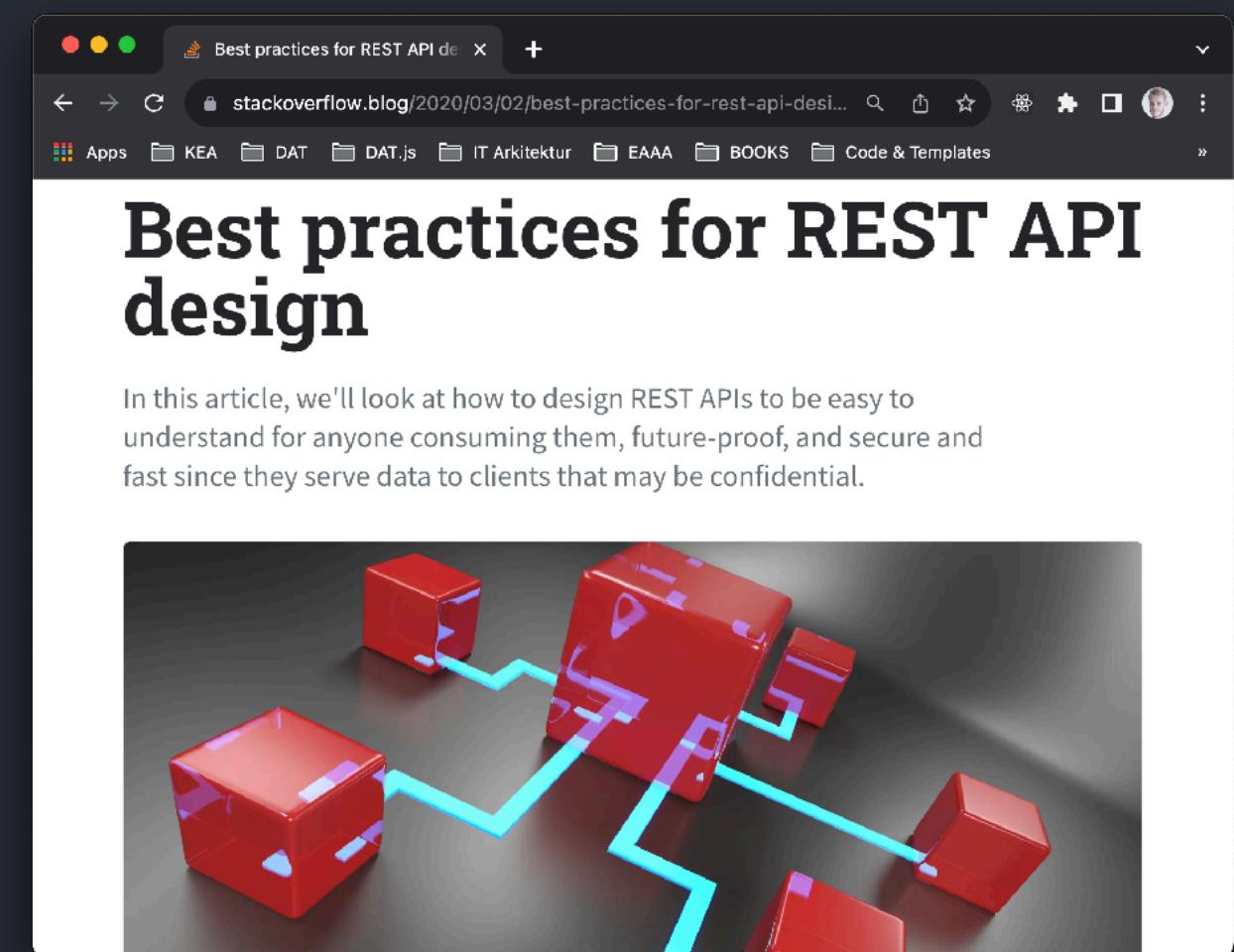
<https://stackoverflow.blog/2020/03/02/best-practices-for-rest-api-design/#h-cache-data-to-improve-performance>



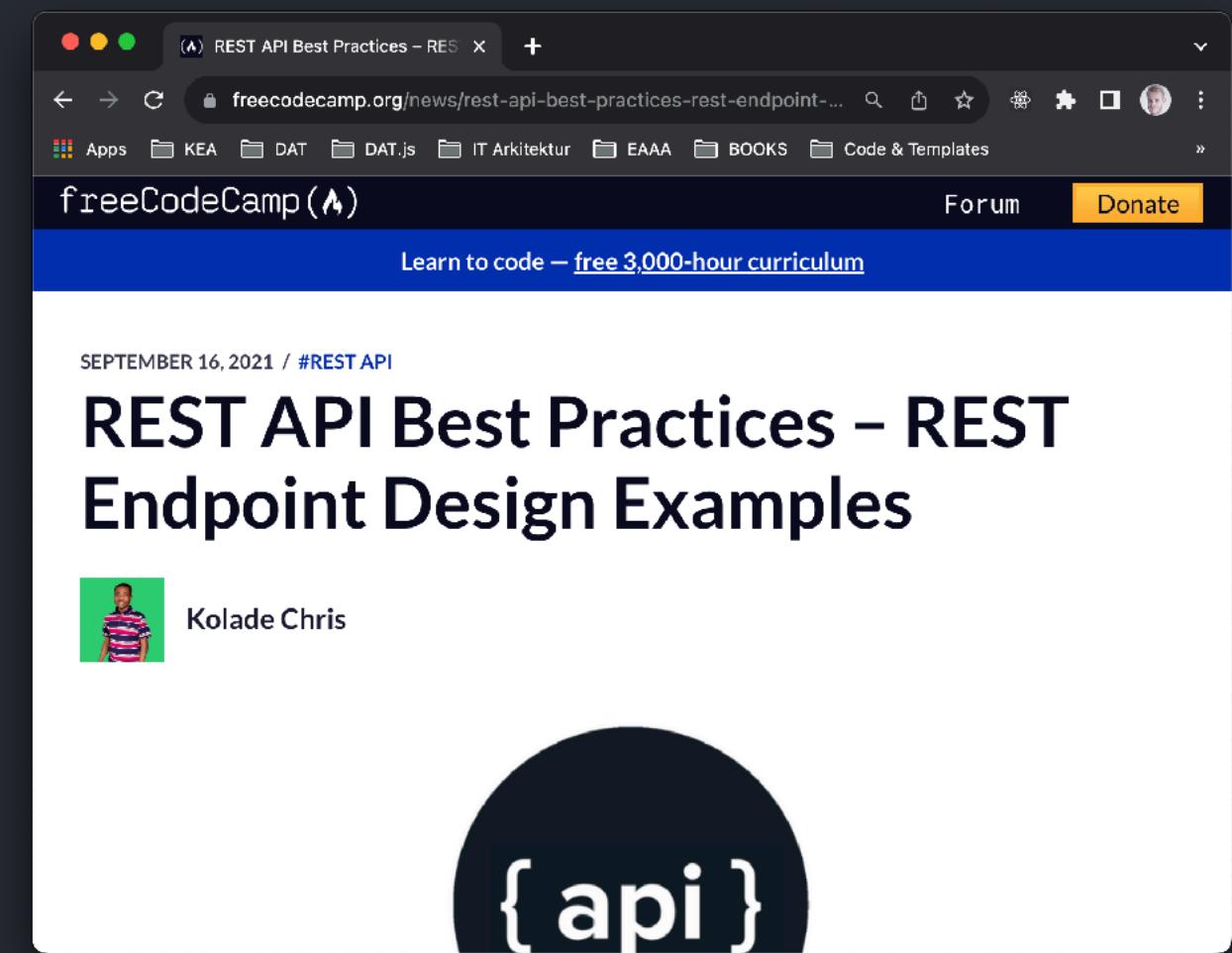
<https://www.freecodecamp.org/news/rest-api-best-practices-rest-endpoint-design-examples/>

Best Practices for REST API Design

- Based on the 10 Best Practices and the articles, improve your implementation of REST
 - Prioritize by
 1. Must-Have: 1, 2, 3 & 4
 2. Should-Have: 10, 6 & 5
 3. Could-Have: 7, 8 & 9
 - See Examples of Express.js implementation in [Best practices for REST API design](#)

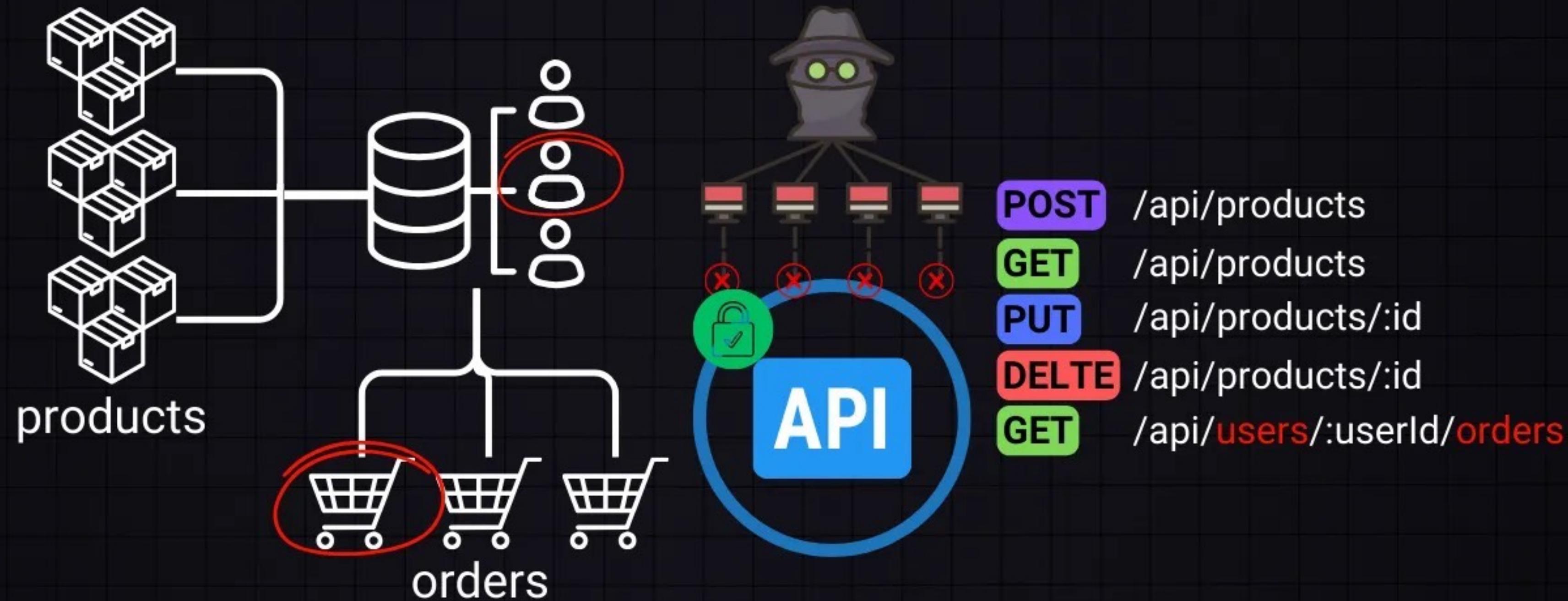


<https://stackoverflow.blog/2020/03/02/best-practices-for-rest-api-design/#h-cache-data-to-improve-performance>



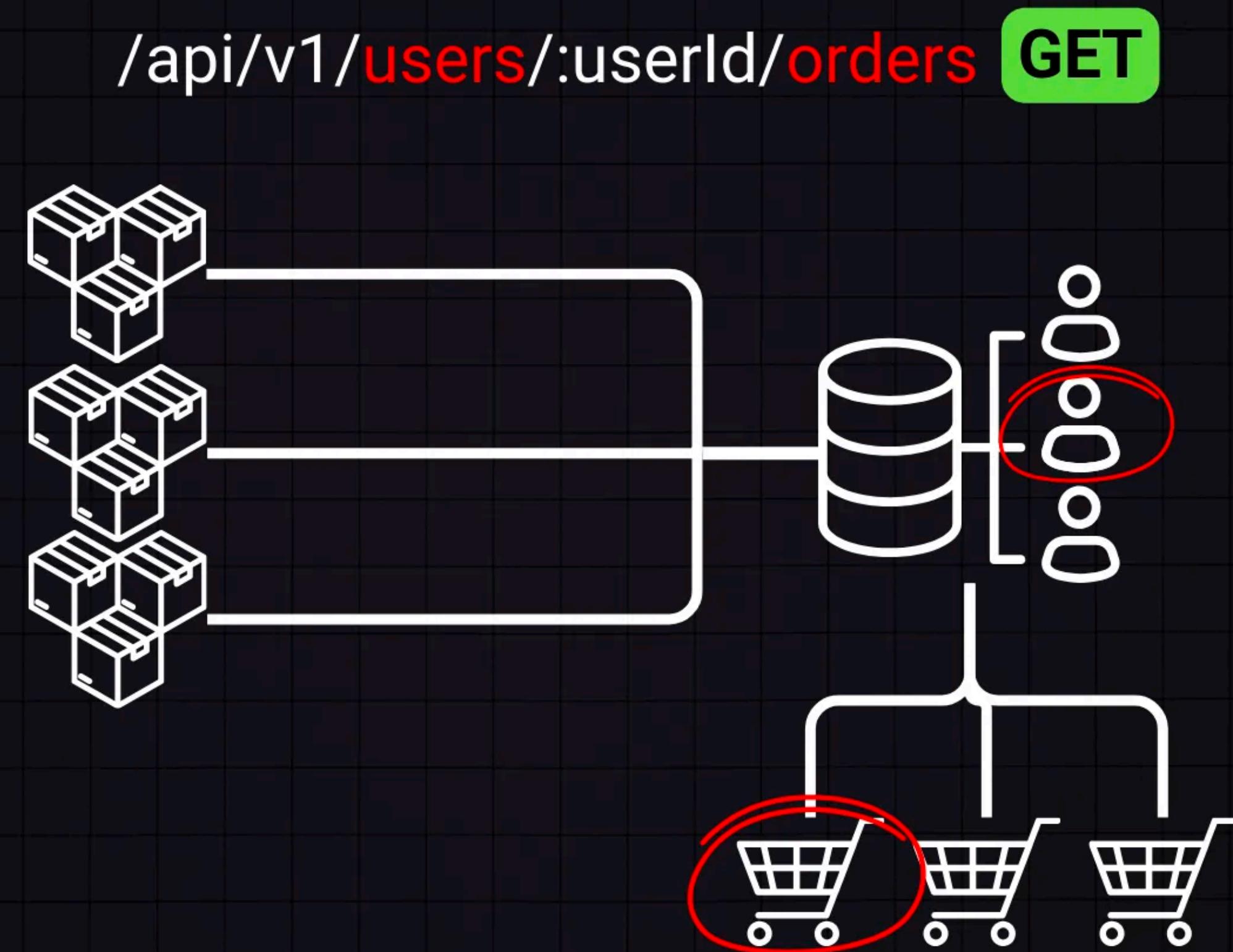
<https://www.freecodecamp.org/news/rest-api-best-practices-rest-endpoint-design-examples/>

API DESIGN



API Design 101: From Basics to Best Practices

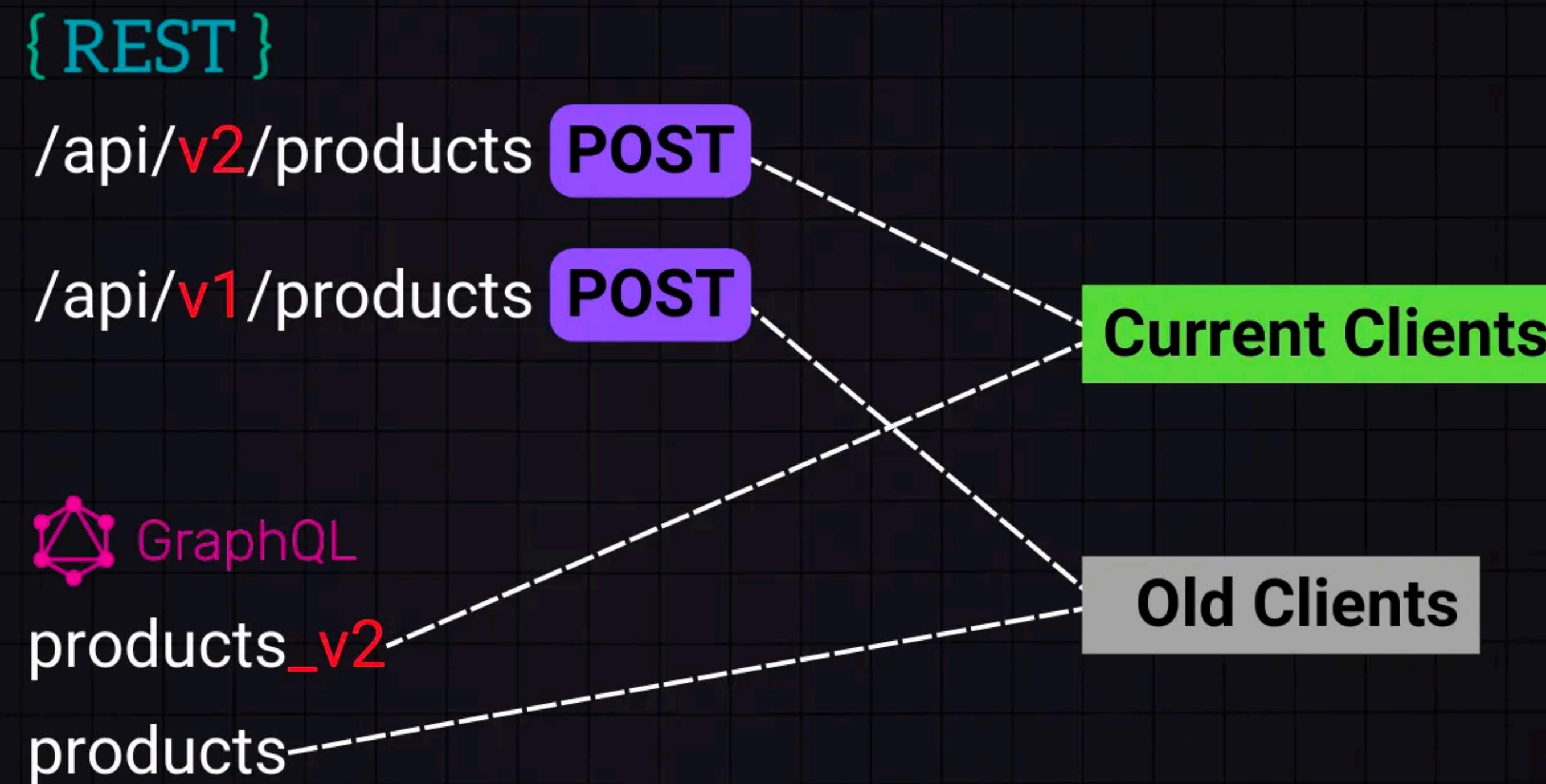
Relationships in API Design



Queries, Limit, and Idempotence of GET Requests



Backward Compatibility and Versioning



DOS & DON'TS

1. Let's just use POST for everything!
2. While we're at it...
3. Let's use PUT... What do you mean "404"!?
4. Everything is "OK"
5. Server: "400 Bad Request. Body: Empty..." — Client: "Thanks..."
6. "400 Bad Request. Body: Some random error" — Client: "But... this is a GET request"
7. 404 (Not Found) vs 410 (Gone)
8. Get your priorities right
9. "We want our API to be public, completely open, and free to use!"
— The Care Bears
10. "500 Internal Server Error" more like "500 Whoops, code broke"

The screenshot shows a Medium article titled "RESTful APIs. 10 DOs and DON'Ts" by Julien Garrigues. The article was published in Creative by Black Pug Studio on May 16, 2018, and has 163 views. The content discusses common REST API anti-patterns, such as using POST for everything or returning empty bodies for errors. It includes a quote from the Care Bears and a note about missing documentation.

RESTful APIs
10 DOs and DON'Ts
Julien Garrigues · Follow
Published in Creative by Black Pug Studio · 8 min read · May 16, 2018

In a world where RESTful APIs are used every day to feed our lives with more data than ever possible, it is necessary to understand how they work, or at least how they **SHOULD** work.

"Most APIs have unclear, incomplete, or simply missing documentation!"

As a developer, I was shocked to discover how few APIs actually follow the basic rules and guidelines in place. So, I thought I would show a few general DOs and DON'Ts based on live APIs that I've encountered or used in the past (without mentioning where they were found, no need for finger pointing of course!).

I won't go into extensive details about everything. In fact, I'll try to keep it short and straight to the point, and if you are interested after reading my

HTTP REQUEST METHODS (verbs)

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>

CRUD vs REST & HTTP Verbs

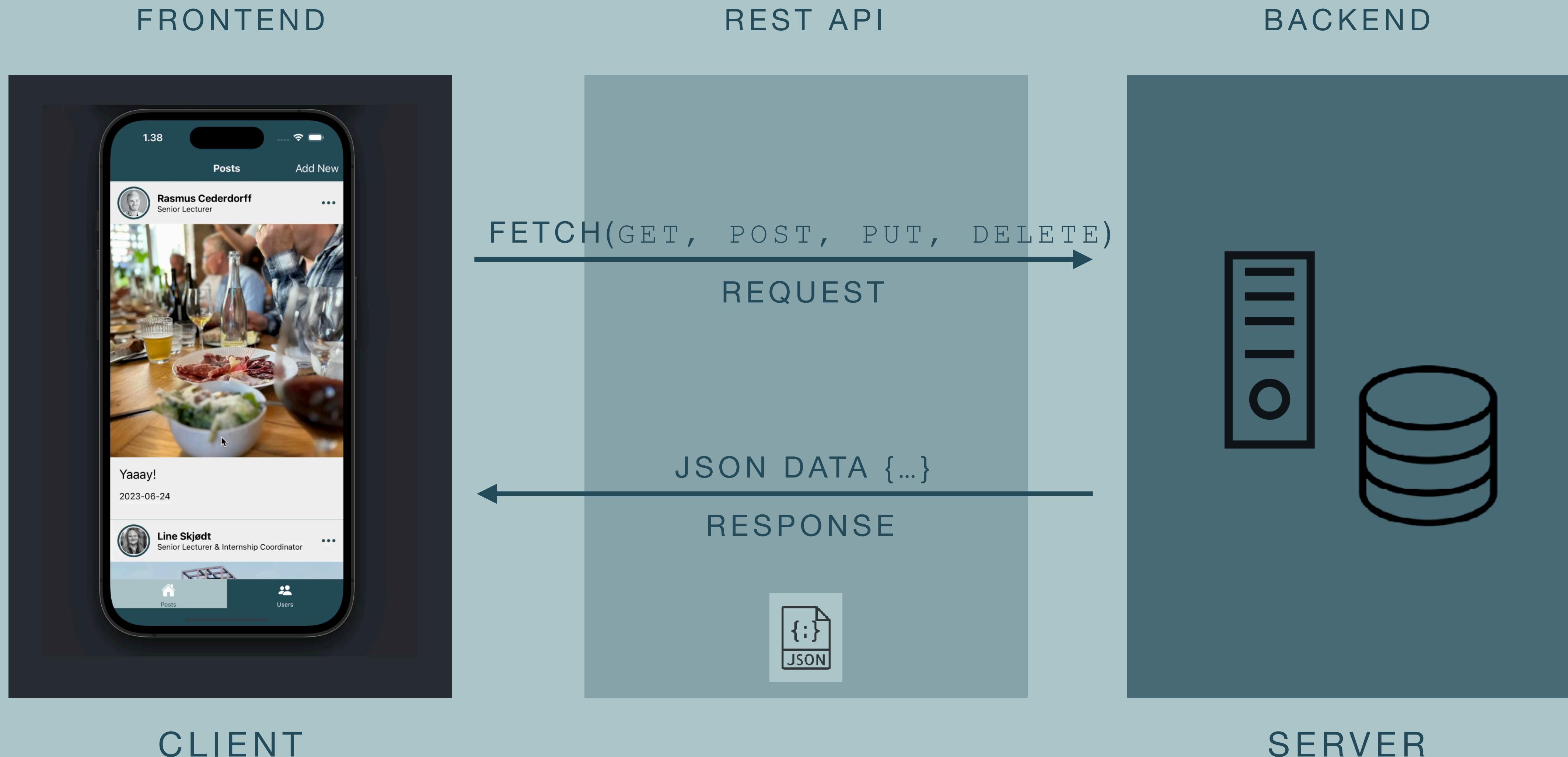
CREATE -> POST: create a new resource (object)

READ -> GET: retrieve a specific resource or a collection

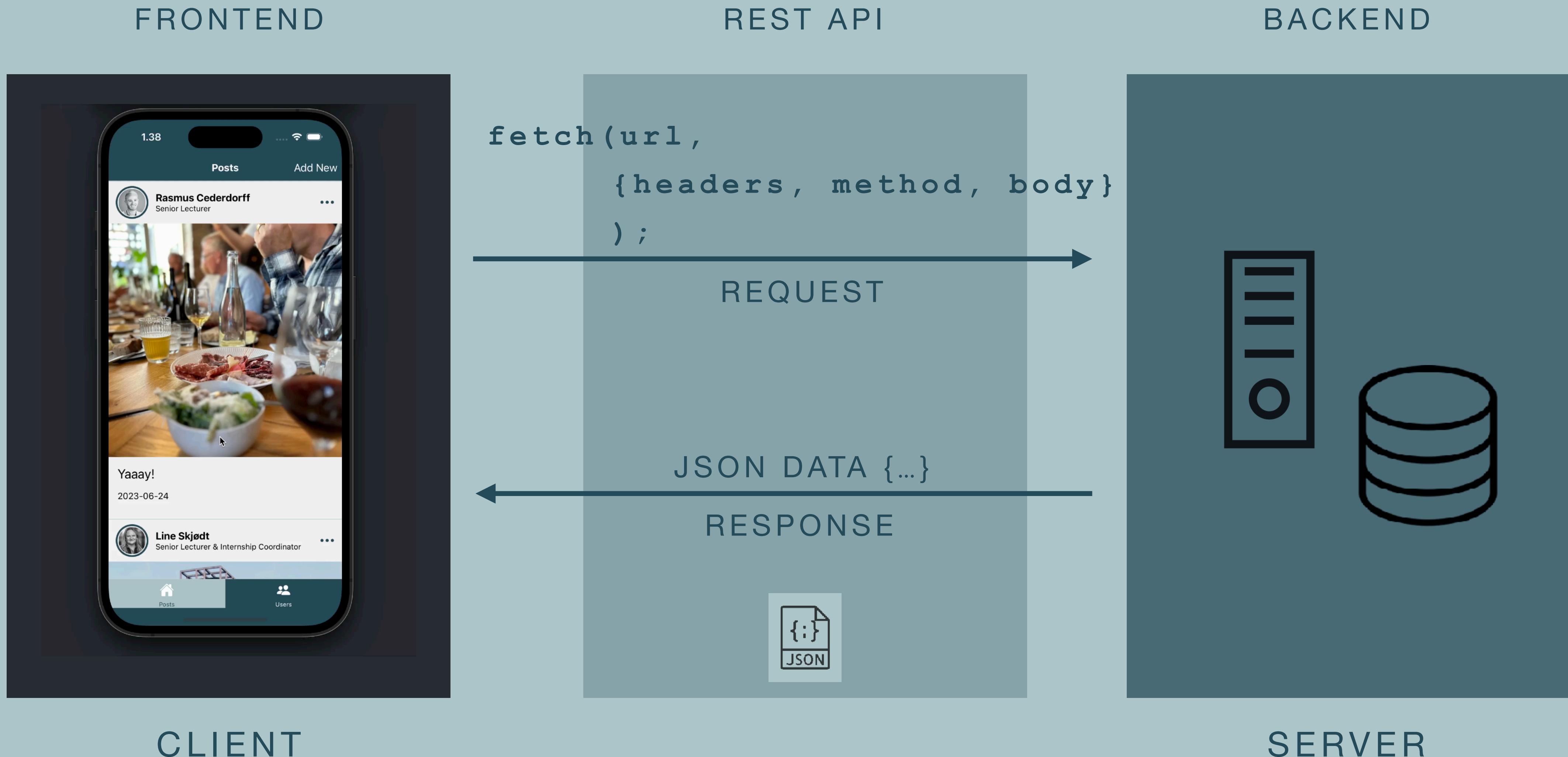
UPDATE -> PUT: update a specific resource (by id)

DELETE -> DELETE: remove a specific resource by id

HTTP Request & Response



HTTP Request & Response

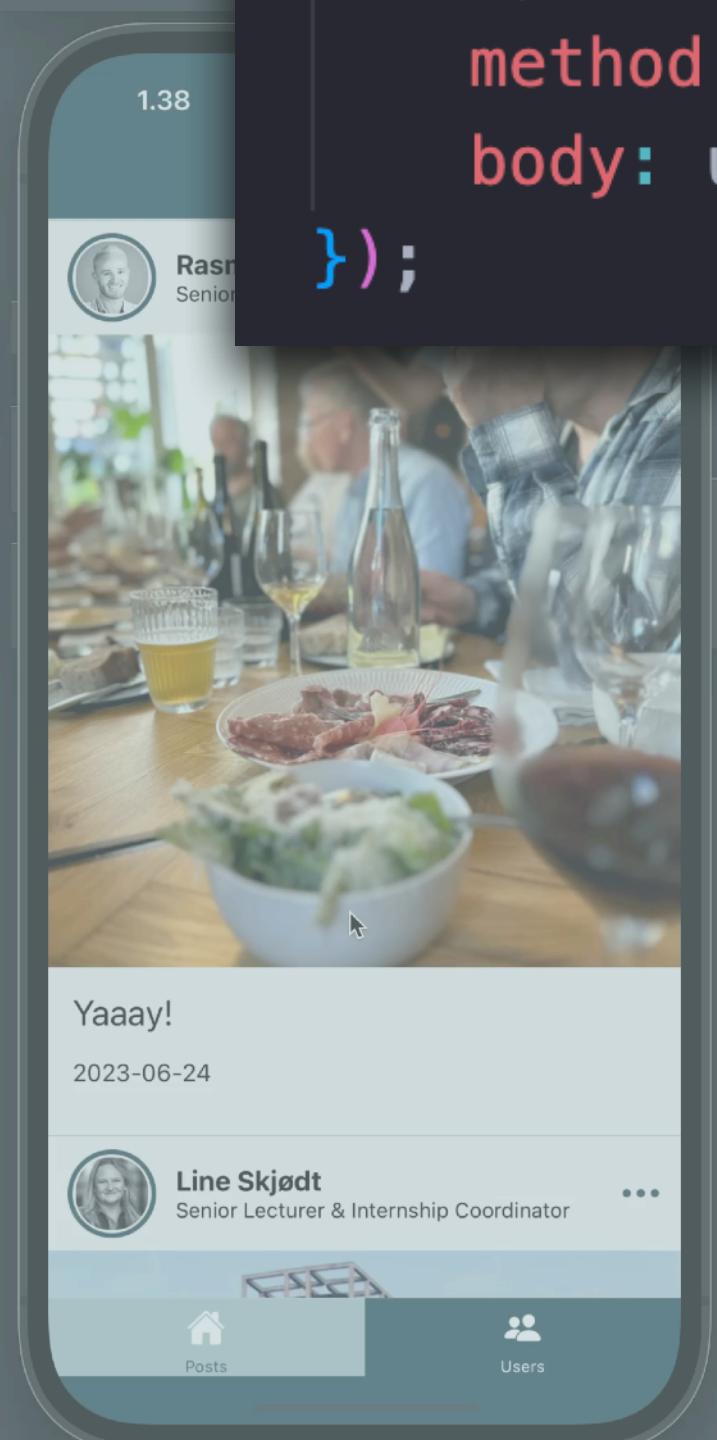


HTTP Requests & Response

```
const newUser = { name, title, mail, image };
const userAsJson = JSON.stringify(newUser);

const response = await fetch("http://localhost:3333/users", {
  headers: {
    "Content-Type": "application/json"
  },
  method: "POST",
  body: userAsJson
});
```

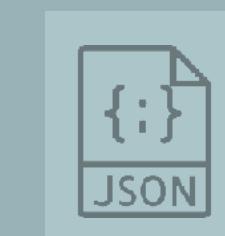
FR



REQUEST

JSON DATA { ... }

RESPONSE



BACKEND



CLIENT

SERVER

URL



Specifies the web address of the resource you want to access.

Headers



Provide metadata and instructions for the request or response.

Body



Carries data, often used to send information to the server in requests.

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 headers

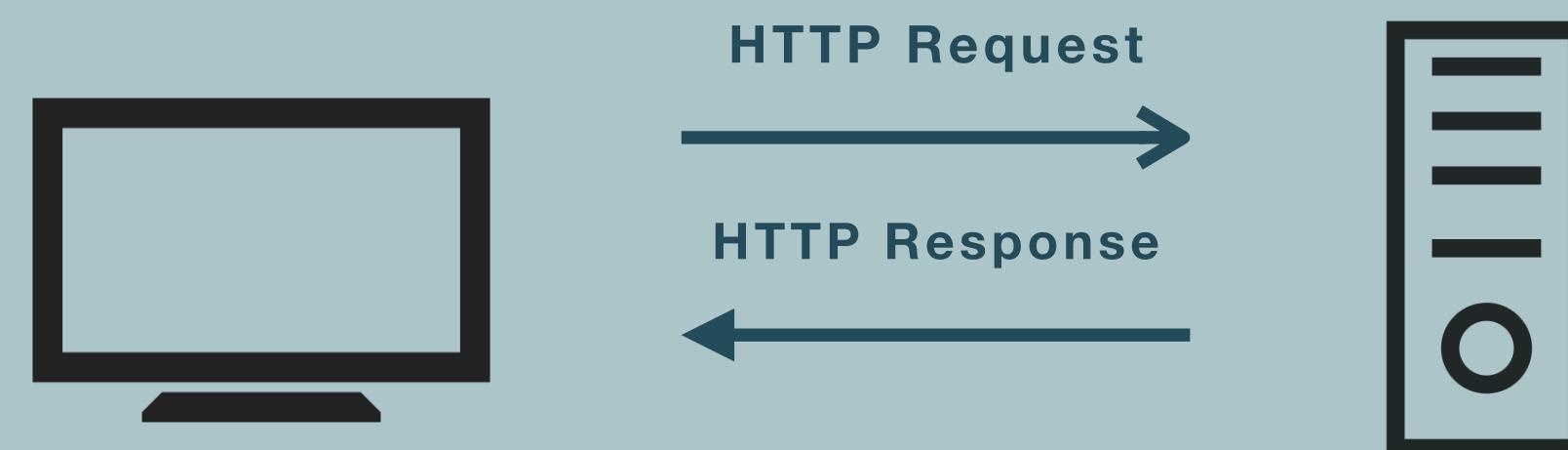
```
const response = await fetch("http://localhost:3333/users", {
  headers: {
    "Content-Type": "application/json"
  },
  method: "POST",
  body: userAsJson
});
```

```
const headers = {
  "Content-Type": "application/json"
};

// ===== CREATE ===== //
// Create (POST) user to backend using REST API
async function createUser(event) {
  event.preventDefault();
  const name = event.target.name.value;
  const title = event.target.title.value;
  const mail = event.target.mail.value;
  const image = event.target.image.value;
  // create a new user
  const newUser = { name, title, mail, image };
  const userAsJson = JSON.stringify(newUser);
  const response = await fetch(` ${endpoint}/users`, {
    headers: headers,
    method: "POST",
    body: userAsJson
  });

  if (response.ok) {
    // if success, update the users grid
  }
}
```

REQUEST body



In HTTP requests, there are occasions when data must be transmitted. This data is enclosed within what we call the request body.

Different types of request body:

- **JSON Request Body:** Data sent as a structured JSON-encoded string.
- **Form Data Request Body:** Data transmitted in a structured format, commonly used for web forms.
- **Binary Data Request Body:** Non-textual data, such as files or images, sent in its raw binary form.
- **URL Search Params Request Body:** Simple data, like key-value pairs, sent as URL search parameters.

REQUEST body

```
const url = "http://localhost:3000/json-api/posts/";
await fetch(url, {
  method: "POST",
  body: JSON.stringify(newPost)
});
```

```
async function savePost(postToUpdate) {
  const response = await fetch(url, {
    method: "PUT",
    body: JSON.stringify(postToUpdate)
  });
  const data = await response.json();
  console.log(data);
  navigate("/");
}
```

Fetch and create post

Using POST to create a new post object in the resource.

```
// new post object
const newPost = {
  title: "My new post",
  body: "Body description of a new post",
  image: "image url or image data string"
};

const url = "https://race-rest-default-firebaseio.com/posts.json";
const response = await fetch(url, {
  method: "POST", // fetch request using POST
  body: JSON.stringify(newPost) // newPost object to JSON
});
```

Fetch and update post

Using PUT to an existing post object by given id.

```
const postId = "5tl4jHHSRaKEB0UW9nQd"; // id of the object to update
const postToUpdate = { title: "...", body: "...", image: "..." };

const url = `https://race-rest-default-firebase.firebaseio.com/posts/${postId}.json`;
const response = await fetch(url, {
  method: "PUT", // using HTTP method PUT
  body: JSON.stringify(postToUpdate)
});
```

Fetch and delete post

Using DELETE to an object by given id.

```
const postId = "5tl4jHHSRaKEB0UW9nQd"; // id of the object to update
const url = `https://race-rest-default-rtdb.firebaseio.com/posts/${postId}.json`;

const response = await fetch(url, {
  method: "DELETE"
});
```

DOM Manipulation

Application Programming Interface

Headless WordPress

Access your content through WP REST API

What's a Headless CMS?



A CMS without a head 🤔

I'm not kidding

It's a CMS without any frontend



Headless CMS

...is a system to control the content that is separated from the display layer or the front-end user experience.

What
Is Headless CMS?



Traditional CMS

FRONTEND

CMS

BACKEND

Headless CMS

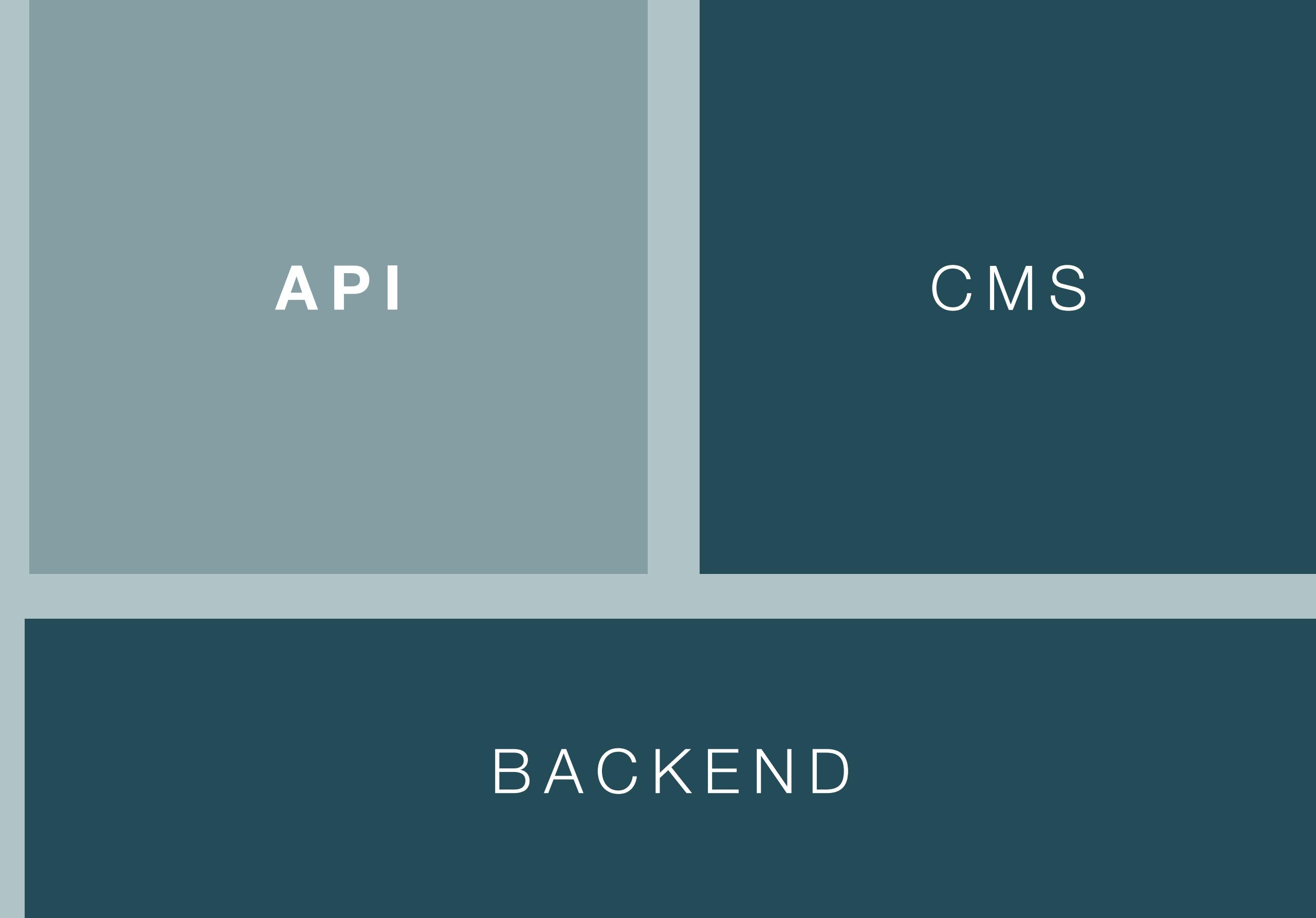
The diagram illustrates the architecture of a Headless CMS. It features three rectangular boxes arranged horizontally. The top-left box is light gray and contains the word "FRONTEND". The top-right box is dark teal and contains the letters "CMS". The bottom box is also dark teal and contains the word "BACKEND".

FRONTEND

CMS

BACKEND

Headless CMS



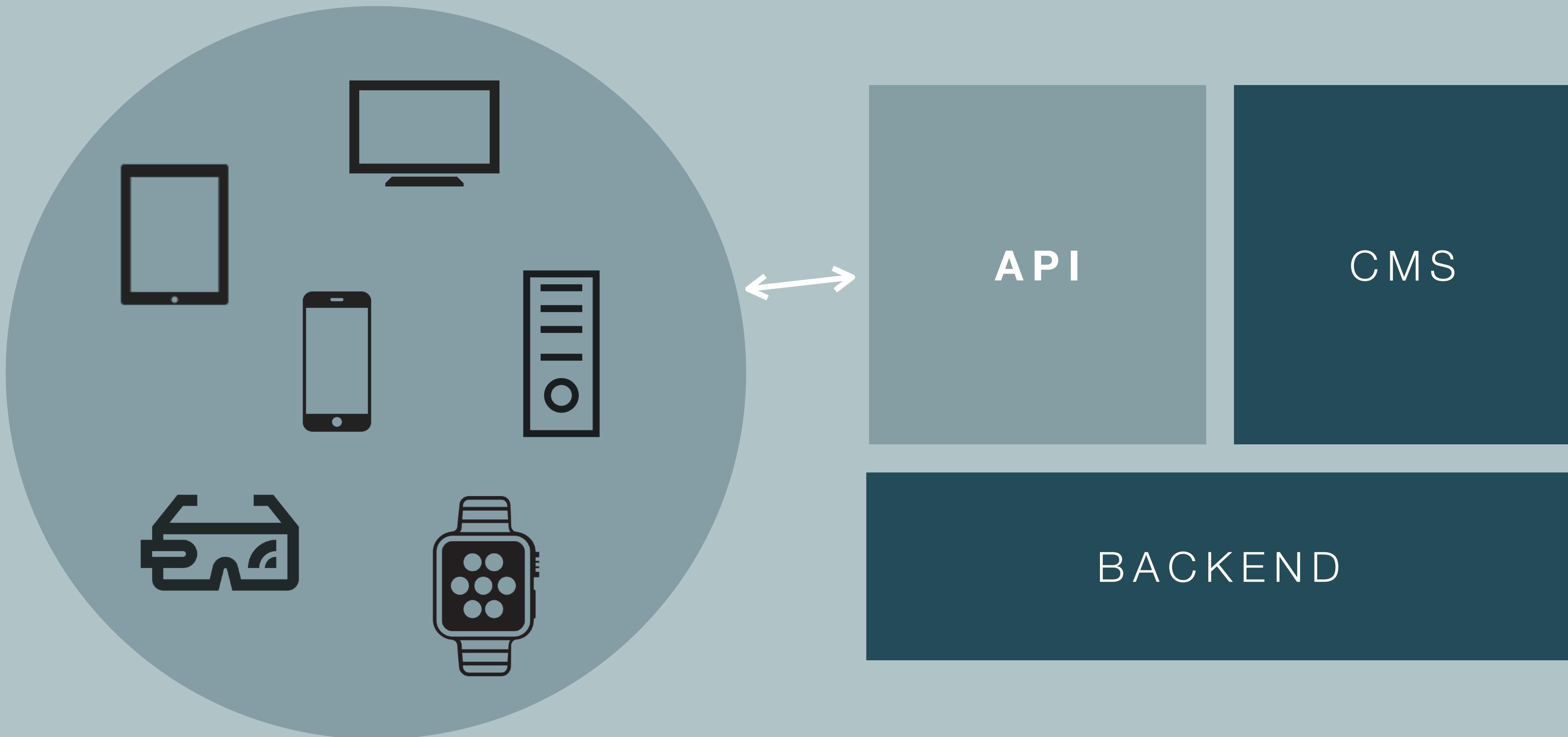
The diagram illustrates the architecture of a Headless CMS. It features three main components arranged vertically: a light gray rectangle labeled "API" at the top, a dark teal rectangle labeled "CMS" in the middle, and a dark teal rectangle labeled "BACKEND" at the bottom. The "API" and "CMS" boxes are positioned side-by-side, while the "BACKEND" box is centered below them, suggesting a layered or integrated system.

API

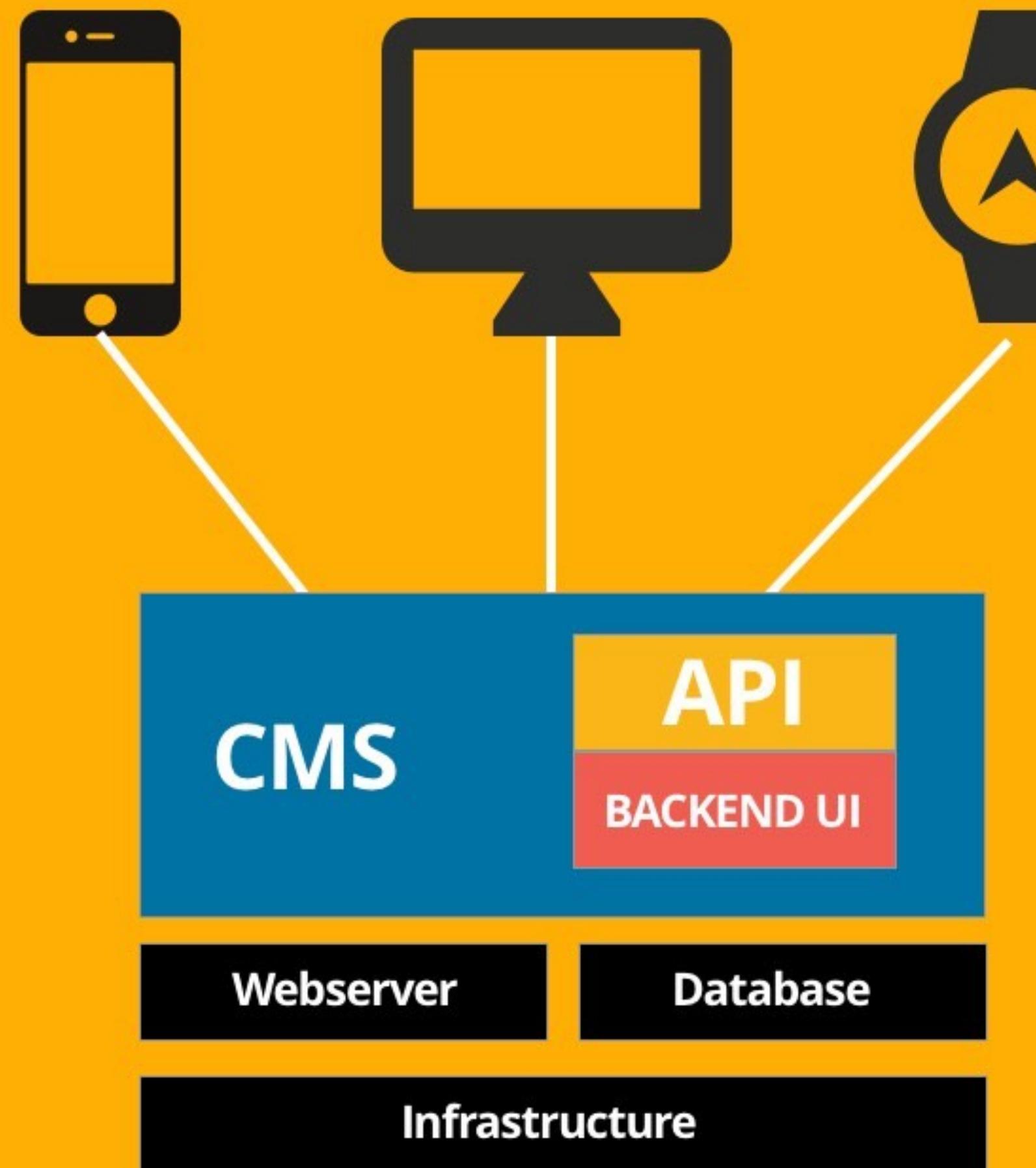
CMS

BACKEND

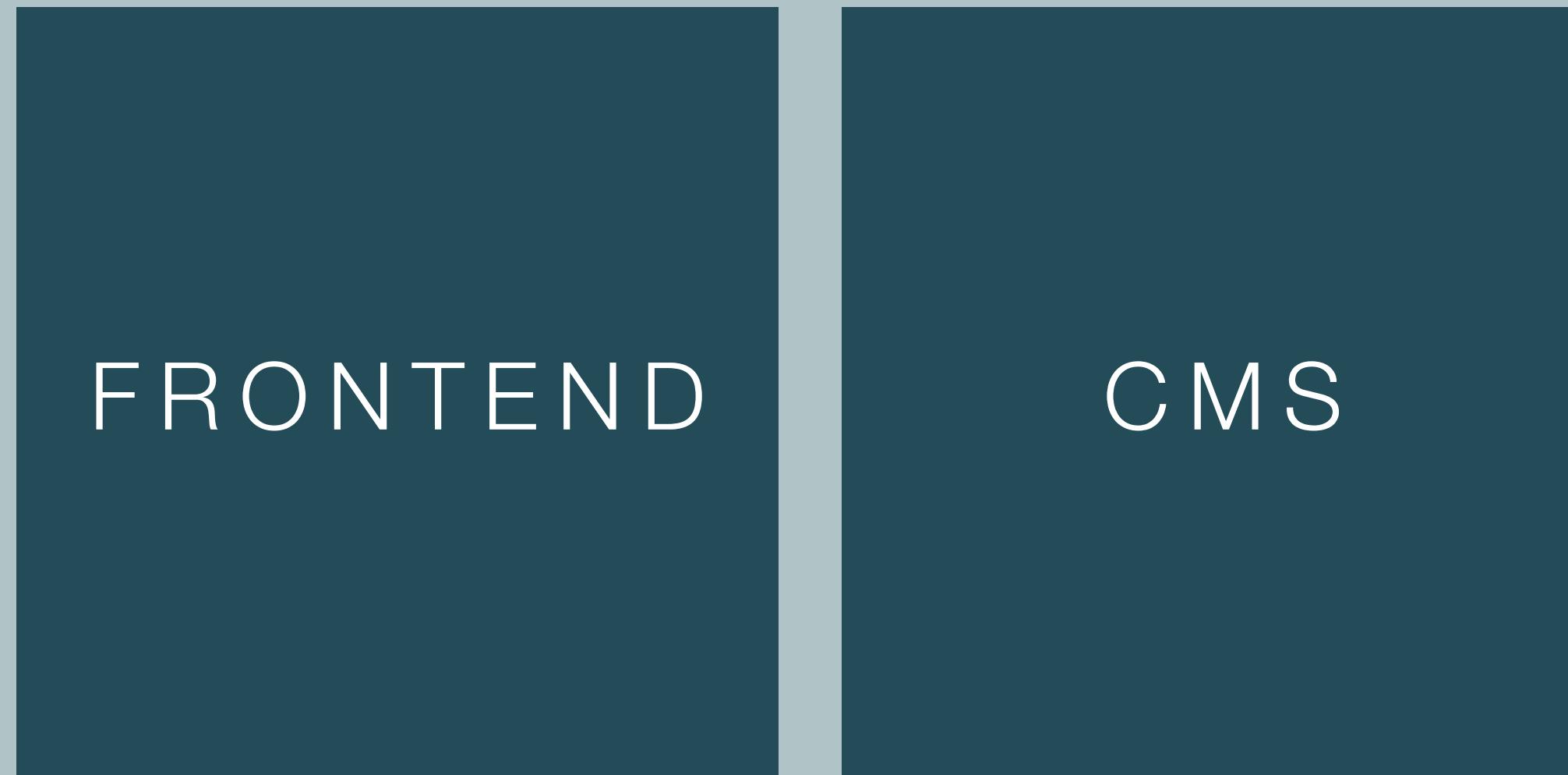
Headless CMS



Headless CMS



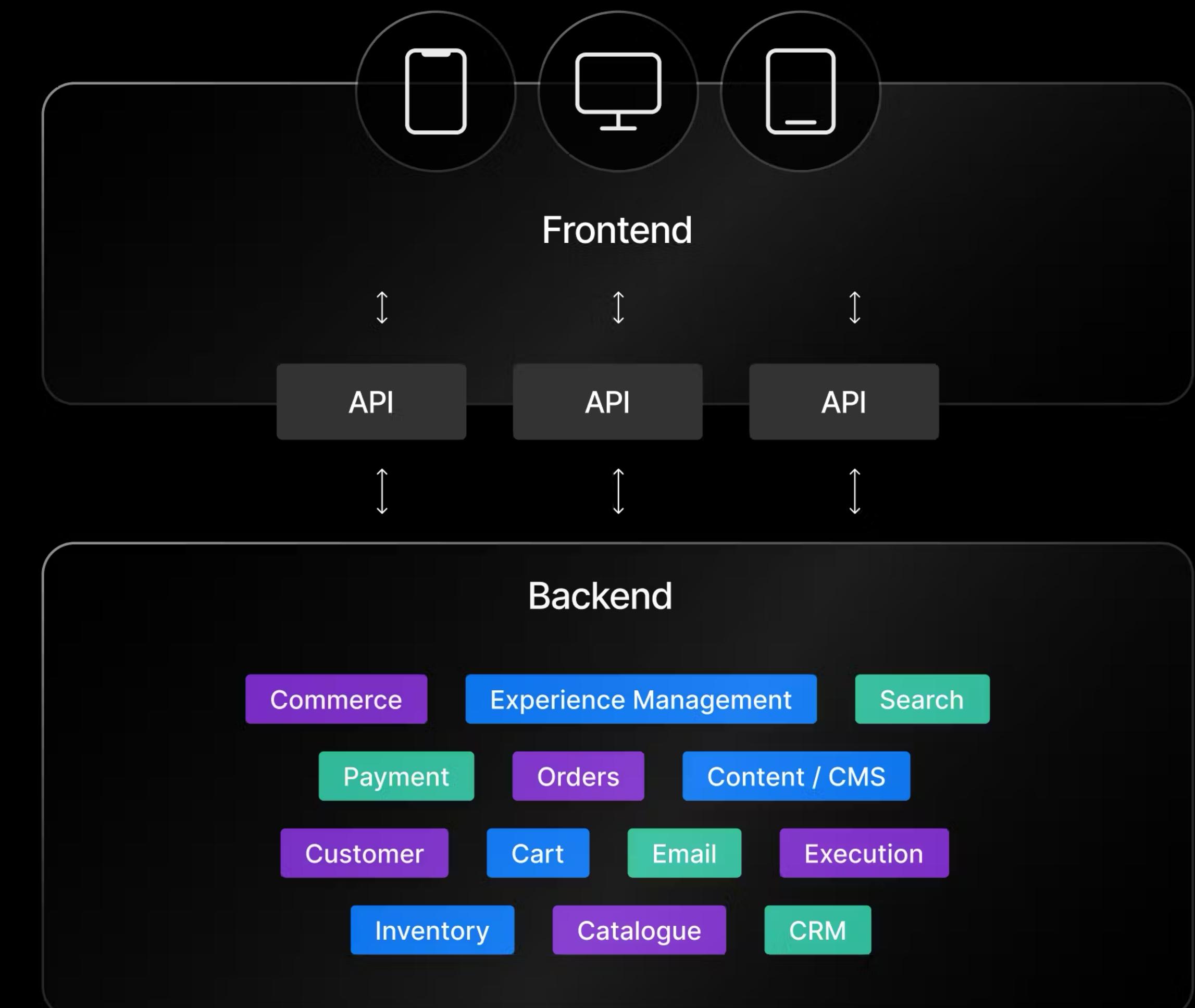
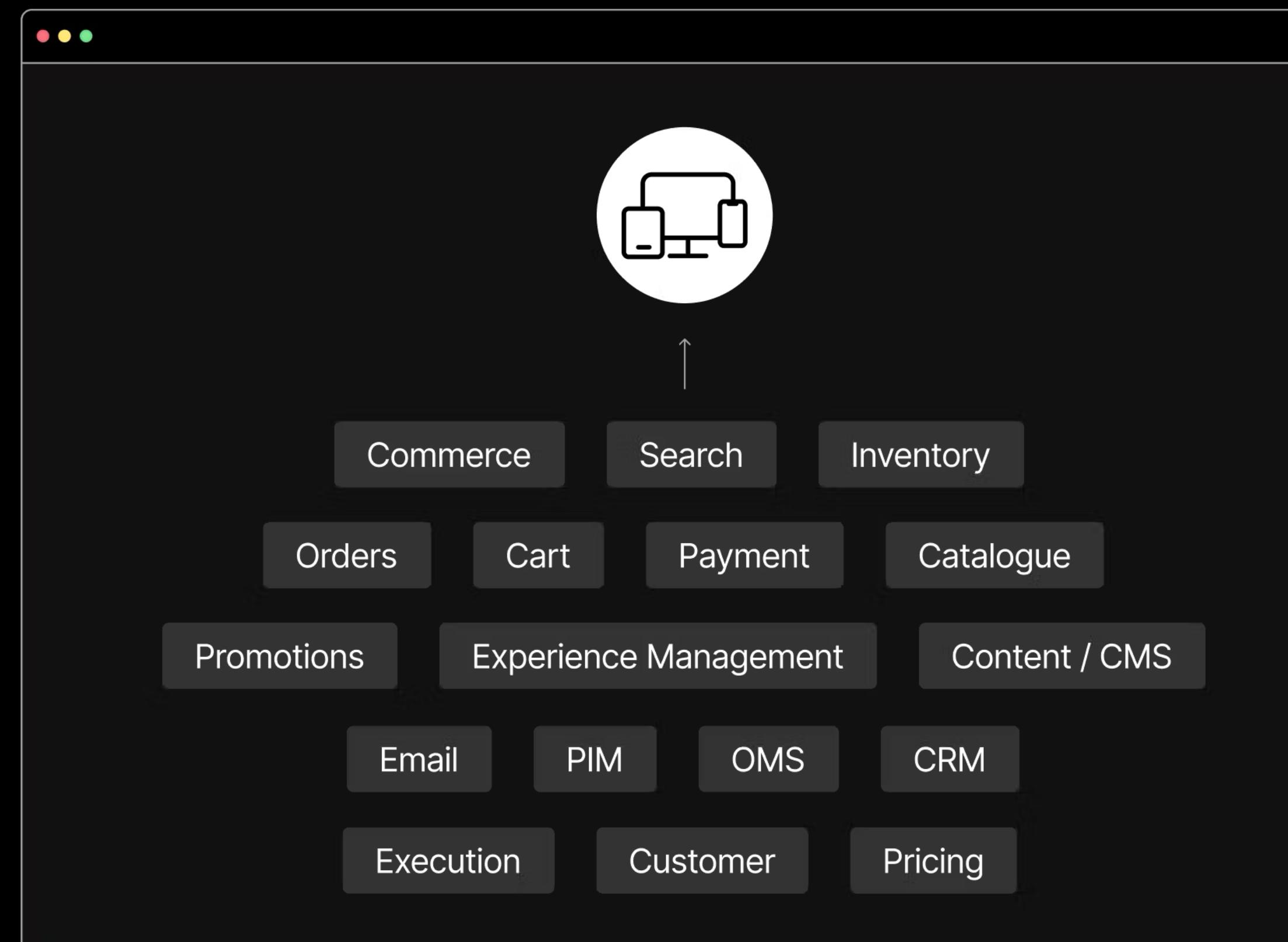
Traditional CMS



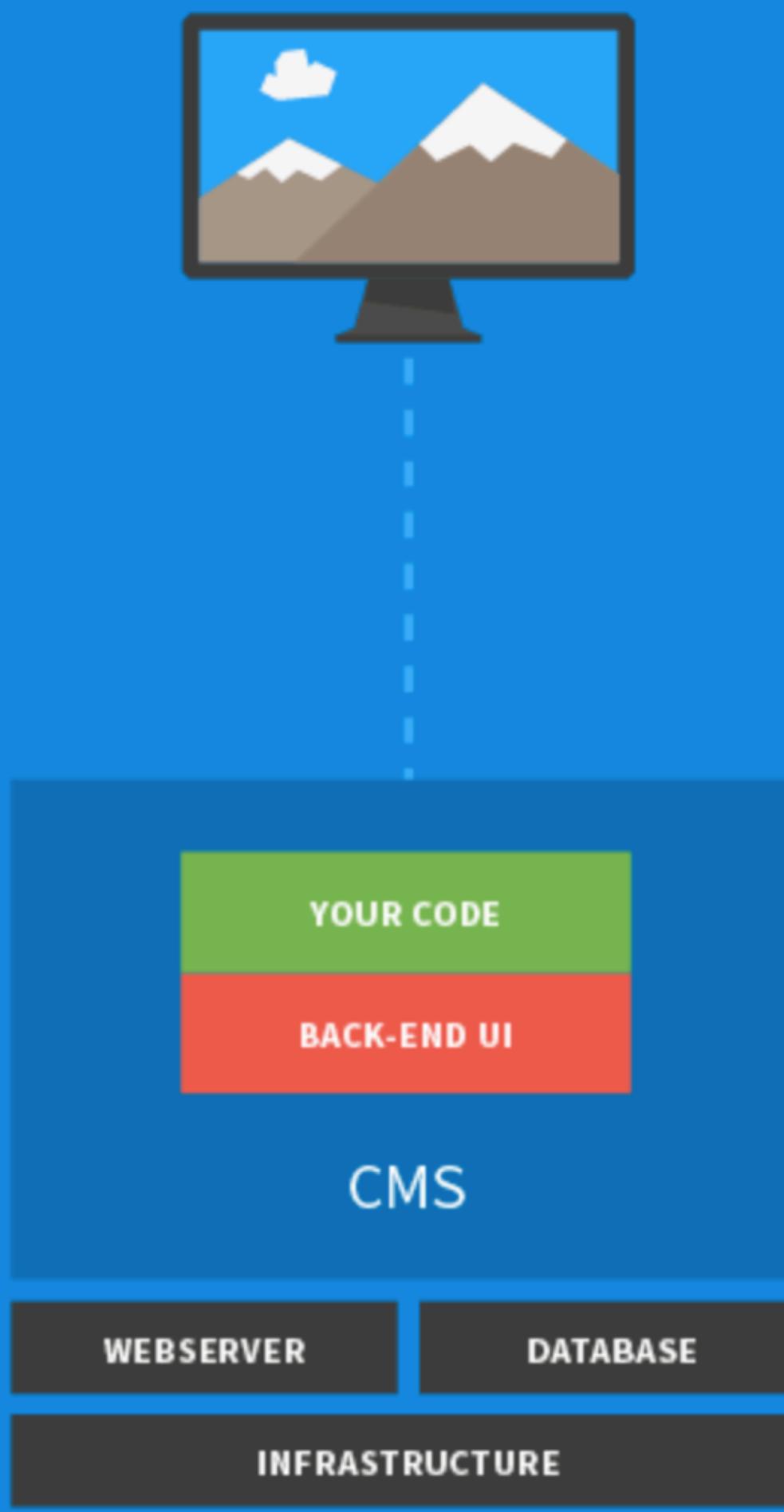
Headless CMS



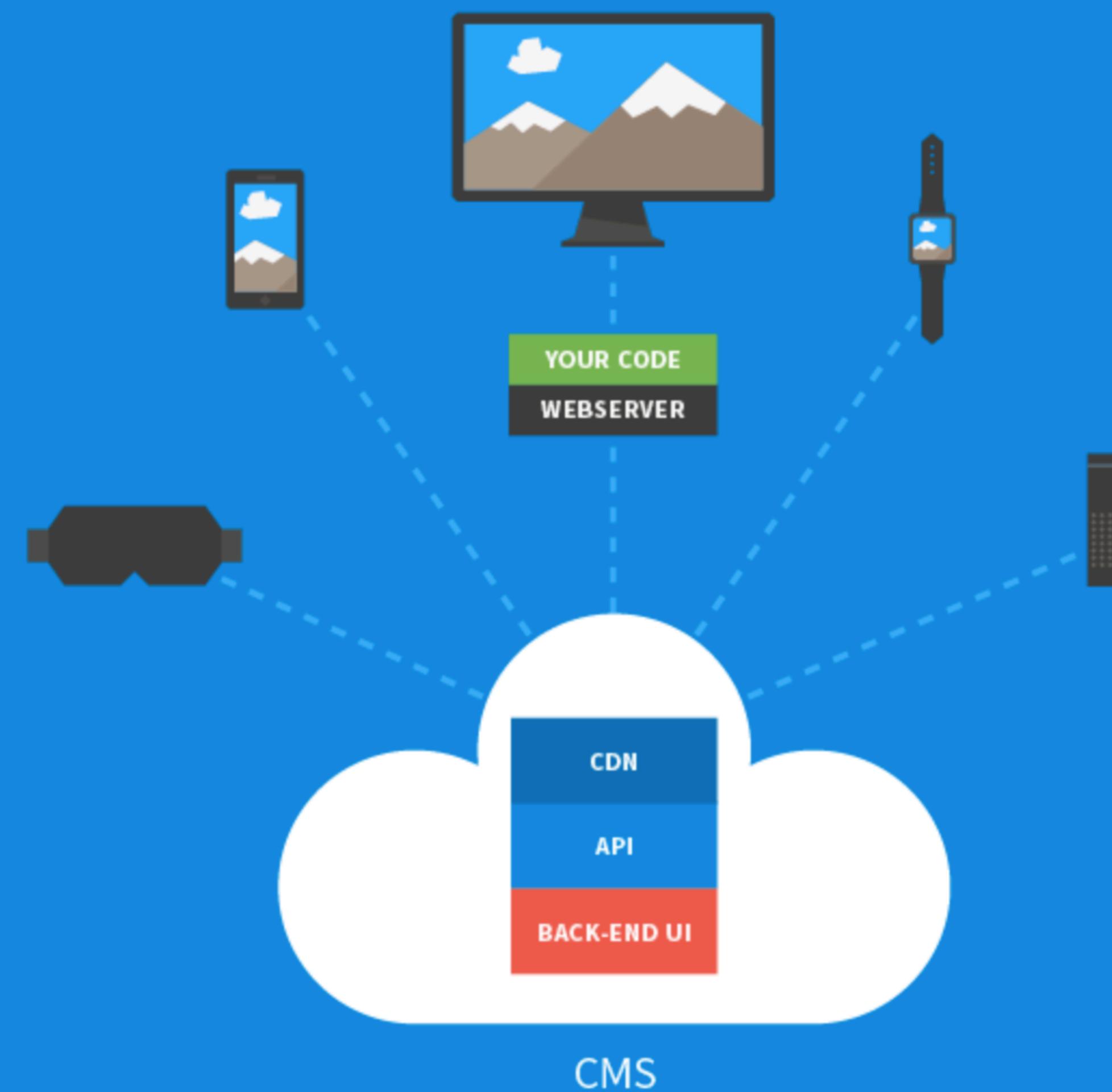
Monolith vs. Headless



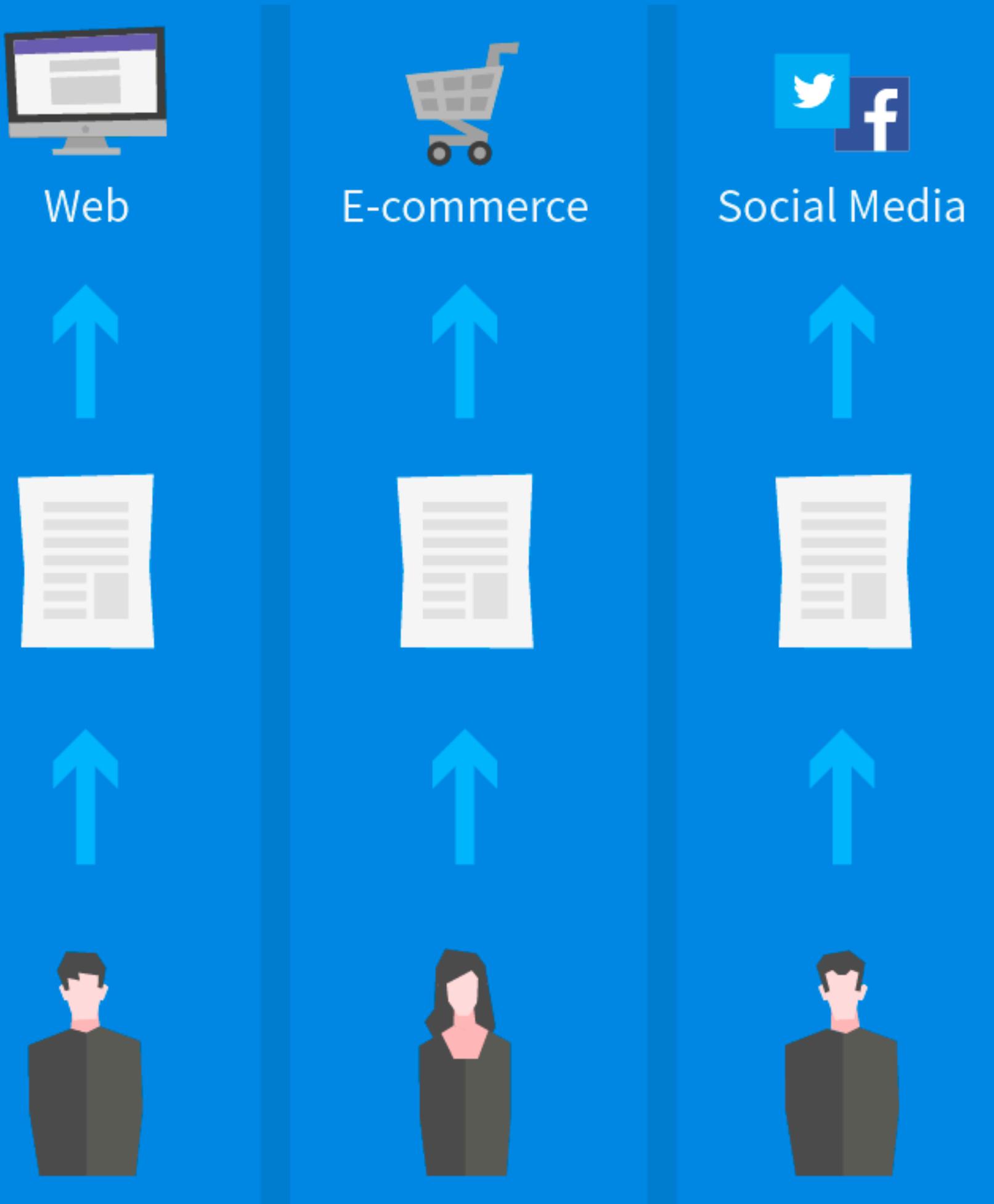
TRADITIONAL CMS



HEADLESS CMS



TRADITIONAL CMS



HEADLESS CMS



TRADITIONAL CMS

A WAY TO STORE DATA

A CRUD UI

A WAY TO DISPLAY THE DATA

CRUD: CREATE, READ, UPDATE & DELETE

HEADLESS CMS

A WAY TO STORE DATA

A CRUD UI

AN API TO ACCESS THE DATA

WordPress

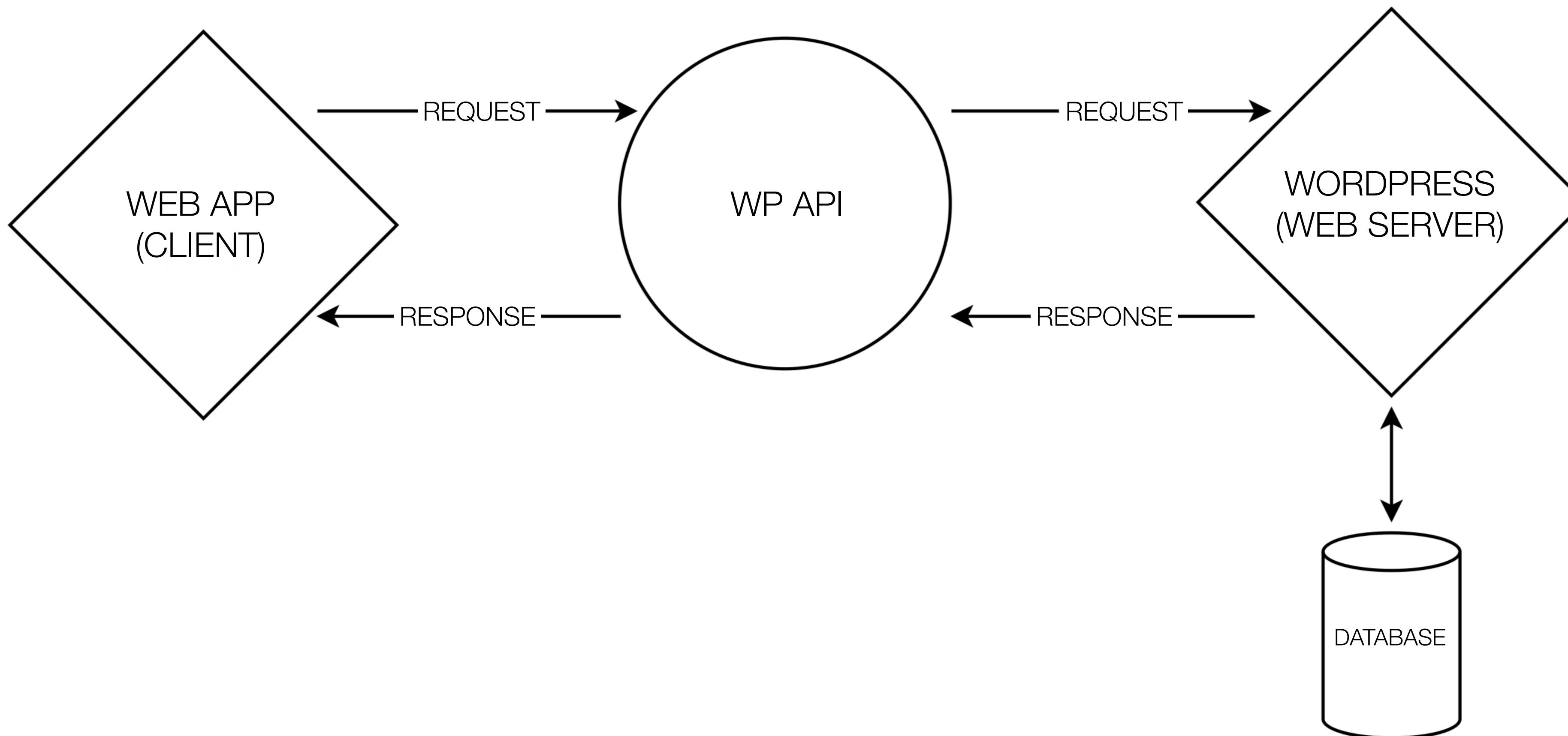
FRONTEND

API

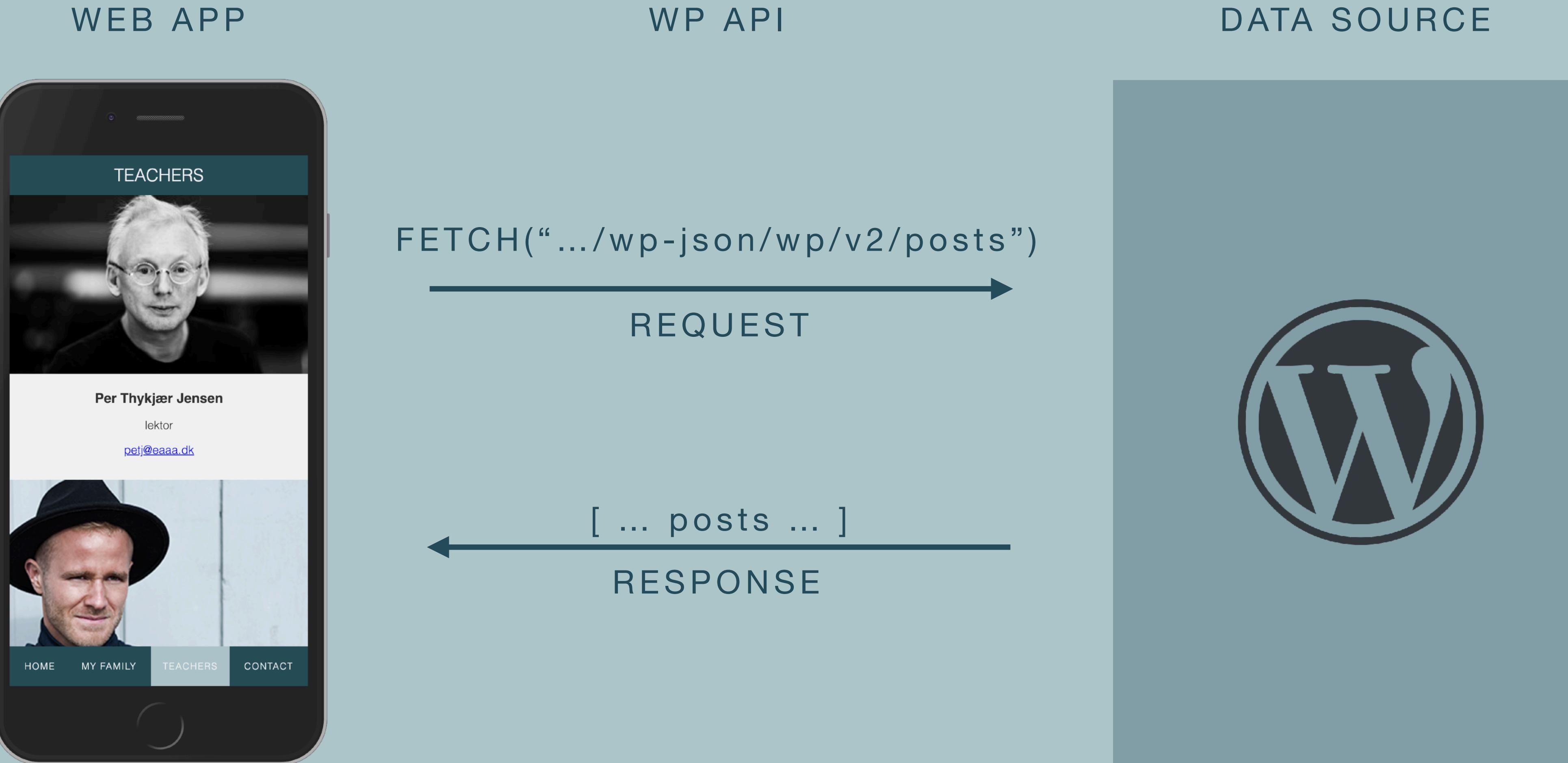
CMS

BACKEND

APPLICATION PROGRAMMING INTERFACE



Fetch & Headless CMS



Posts < MY HEADLESS CMS — +

← → C 🔒 headlesscms.cederdorff.com/wp-admin/edit.php

Howdy, mdu-front

Dashboard Posts All Posts Add New Categories Tags

WordPress 5.8 is available! Please notify the site administrator.

Posts Add New

All (8) | Published (8) | Trash (2)

Bulk actions ▾ Apply All dates ▾ All Categories ▾ Filter

	Title	Author	Categories	Tags	Comment	Date
<input type="checkbox"/>	Martin	admin	Persons	—	0 1	Published 2020/08/31 at 6:37 am
<input type="checkbox"/>	Peter Cederdorff	admin	Family Members	—	—	Published 2019/02/07 at 4:14 pm
<input type="checkbox"/>	Rasmus Cederdorff	admin	Family Members	—	0 3	Published 2019/02/07 at 4:12 pm
<input type="checkbox"/>	Alicia	admin	Family Members	—	0 3	Published 2019/02/07 at 11:34 am
<input type="checkbox"/>	Per Thykær	admin	Teachers	—	0 6	Published 2019/02/06 at 7:40 am
<input type="checkbox"/>	Rasmus Cederdorff	admin	Teachers	—	0 4	Published 2019/02/05 at 6:35 pm
<input type="checkbox"/>	Michael Hvidtfeldt	admin	Teachers	—	0 2	Published 2019/02/05 at 6:34 pm
<input type="checkbox"/>	Birgitte Kirk Iversen	admin	Teachers	—	0 8	Published 2019/02/05 at 6:32 pm
	Title	Author	Categories	Tags	Comment	Date

Bulk actions ▾ Apply

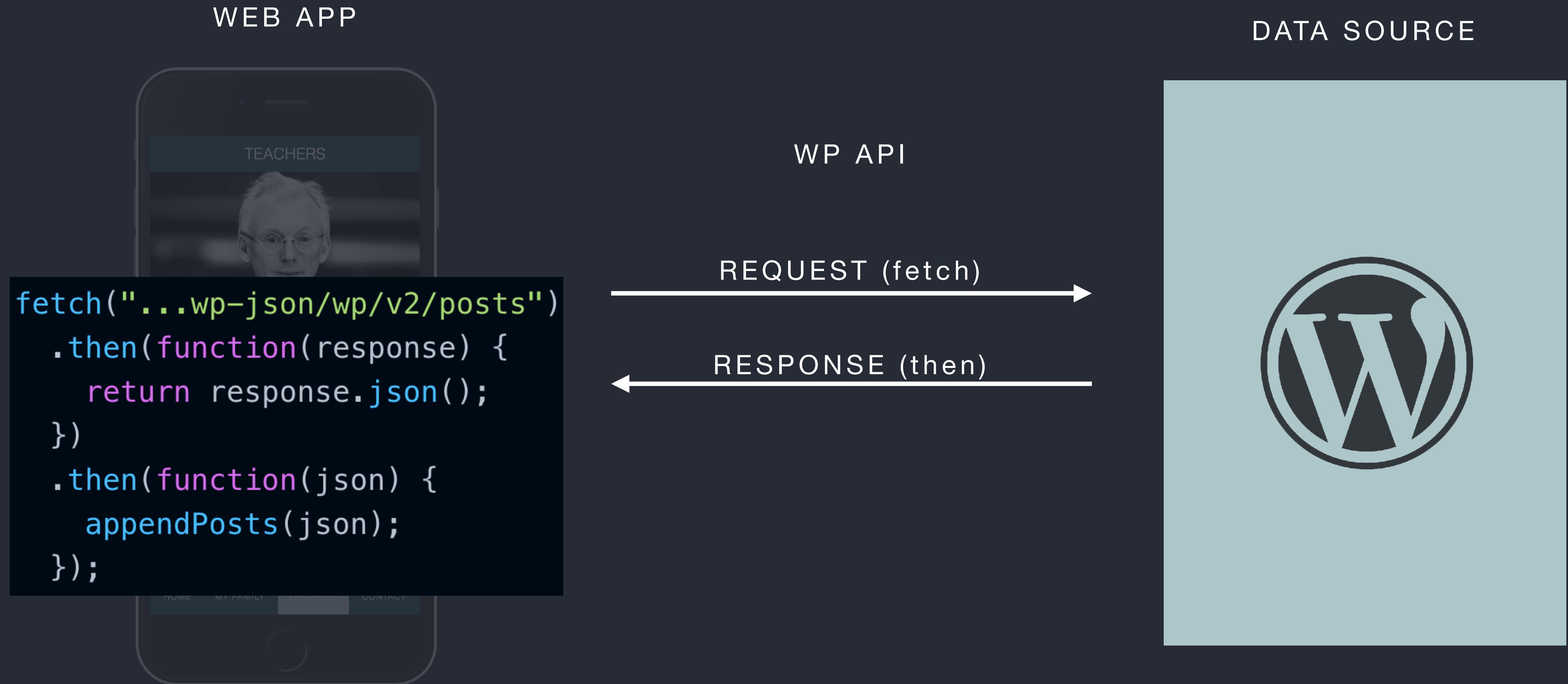
Screen Options ▾ Help ▾

8 items

Thank you for creating with WordPress.

Version 5.5.5

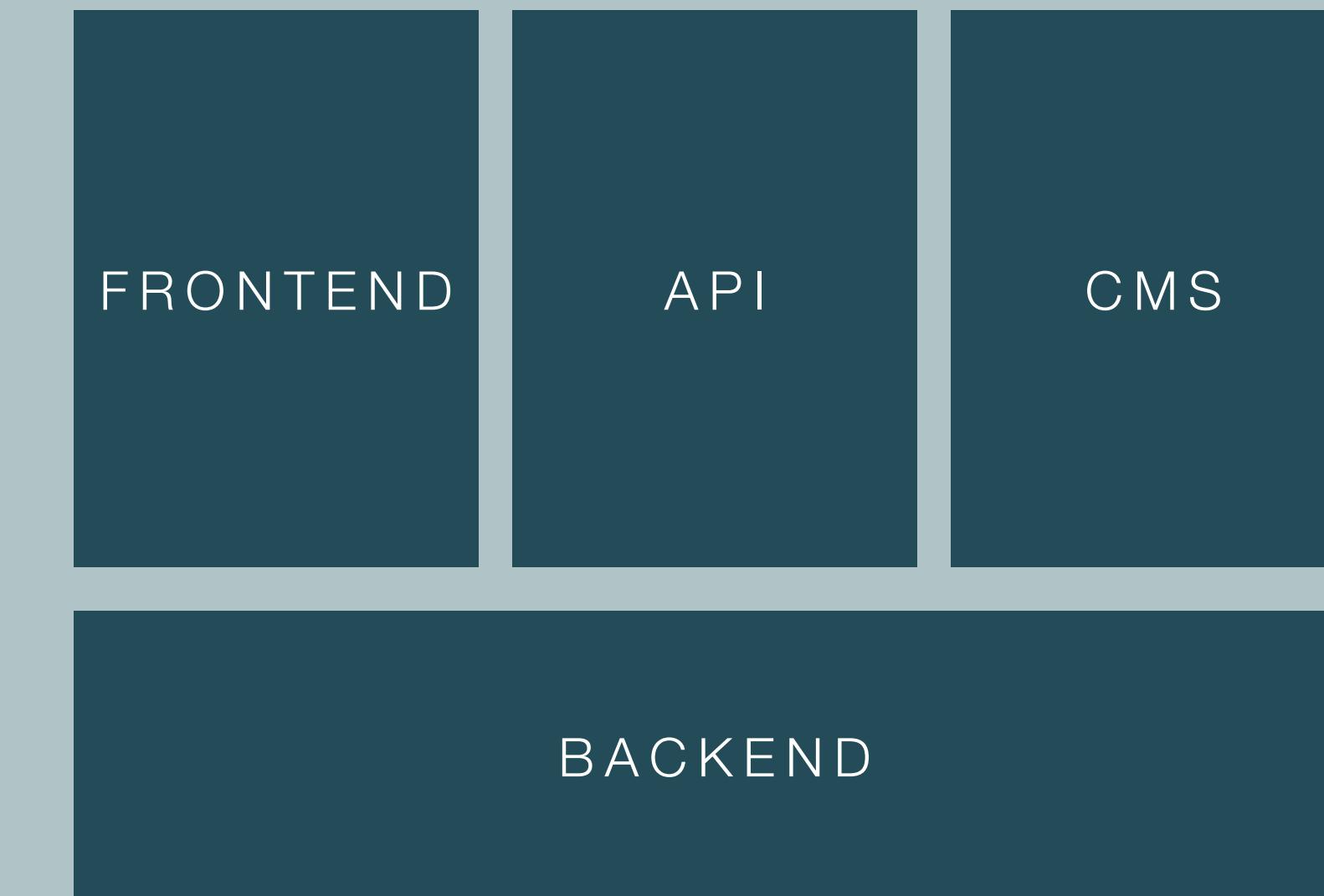
Fetch & Headless CMS



WEB APP



HEADLESS WP



movie-api.cederdorff.com

Fetch posts

Screenshot of the WordPress admin dashboard showing the Posts screen.

The dashboard header includes the WordPress logo, site title "Movie API", a refresh icon with "7", a comment icon with "0", and a "+ New" button.

A notification bar at the top right says "WordPress 5.3.2 is available! [Please update now.](#)"

The left sidebar menu is visible, with "Posts" selected and highlighted in blue. Other menu items include Dashboard, All Posts, Add New, Categories, Tags, Media, Pages, Comments, Appearance, Plugins (with a red "3" badge), Users, Tools, Settings, Custom Fields, and Collapse menu.

The main content area is titled "Posts" with a "Add New" button. It shows a list of published posts:

	Title	Author	Categories	Tags
<input type="checkbox"/>	Frozen	admin	Animation, Comedy	—
<input type="checkbox"/>	Don't Let Go	admin	Drama, Horror	—
<input type="checkbox"/>	Lion King	admin	Animation	—
<input type="checkbox"/>	Title ▲			

Below the table are "Bulk Actions" dropdown and "Apply" buttons.

Fetch posts

```
let _movies = [];

// fetch all movies from WP
async function getMovies() {
  let response = await fetch("https://movie-api.cederdorff.com/wp-json/wp/v2/posts?_embed");
  let data = await response.json();
  _movies = data;
  appendMovies(data);
  showLoader(false);
}

getMovies();
```

API Routes

Meta data: <http://your-domain.com/wp-json/>

Posts: <http://your-domain.com/wp-json/wp/v2/posts>

Pages: <http://your-domain.com/wp-json/wp/v2/pages>

Media: <http://your-domain.com/wp-json/wp/v2/media>

Types: <http://your-domain.com/wp-json/wp/v2/types>

Taxonomies: <http://your-domain.com/wp-json/wp/v2/taxonomies>

Categories: <http://your-domain.com/wp-json/wp/v2/categories>

Tags: <http://your-domain.com/wp-json/wp/v2/tags>

Users: <http://your-domain.com/wp-json/wp/v2/users>

Comments: <http://your-domain.com/wp-json/wp/v2/comments>

DOCS: <https://developer.wordpress.org/rest-api/>

Filter, sort & order

[https://api.cederdorff.com/wp-json/wp/v2/posts? embed](https://api.cederdorff.com/wp-json/wp/v2/posts?embed)

[https://api.cederdorff.com/wp-json/wp/v2/posts?
embed&categories=2](https://api.cederdorff.com/wp-json/wp/v2/posts?embed&categories=2)

[https://api.cederdorff.com/wp-json/wp/v2/posts?
orderby=date&order=desc&after=2019-00-10T17:00:00](https://api.cederdorff.com/wp-json/wp/v2/posts?orderby=date&order=desc&after=2019-00-10T17:00:00)

[https://api.cederdorff.com/wp-json/wp/v2/posts?
orderby=date&order=asc](https://api.cederdorff.com/wp-json/wp/v2/posts?orderby=date&order=asc)

Recap on JS

<https://cederdorff.com/race/slides/js-concepts.pdf>



Code
Every
Day