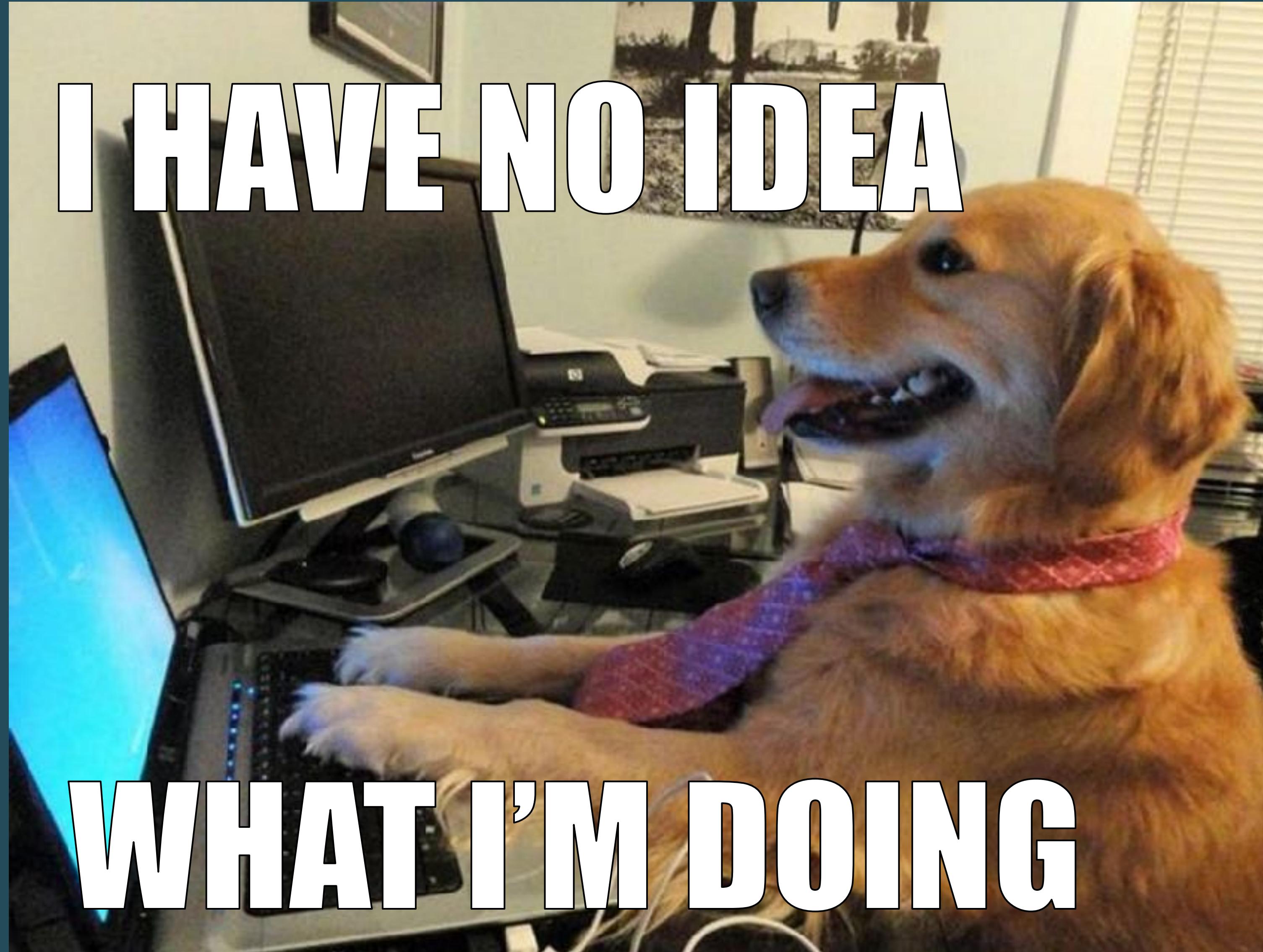
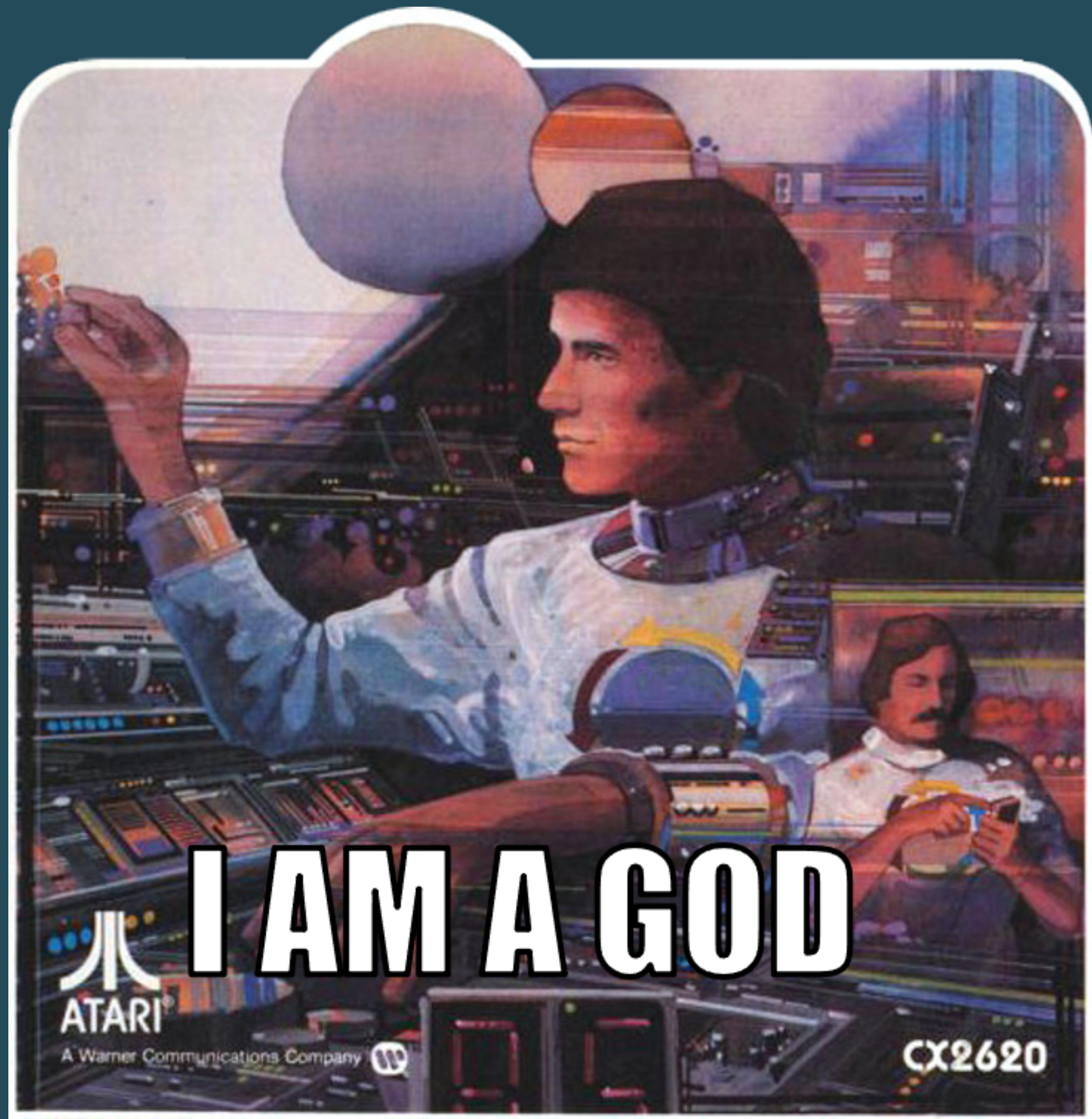


Fetch, JSON, CRUD & REST

I HAVE NO IDEA

WHAT I'M DOING





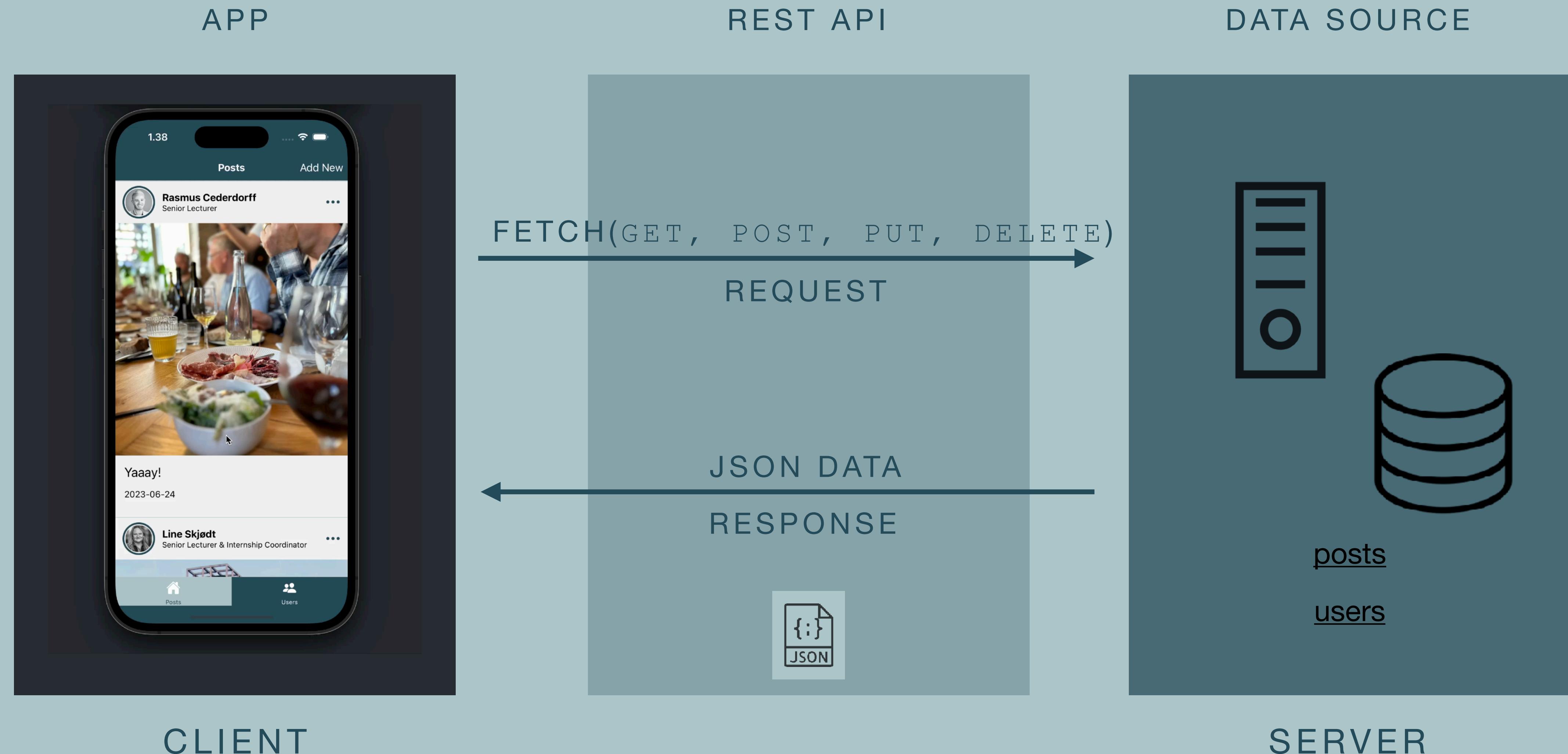
Content

- Client-Server Model
- Web Development
- Fetch
- Async JS & await/sync
- What's a Data Source?
- API
- JSON
- CRUD
- REST
- HTTP Request Methods & Verbs
- CRUD vs REST & HTTP Verbs
- REST API Best Practices
- What's Firebase?

Turn on the Database

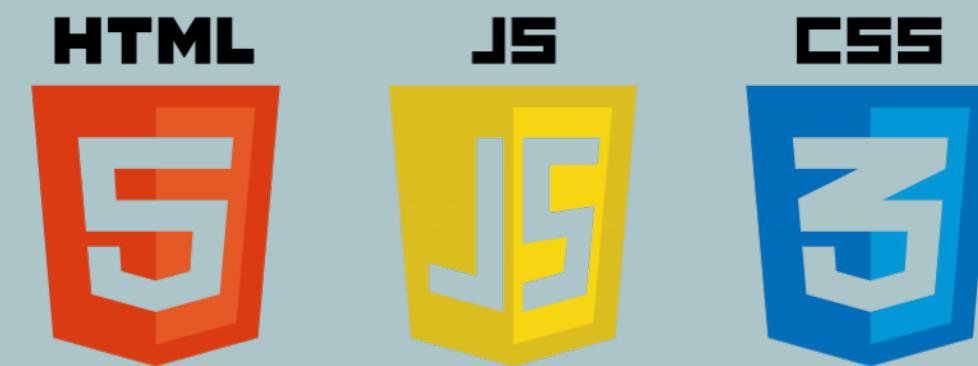


Fetch, HTTP Request & Response



Webudvikling

FRONTEND

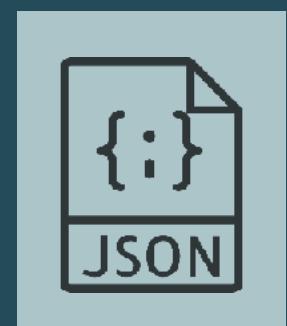


REST API

FETCH(GET, POST, PUT, DELETE)

REQUEST

JSON DATA
RESPONSE

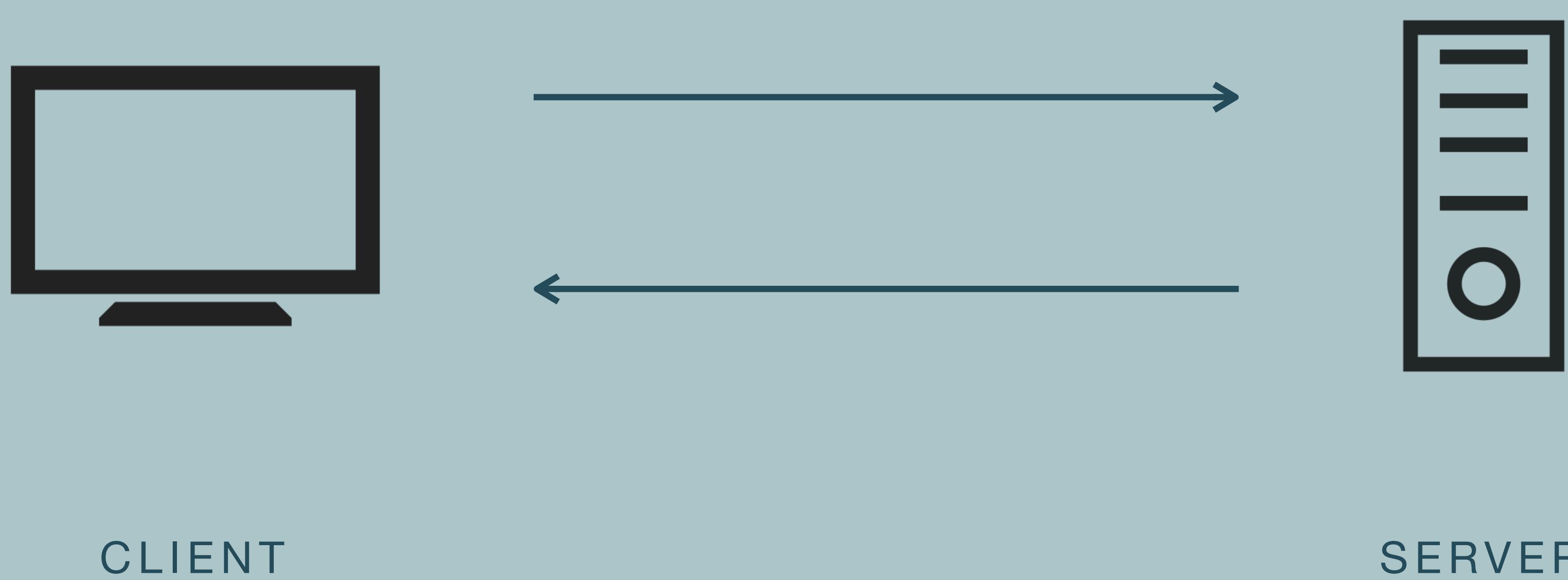


BACKEND



Client-Server Model

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



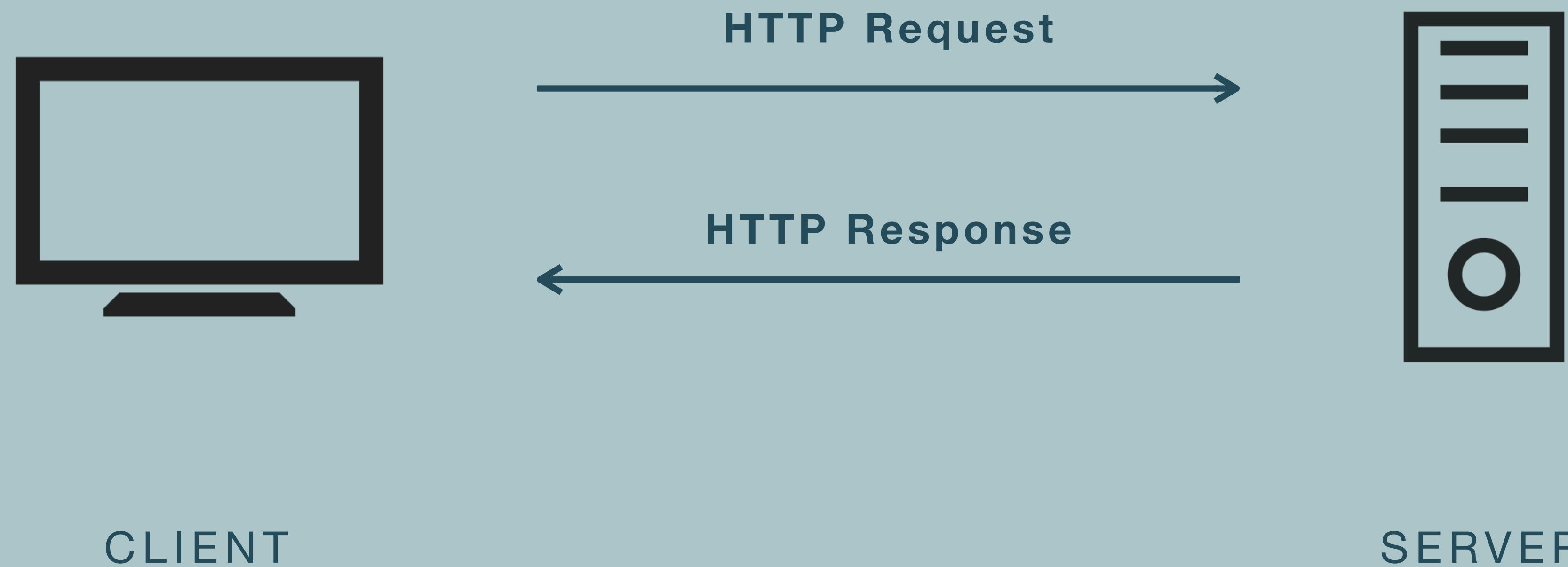
Client-Server Model

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



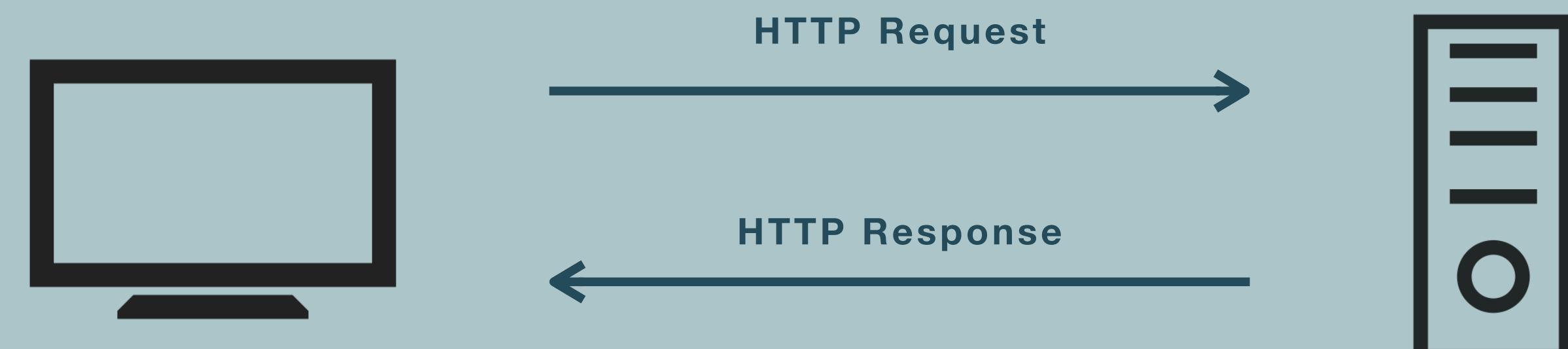
Client-Server Model

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**:

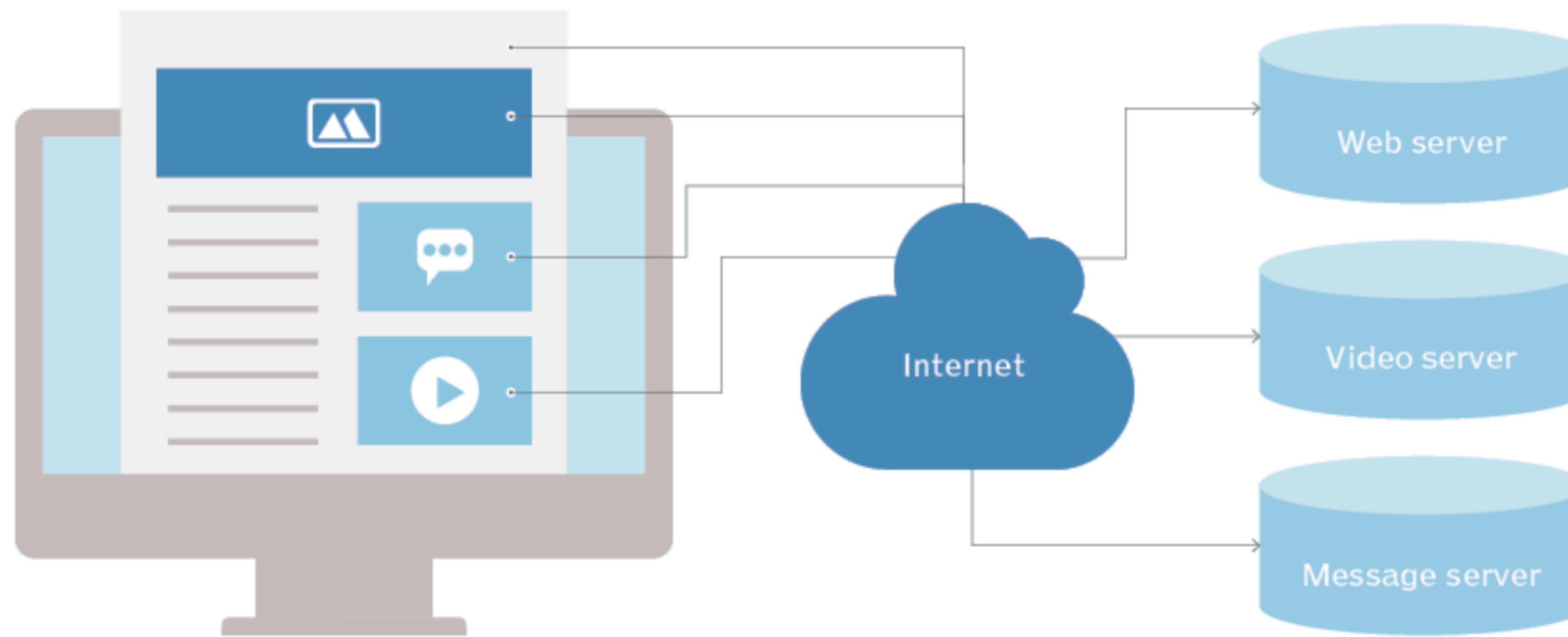
1. A client (a browser) sends an **HTTP request** to the web
2. A web server receives the request
3. The server runs an application to process the request
4. The server returns an **HTTP response** (output) to the browser
5. The client (the browser) receives the response

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>

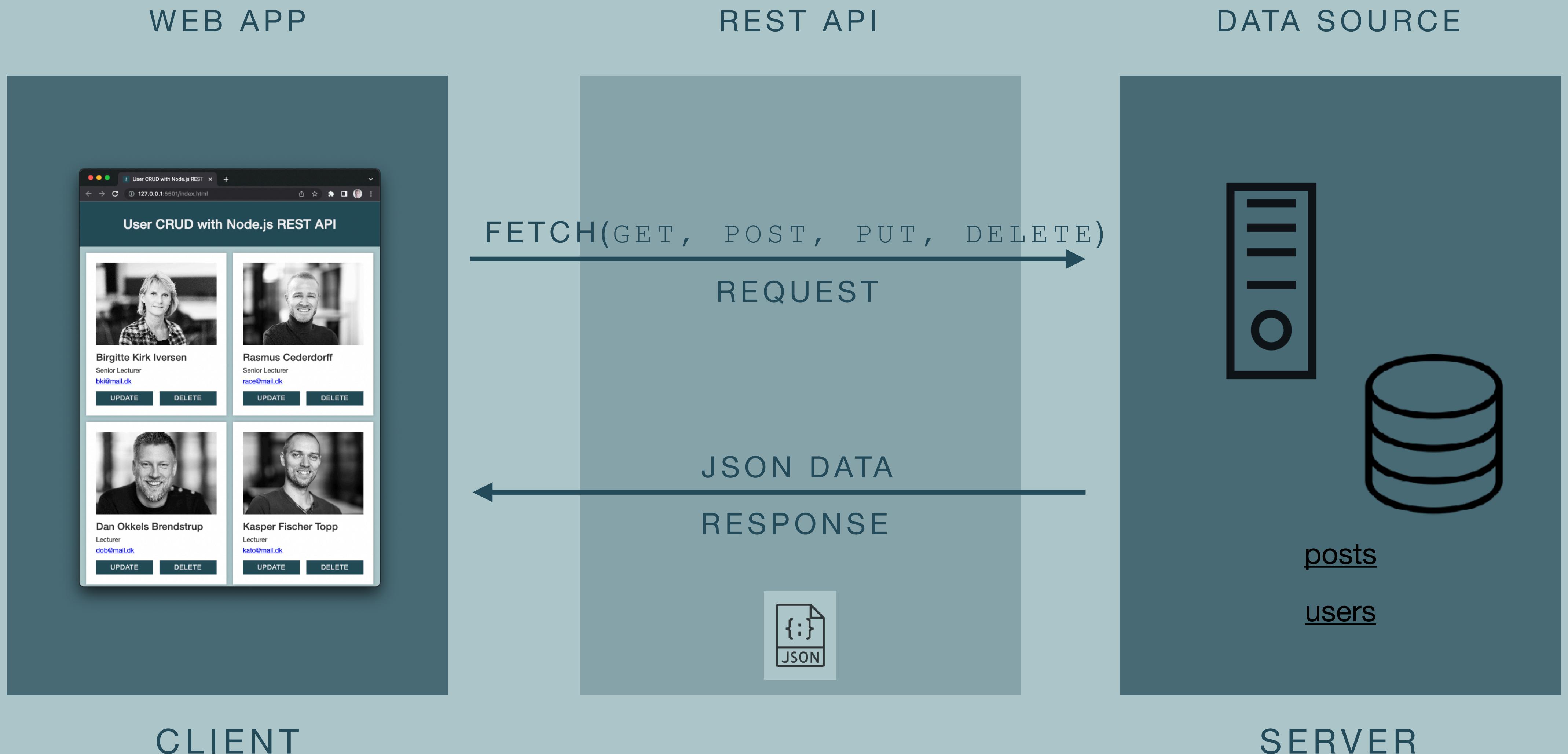
Network Tab

The screenshot shows a web browser window with the URL kompetence.kea.dk/kurser-fag/webudvikling-frontend. The main content area displays information about the 'WEBUDVIKLING FRONTEND' course, including a 'SE HOLDSTART OG TILMELD DIG →' button and a 'LAV DIN EGEN PDF-BROCHURE' section with a 'FØJ TIL PDF' button. A red sidebar on the left contains a 'NY UDDANNELSE: DIPLOM I WEBUDVIKLING' section with text about the diploma program.

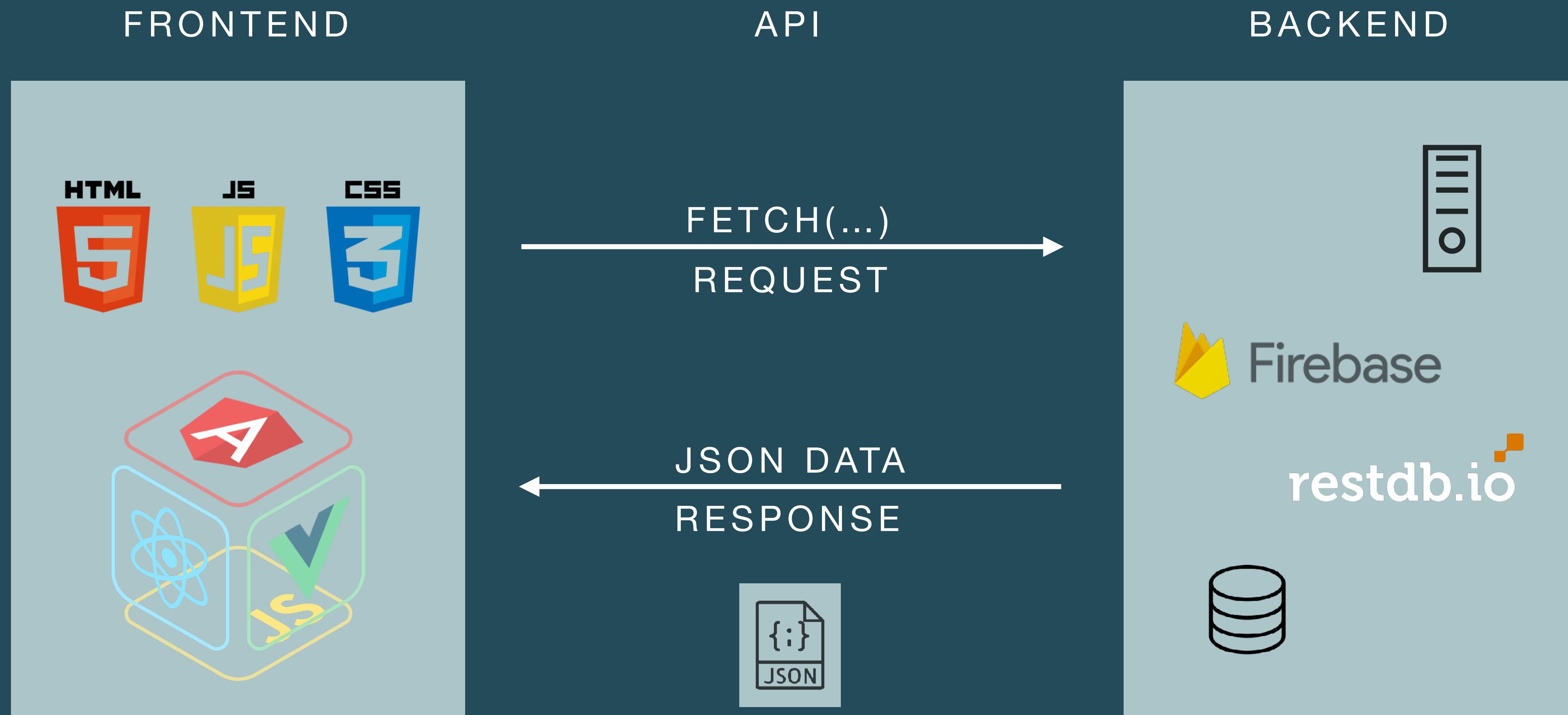
Network Tab Details:

- Filter:** All | Fetch/XHR | JS | CSS | Img | Media | Font | Doc | WS | Wasm | Manifest | Other
- Has blocked cookies:**
- Blocked Requests:**
- 3rd-party requests:**
- Timeline:** 500 ms, 1000 ms, 1500 ms, 2000 ms
- Table Headers:** Name, Status, Type, Initiator, Size, Time, Waterfall
- Table Data:** A list of 46 requests, mostly 200 status, including files like 'webudvikling-fro...', 'quixtrap.css', 'quix.css', 'jquery.min.js?77...', 'style.css?77958...', etc.
- Metrics:** 46 requests | 28.7 kB transferred | 1.9 MB resources | Finish: 1.59 s | DOMContentLoaded

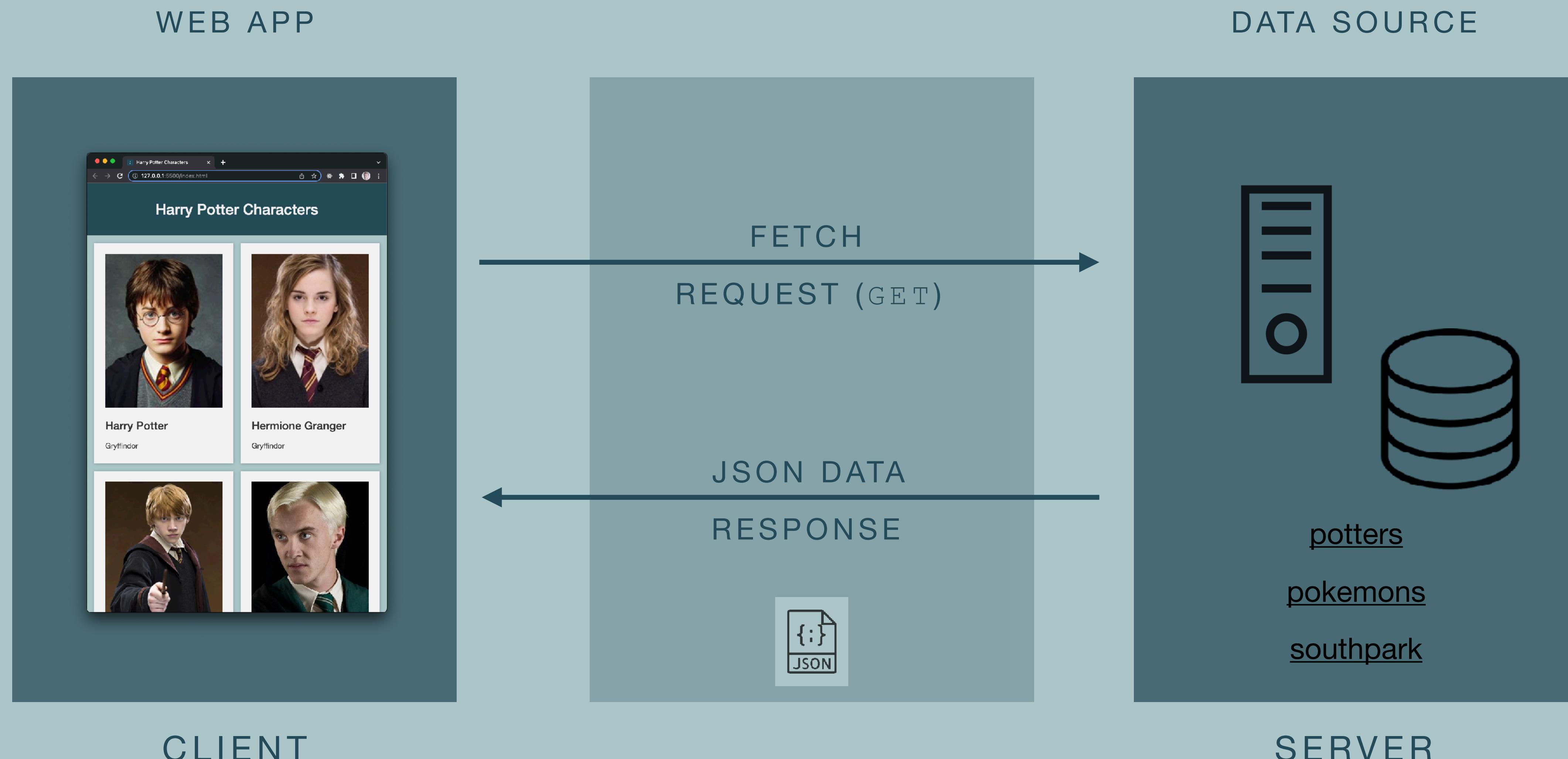
Fetch, HTTP Request & Response



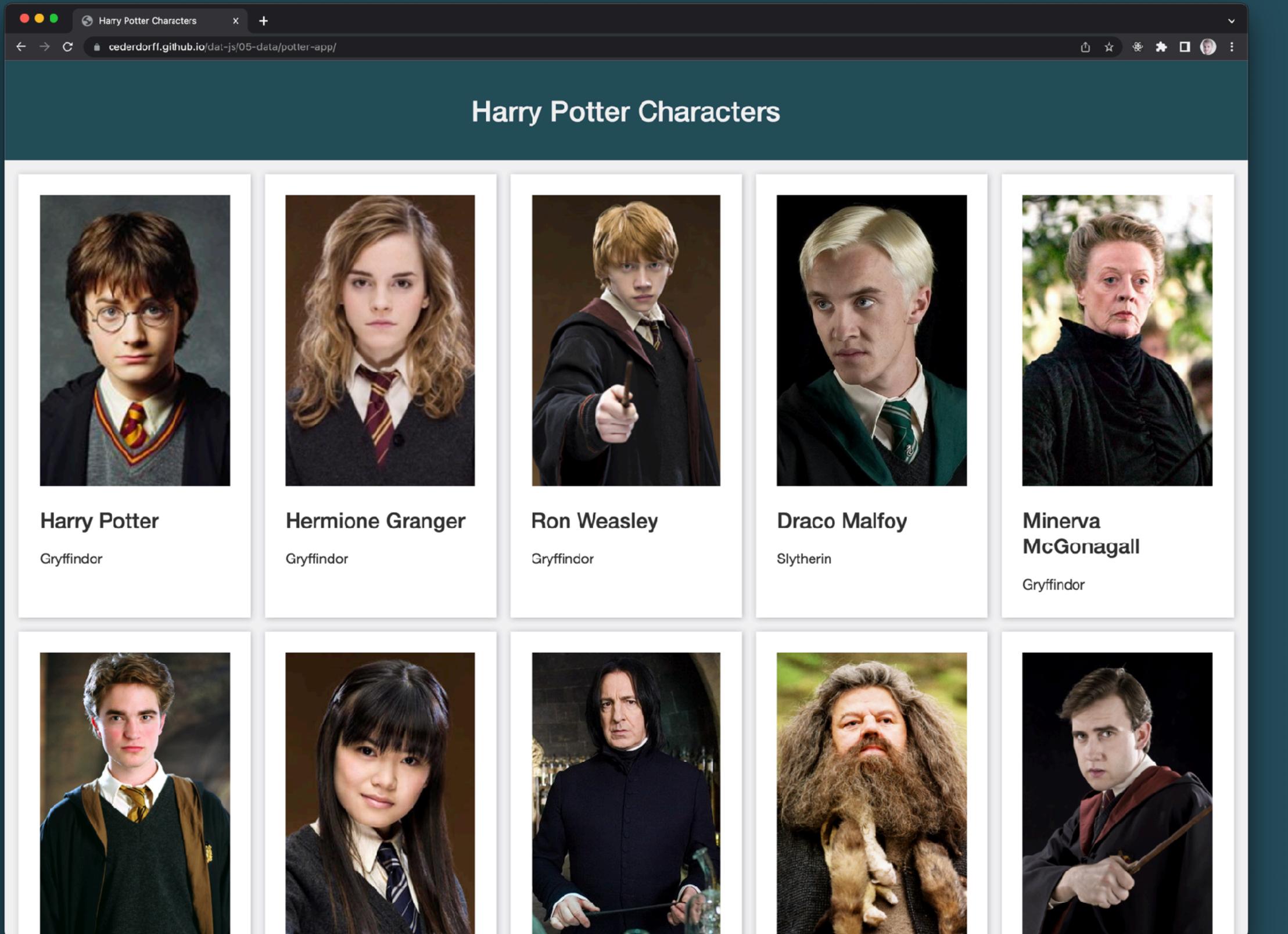
Web Development



Fetch, HTTP Request & Response



Frontend (client)



<https://cederdorff.github.io/dat-js/05-data/potter-app/>

JSON Data Source (Server)

The screenshot shows a browser window displaying a JSON object representing the Harry Potter characters. The JSON structure includes properties for each character's name, species, gender, house, date of birth, year of birth, ancestry, eye color, hair color, and wand details. The "Raw" tab shows the unprocessed JSON code, and the "Parsed" tab shows the JSON object with its properties and values.

```
Raw
{
  "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
  }
}

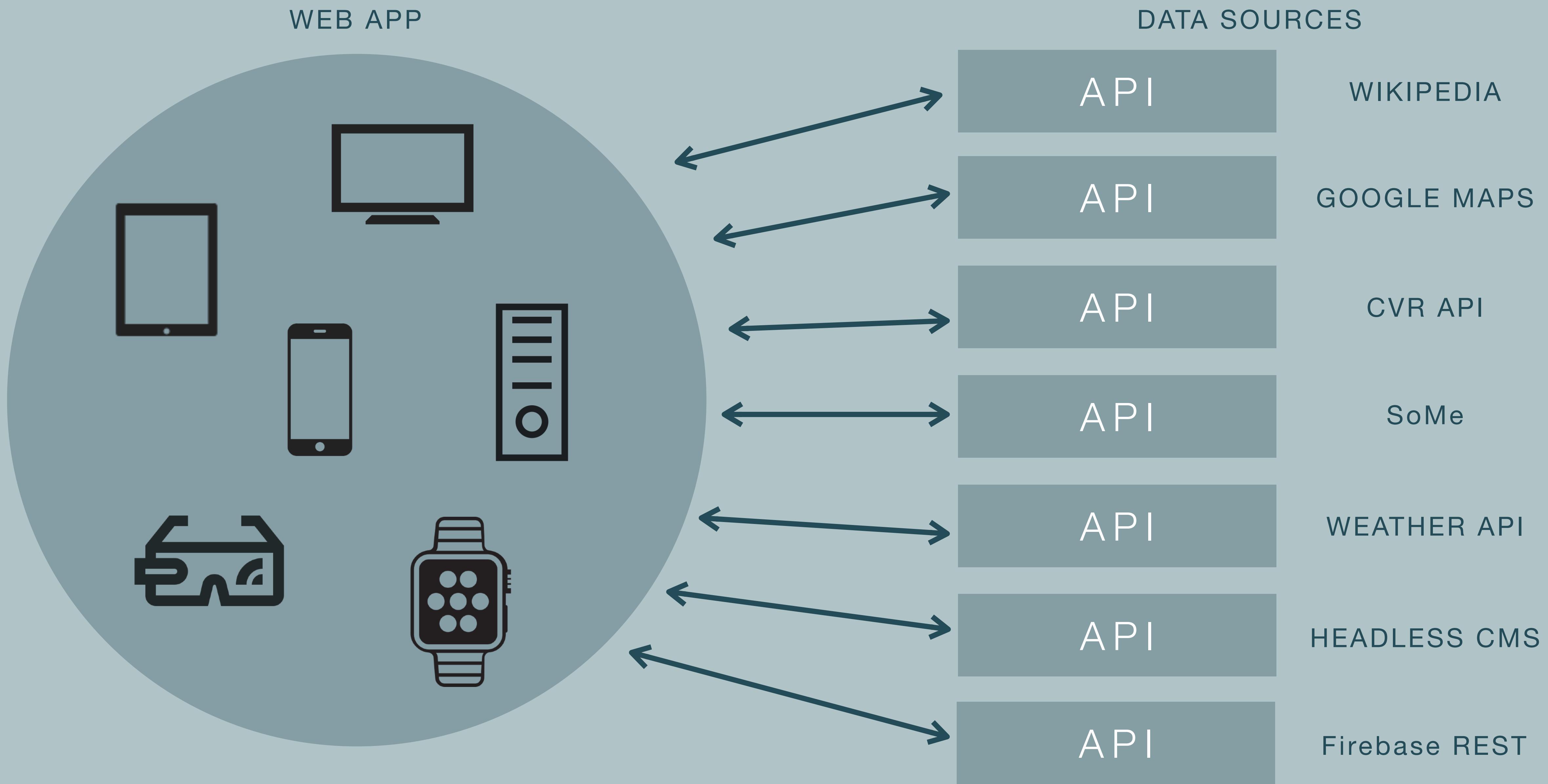
```

Parsed

```
{
  "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
  }
}
```

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

API



React CRUD

React Firebase REST Post App https://race-rest.web.app

POSTS CREATE

 Morten Algy Bonderup
Senior Lecturer



Qui est esse

Est rerum tempore vitae sequi sint nihil reprehenderit dolor beatae ea dolores neque fugiat blanditiis voluptate porro vel nihil molestiae ut reiciendis qui aperiam non debitis possimus qui neque nisi nulla

 Dan Okkels Brendstrup
Lecturer



Consequuntur deleniti eos quia temporibus ab aliquid at

Voluptatem cumque tenetur consequatur expedita ipsum nemo quia explicabo aut eum minima consequatur tempore cumque quae est et et in consequuntur voluptatem voluptates aut

 Kim Elkjær Marcher-Jepsen
Senior Lecturer



At nam consequatur ea labore ea harum

Cupiditate quo est a modi nesciunt soluta ipsa voluptas error itaque dicta in autem qui minus magnam et distinctio eum accusamus ratione error aut

 Birgitte Kirk Iversen
Senior Lecturer



Jes Arbov
Lecturer

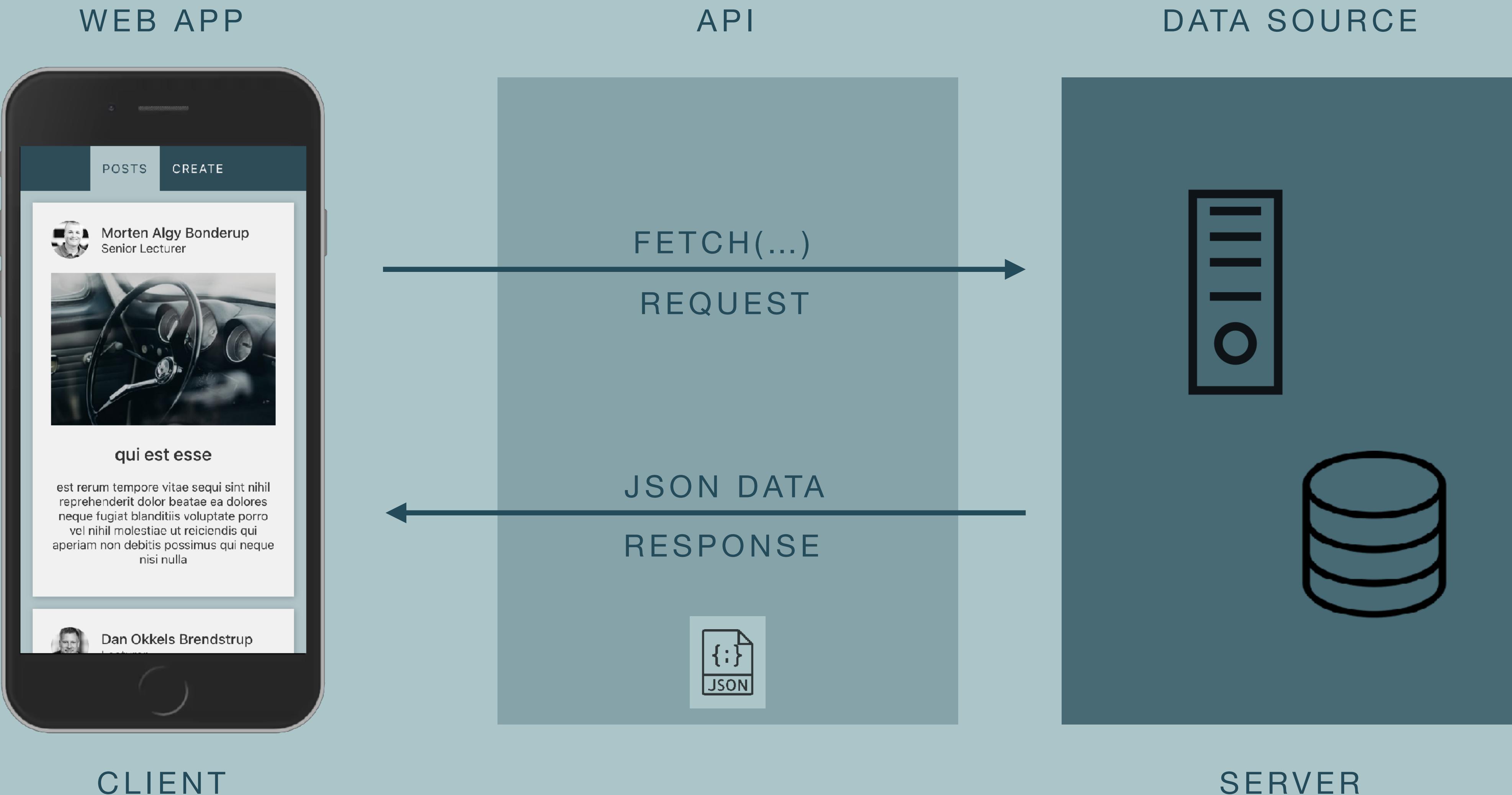


 Maria Louise Bendixen
Senior Lecturer



<https://race-rest.web.app/>

Web Development



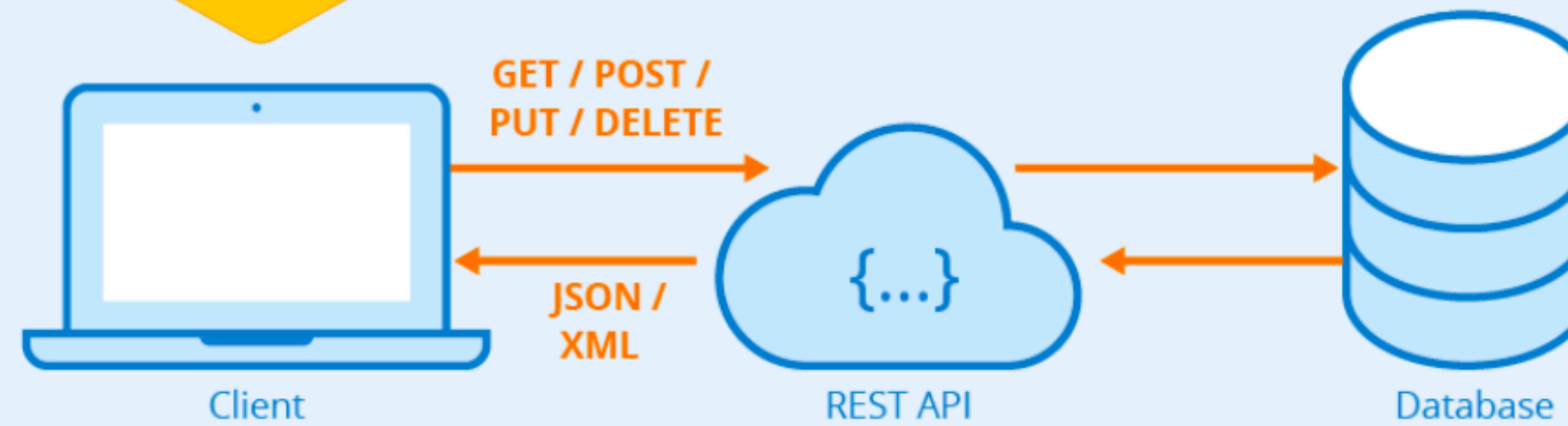
When someone asks you how to get data from a database in a js code



made with mematic

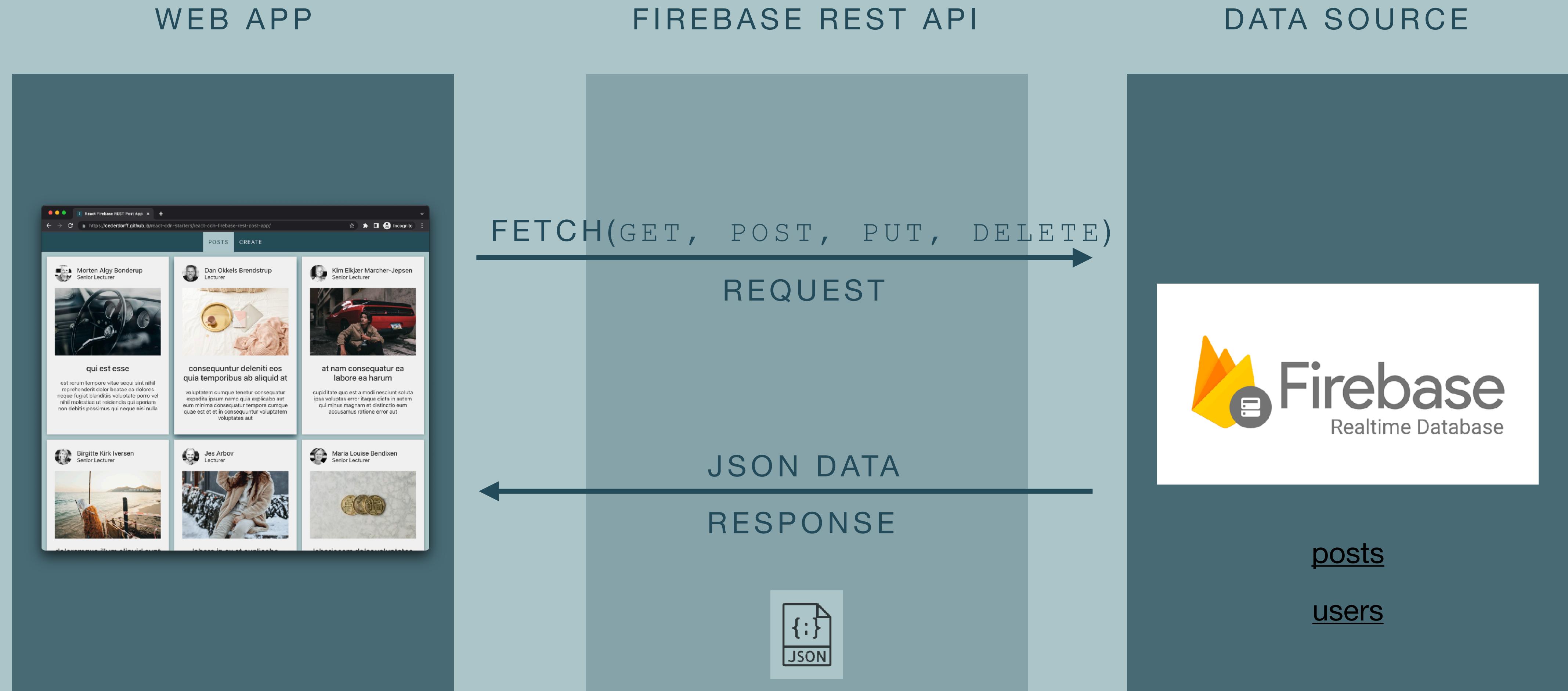


Firebase





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(...)

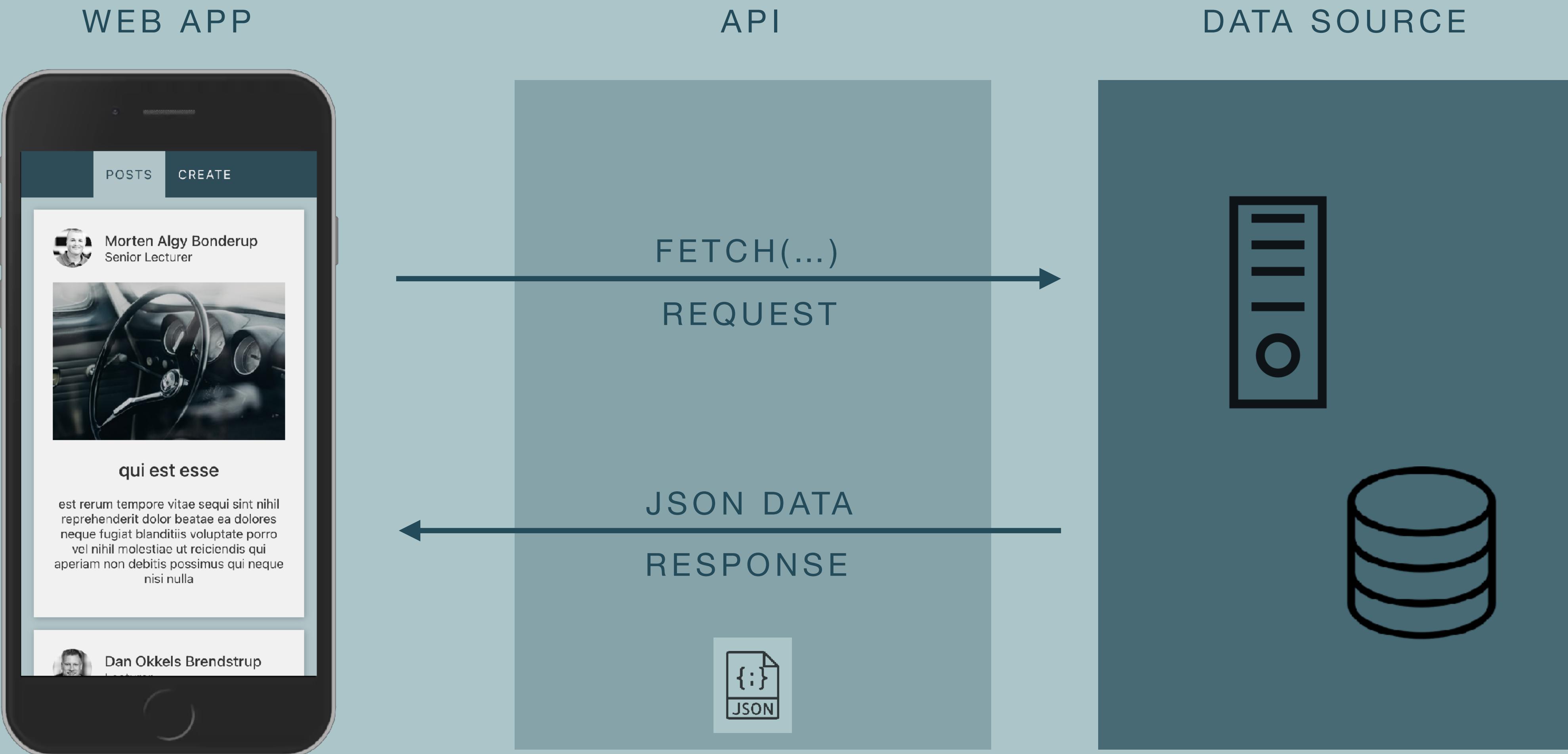
... get & post data from and to a data source
... can perform network requests to a server

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

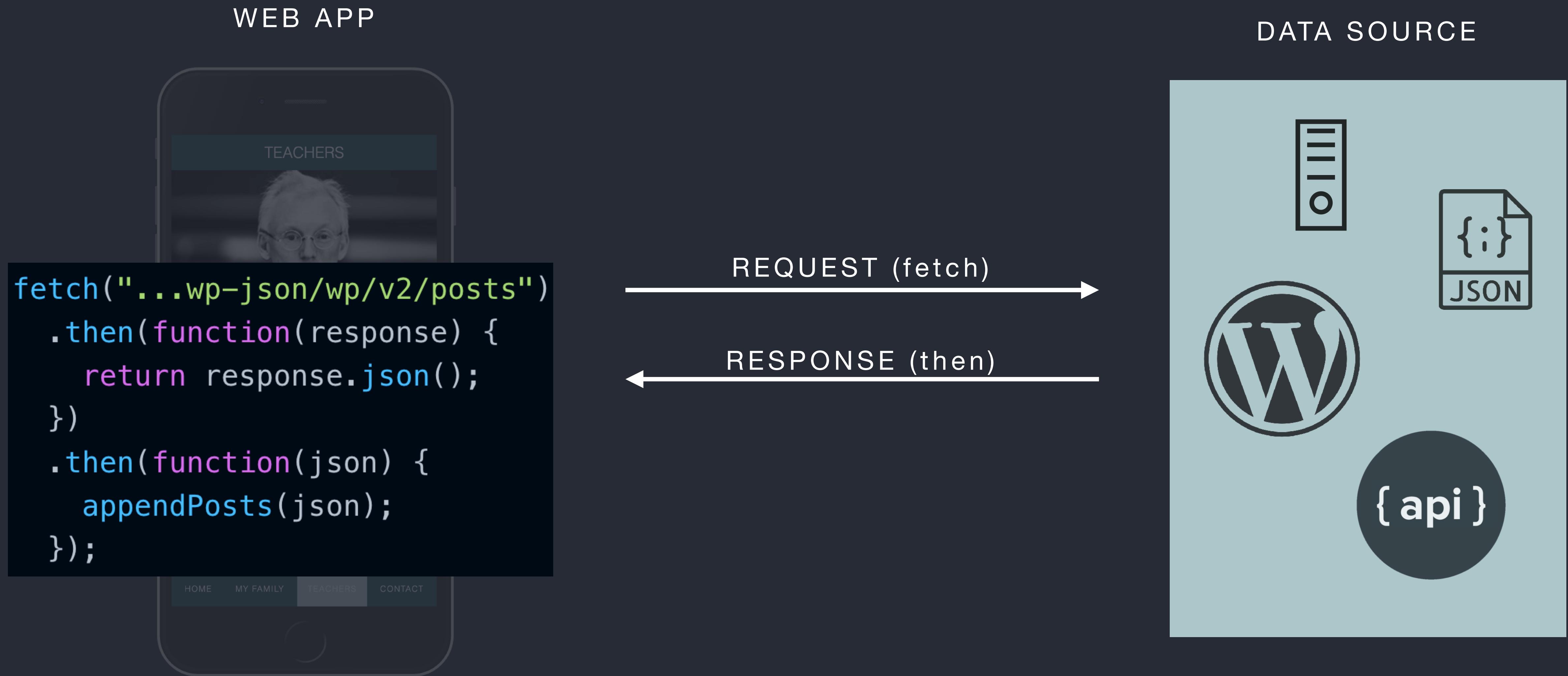
- `url` – the URL to access.
- `options` – optional parameters: method, headers etc.

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;
}

```



```
[
  {
    "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*/
      `


        <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)

The diagram illustrates the flow of data from the `app.js` code to the resulting JSON response. A curved arrow originates from the URL in the `fetch` statement in the `getPerson` function and points to the JSON data on the right. Another curved arrow originates from the `persons` variable in the `displayPersons` function and points back to the `app.js` code.

```
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


```

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

A screenshot of a browser window showing the JSON data received from the URL `https://raw.githubusercontent.com/cederdorff/web-diplom-frontend/main/_data/persons.json`. The JSON array contains eight objects, each representing a person with properties: name, mail, title, and img.

```
[{"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": ""}]
```

request (fetch)

The diagram illustrates the flow of data from the `app.js` code to the final JSON response. A curved arrow labeled "request (fetch)" points from the `fetch` call in the `getPerson` function to the browser window. Another curved arrow labeled "response (JSON)" points from the `response.json` call to the JSON data displayed in the browser.

```
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


```

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

A screenshot of a browser window showing the JSON response from the fetch request. The URL is `https://raw.githubusercontent.com/cederdorff/web-diplom-frontend/main/_data/persons.json`. The response is a JSON array containing eight objects, each representing a person with properties: name, mail, title, and img.

```
[{"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": ""}]
```

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.

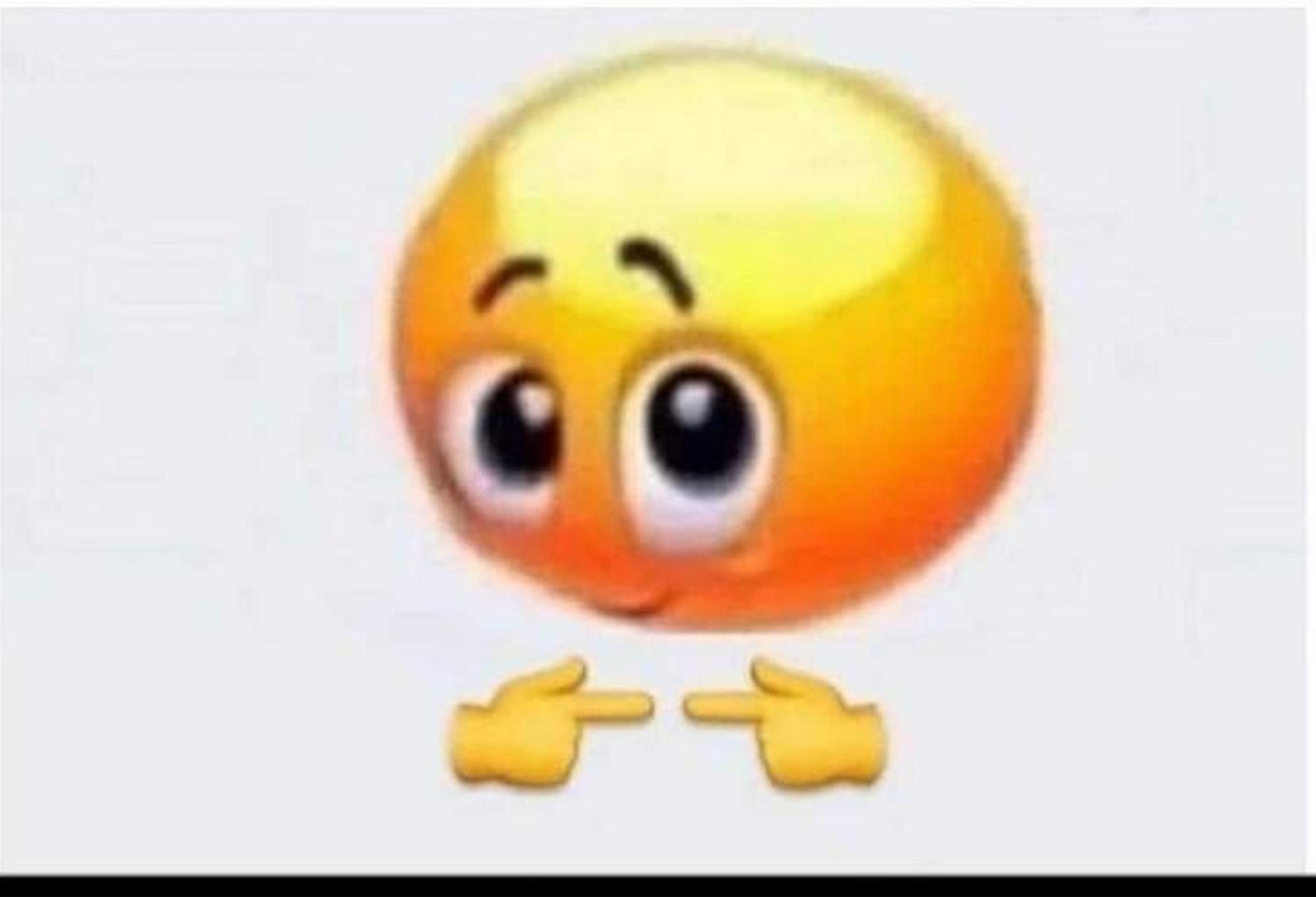
```
// fetch with callbacks
fetch("https://cederdorff.github.io/web-frontend/canvas-users/data.json")
  .then(function (response) {
    return response.json();
  })
  .then(function (data) {
    console.log(data);
  });

```

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

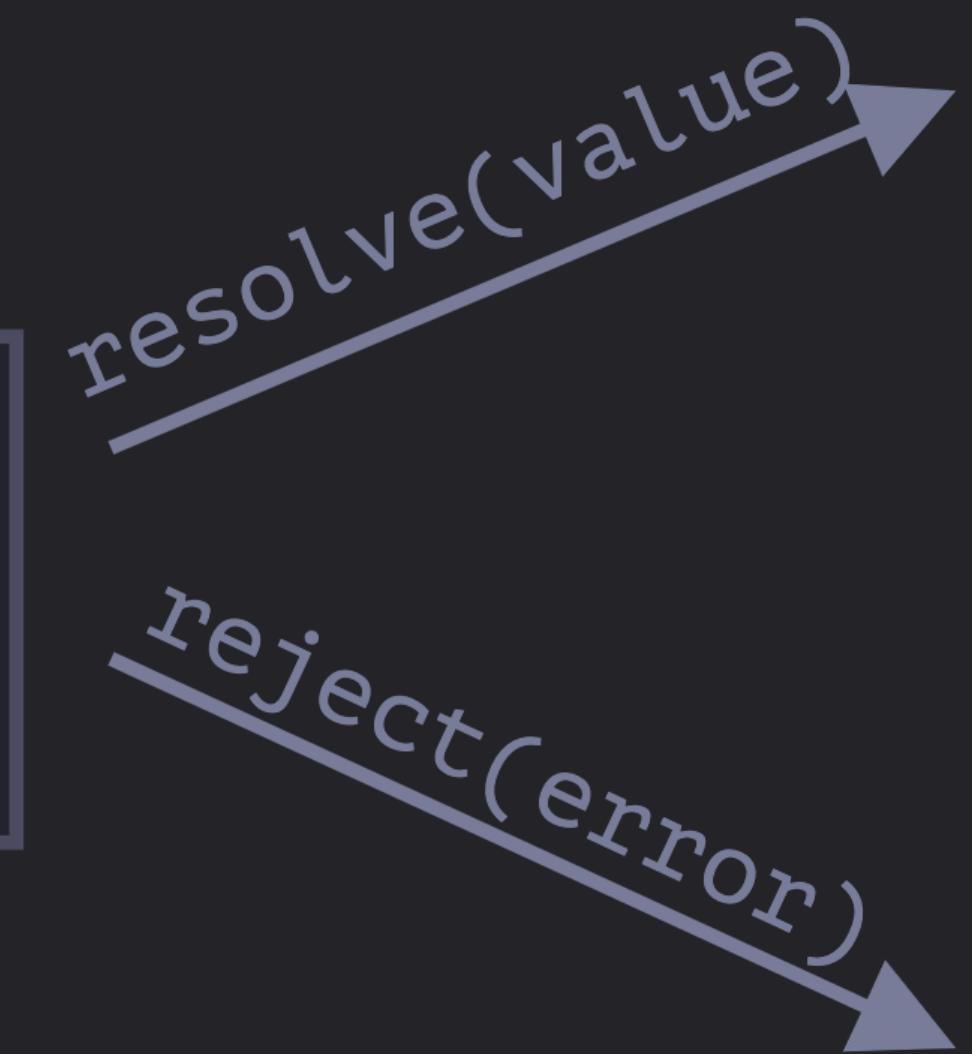
We can use `await` to wait for `fetch` to finish



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.

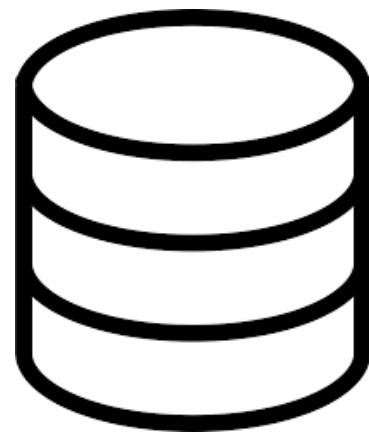
```
function PostsPage() {
  const [posts, setPosts] = useState([]);

  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>
    </section>
  );
}
```

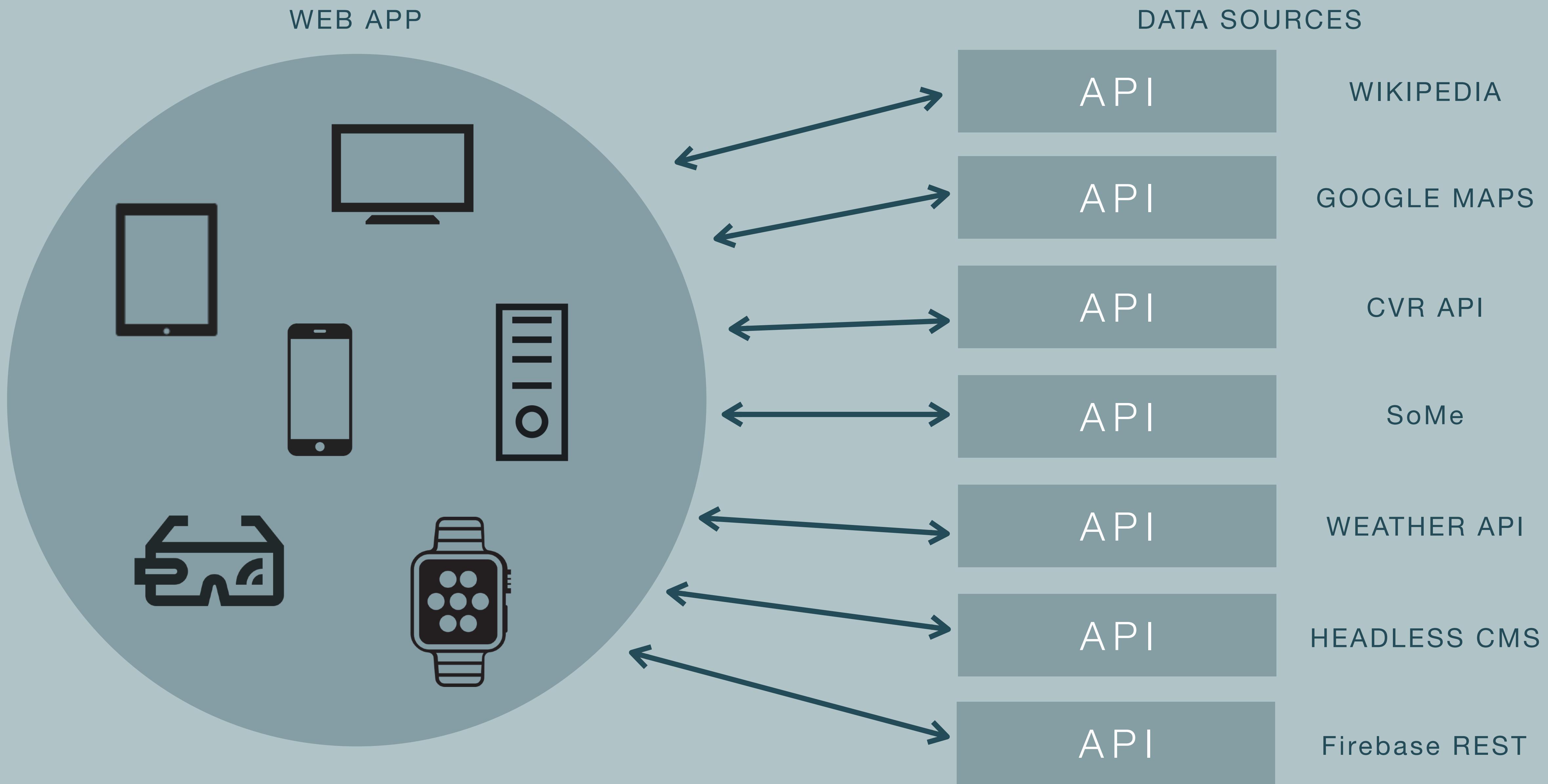
What's a Data Source?

- Location of data
- Where data is coming from
- Can be any kind of data of any file format
- Database, a file, data sheet, spreadsheet, XML, JSON



{JSON}

API



API

Application Programming Interface

A way for systems to communicate - get and post data.

API

APPLICATION PROGRAMMING INTERFACE

The screenshot shows a web browser window titled "API'er - definition og funktion." with the URL "chat.openai.com". The page content is a conversation with AI, starting with a user question "Hvad er en API?" followed by AI responses explaining what APIs are, their functions, and types.

Hvad er en API?

API står for "Application Programming Interface", på dansk "Programmeringsgrænseflade". En API er en samling af software-komponenter, protokoller og værktøjer, der gør det muligt for forskellige programmer eller systemer at kommunikere med hinanden og udveksle data på en standardiseret måde.

API'er kan fungere som en bro mellem forskellige applikationer, platforme eller tjenester, hvilket gør det muligt for udviklere at udnytte funktionalitet eller data fra andre systemer uden at skulle skrive alt koden fra bunden. API'er gør det også muligt at integrere forskellige tjenester og applikationer for at skabe mere komplekse og avancerede systemer.

Der findes mange forskellige typer af API'er, inklusiv web-API'er, biblioteks-API'er og operativsystem-API'er. Web-API'er er en af de mest almindelige typer, og disse gør det muligt for applikationer at kommunikere via internettet ved hjælp af standardiserede protokoller såsom HTTP eller REST.

Send a message...

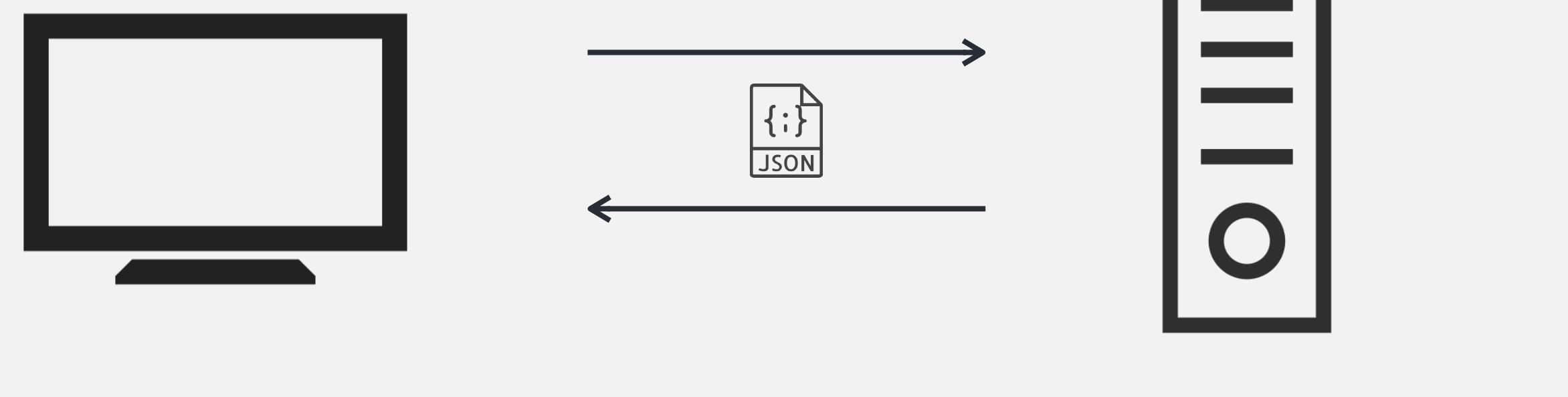
ChatGPT Mar 23 Version. Free Research Preview. ChatGPT may produce inaccurate information about people, places, or facts.

What's an API?

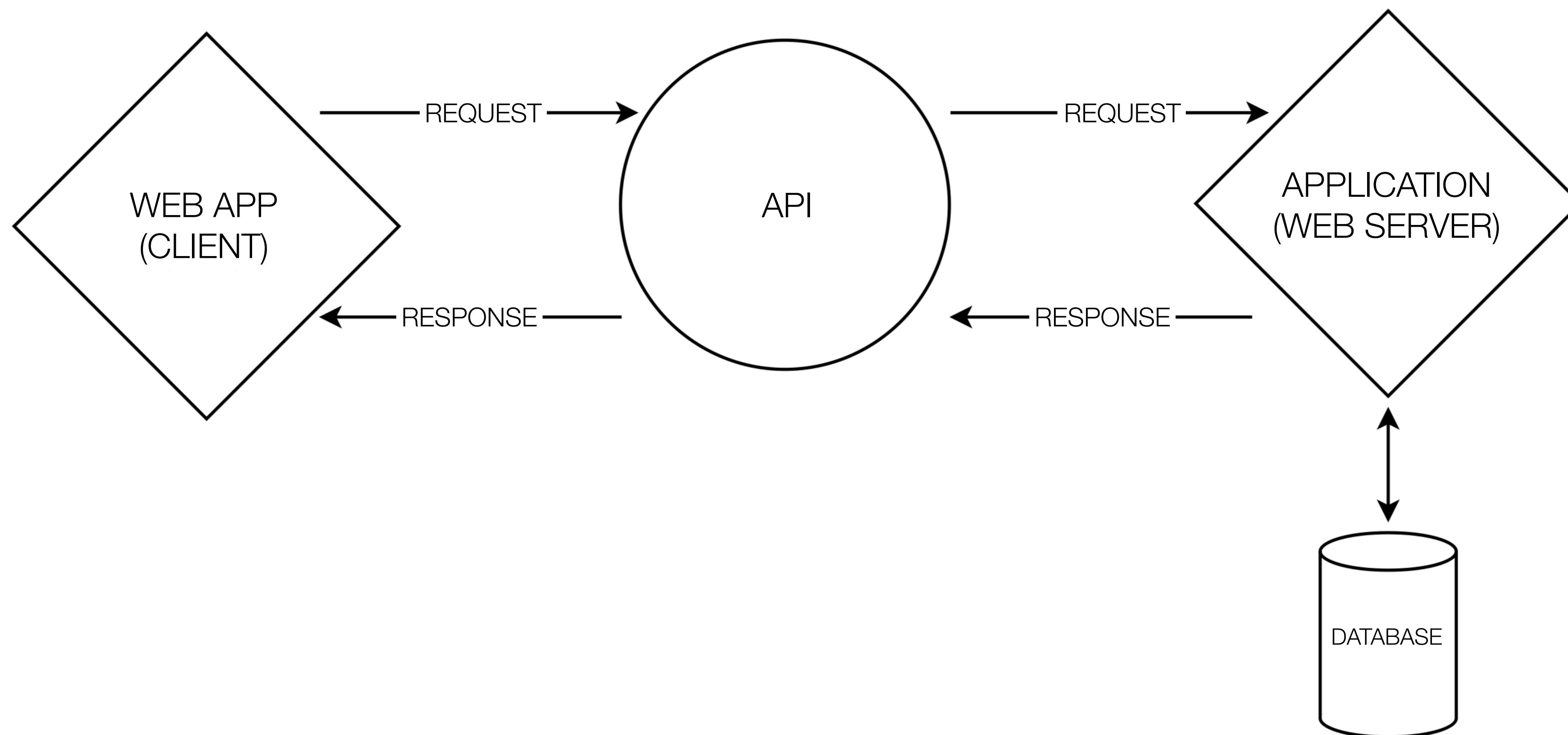
<https://www.youtube.com/watch?v=s7wmiS2mSXY>

Application Programming Interface

- An interface that makes it possible for two systems to communicate and exchange data in a standardised way.
- Ex communication between a web app and a web server.
- An API is the glue or bridge between different applications, platforms, or services.
- Platform independent: Can be used by different clients, devices and users: websites, web apps, mobile apps, webshops and other clients.
- The client does not need to know anything about the service or program provided by the API and vice versa.
- There are many different types of API's: web APIs, library APIs, operating system APIs, etc.
- Web APIs are one of the most common types, allowing applications to communicate over the web using standardised protocols such as HTTP or REST.

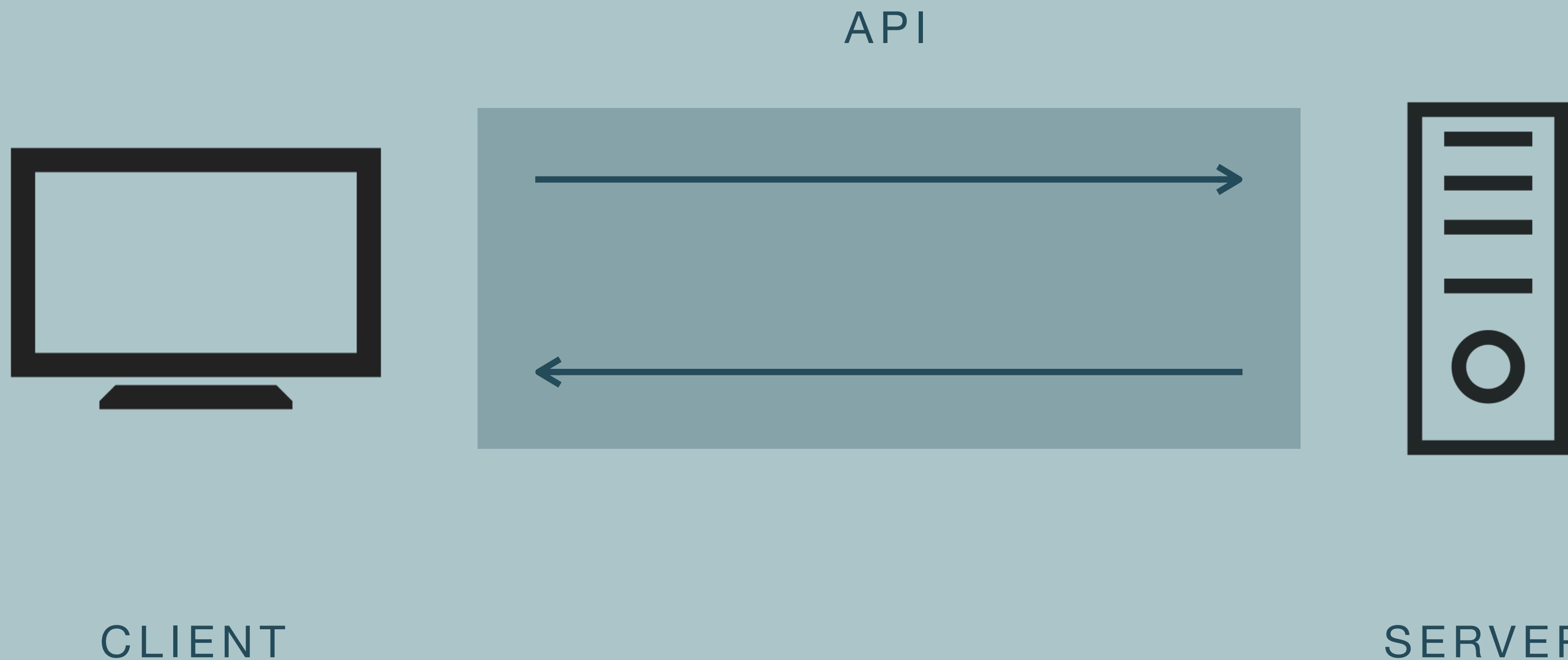


Application Programming Interface



Application Programming Interface

Communication between two systems



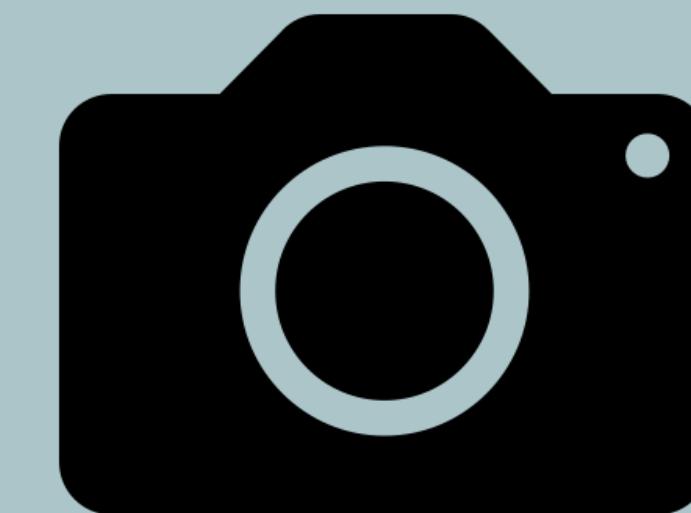
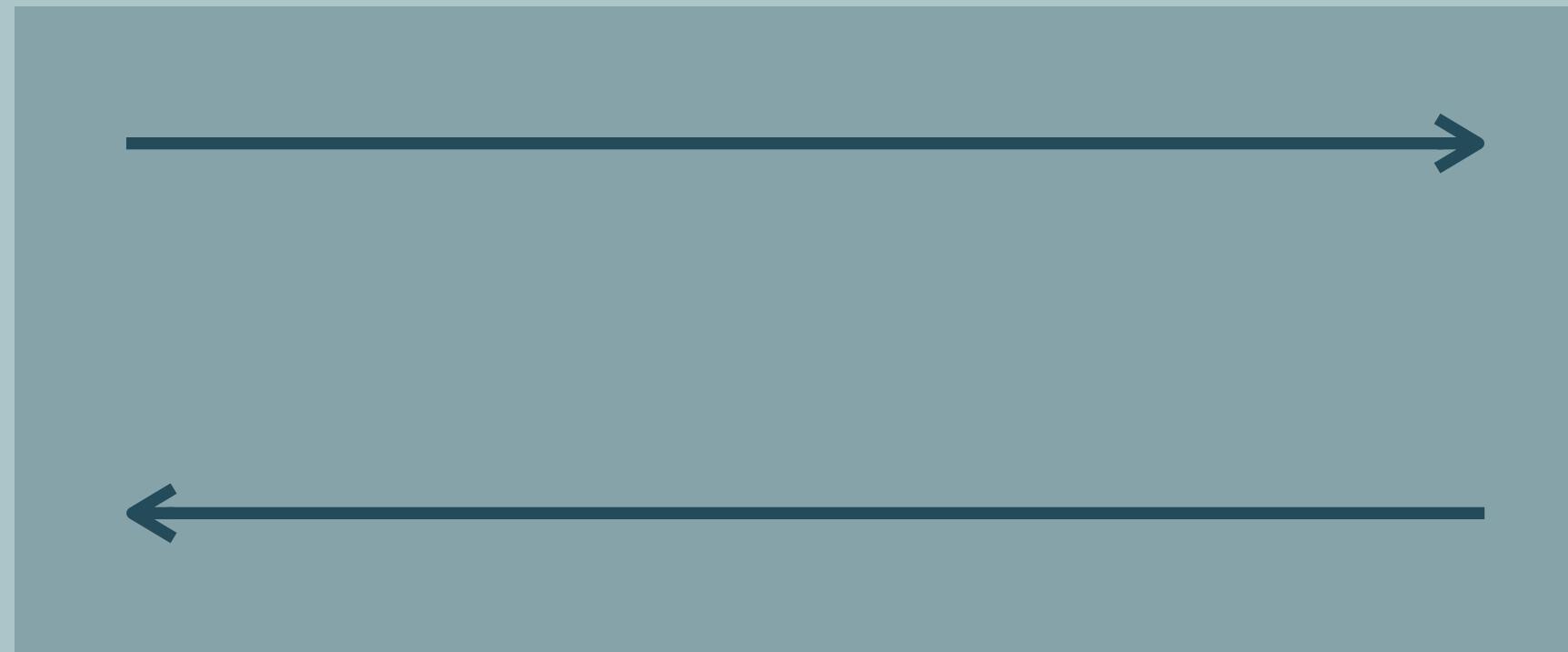
Application Programming Interface

Communication between two systems



Web App

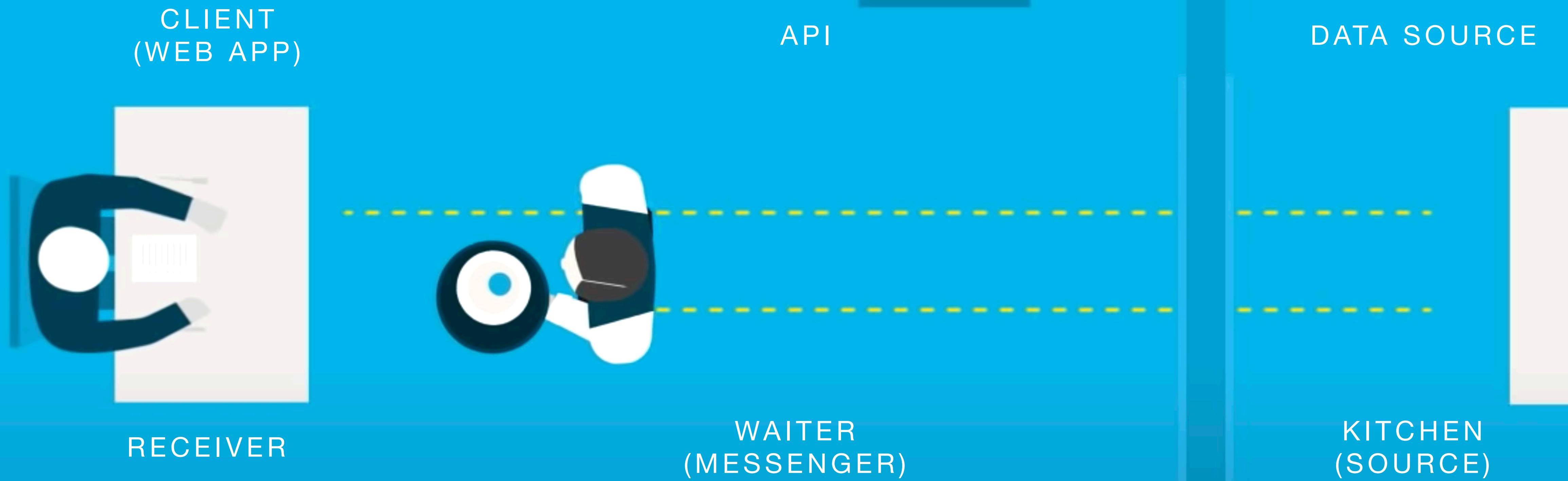
Camera API

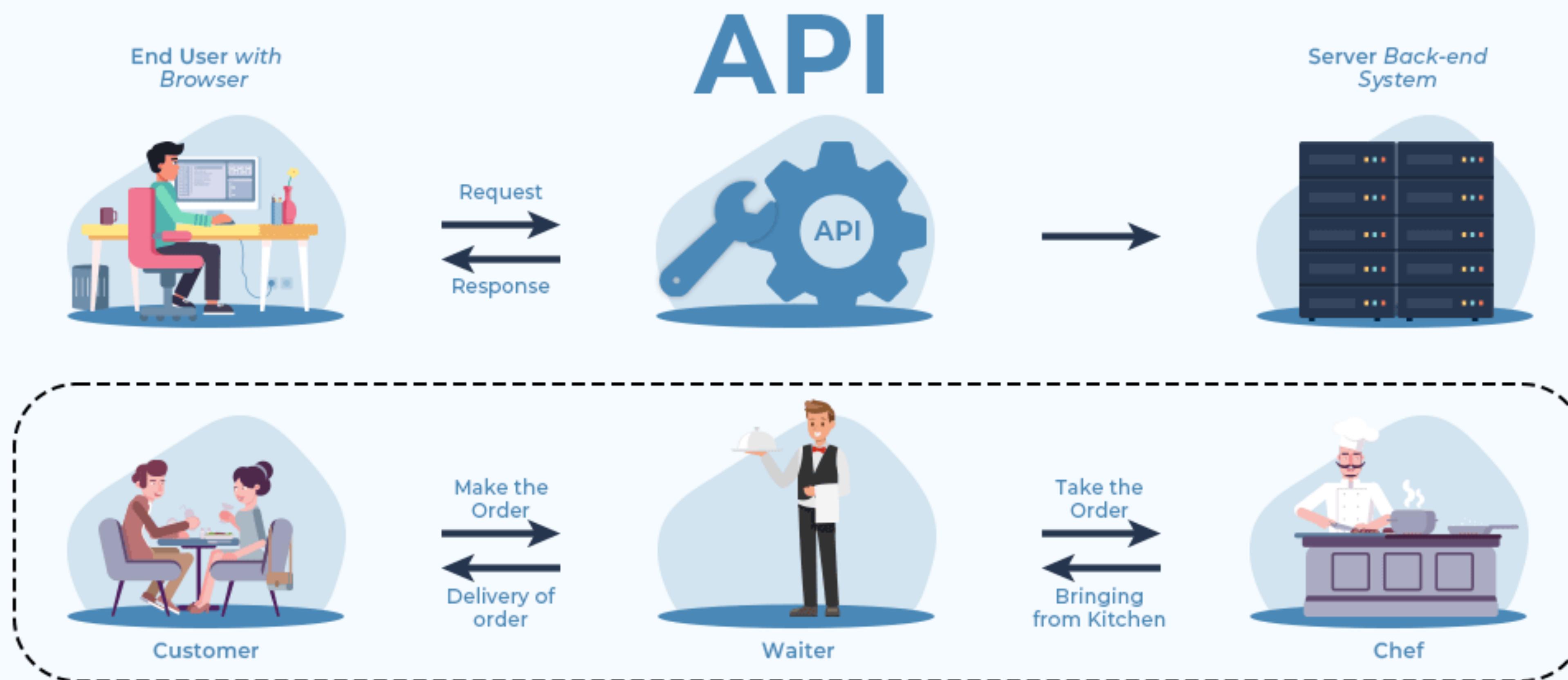


Camera



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

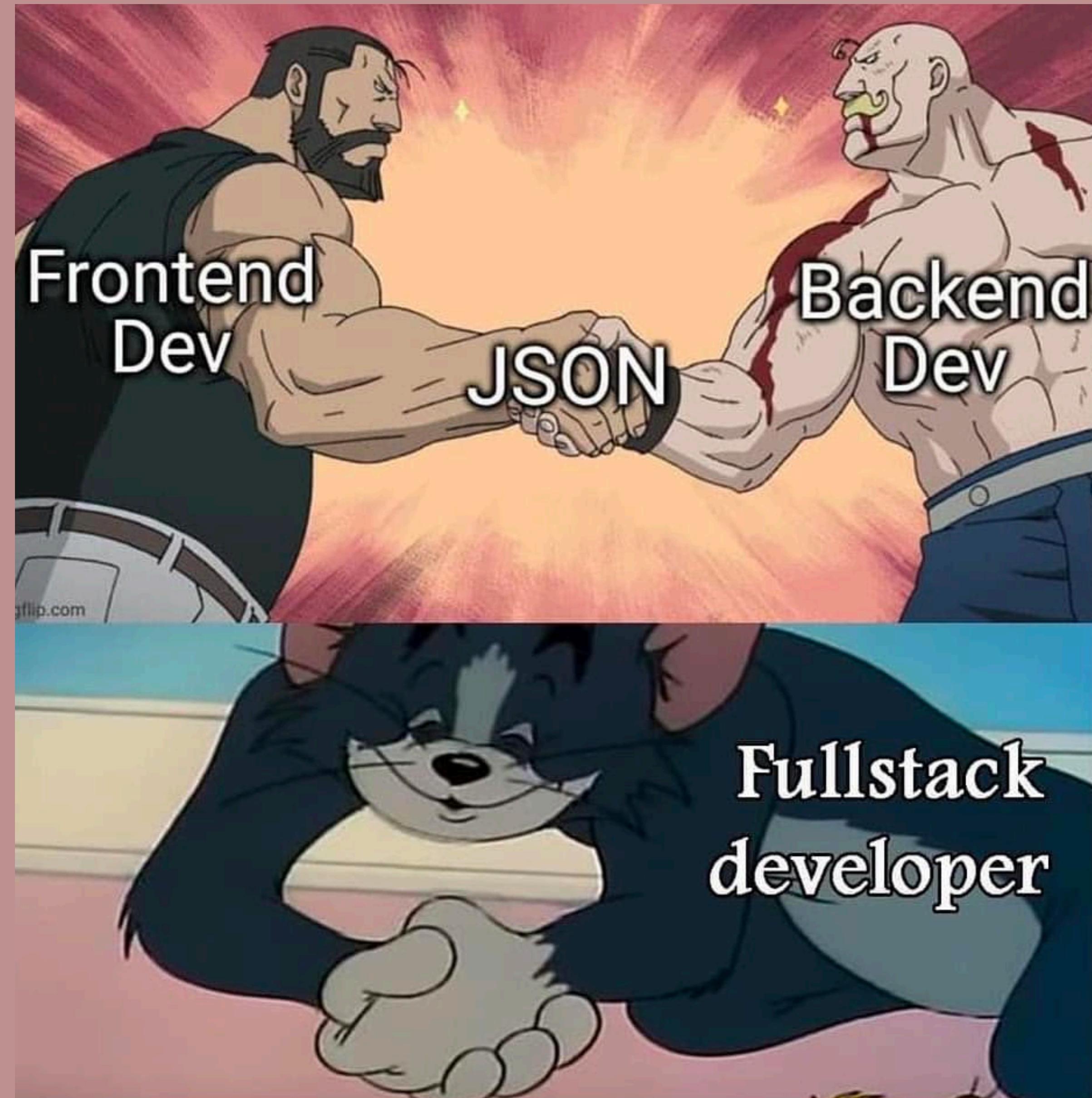




JSON

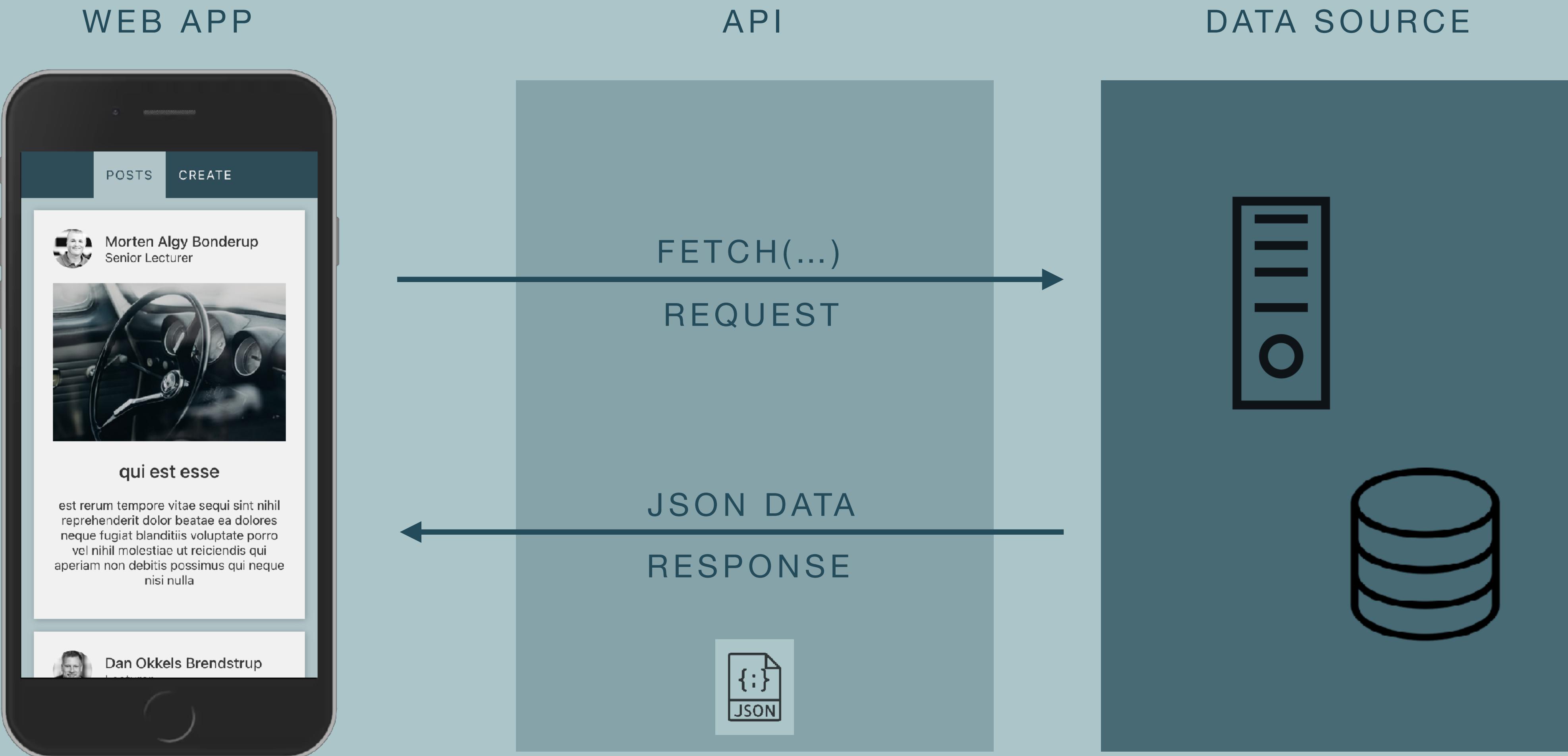
JavaScript Object Notation

... 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

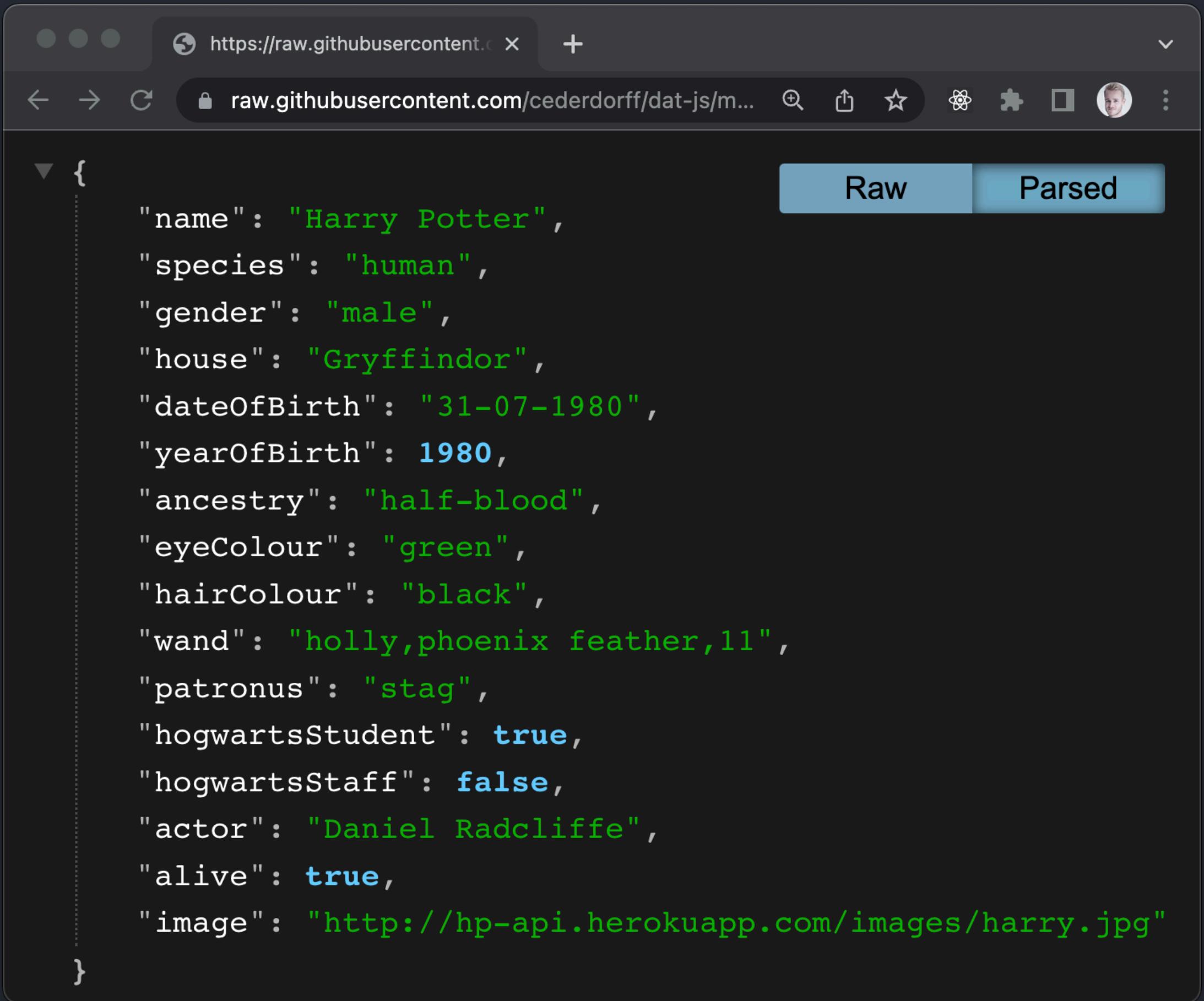
```
{  
  "name": "Alicia",  
  "age": 6  
}
```

JSON OBJECT

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

LIST OF JSON OBJECTS

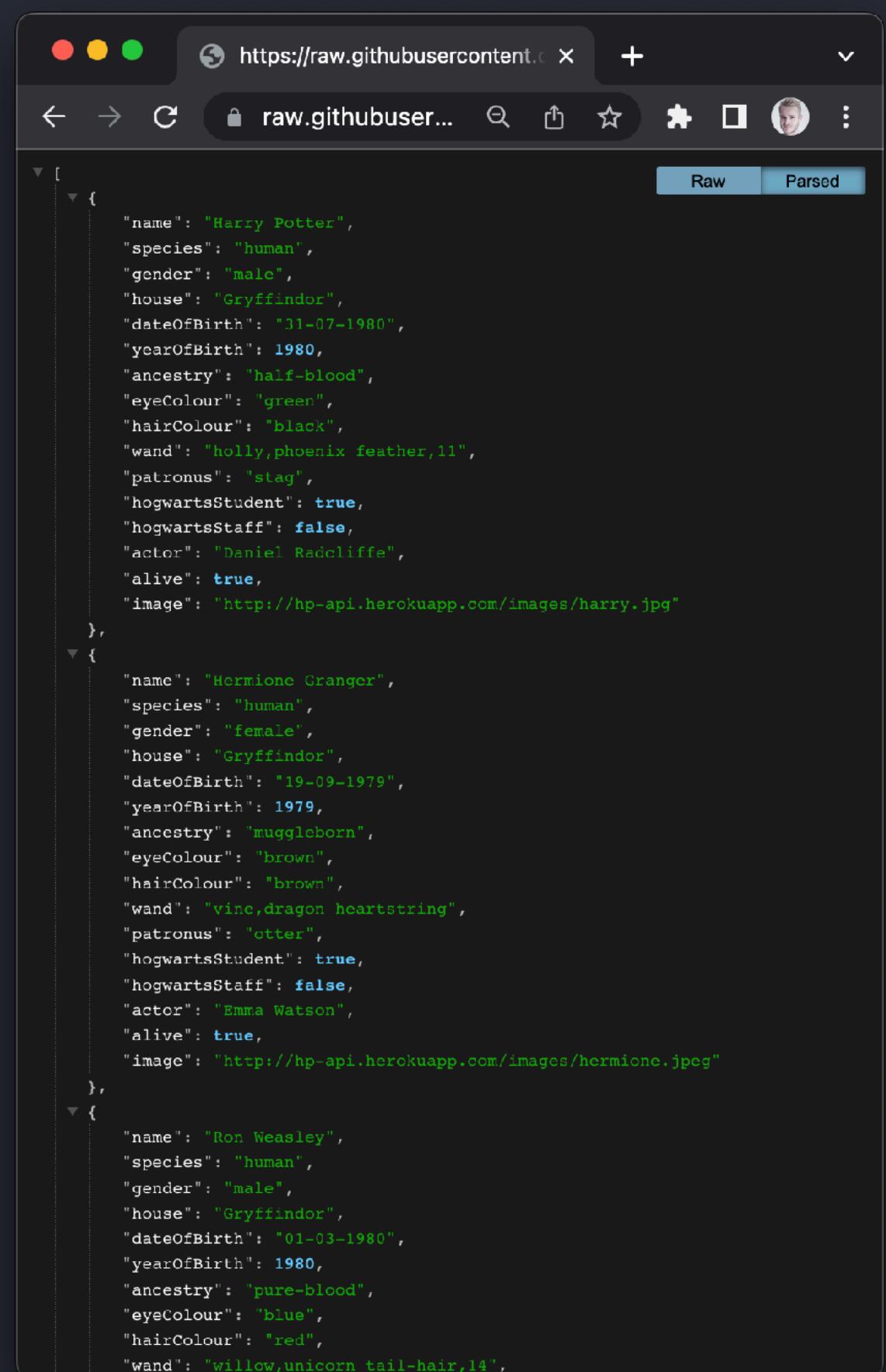
JSON



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 object represents Harry Potter's character details. The 'Raw' tab is selected, showing the raw JSON code, while the 'Parsed' tab shows the same data as an expandable tree structure.

```
{  
  "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"  
}
```

JSON OBJECT



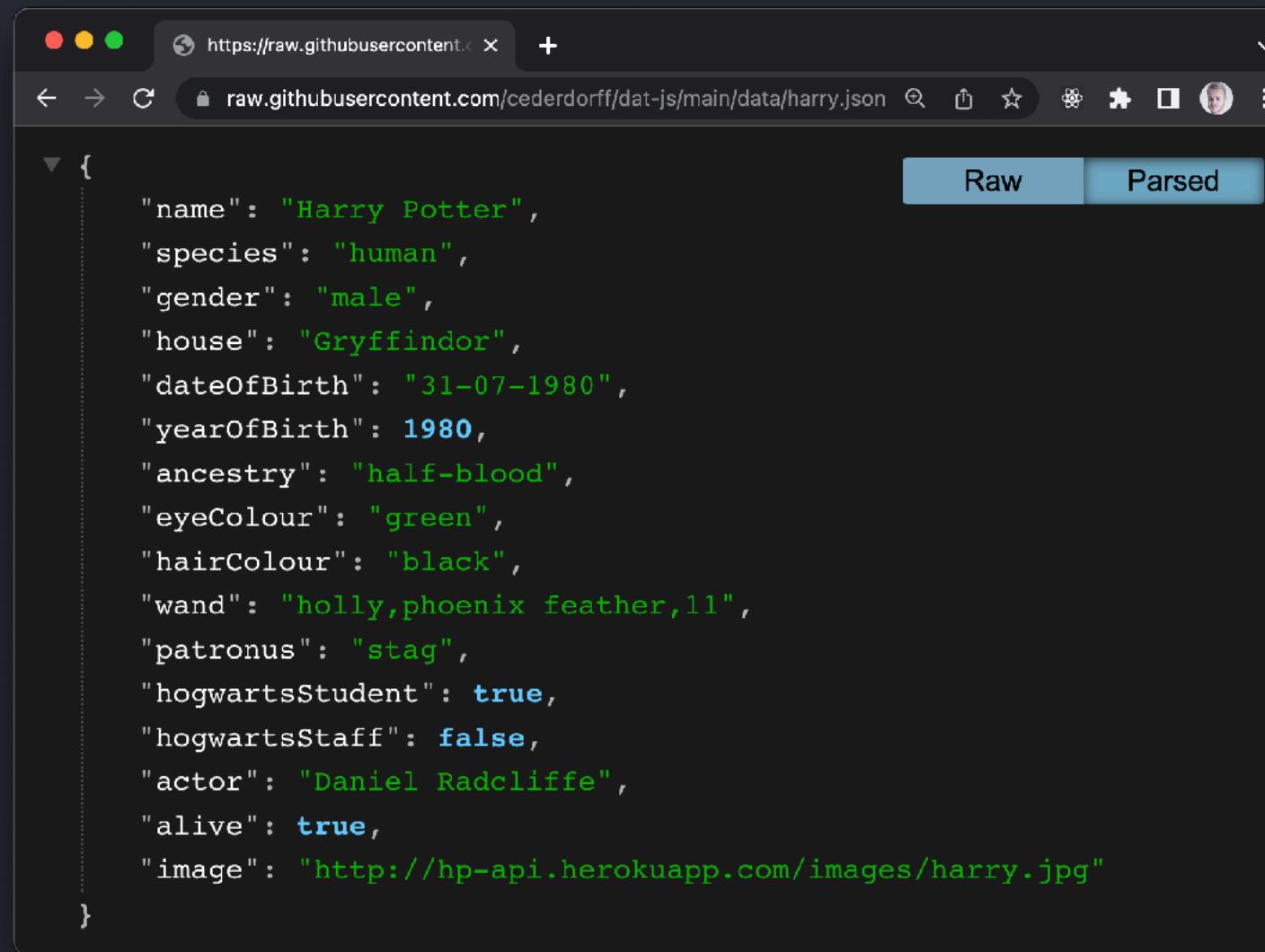
A screenshot of a web browser window displaying a list of three JSON objects. The URL is <https://raw.githubusercontent.com/cederdorff/dat-js/master/data/characters.json>. The JSON array contains three objects: Harry Potter, Hermione Granger, and Ron Weasley. The 'Raw' tab is selected, showing the raw JSON code, while the 'Parsed' tab shows the data as an expandable tree structure.

```
[  
  {  
    "name": "Harry Potter",  
    "species": "human",  
    "gender": "male",  

```

LIST OF JSON OBJECTS

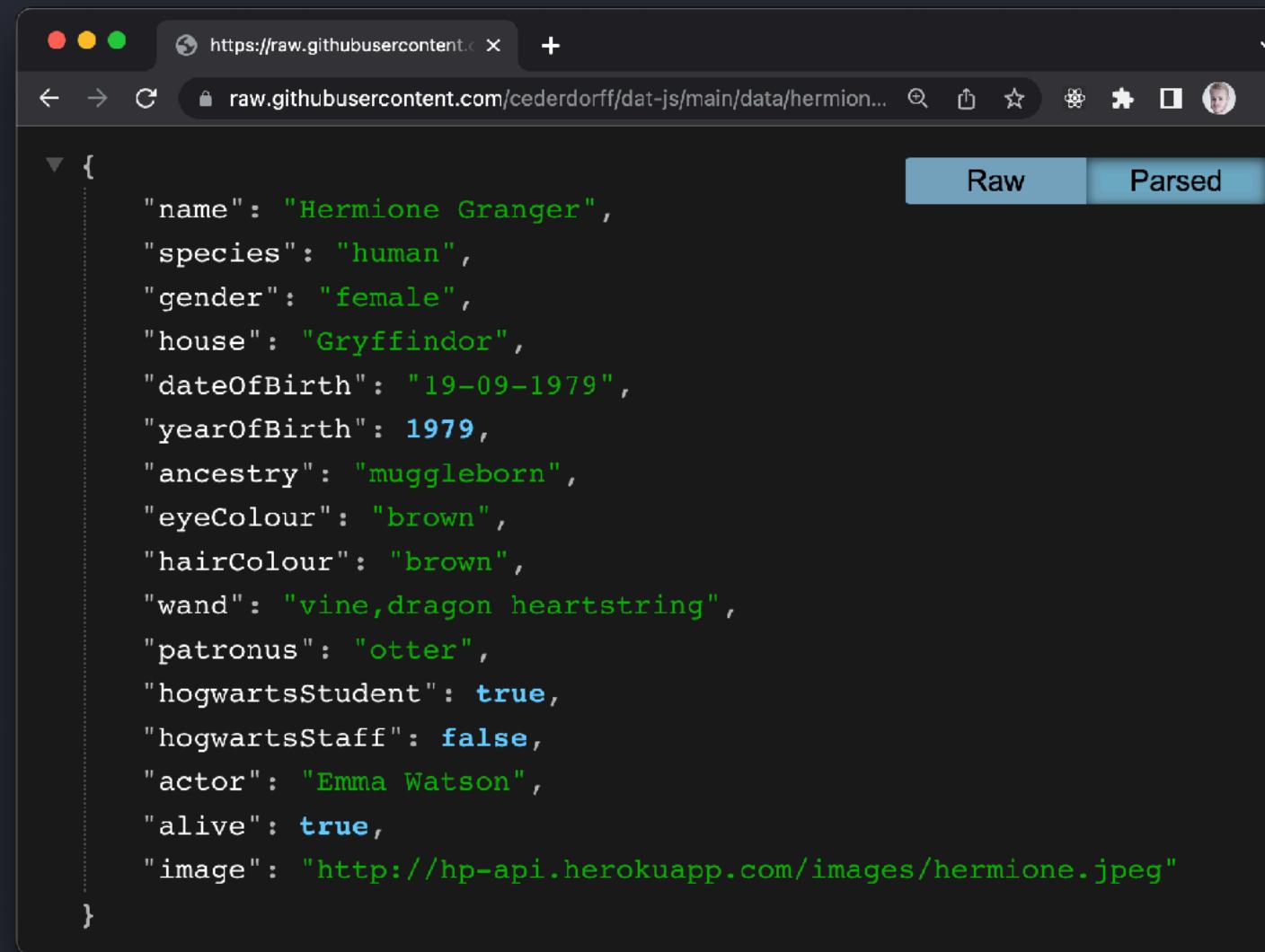
JSON Objects



A screenshot of a web browser displaying a JSON object for Harry Potter. The JSON is shown in a dark-themed code editor interface with syntax highlighting. The object contains properties such as name, species, gender, house, date of birth, year of birth, ancestry, eye colour, hair colour, wand, patronus, Hogwarts student status, staff status, actor, alive status, and image URL. A 'Raw' and 'Parsed' button is visible at the top right.

```
Raw Parsed
{
  "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"
}
```

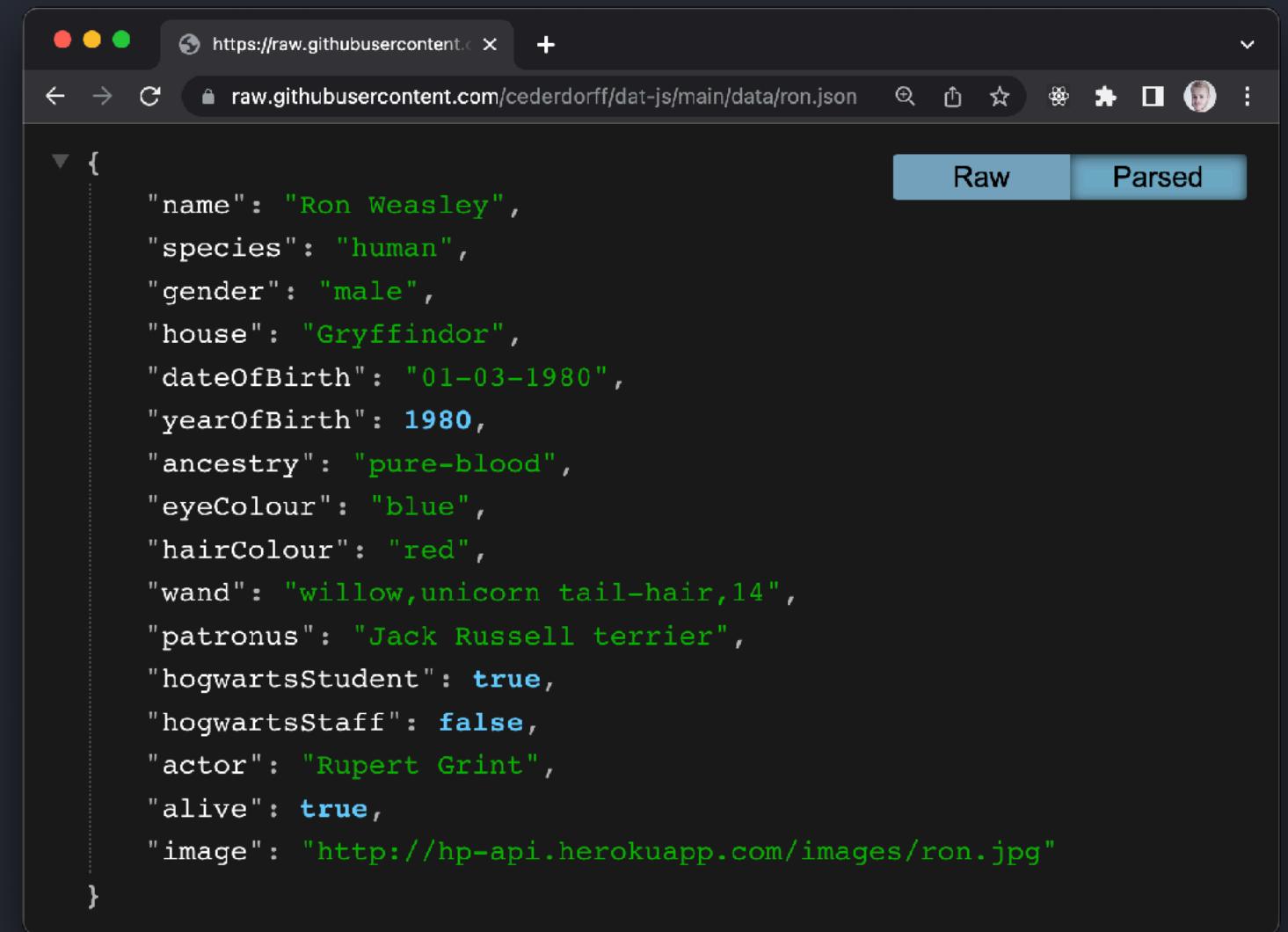
<https://raw.githubusercontent.com/cederdorff/dat-js/main/data/harry.json>



A screenshot of a web browser displaying a JSON object for Hermione Granger. The JSON is shown in a dark-themed code editor interface with syntax highlighting. The object contains properties such as name, species, gender, house, date of birth, year of birth, ancestry, eye colour, hair colour, wand, patronus, Hogwarts student status, staff status, actor, alive status, and image URL. A 'Raw' and 'Parsed' button is visible at the top right.

```
Raw Parsed
{
  "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"
}
```

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



A screenshot of a web browser displaying a JSON object for Ron Weasley. The JSON is shown in a dark-themed code editor interface with syntax highlighting. The object contains properties such as name, species, gender, house, date of birth, year of birth, ancestry, eye colour, hair colour, wand, patronus, Hogwarts student status, staff status, actor, alive status, and image URL. A 'Raw' and 'Parsed' button is visible at the top right.

```
Raw Parsed
{
  "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",
  "patronus": "Jack Russell terrier",
  "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/ron.json>

JSON Object

The diagram illustrates the relationship between a JSON object and a web application. On the left, a screenshot of a browser window shows a character card for Harry Potter from the Harry Potter Characters application. The card displays a portrait of Harry Potter, his name, house affiliation (Gryffindor), and a link to his JSON representation. A red arrow points from the JSON object on the right to this link. On the right, a screenshot of a browser window shows the raw JSON data for Harry Potter. The JSON object contains detailed information about Harry, including his name, species, gender, house, date of birth, year of birth, ancestry, eye and hair color, wand details, patronus, Hogwarts status, actor, and current status. The JSON is presented in a collapsible tree view with a 'Raw' and 'Parsed' tab.

```
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>

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

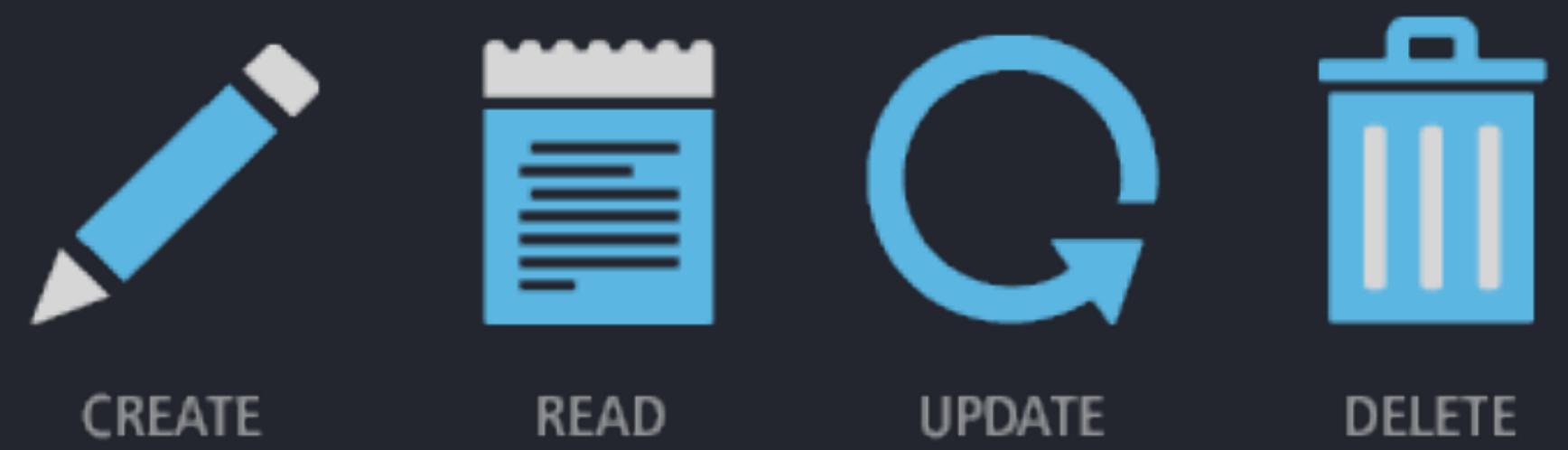
```
const user = {  
    name: "John",  
    age: 30,  
    gender: "male",  
    lookingFor: "female"  
};  
  
// === JSON.stringify === //  
const jsonUser = JSON.stringify(user);  
console.log(jsonUser); // {"name":"John","age":30,"gender":"male","lookingFor":"female"}  
  
// === JSON.parse ===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

Fetch and create post

newPost object is parsed
to JSON and posted to the
ressource wrapped on
body

```
// 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
});
```



C R U D

What's CRUD?

- CREATE objects like a post, user, movie, product, etc.
- READ objects like an array (or object) of objects (posts, users, movies, products, etc)
- UPDATE an object, often given by a unique id.
- DELETE an object, often given by a unique id.

REST API Design & CRUD



CRUD operations

Create

/api/products **POST**

Read

/api/products **GET**

Update

/api/products/:id **PUT**

Delete

/api/products/:id **DELETE**

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



**What are
REST APIs?**

Dataforsyningen REST API

... an example of a REST API



<https://dataforsyningen.dk/>

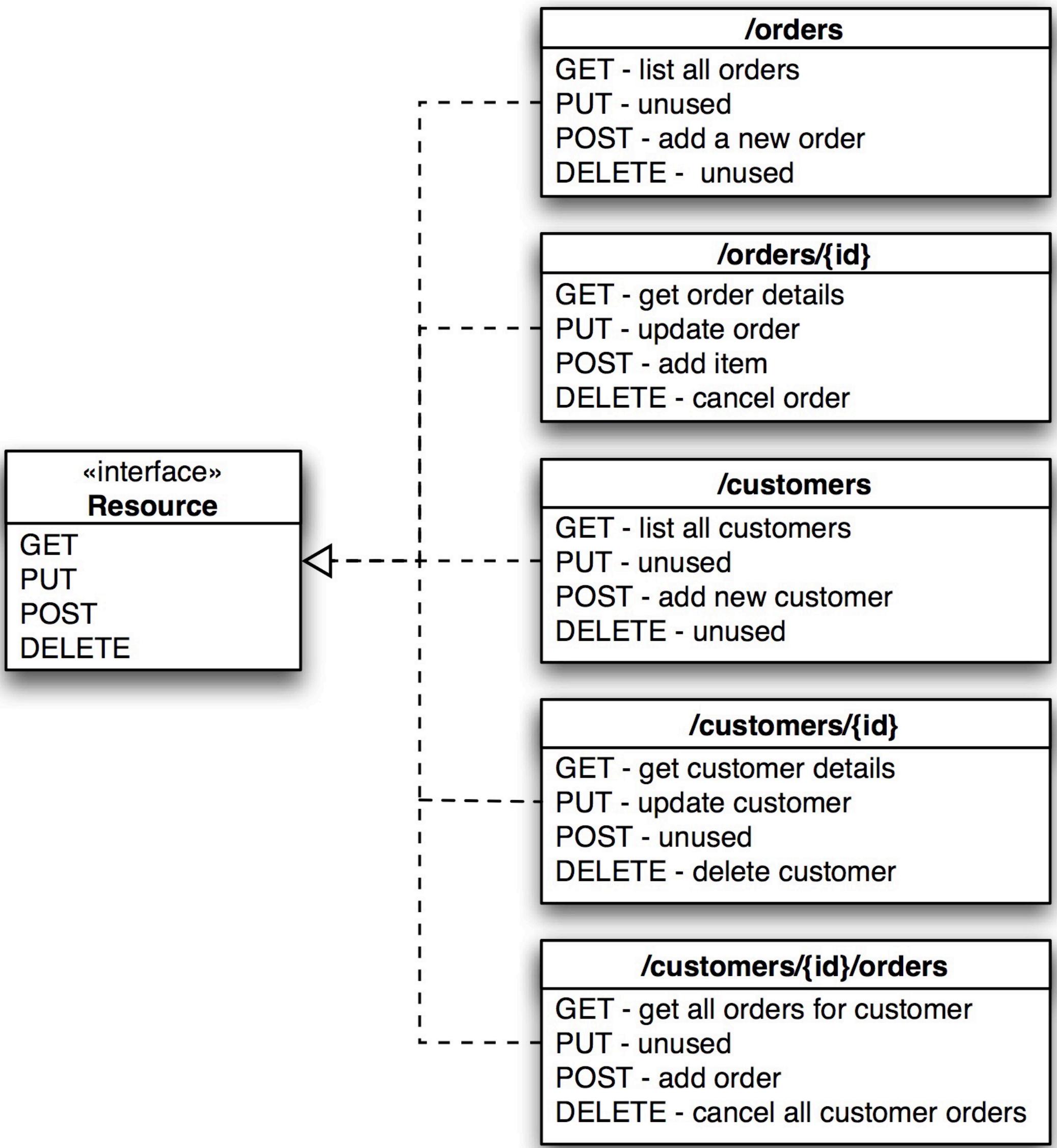
The screenshot shows the REST API documentation page. The title "Dataforsyningen REST API" is at the top. Below it, there are links to the API (<https://api.dataforsyningen.dk/>), documentation (<https://dataforsyningen.dk/>), and a data overview (<https://dataforsyningen.dk/data>). A section titled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

[Dataforsyningen REST API](#)

REpresentational State Transfer

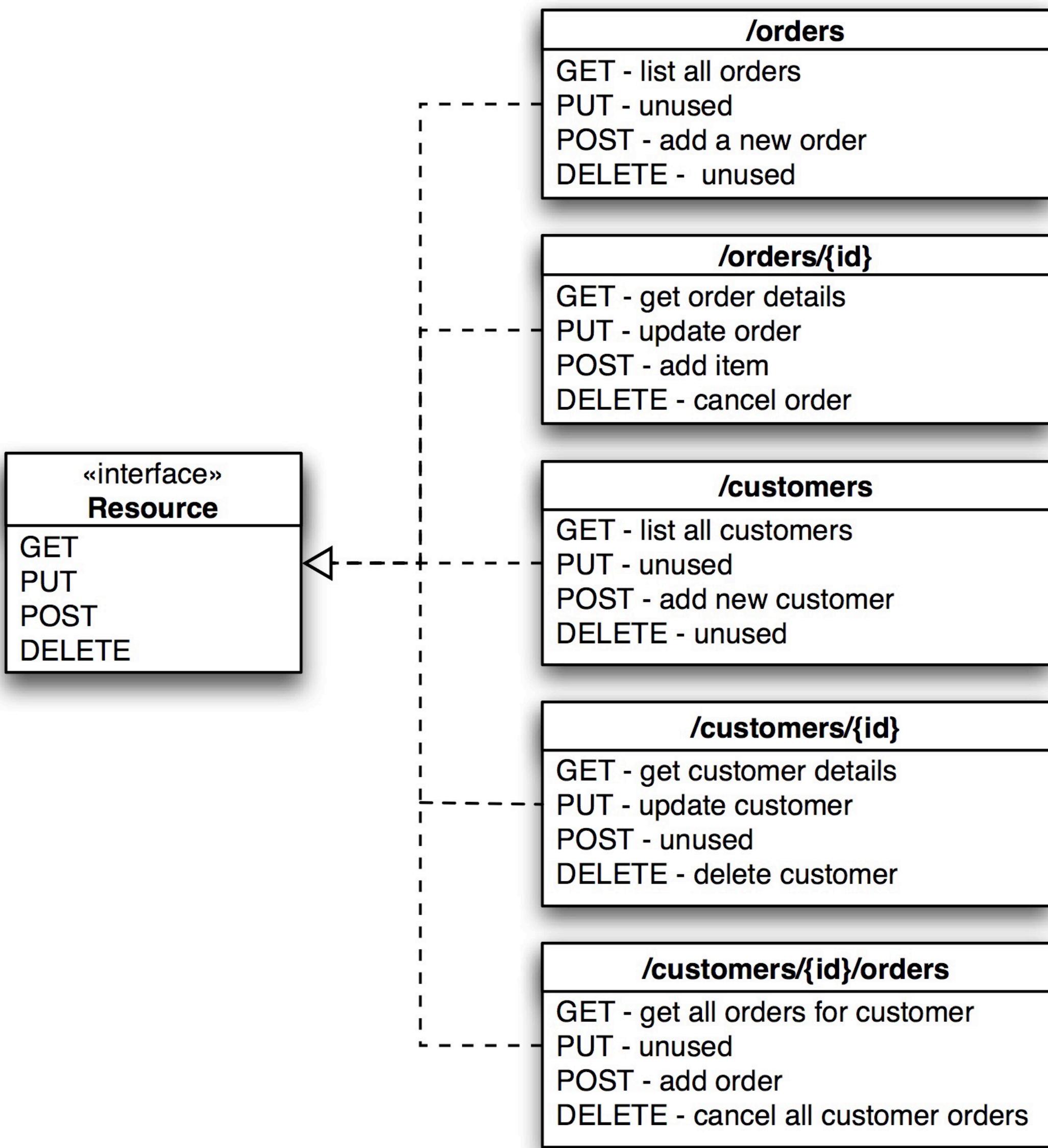
- Clean, semantic URL:
 - <http://example.com/products/2/25> instead of:
 - [http://example.com/products?
category=2&id=25](http://example.com/products?category=2&id=25)
- Basic HTTP requests to perform create, read, update and delete with the HTTP methods GET, POST, PUT and DELETE

Principles of REST



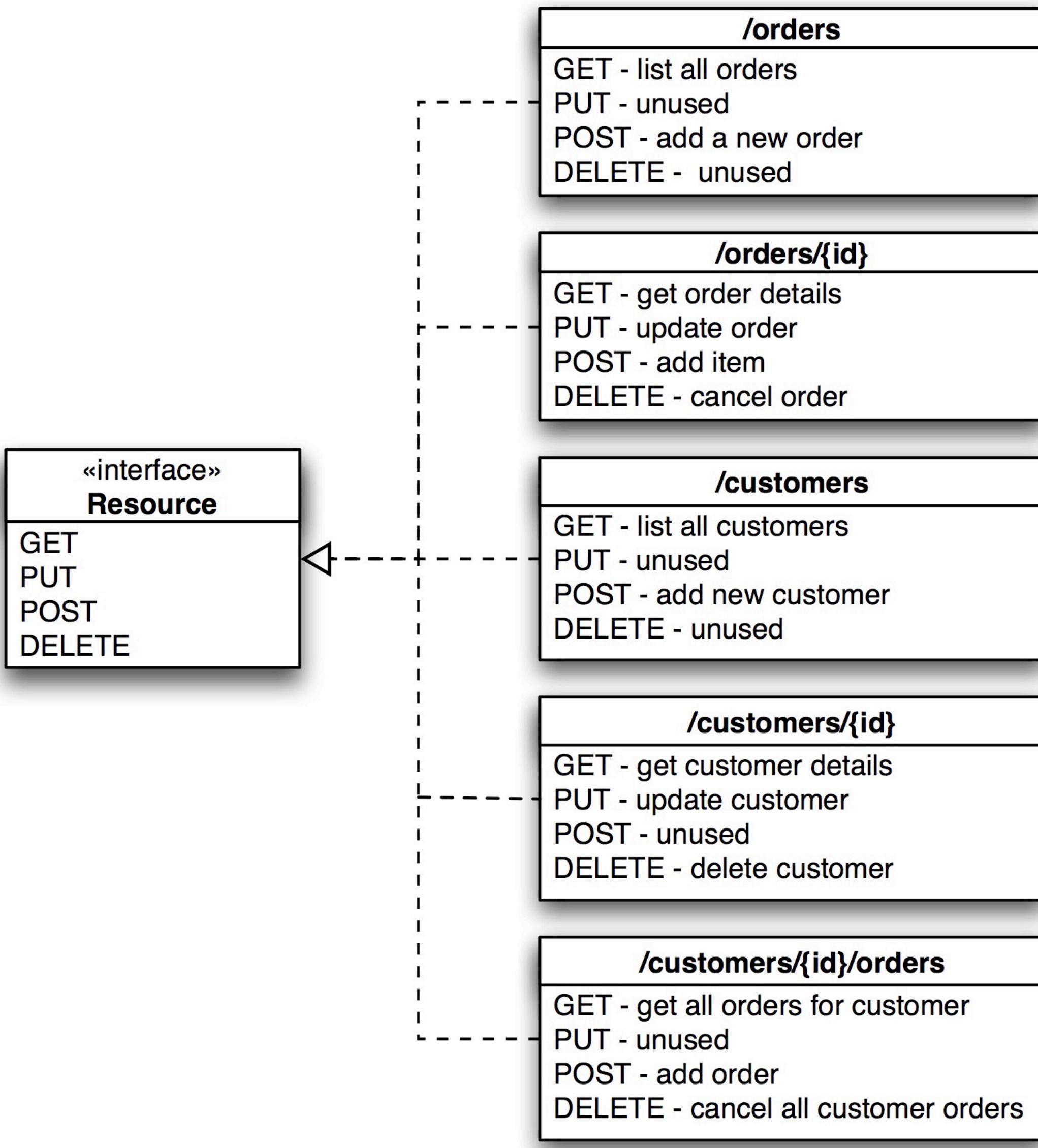
- REST defines a set of rules and guidelines on how you can interact with an API:
 - With REST, each piece of data in the database is treated as a resource. It has a unique id and URL. The URL structure represents the hierarchy of the data, allowing you to access specific data nodes.
 - You can use standard 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



- Resource identification:** Each piece of data in the Firebase Realtime Database 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.
- HTTP methods:** RESTful APIs utilize standard HTTP methods to perform operations on resources. The Firebase Realtime Database 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.
- Stateless communication:** Each request sent to the Firebase Realtime Database 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.
- Data format:** The data exchanged with the Firebase Realtime Database REST API is typically in JSON format. JSON provides a lightweight and flexible way to represent structured data.

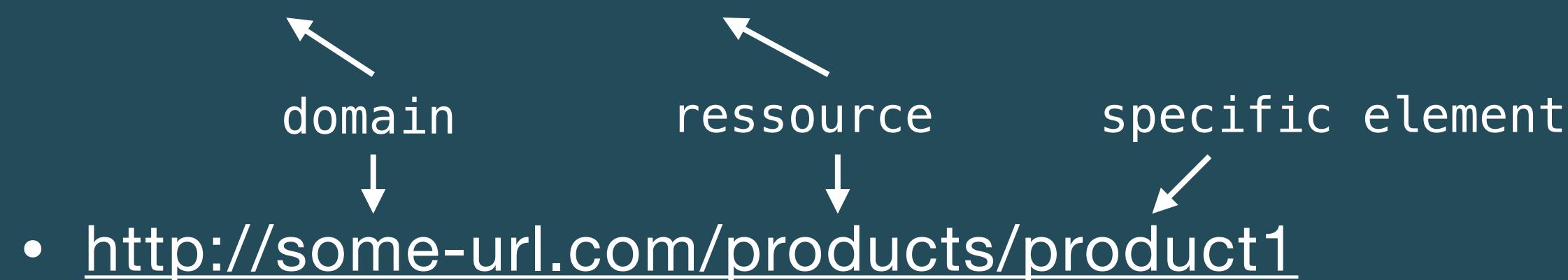
Principles of REST



- 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

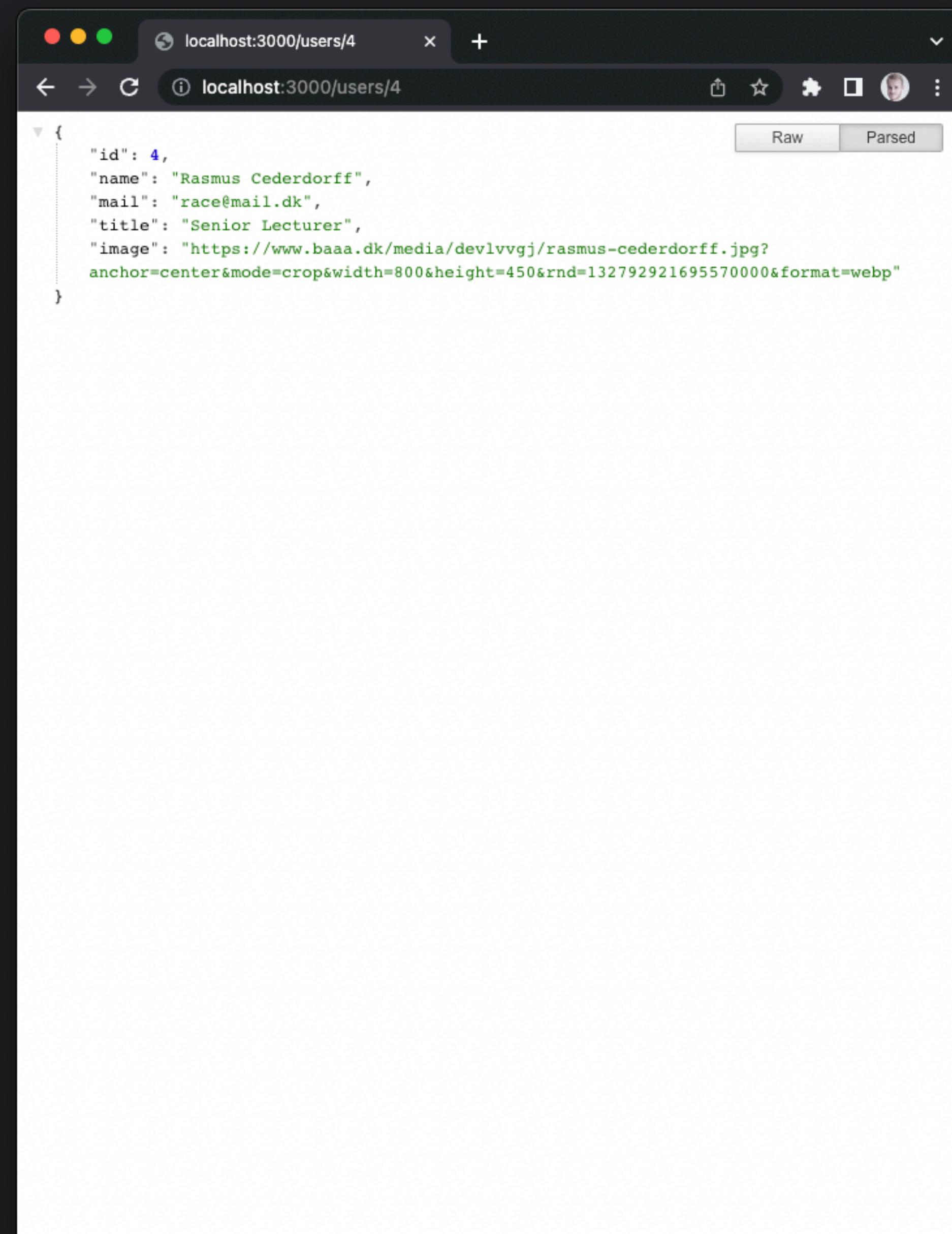
Collections (JSON Array)



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"}
```

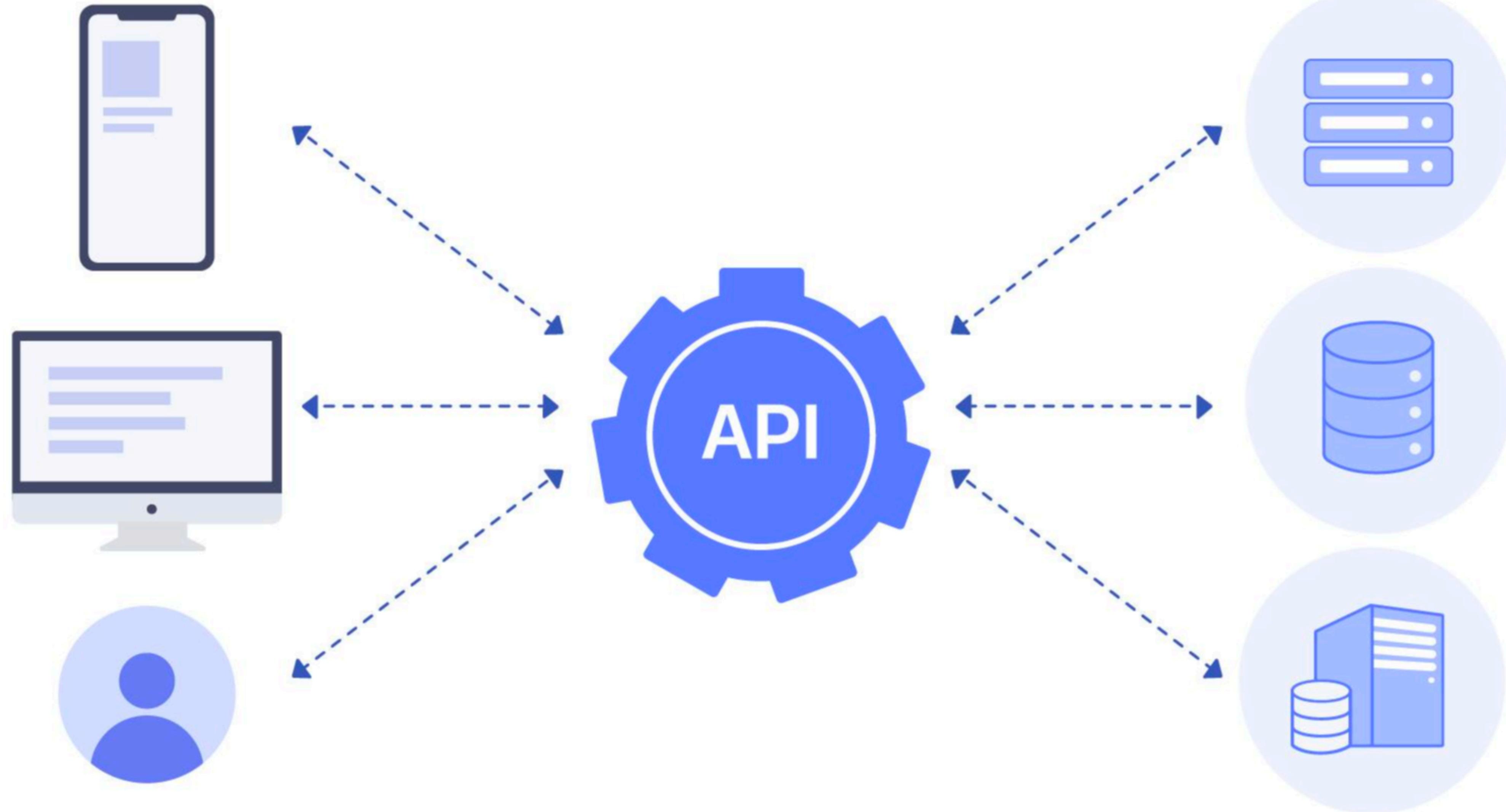
Advantages of REST

- 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.

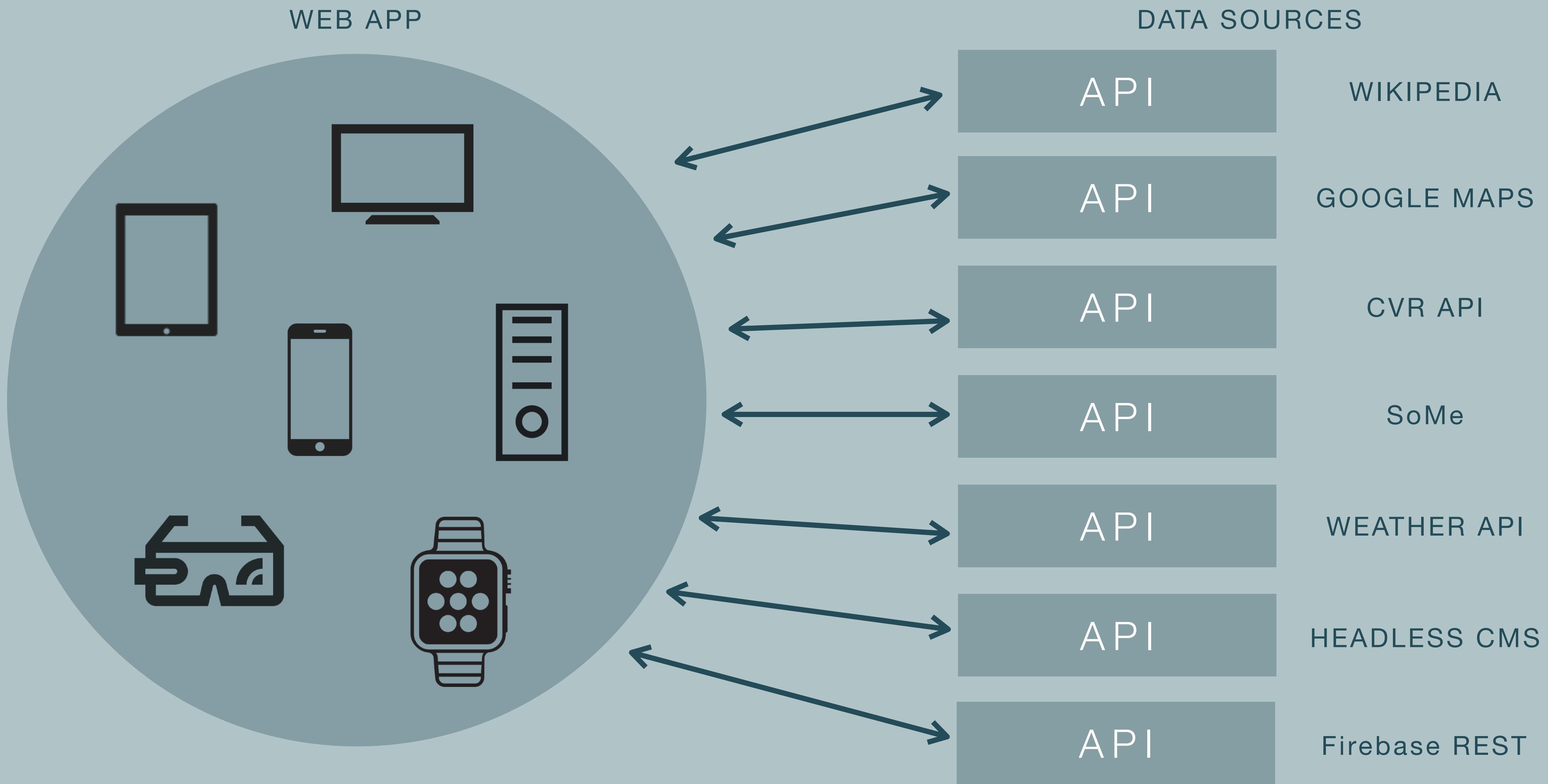
Advantages of REST

- 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

API



API



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

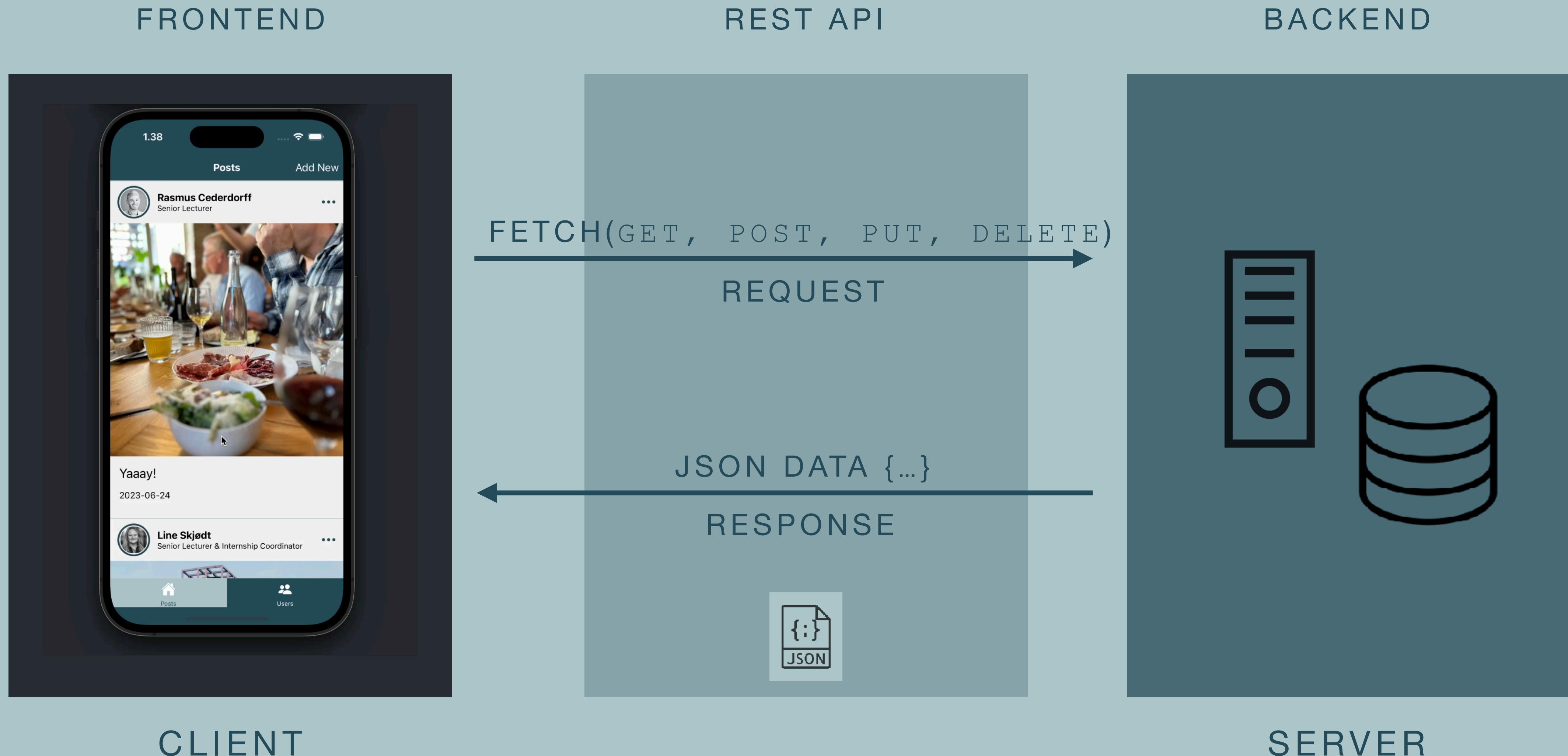
CREATE -> POST: create a new resource (object)

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

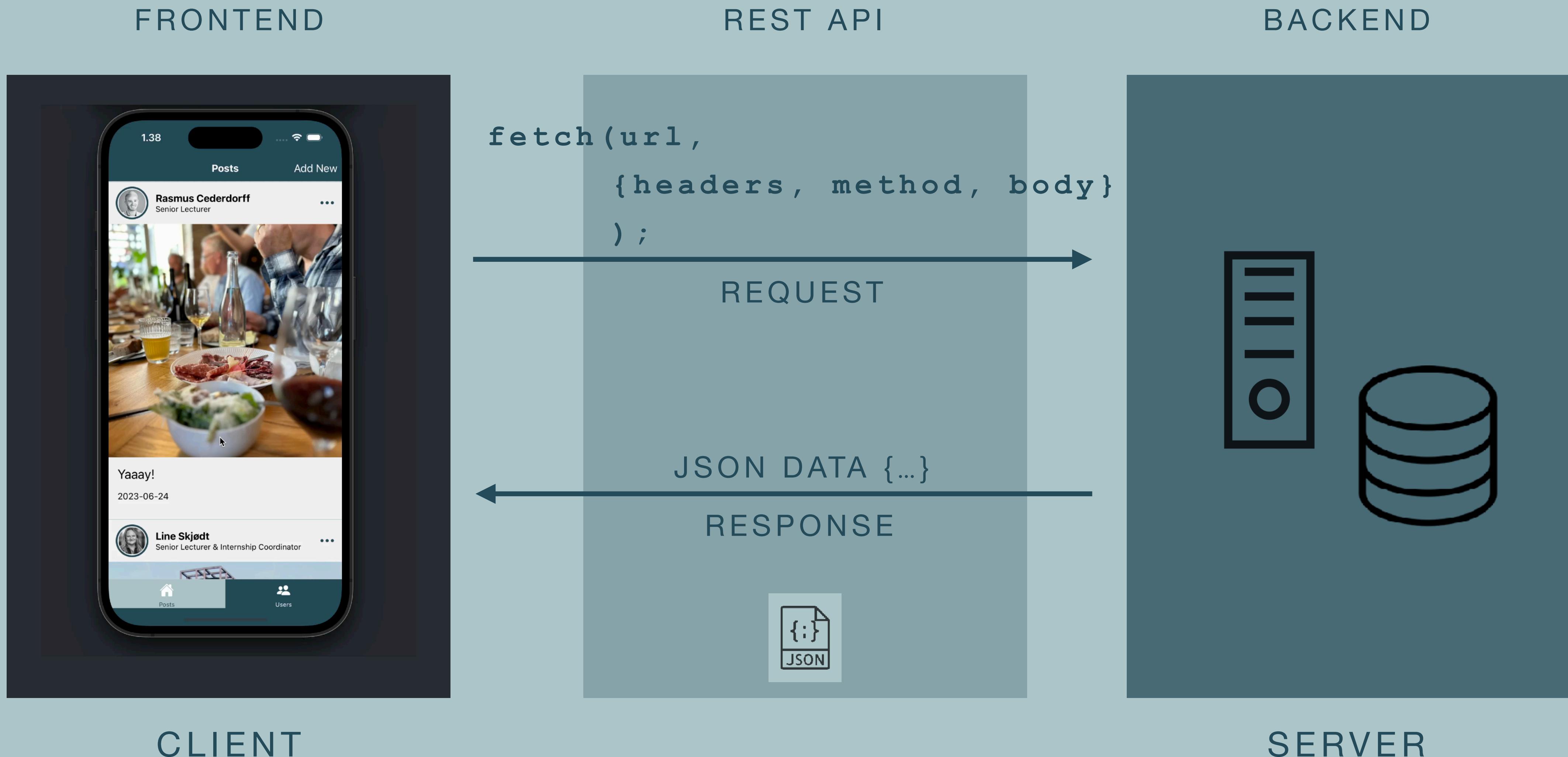


Fetch and read posts

GET is the default request method for fetch.

```
async function getPosts() {  
  const url = "https://race-rest-default-rtbd.firebaseio.com/posts.json";  
  const response = await fetch(url);  
  const data = await response.json();  
  const postsArray = Object.keys(data).map(key => ({ id: key, ...data[key] })); // from object to array  
  return postsArray  
}
```

HTTP Request & Response

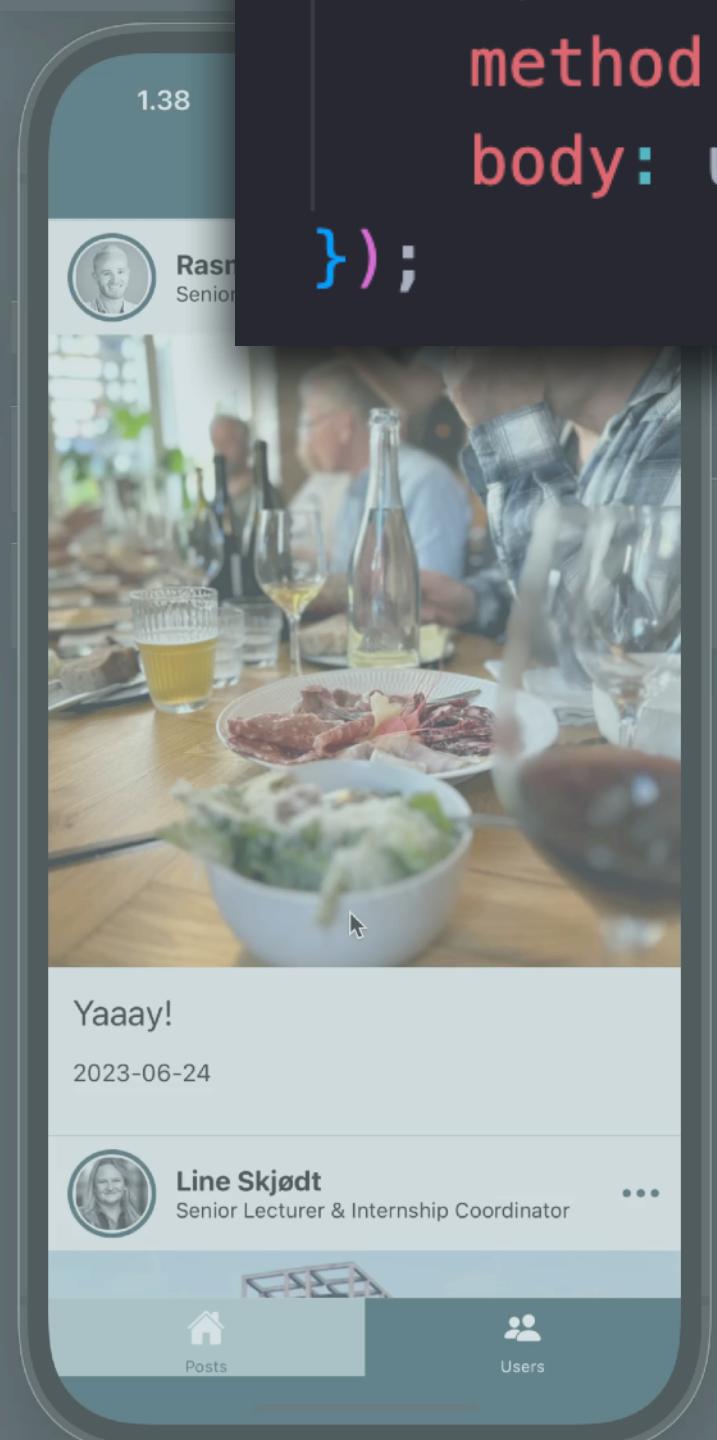


HTTP Requests & Response

```
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

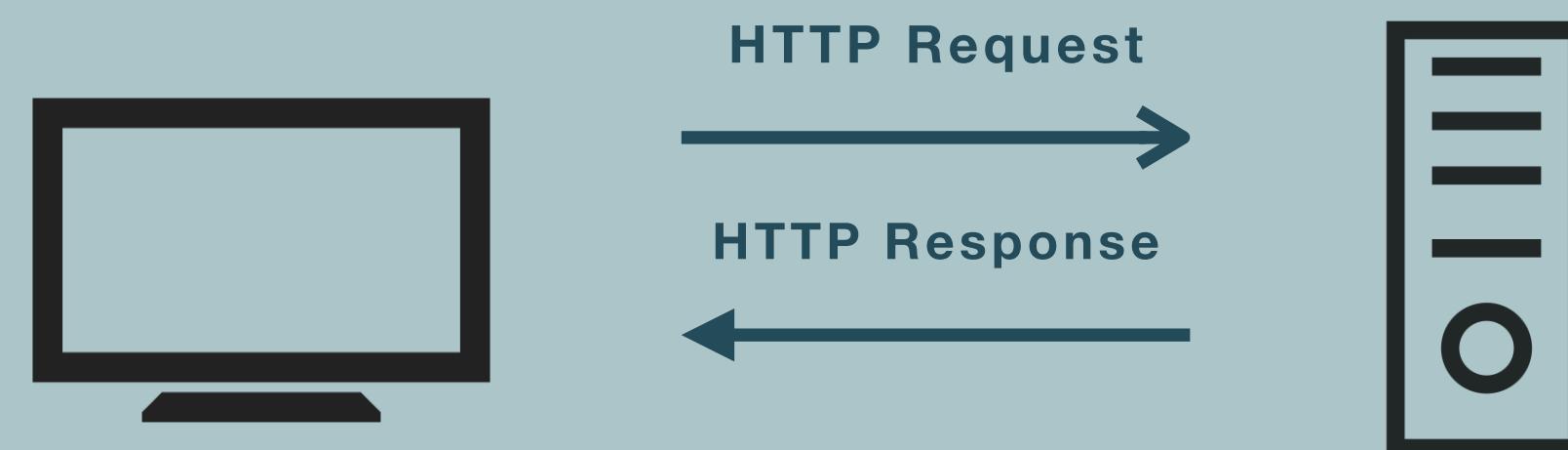
```
const response = await fetch("http://localhost:3333/users", {
  headers: {
    "Content-Type": "application/json"
  },
  method: "POST",
  body: userAsJson
});
```

```
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.

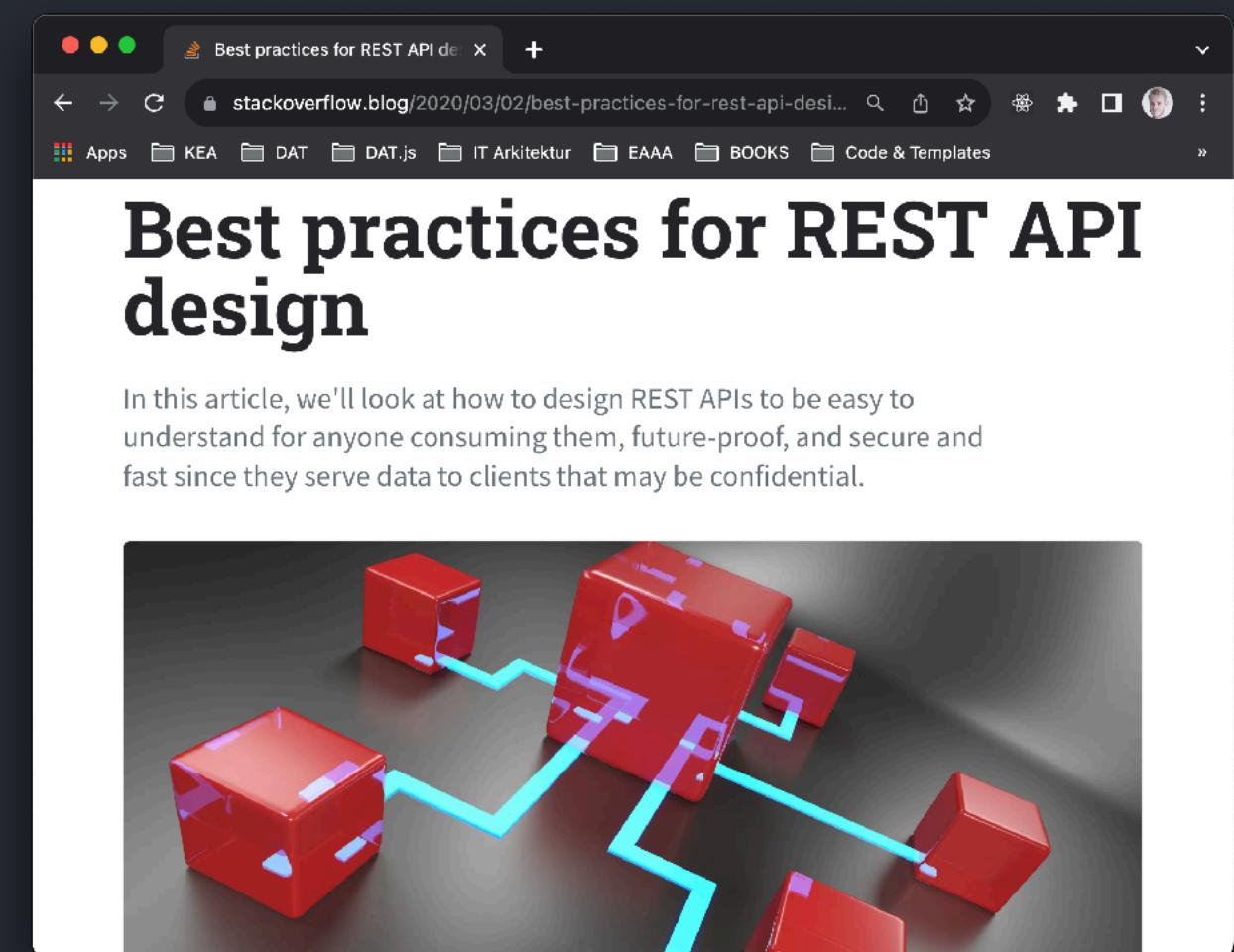
REQUEST body

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

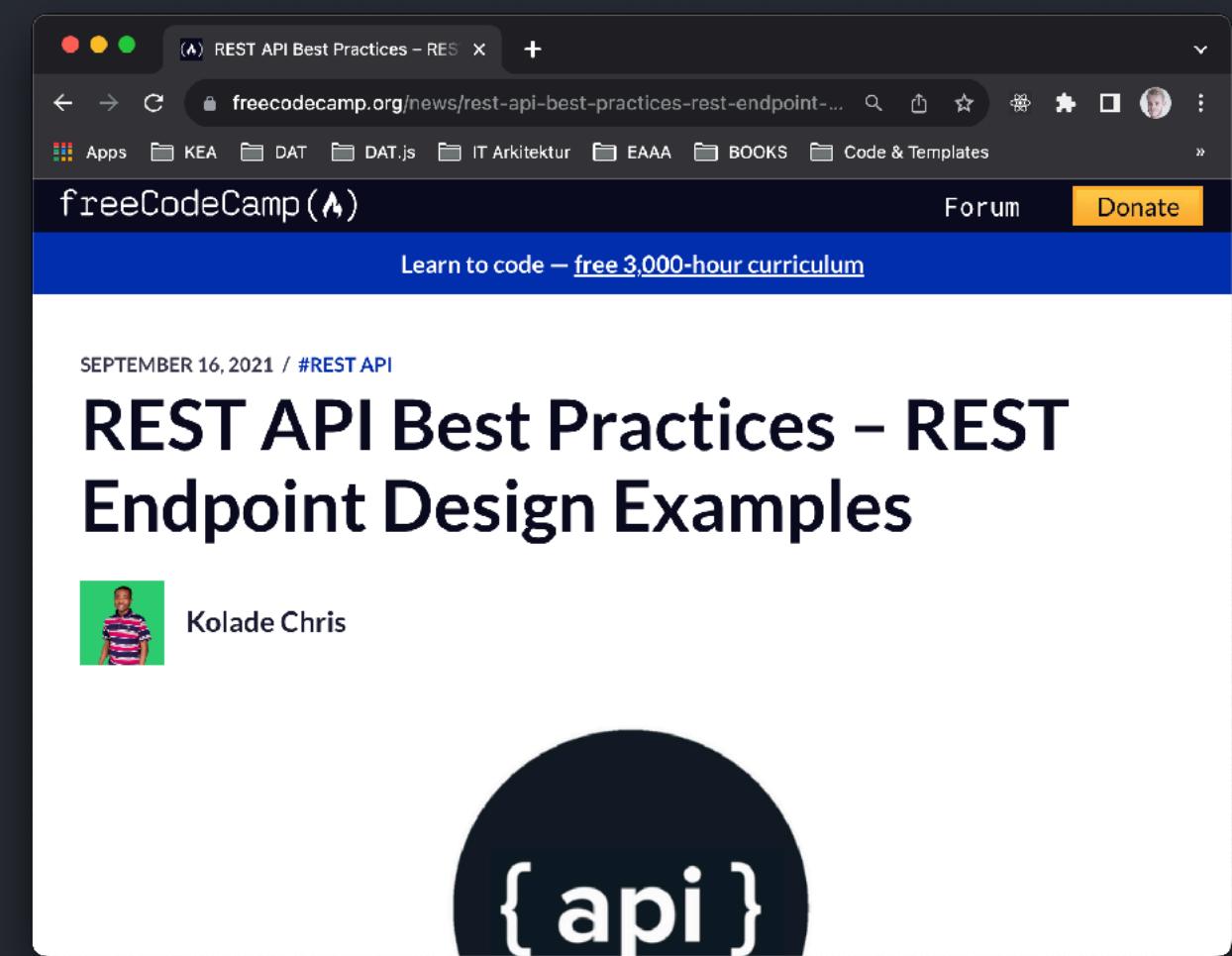
```
async function savePost(postToUpdate) {
  const response = await fetch(url, {
    method: "PUT",
    body: JSON.stringify(postToUpdate)
  });
  const data = await response.json();
  console.log(data);
  navigate("/");
}
```

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



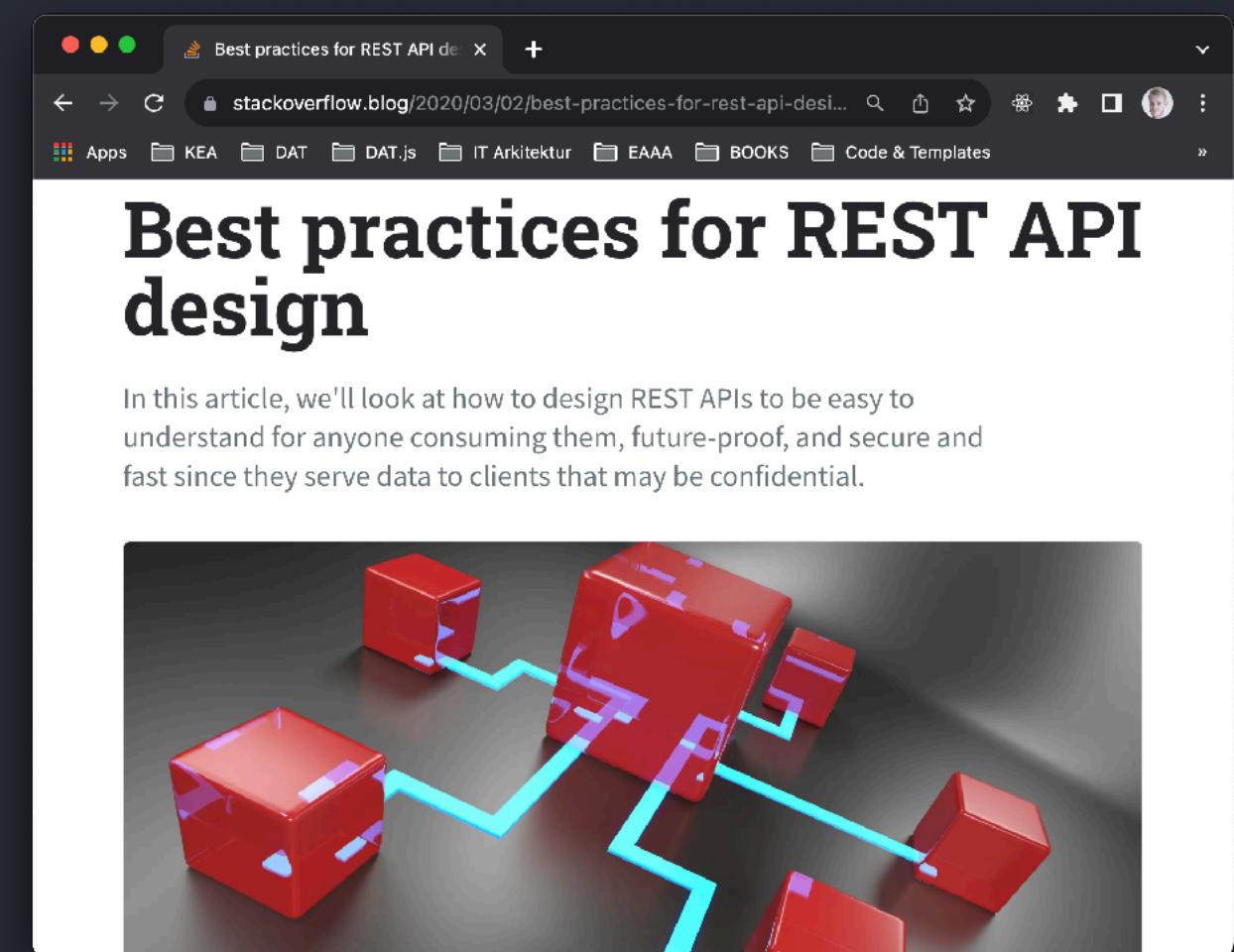
<https://stackoverflow.blog/2020/03/02/best-practices-for-rest-api-design/#h-cache-data-to-improve-performance>



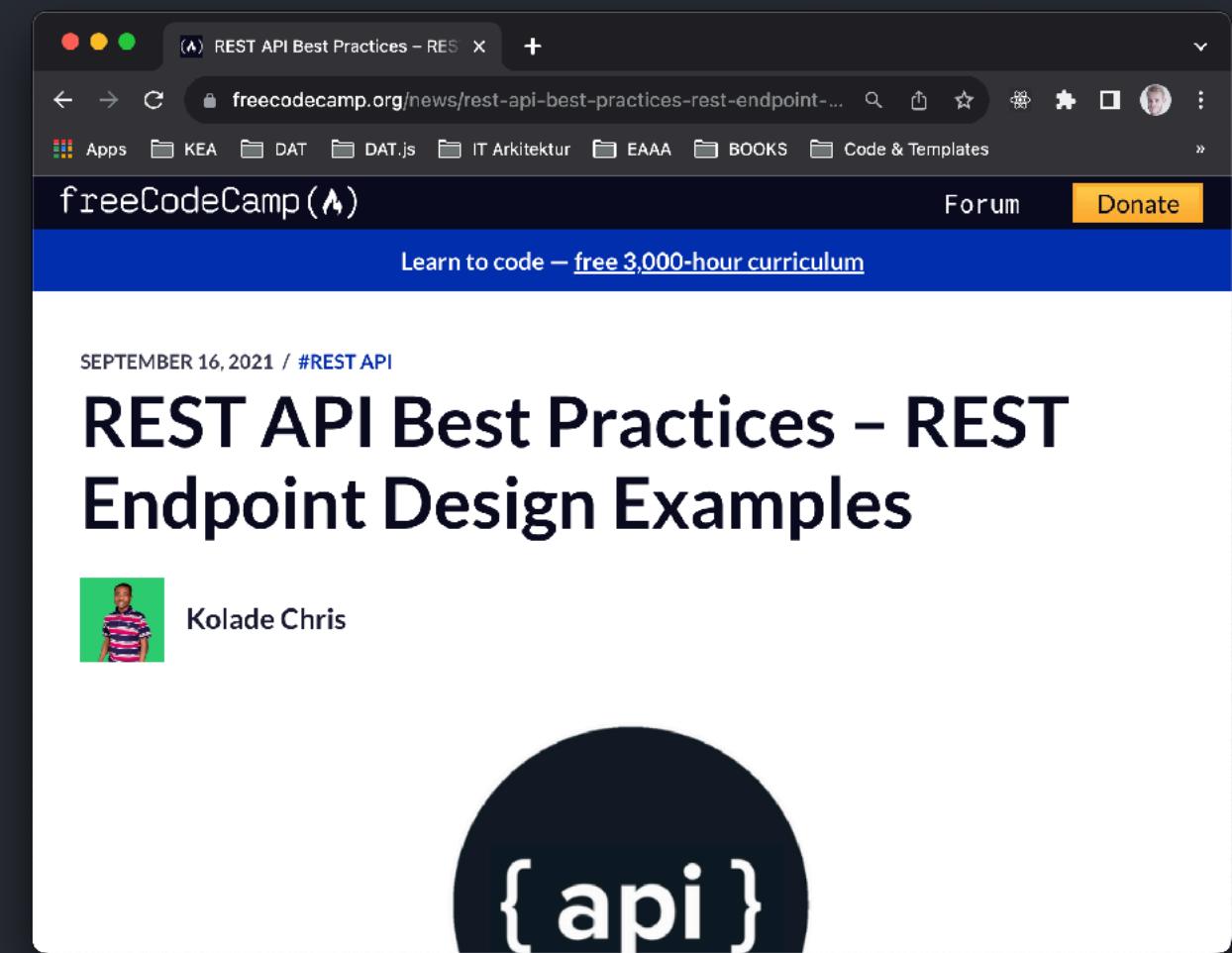
<https://www.freecodecamp.org/news/rest-api-best-practices-rest-endpoint-design-examples/>

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](#)

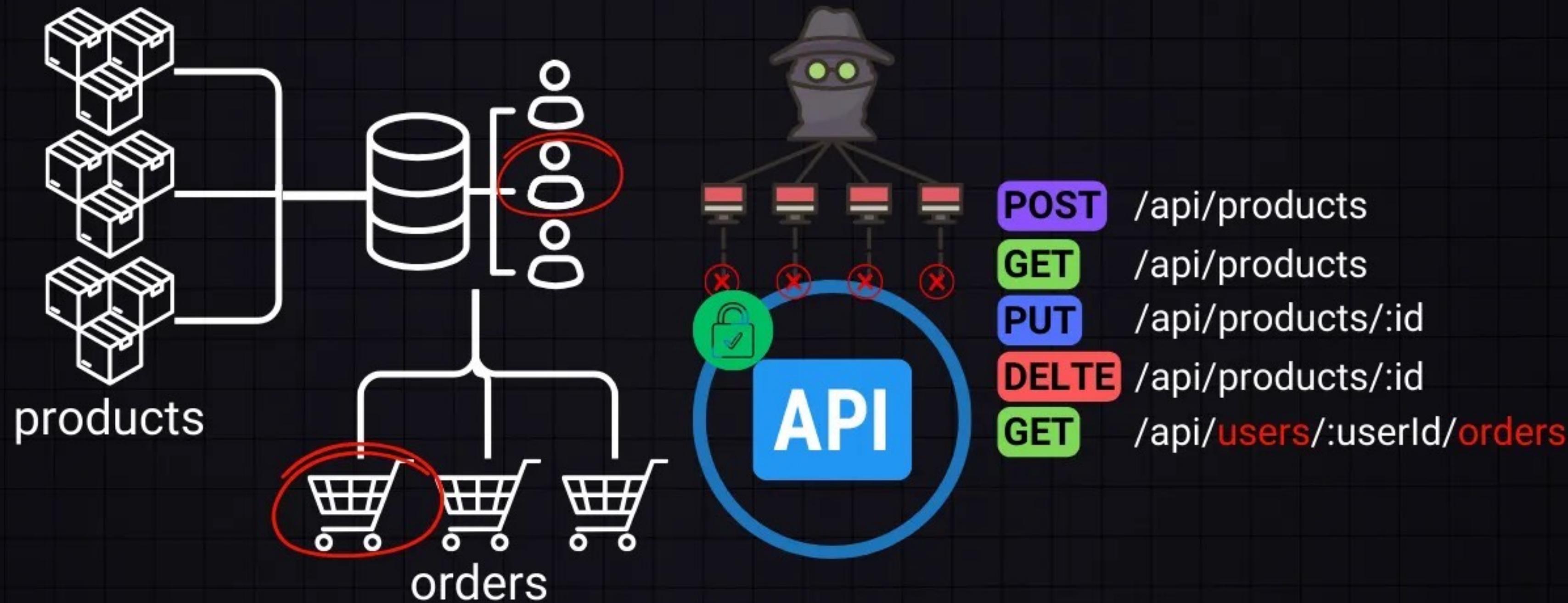


<https://stackoverflow.blog/2020/03/02/best-practices-for-rest-api-design/#h-cache-data-to-improve-performance>



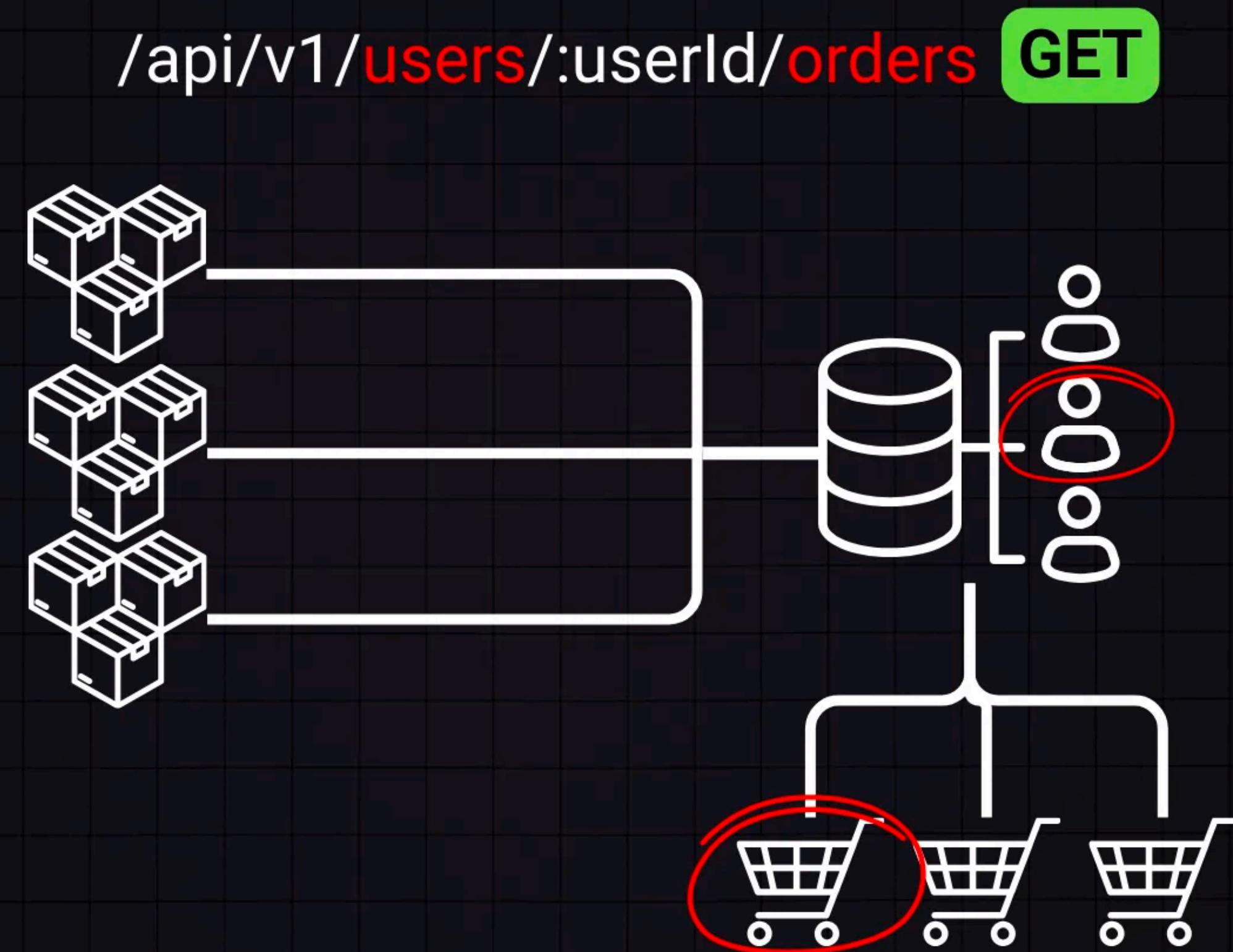
<https://www.freecodecamp.org/news/rest-api-best-practices-rest-endpoint-design-examples/>

API DESIGN



API Design 101: From Basics to Best Practices

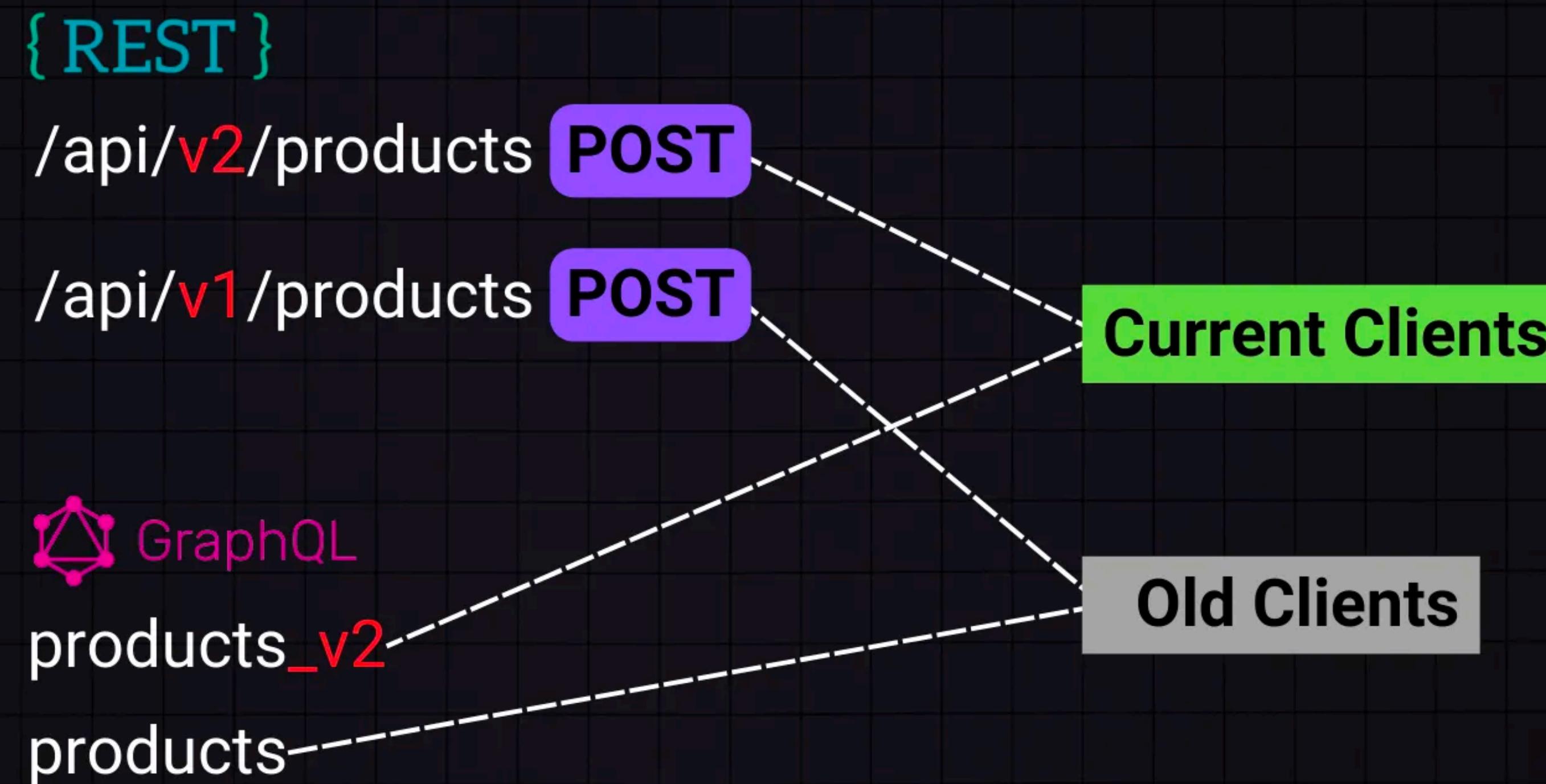
Relationships in API Design



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

Today the slides is your documentation

Read and use
them carefully

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



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"
});
```



What is Firebase?

Platform, a suite of tools & Backend-as-a-Service
for Web & App Development

100 *SECONDS OF*

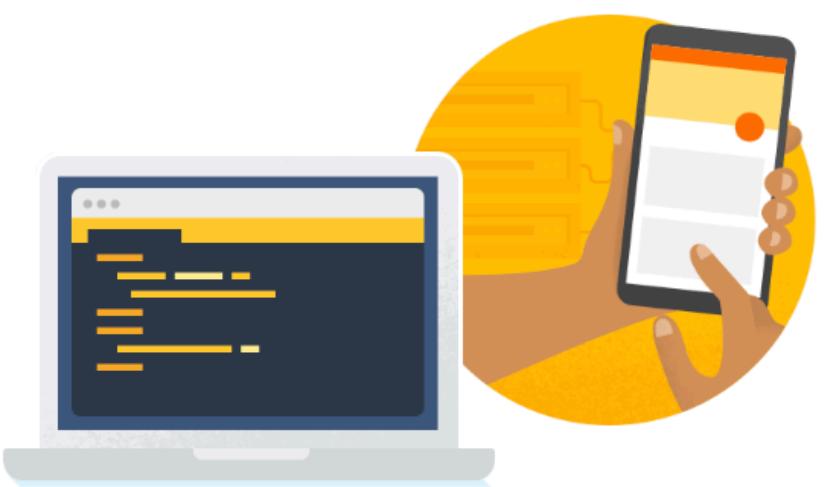


<https://www.youtube.com/watch?v=vAoB4VbhRzM>



Introducing Firebase

<https://www.youtube.com/watch?v=iosNuldQoy8>



Build better apps



Cloud Firestore

Store and sync app data at global scale



ML Kit BETA

Machine learning for mobile developers



Cloud Functions

Run mobile backend code without managing servers



Authentication

Authenticate users simply and securely



Hosting

Deliver web app assets with speed and security



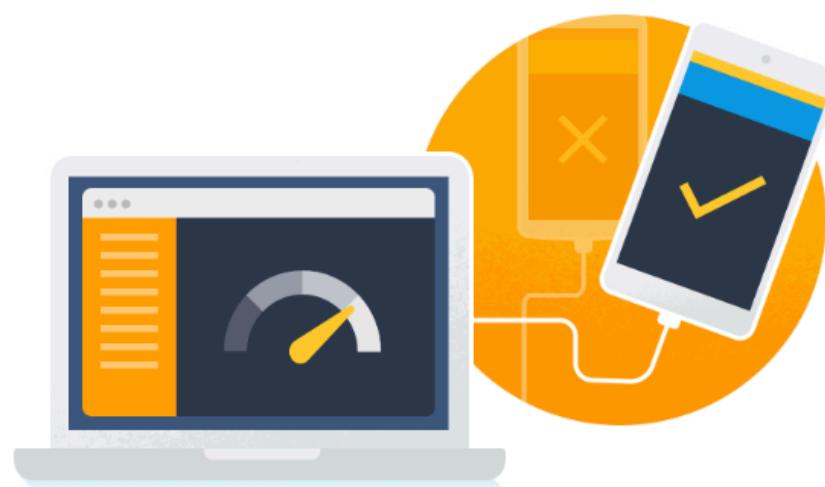
Cloud Storage

Store and serve files at Google scale



Realtime Database

Store and sync app data in milliseconds



Improve app quality



Crashlytics

Prioritize and fix issues with powerful, realtime crash reporting



Performance Monitoring

Gain insight into your app's performance



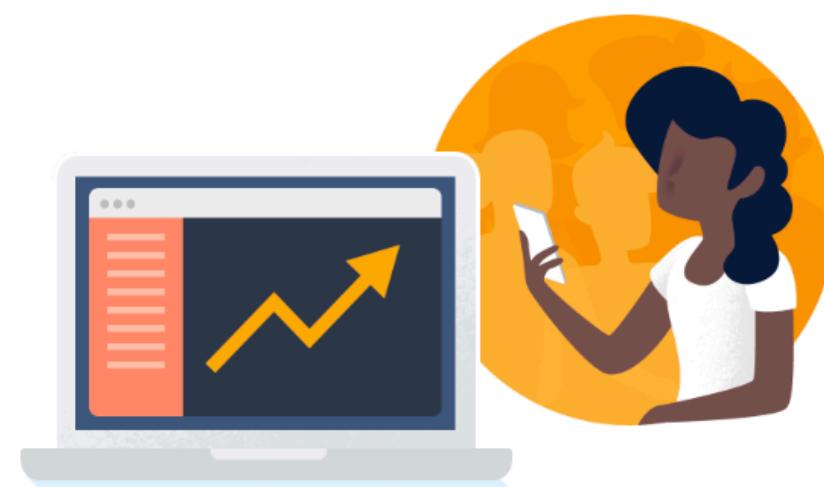
Test Lab

Test your app on devices hosted by Google



App Distribution BETA

Distribute pre-release versions of your app to your trusted testers



Grow your business



In-App Messaging BETA

Engage active app users with contextual messages



Google Analytics

Get free and unlimited app analytics



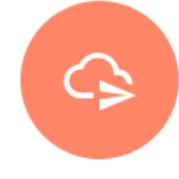
Predictions

Smart user segmentation based on predicted behavior



A/B Testing BETA

Optimize your app experience through experimentation



Cloud Messaging

Send targeted messages and notifications



Remote Config

Modify your app without deploying a new version



Dynamic Links

Drive growth by using deep links with attribution



Realtime Database & REST API

Store and sync data in real time
REST API or SDK

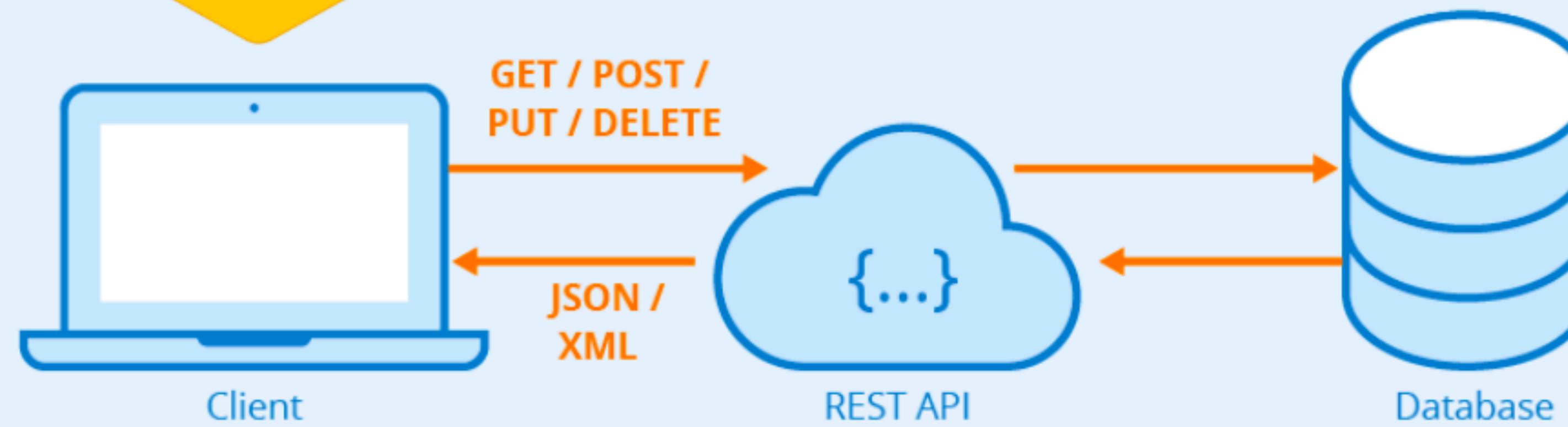
<https://firebase.google.com/products/realtime-database>



Database



Firebase



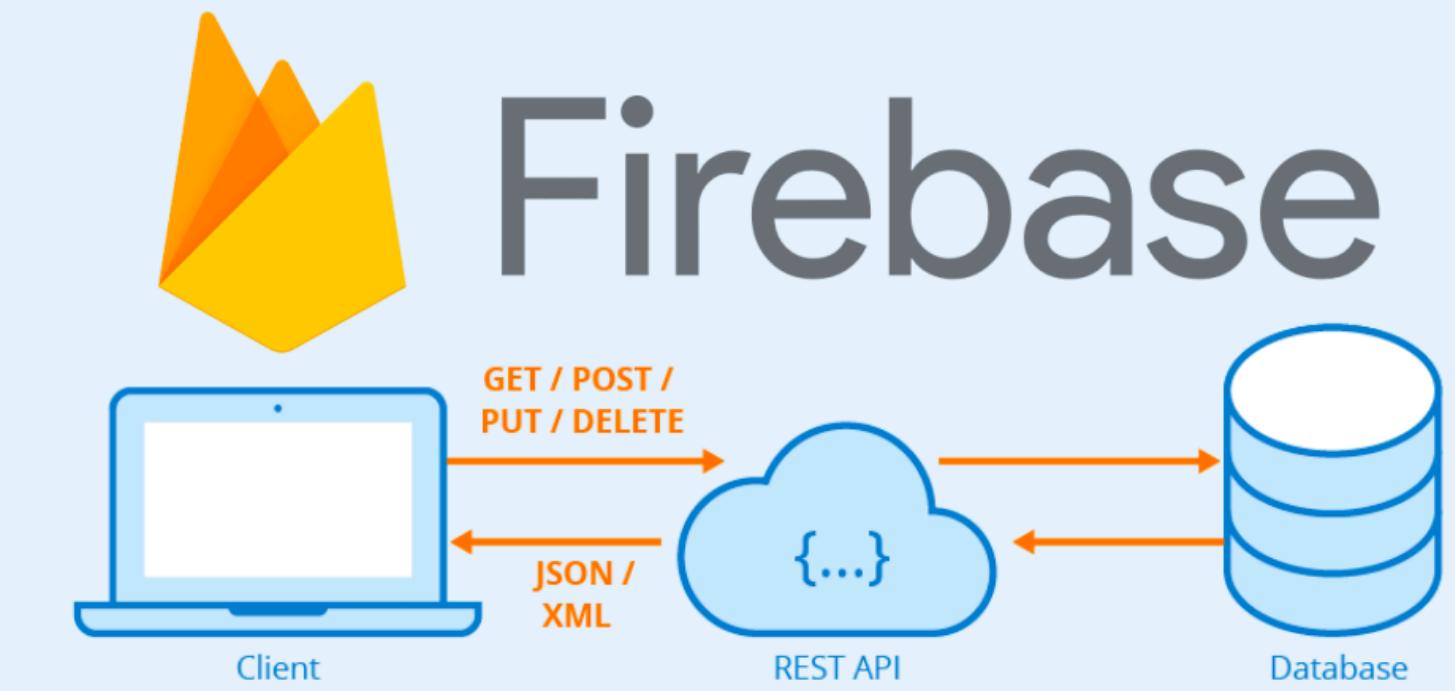
“

The Firebase Realtime Database is a cloud-hosted database. Data is stored as JSON and synchronized in realtime to every connected client.

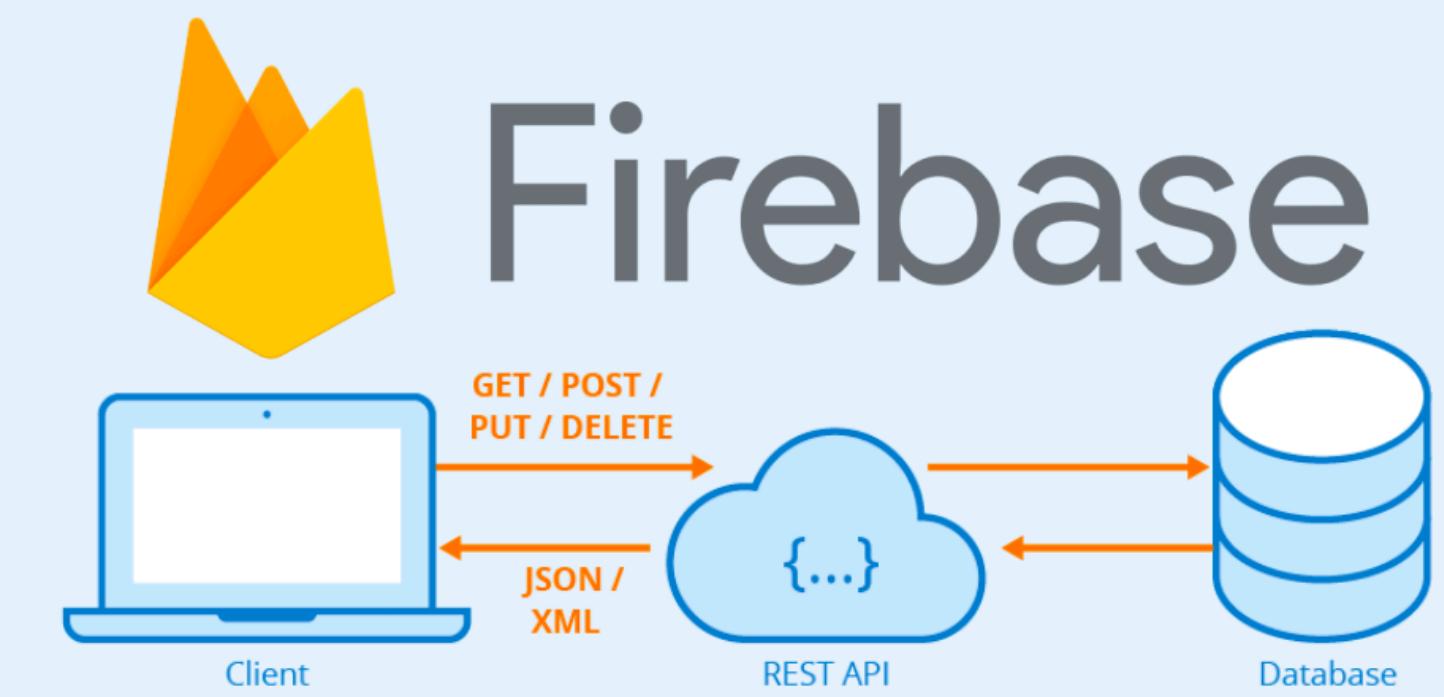
We can use any Firebase Realtime Database URL as a REST endpoint. All we need to do is append .json to the end of the URL and send a request from our favorite HTTPS client.

”

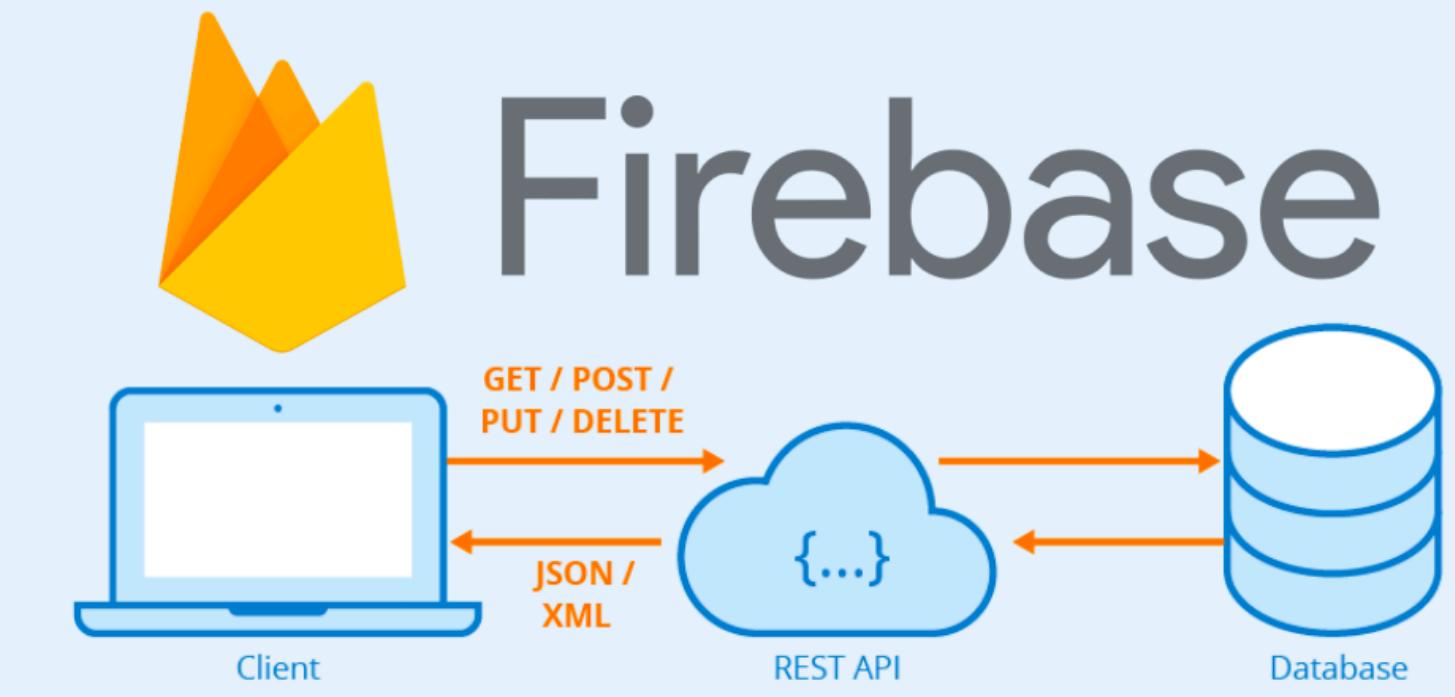
<https://firebase.google.com/docs/database/rest/start>



By following these REST principles, the Firebase Realtime Database REST API provides a simple and consistent way to work with the database. 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.



In the context of Firebase Realtime Database REST API, you can perform CRUD operations using HTTP methods like POST, GET, PUT, PATCH, and DELETE to create, read, update, and delete data in the Firebase Realtime Database. This allows you to interact with the database and manipulate data effectively based on your application's needs.





**my firebase doesn't
work. Can you help me?**

**read the
documentation**

But...

**read the
documentation**

How data is structured

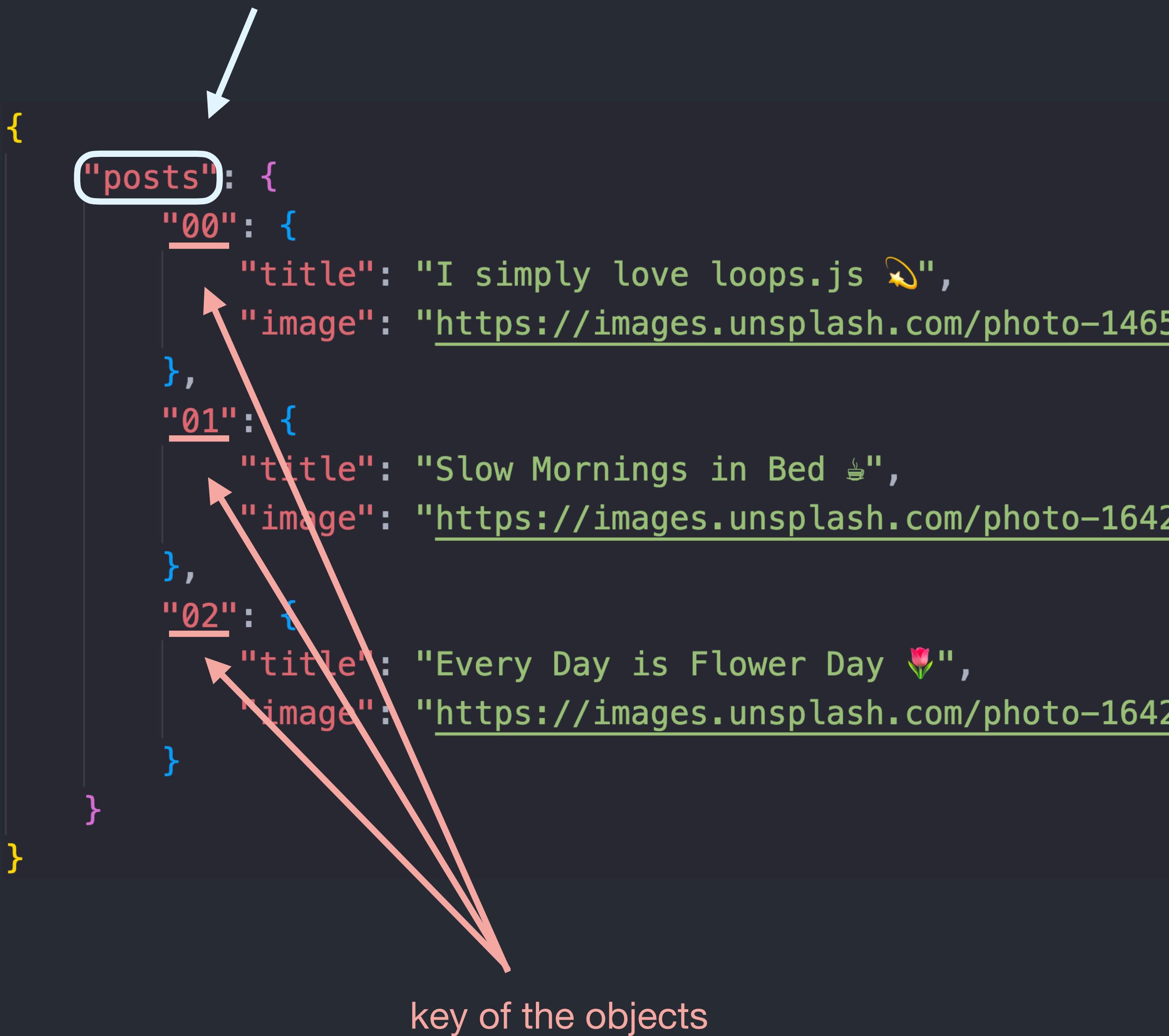
```
{  
  "posts": {  
    "00": {  
      "title": "I simply love loops.js 🌟",  
      "image": "https://images.unsplash.com/photo-1465...  
    },  
    "01": {  
      "title": "Slow Mornings in Bed ☕",  
      "image": "https://images.unsplash.com/photo-1642...  
    },  
    "02": {  
      "title": "Every Day is Flower Day 🌸",  
      "image": "https://images.unsplash.com/photo-1642...  
    }  
  }  
}
```

- One big object with objects.
- All objects have a unique key
- “posts”, “00”, “01” and “02” are all keys.
- “posts” is the key of the collection of post objects (list of posts).
- “00”, “01” and “02” are keys of a post object.
- “title” and “image” are keys inside of every post object. “title” and “image” contains values.
- Remember a property contains a key and a value.

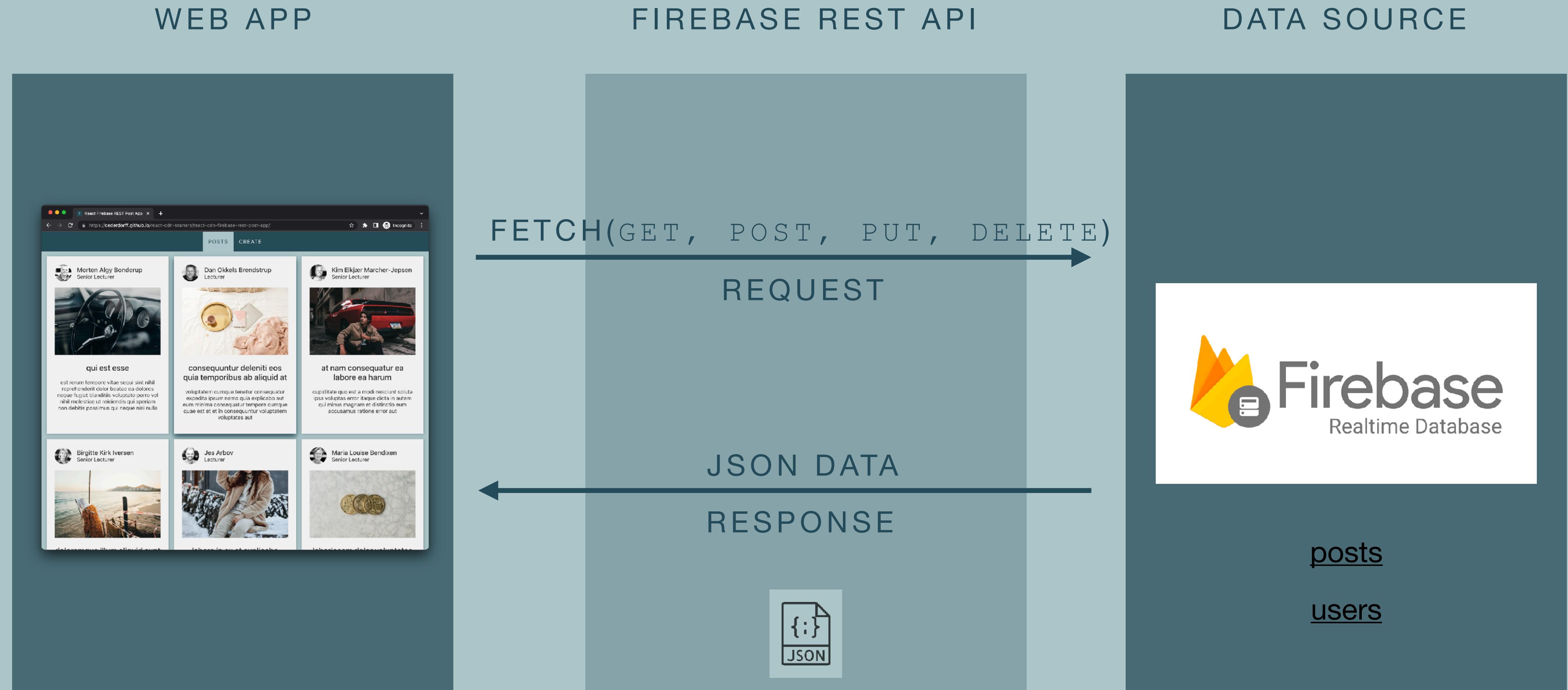
```
{  
  "posts": {  
    "00": {  
      "title": "I simply love loops.js 🌟",  
      "image": "https://images.unsplash.com/photo-1465...  
    },  
    "01": {  
      "title": "Slow Mornings in Bed ☕",  
      "image": "https://images.unsplash.com/photo-1642...  
    },  
    "02": {  
      "title": "Every Day is Flower Day 🌸",  
      "image": "https://images.unsplash.com/photo-1642...  
    }  
  }  
}
```

- Just think of an array of objects (posts array with post objects).
- We are just using an object to hold all the objects instead of an array.
- It's kind of a dictionary. The posts object contains unique prop names (keys) which easily can be "looked up".
- Then every post object (with title and image props) can be returned:
 - `["posts"]["02"]`
 - `["posts"]["00"]`
 - `["posts"]["01"]`
 - `["posts"]["01"]["title"]`

key and name of the collections



Fetch, HTTP Request & Response



Realtime Database REST API

[https://firebase.google.com/docs/
database/rest/start](https://firebase.google.com/docs/database/rest/start)

```
export default function PostsPage() {
  const [posts, setPosts] = useState([]);
  const [showLoader, dismissLoader] = useIonLoading();

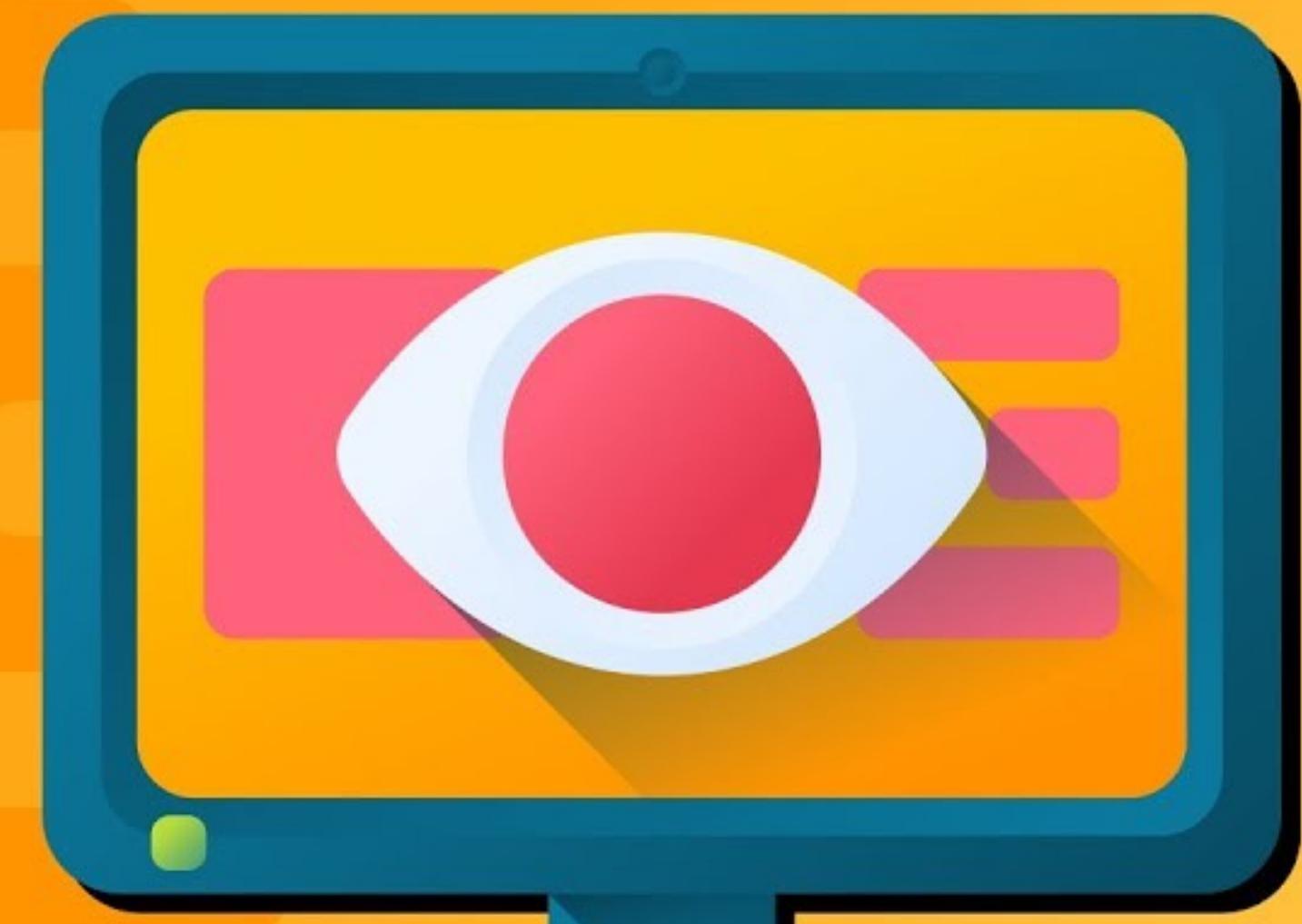
  async function getPosts() {
    const response = await fetch("https://race-rest-default-rtbd.firebaseio.com/posts.json");
    const data = await response.json();
    // map object into an array with objects
    const postsArray = Object.keys(data).map(key => ({ id: key, ...data[key] }));
    return postsArray;
  }

  async function getUsers() {
    const response = await fetch("https://race-rest-default-rtbd.firebaseio.com/users.json");
    const data = await response.json();
    // map object into an array with objects
    const users = Object.keys(data).map(key => ({ id: key, ...data[key] }));
    return users;
  }
}
```



Firebase Security Rules

100 *SECONDS OF*





Firebase Security Rules



Doug Stevenson presents:

Firebase security rules

Introduction to
Firebase security rules

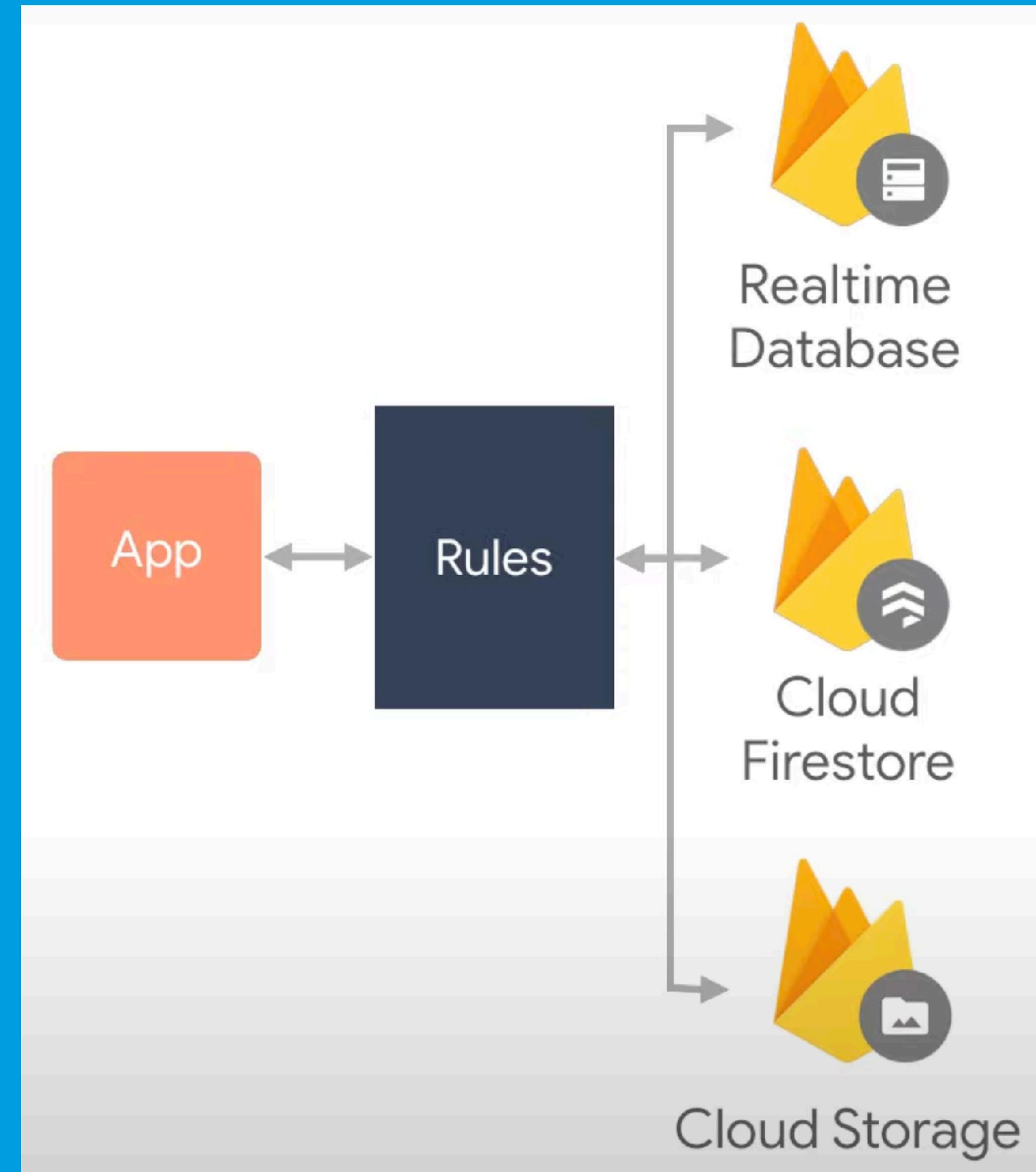


Firecasts

For Firebase Developers



Firebase Security Rules





Code
Every
Day