

# API & REST

Client Server, HTTP, Request & Response,  
JSON, CRUD & REST API

Client Server

JSON

Data Exchange

Communication

REST API

API

HTTP Request & Response

HTTP Verbs

CRUD

PaaS

SaaS

Cloud Services

# Dataforsyningen - REST API

One of many REST APIs



<https://dataforsyningen.dk/>

The screenshot shows the REST API documentation page. The title is "Dataforsyningen REST API". It provides links to the API ([API](https://api.dataforsyningen.dk/)), documentation ([DOCS](https://dataforsyningen.dk/)), and a data overview ([Dataoversigt](https://dataforsyningen.dk/data)). Below this, examples of calls are shown: "Eksempel på kald" (Call example) for "Danmarks adresser og vejnavne" (<https://dataforsyningen.dk/data/4729>) and "Docs (REST Services)" (<https://dawadocs.dataforsyningen.dk/dok/api>). The "Regioner" section lists "alle regioner" (all regions) with a GET call to <https://api.dataforsyningen.dk/regioner> and "en region" (one region) with a GET call to <https://api.dataforsyningen.dk/regioner/1082>.

[Dataforsyningen REST API](#)

The screenshot shows a mobile browser window titled "City Info App" with the URL "127.0.0.1:5500/index.html". The browser interface includes standard controls like back/forward, refresh, and zoom (Dimensions: 414 x 736, 75% zoom). The developer tools are open, specifically the "Console" tab, which displays the following JSON object:

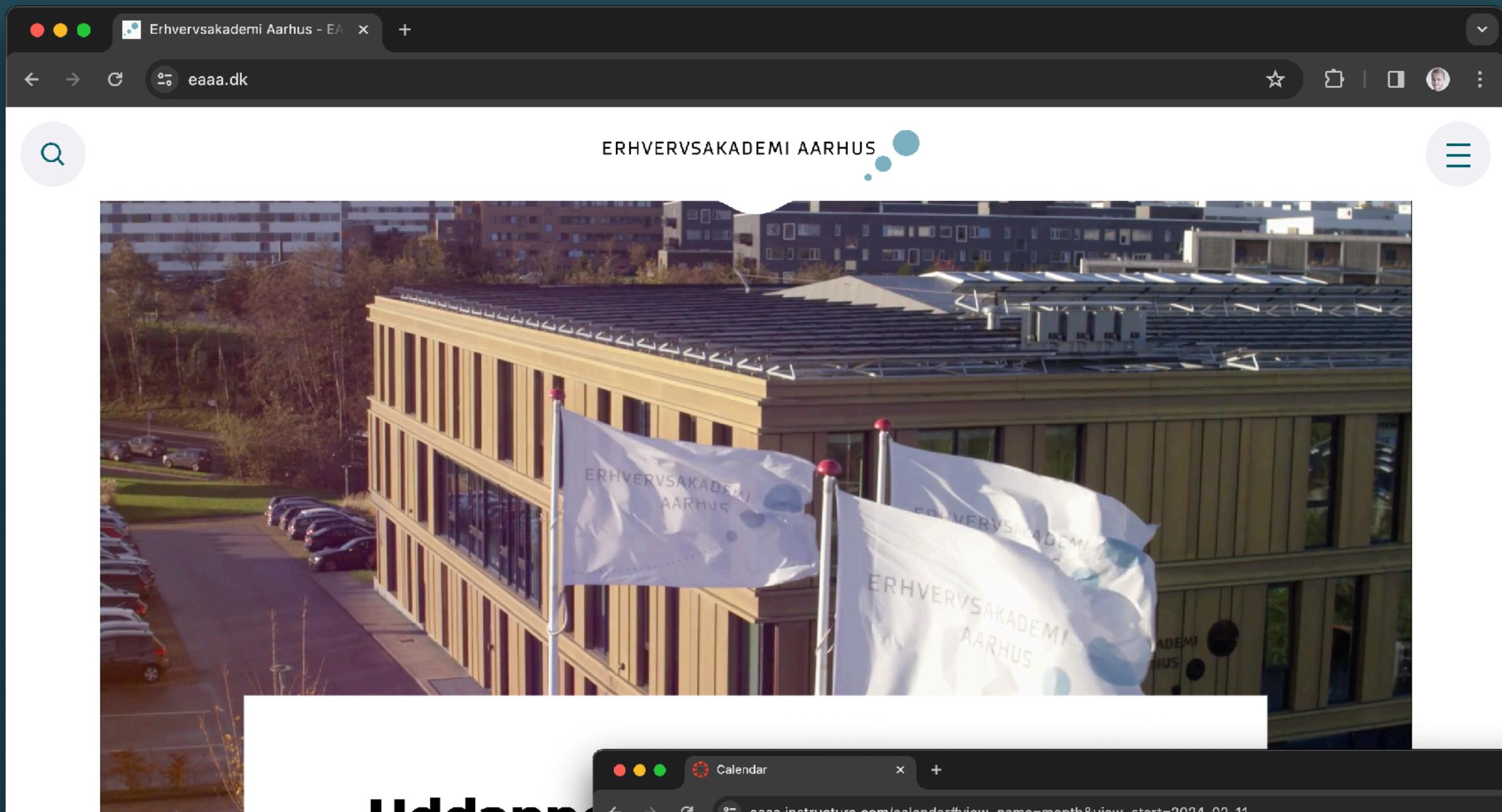
```
app.js:10
{
  href: 'https://api.dataforsyningen.dk/postnumre/8000',
  nr: '8000',
  navn: 'Aarhus C',
  stormodtageradresser: null,
  bbox: Array(4),
  ...
}

bbox: [10.17165345, 56.11626813, 10.33170603, 56.15051496]
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
```

<https://cederdorff.com/city-zip-code/>

# How do systems communicate?

1. How can data get from here to there?
2. How can the same data be used in several contexts?
3. How do we integrate data in our app?
4. How do two systems communicate?



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.

# 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

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

# Communication between two systems

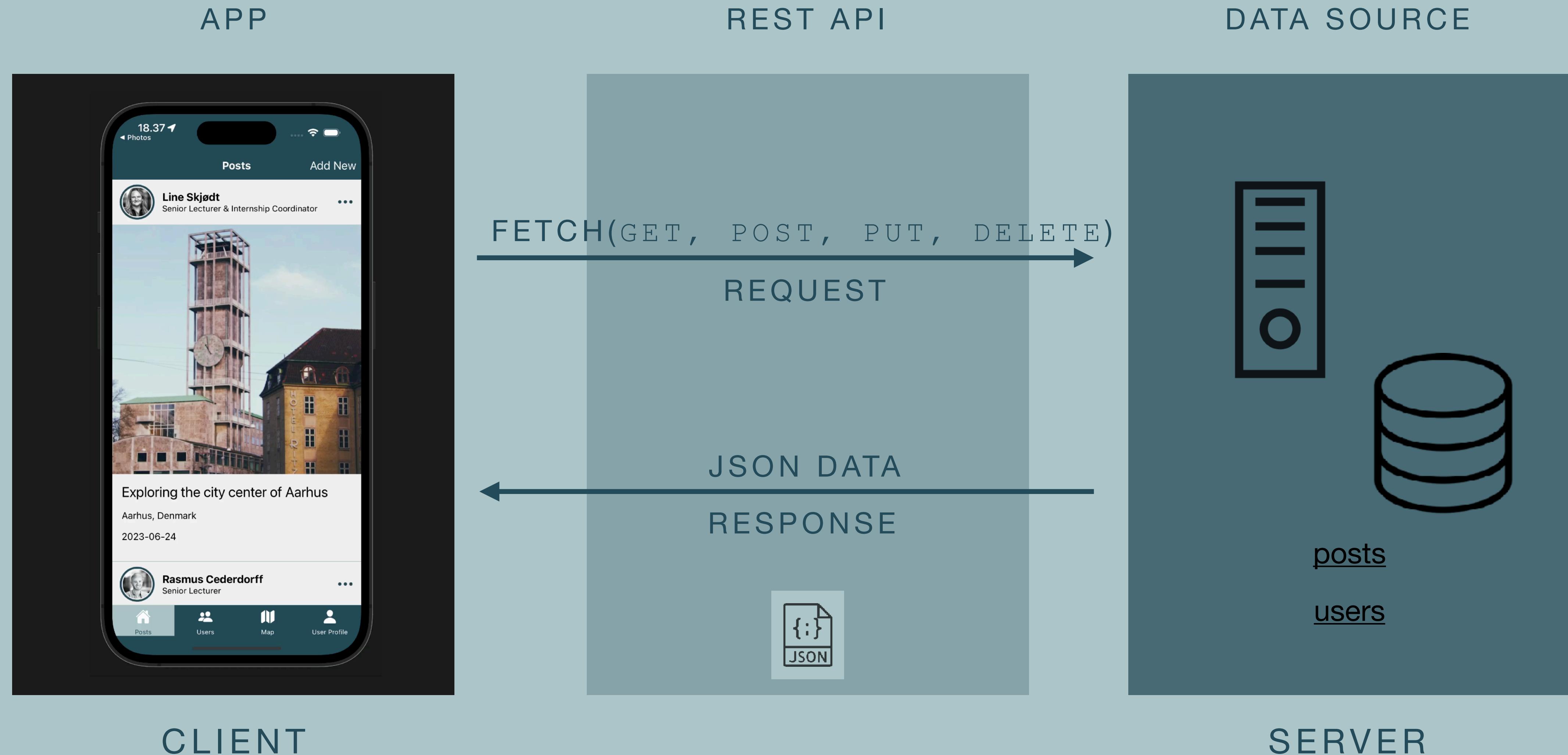
Find some examples



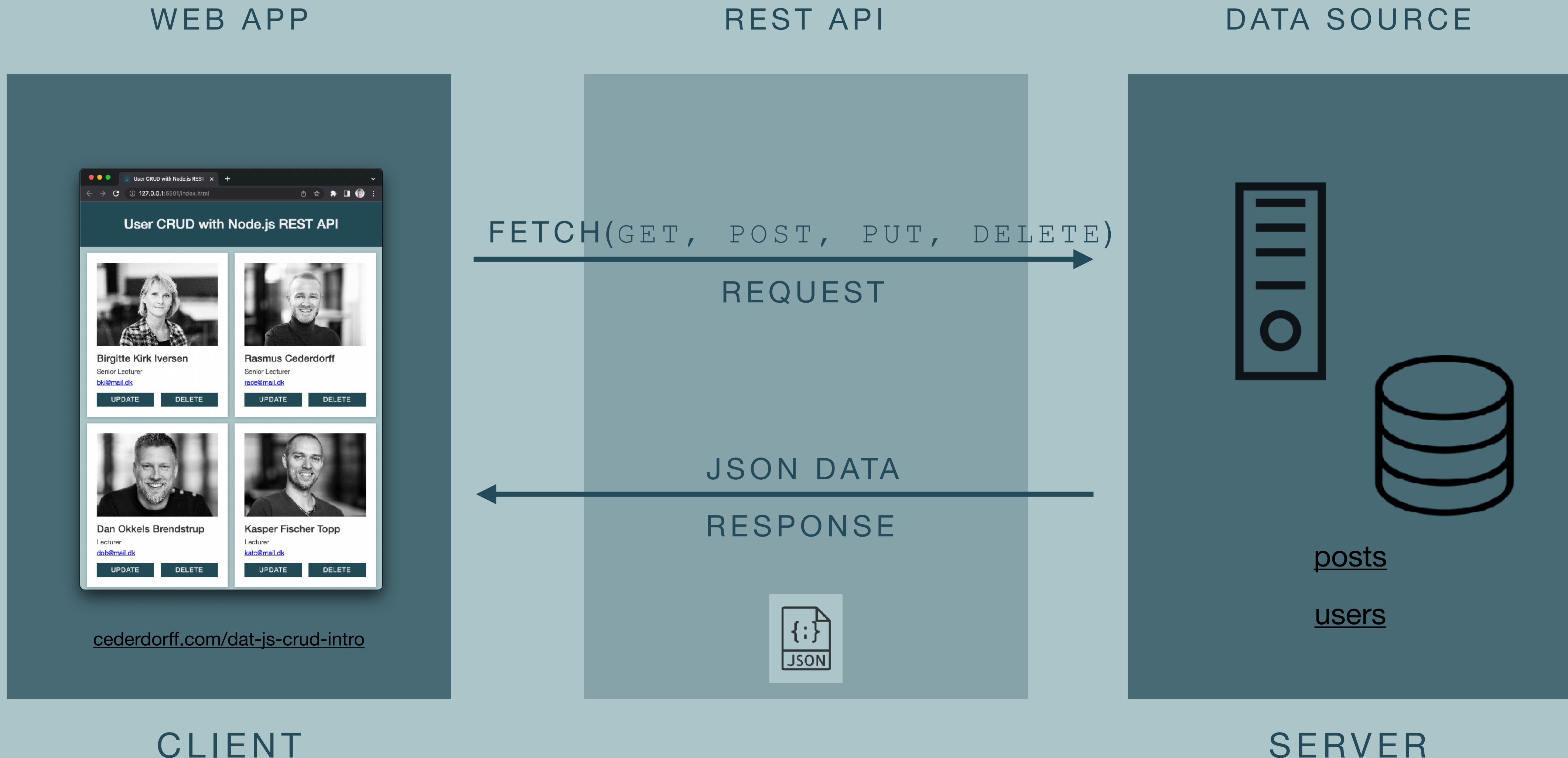
# Client Server

The Basic Architecture of the Web

# Client Server Architecture



# Web Development



# Webudvikling

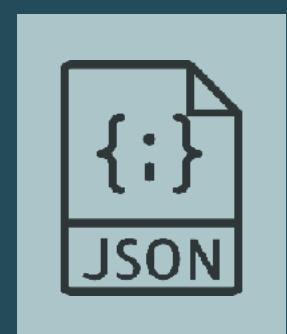
FRONTEND



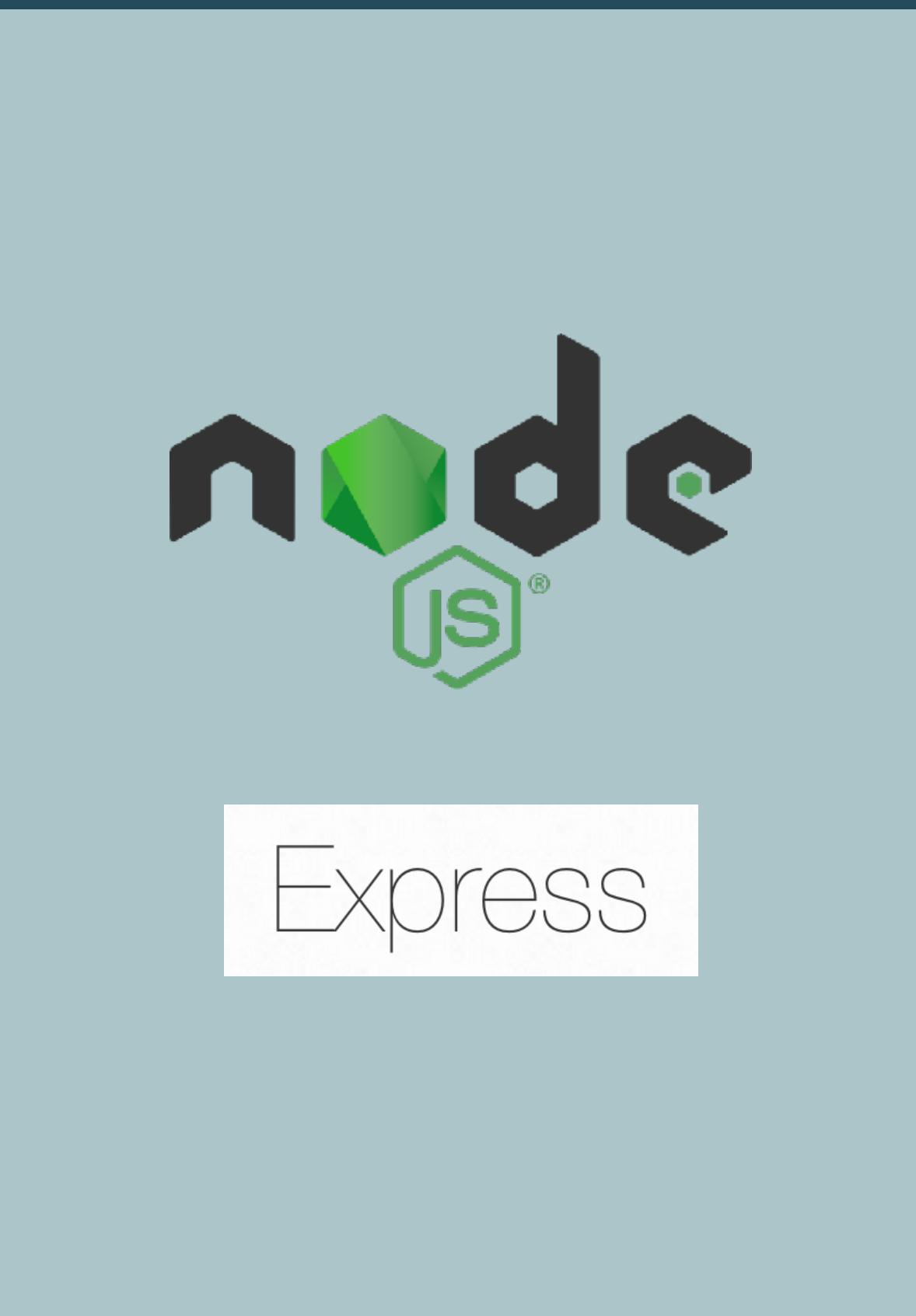
REST API

FETCH(GET, POST, PUT, DELETE)  
REQUEST

JSON DATA  
RESPONSE

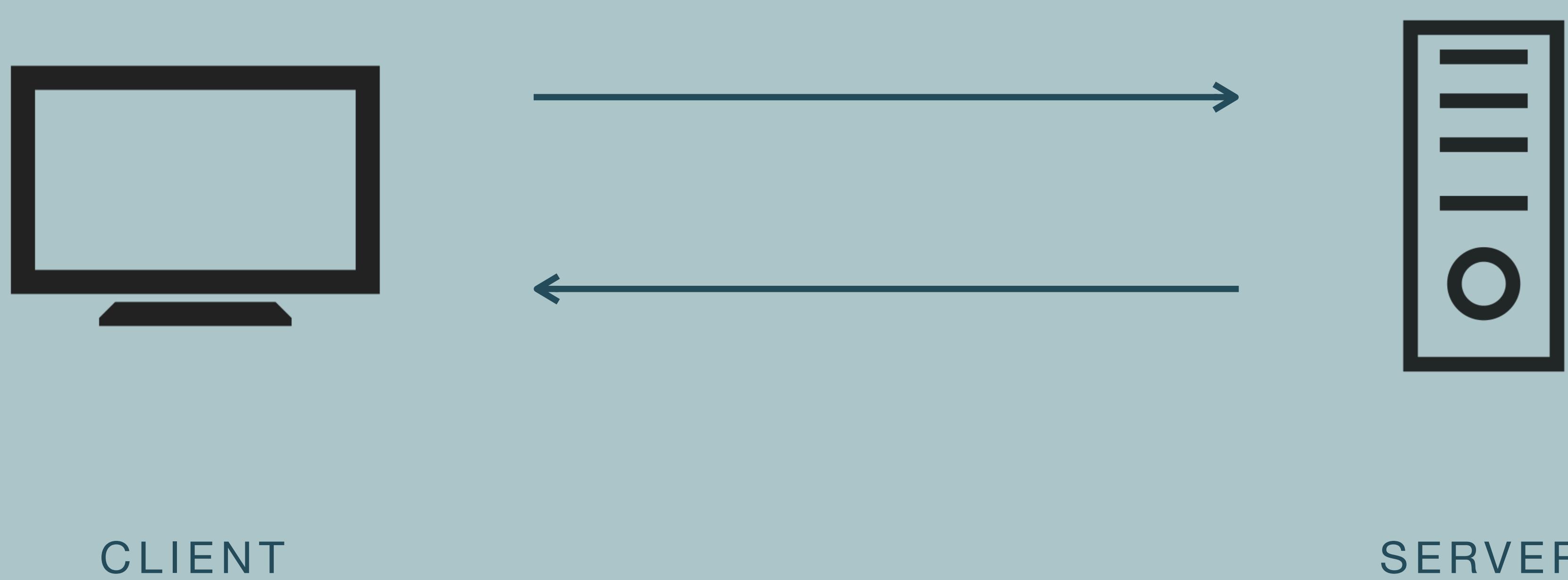


BACKEND



# Client-Server Model

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



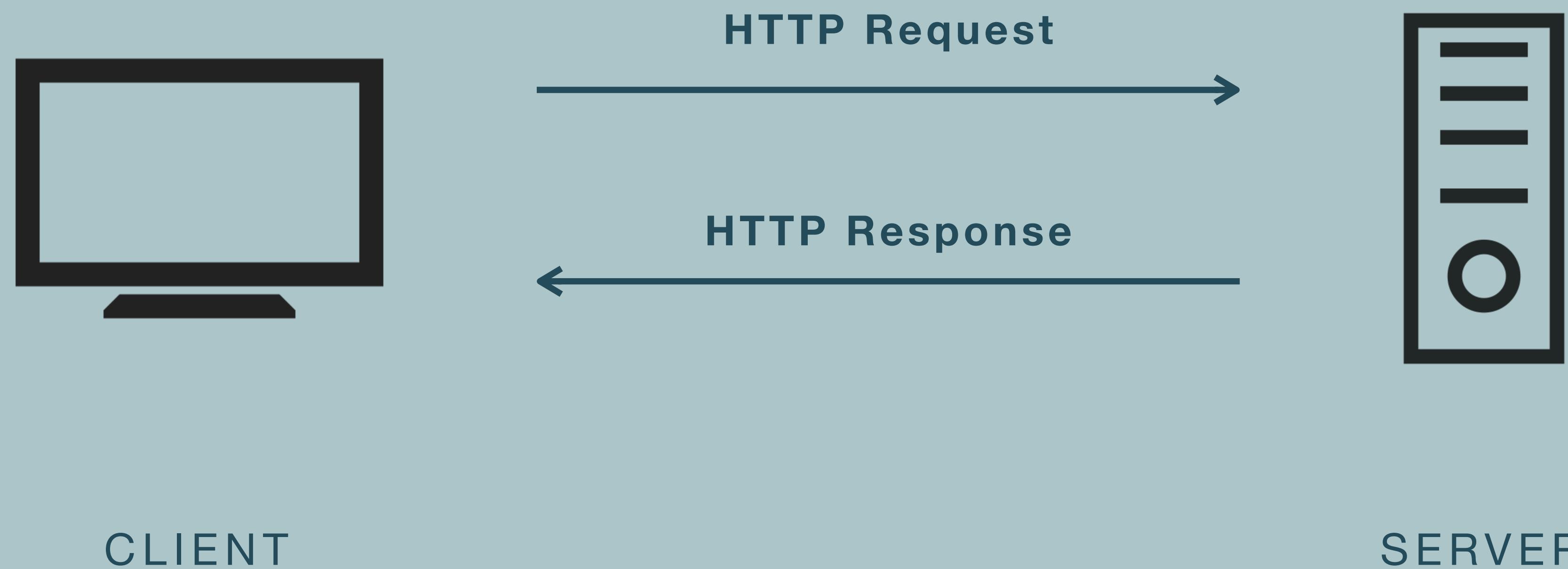
# Client-Server Model

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



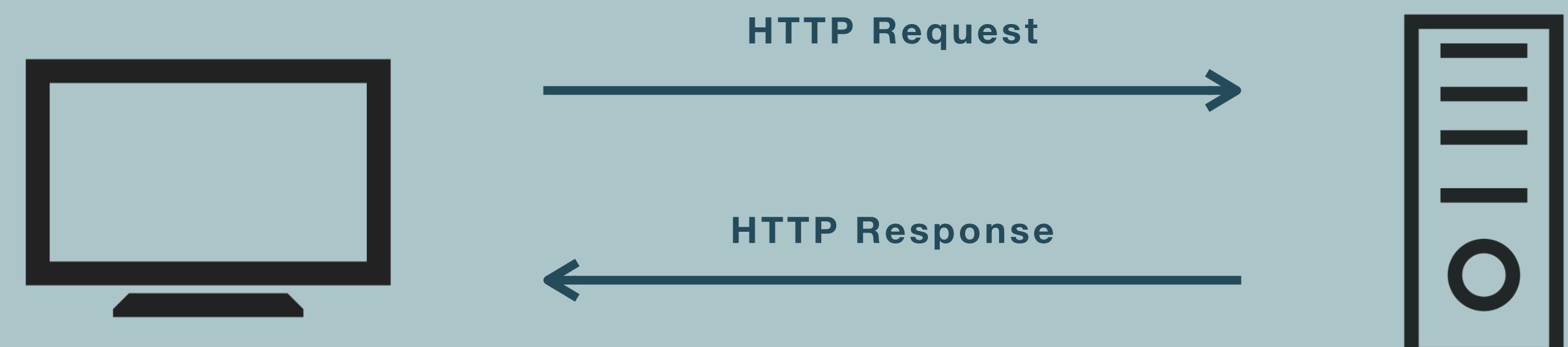
# Client-Server Model

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

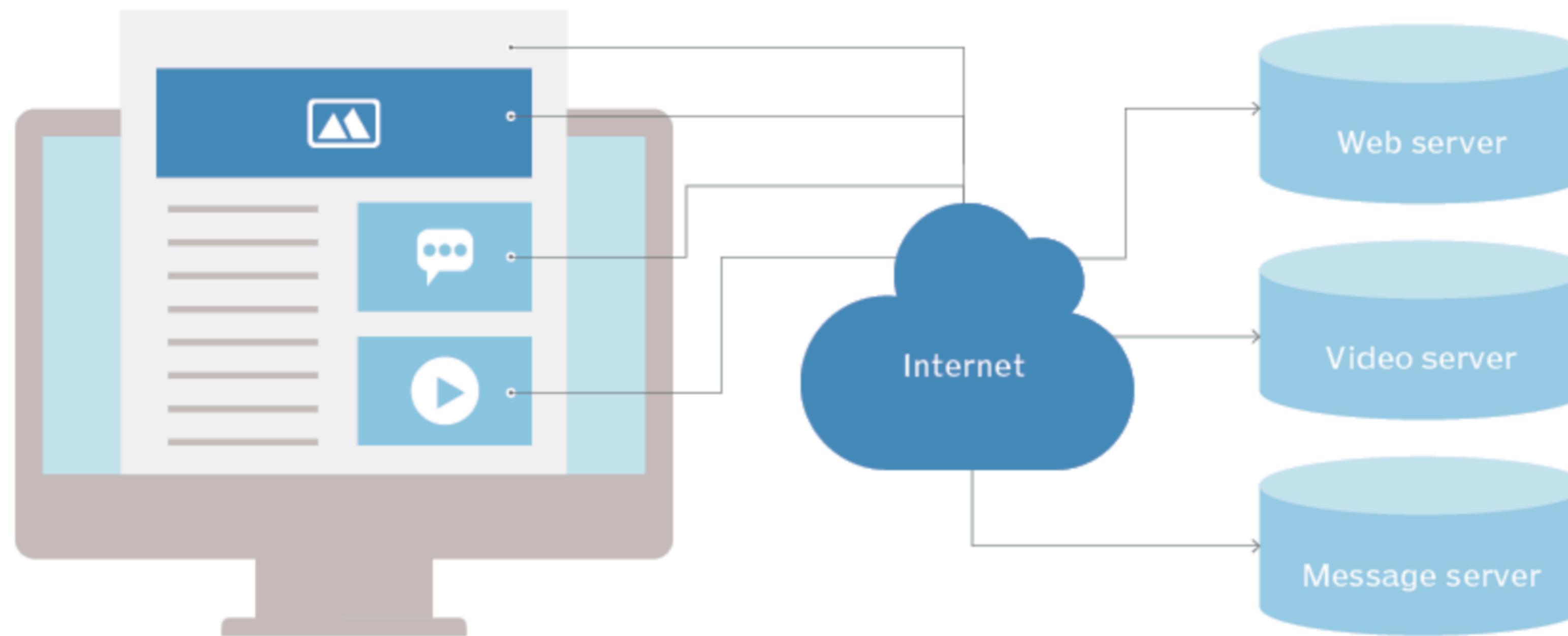
---

## 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>

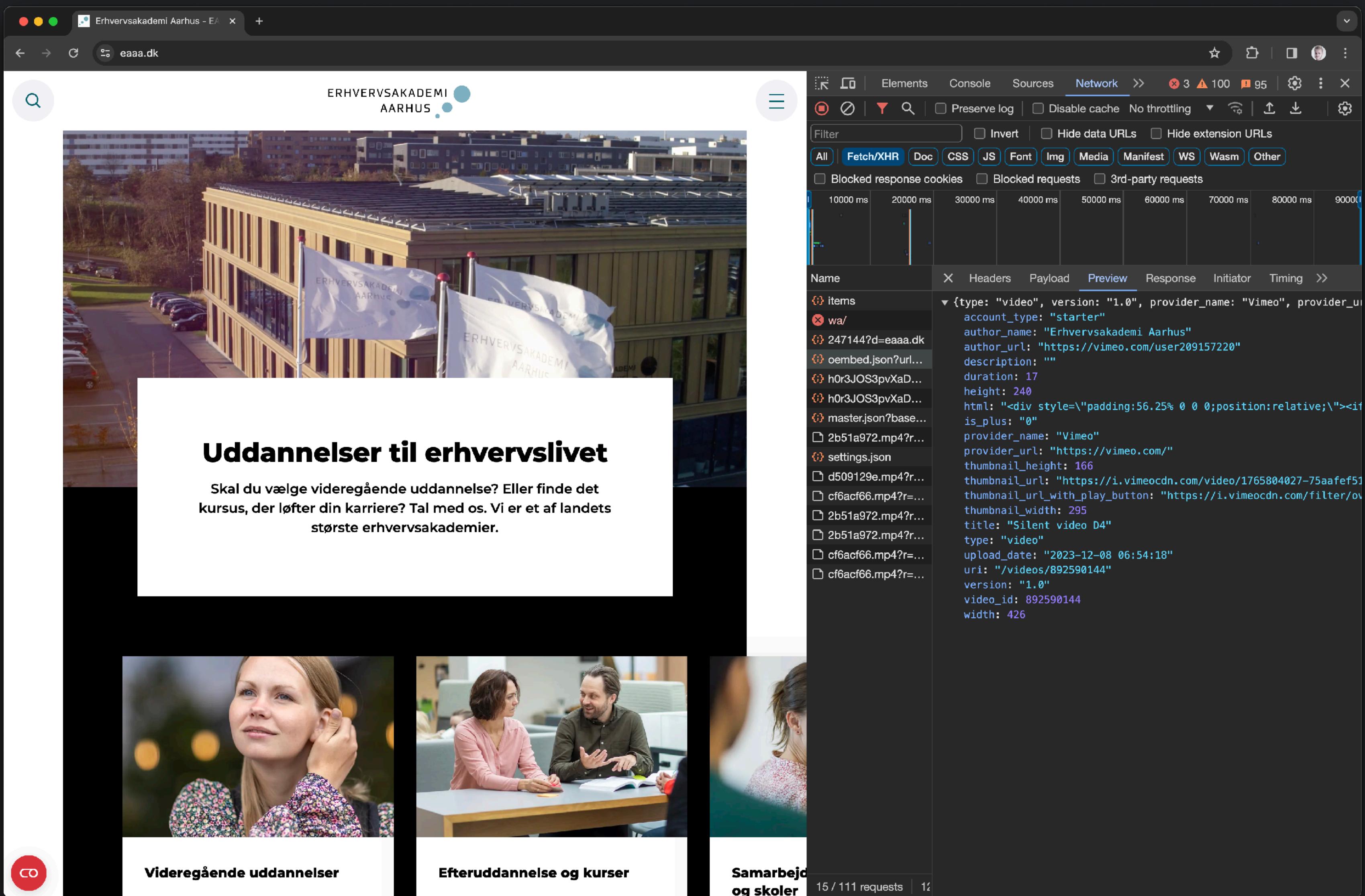
# Network Tab

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

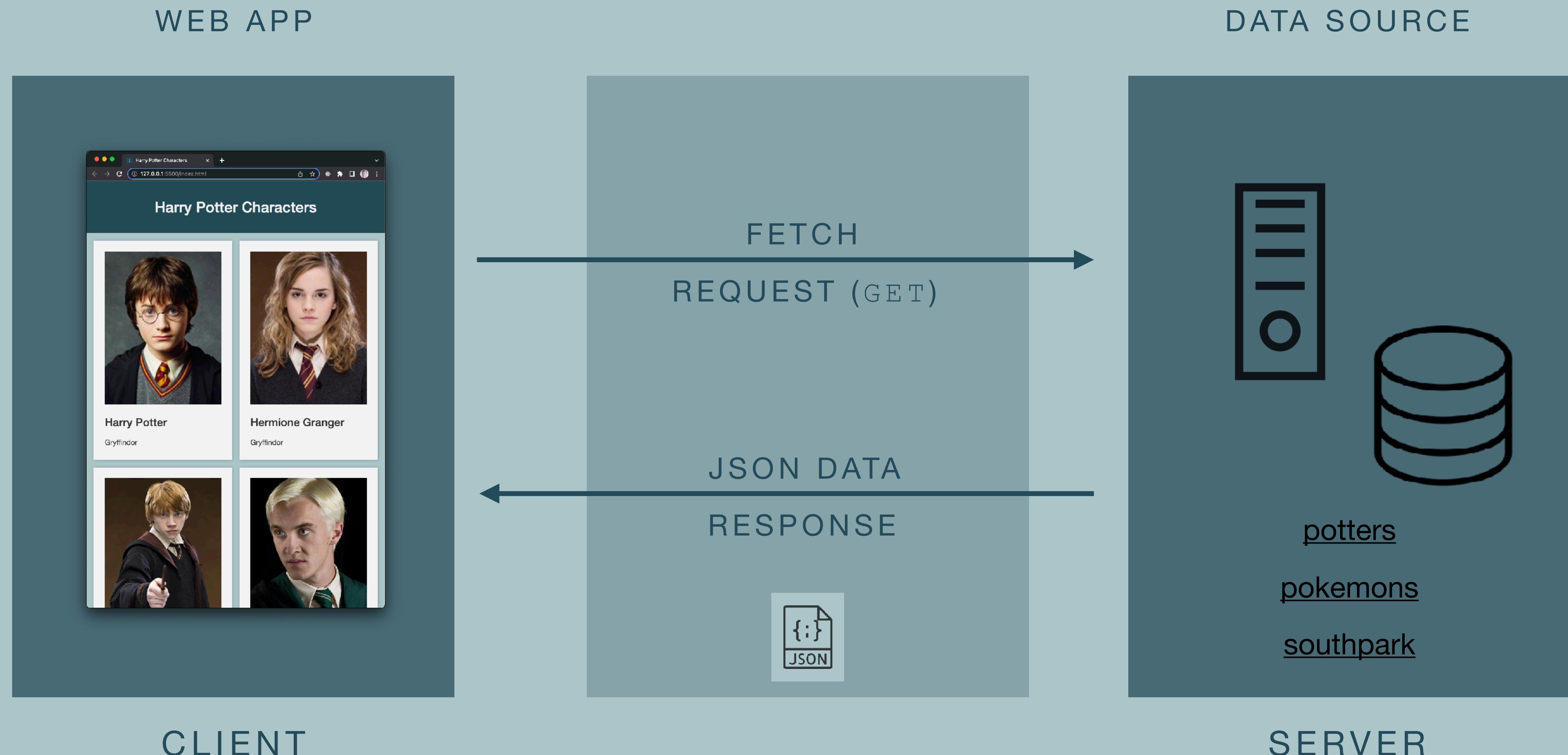
# Network Tab



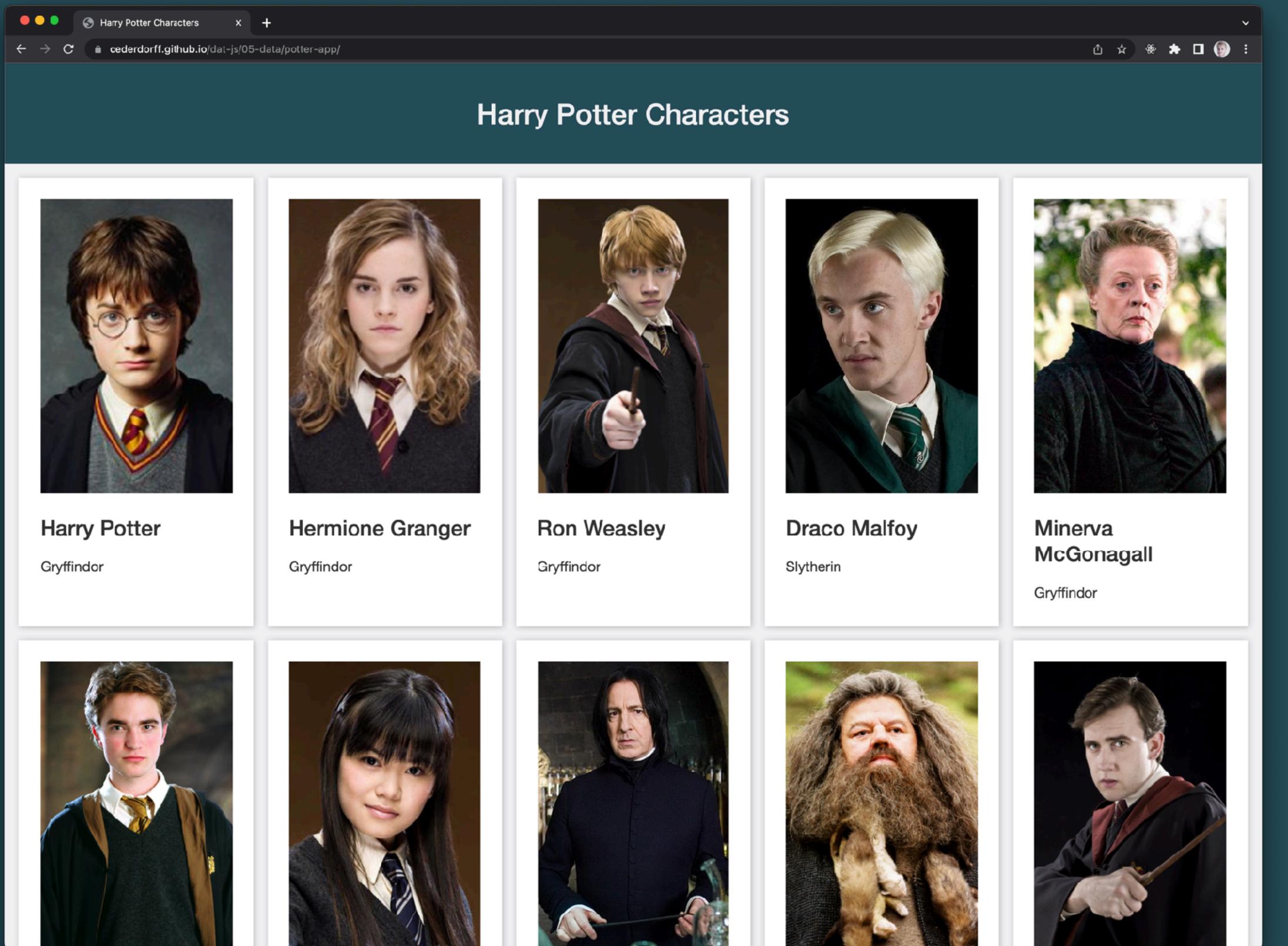
- Use the Network tab to investigate a website (e.g. [kea.dk](#), [dr.dk](#), [google.dk](#), [react-rest-and-auth.web.app](#) or any other).
- Open the website in Google Chrome (or another browser).
- Open the Developer Tool (see: [How to open the dev tool in your browser](#)) and go to Netværk/Network.
- 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-tabben

# Fetch, HTTP Request & Response



# Frontend (client)



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

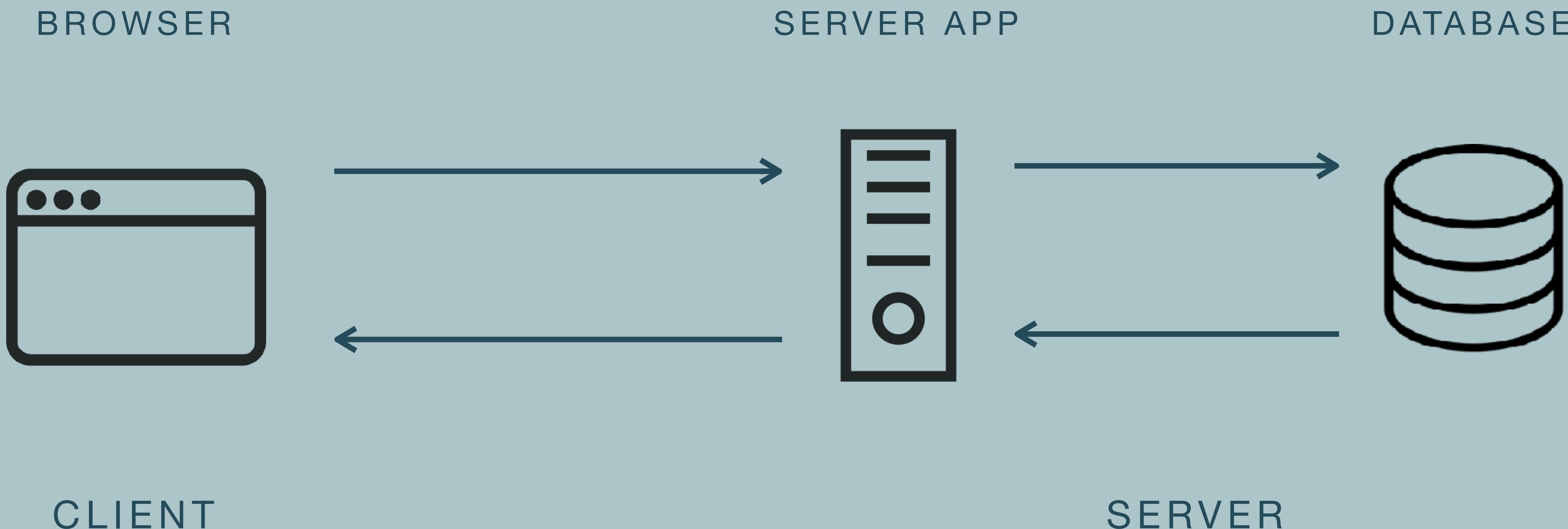
# JSON Data Source (Server)

A screenshot of a web browser window showing a JSON object. The JSON represents the Harry Potter characters from the application. It uses the 'Raw' tab to show the unformatted JSON code, which is color-coded for readability. The 'Parsed' tab shows the JSON structure as a tree. The JSON includes detailed character information such as name, species, gender, house, date of birth, year of birth, ancestry, eye color, hair color, and wand details (wood, core, length, patronus, etc.).

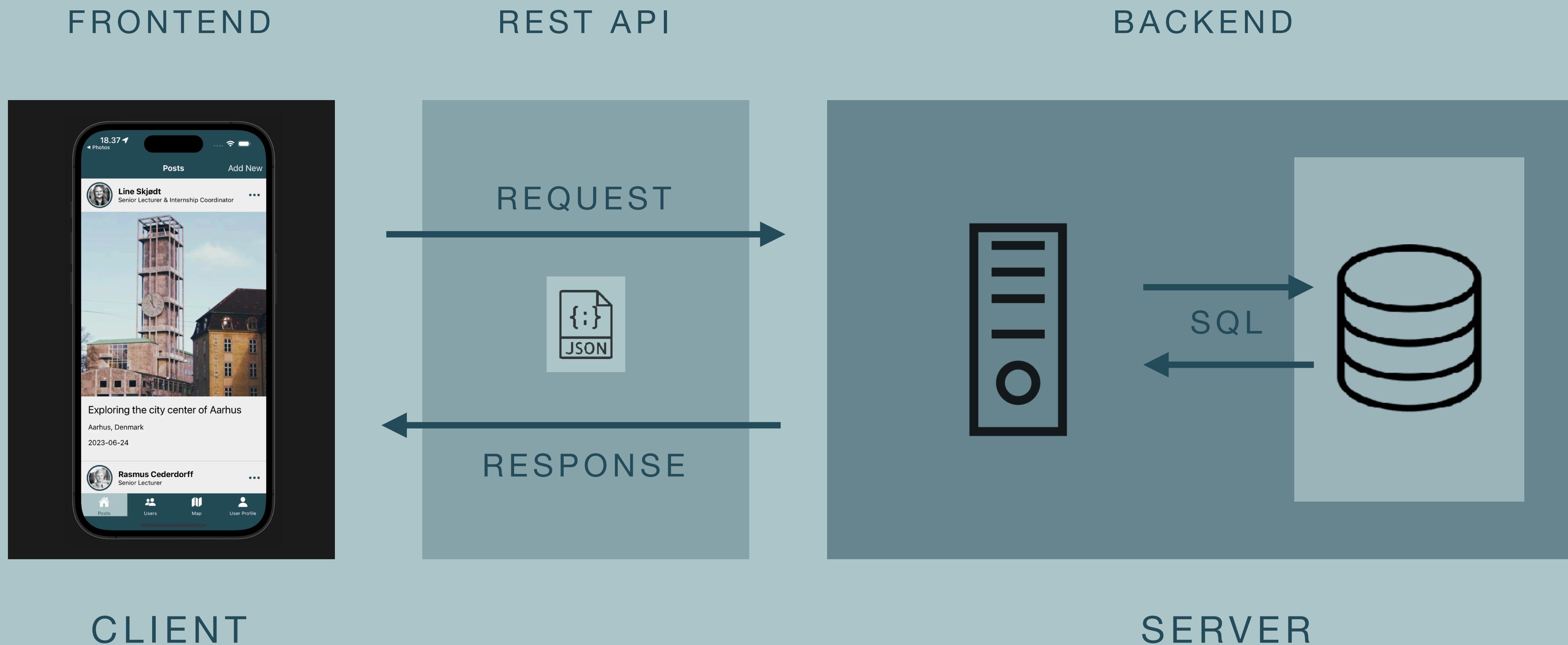
```
name: "Harry Potter",
  species: "human",
  gender: "male",
  house: "Gryffindor",
  dateOfBirth: "31-07-1980",
  yearOfBirth: 1980,
  ancestry: "half-blood",
  eyeColour: "green",
  hairColour: "black",
  wand: {
    wood: "holly",
    core: "phoenix feather",
    length: 11
  },
  patronus: "stag",
  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>

# Web Dev Architecture



# Web Dev Architecture



# API

Application Programming Interface

# API

Application Programming Interface

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

# API

Enables communication between different software applications, specifying how they can request and exchange information.

# API



The image shows a screenshot of a development environment. At the top, there's a menu bar with 'Insomnia' and other options. Below it, there are two tabs: 'routes.py <...' and 'routes.py > ...'. The left tab contains Python code for a Flask application:

```
from flask import Flask, request
app = Flask("python-api-with-flask")
usuarios = []
@app.route("/olamundo", methods=["GET"])
def olamundo():
    return "Olá mundo"
@app.route("/cadastrar_usuario", methods=["POST"])
def cadastrar_usuario(usuario):
    usuarios.append(usuario)
    return {"usuario": usuario}
@app.route("/listar_usuarios", methods=["GET"])
def listar_usuarios():
    return {"usuarios": usuarios}
app.run()
```

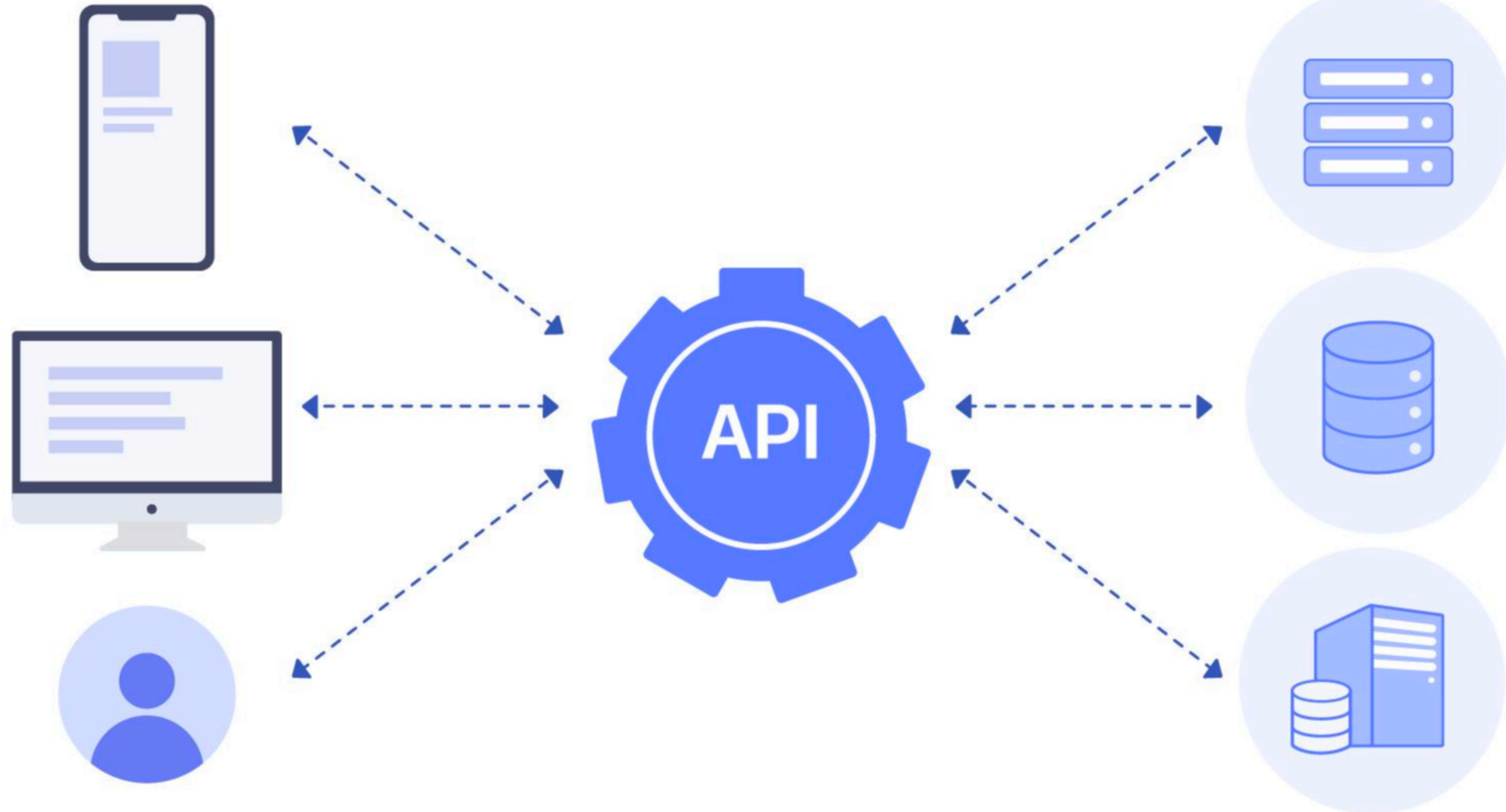
The right tab shows the 'Dashboard / Python with flask' interface with a 'No Environment' section. It lists three routes: 'GET /listar\_usuarios', 'POST /cadastrar\_usuario', and 'GET /olamundo'. Below this, there's a JSON response example:

```
[{"name": "Douglas", "lastName": "Lopes"}, {"name": "Luis", "lastName": "Silva"}, {"name": "Luis", "lastName": "Silva"}]
```

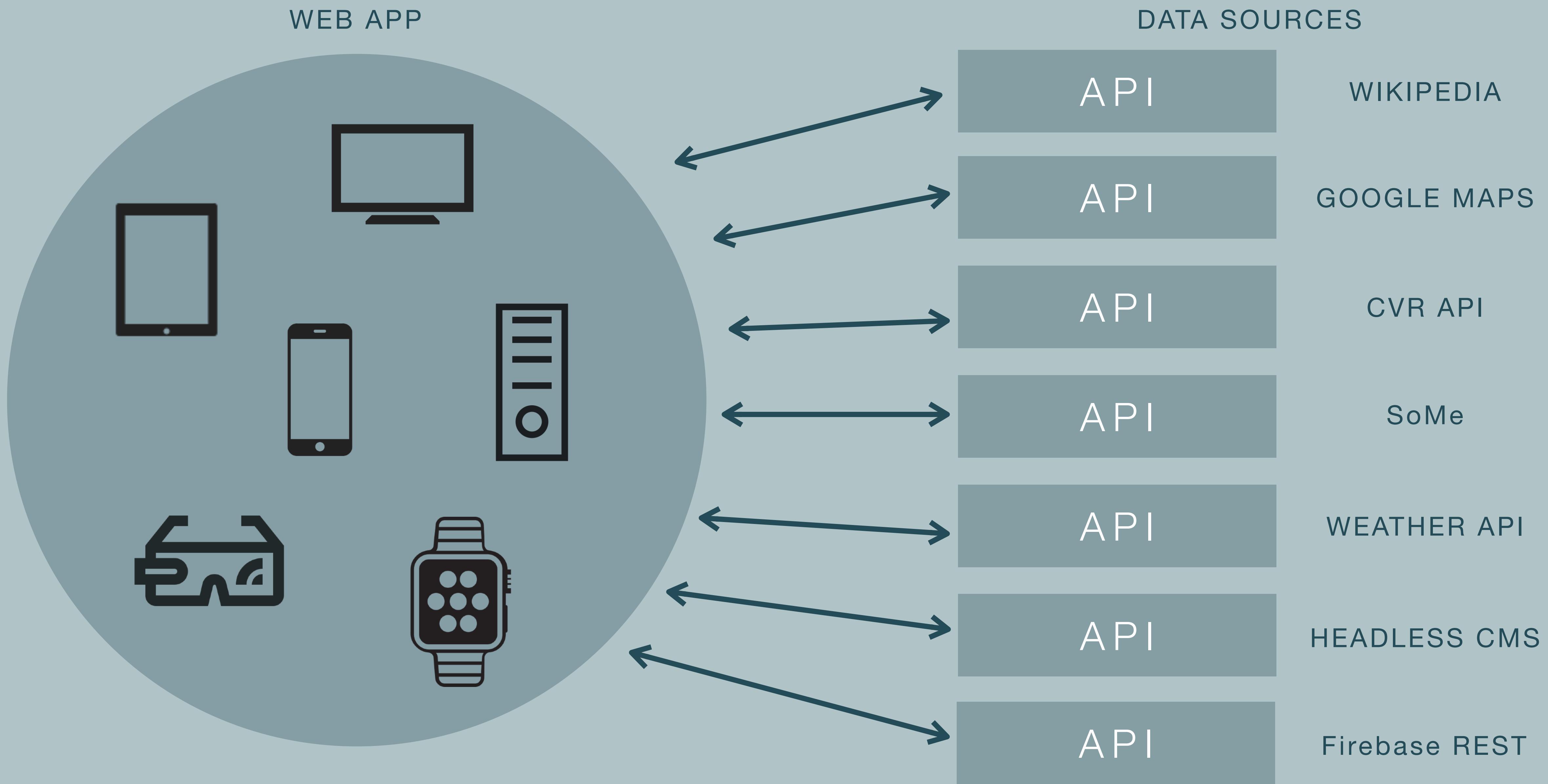
At the bottom, there's a terminal window with the title 'Python 3.8.5 64-bit (base': conda)'. It shows the command 'python routes.py' being run and the output of the application running on port 5000.

- A set of rules and tools that allows different software applications to communicate and interact with each other.
- It defines the methods and data formats that applications can use to request and exchange information.
- APIs are crucial for enabling the integration of different services and systems.

# API

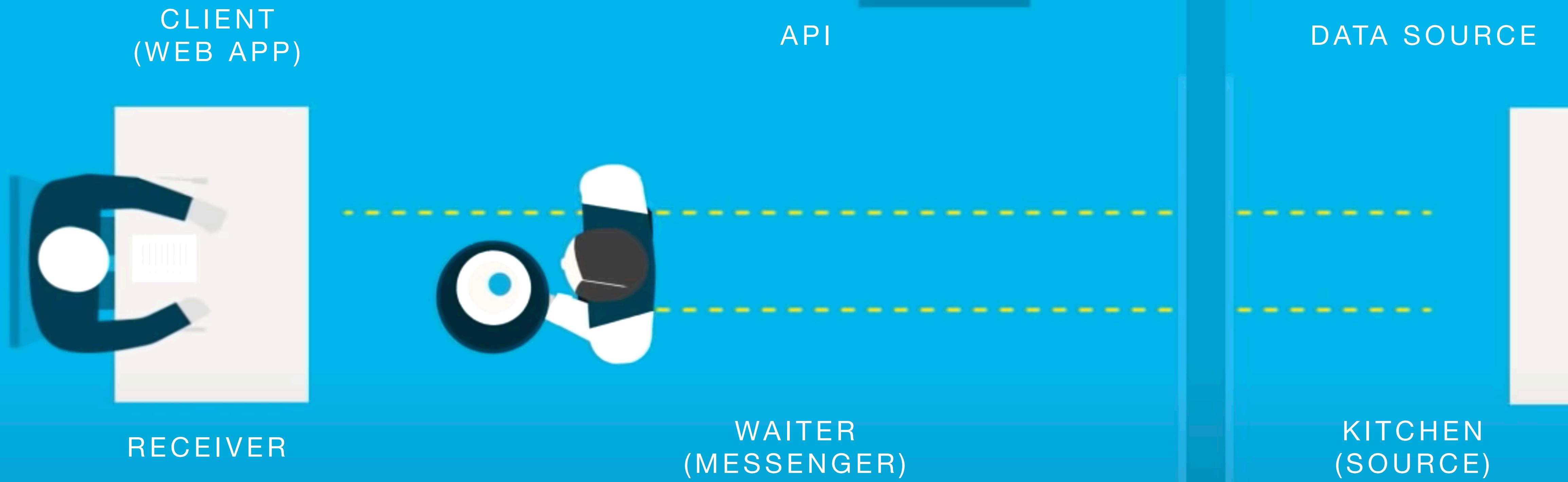


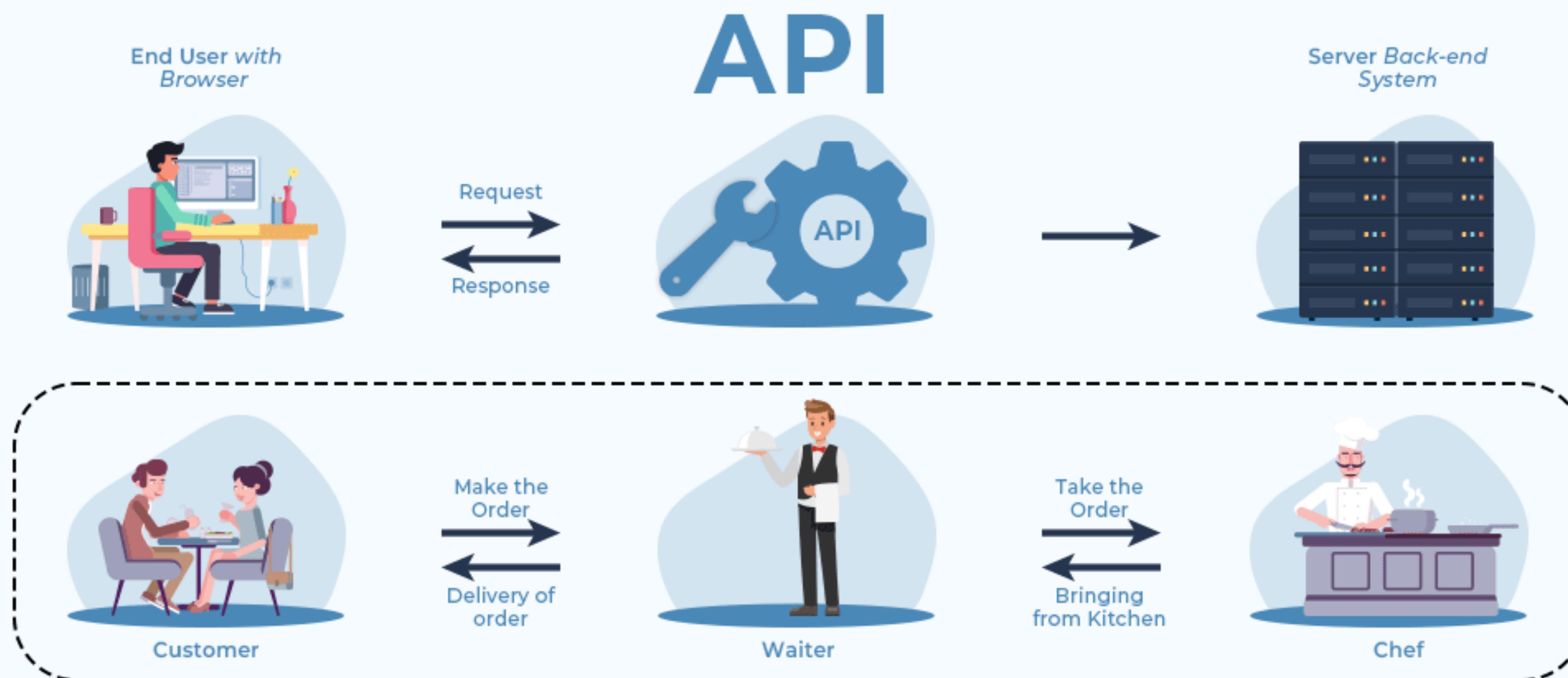
# API





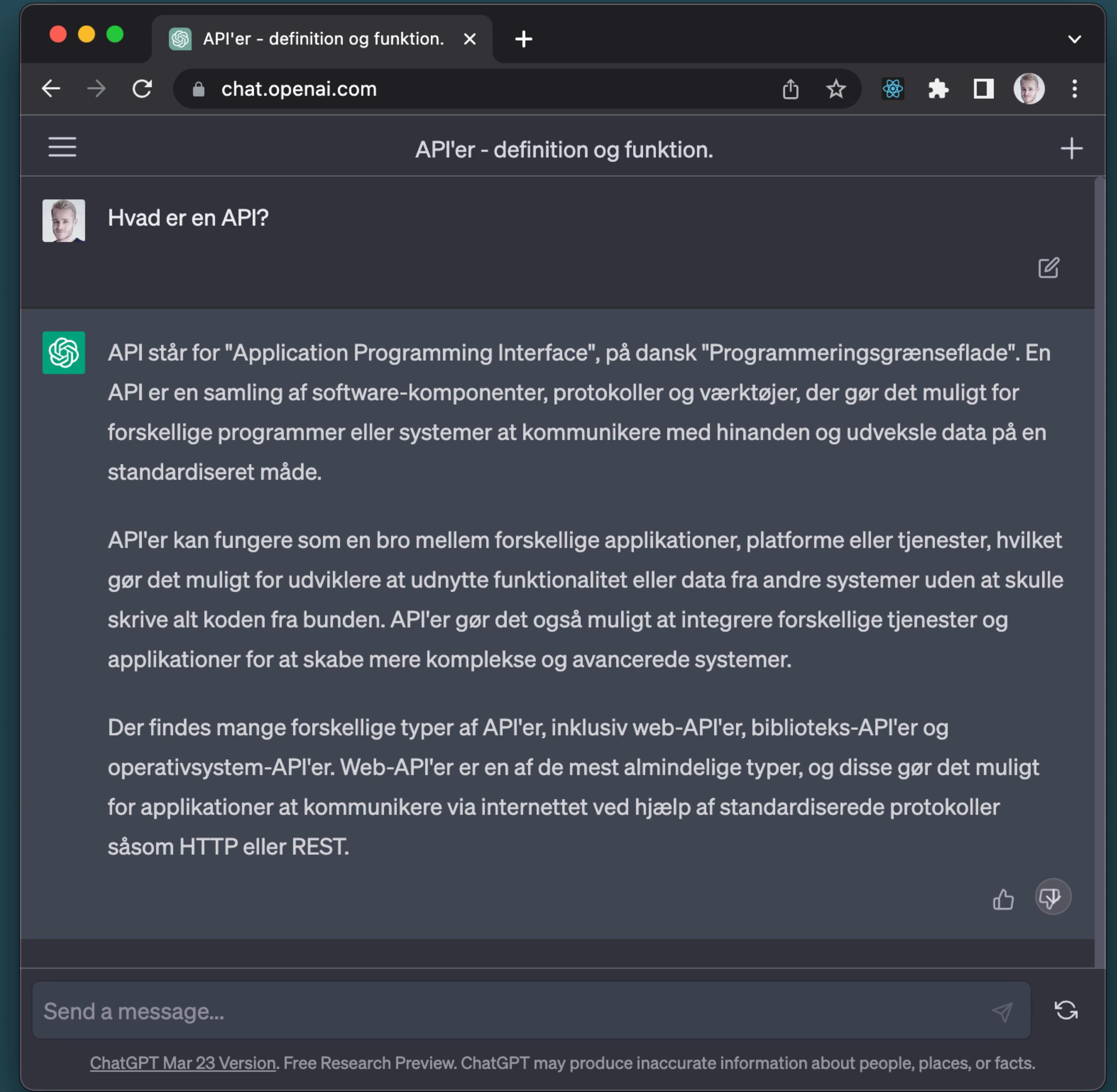
<https://www.youtube.com/watch?v=s7wmiS2mSXY>





# 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 a detailed AI response explaining what an API is and its functions. Below the response, there's a paragraph about the versatility of APIs and how they facilitate integration between different systems. At the bottom, there's a message input field and a note about the ChatGPT version.

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.

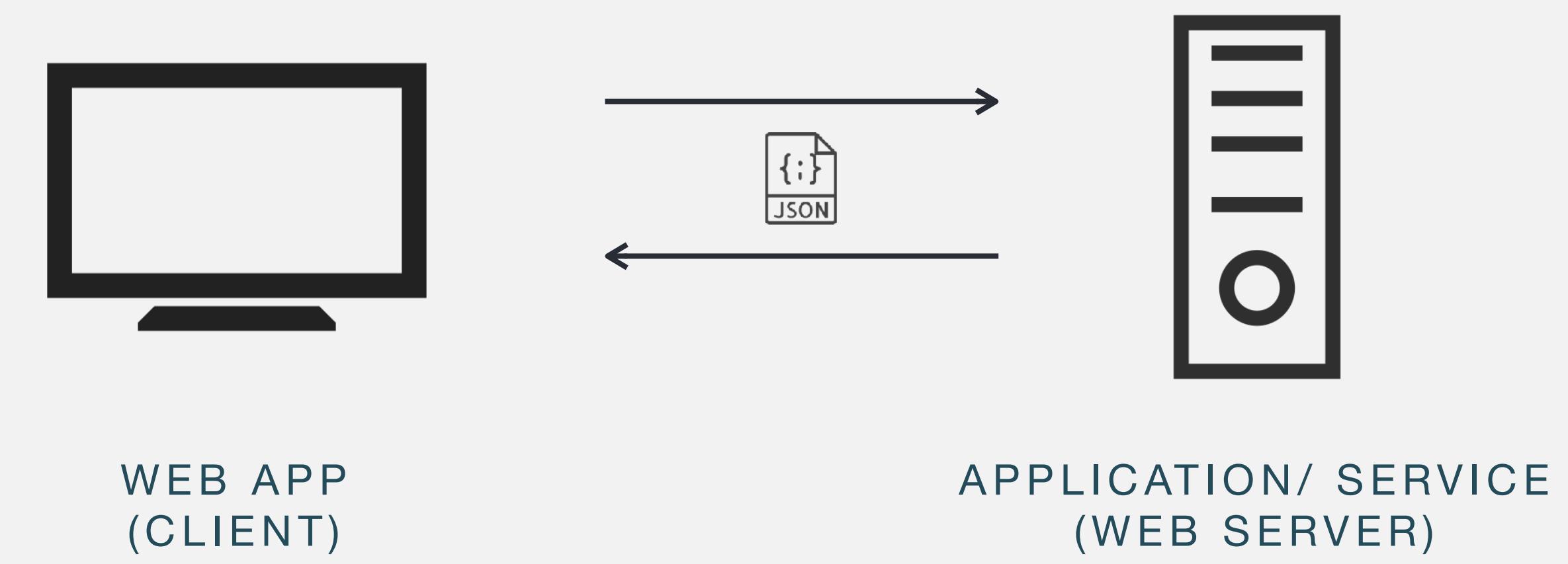
API'er kan fungere som en bro mellem forskellige applikationer, platforme eller tjenester, hvilket gør det muligt for udviklere at udnytte funktionalitet eller data fra andre systemer uden at skulle skrive alt koden fra bunden. API'er gør det også muligt at integrere forskellige tjenester og applikationer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

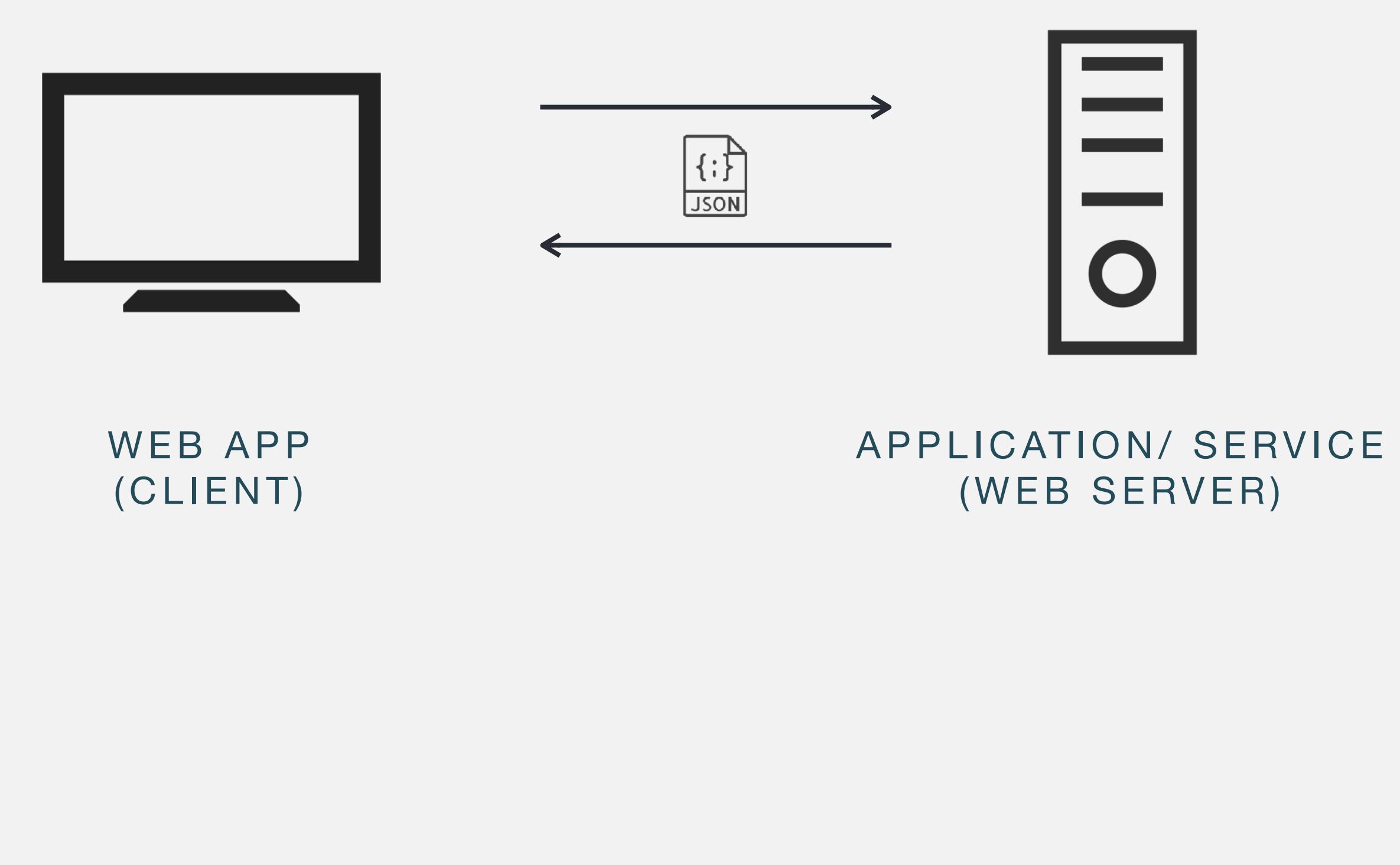
# Application Programming Interface

- **API Definition:** Application Programming Interface
- **Purpose:** Facilitates communication between software applications
- **Functionality:** Defines methods and data formats for information exchange
- **Integration:** Enables seamless interaction between different systems
- **Essential for:** Connecting and integrating diverse software services
- **Examples:** RESTful APIs, GraphQL, SOAP
- **Key Actions:** Requesting, sending, and exchanging data
- **Crucial for:** System interoperability and building modular applications



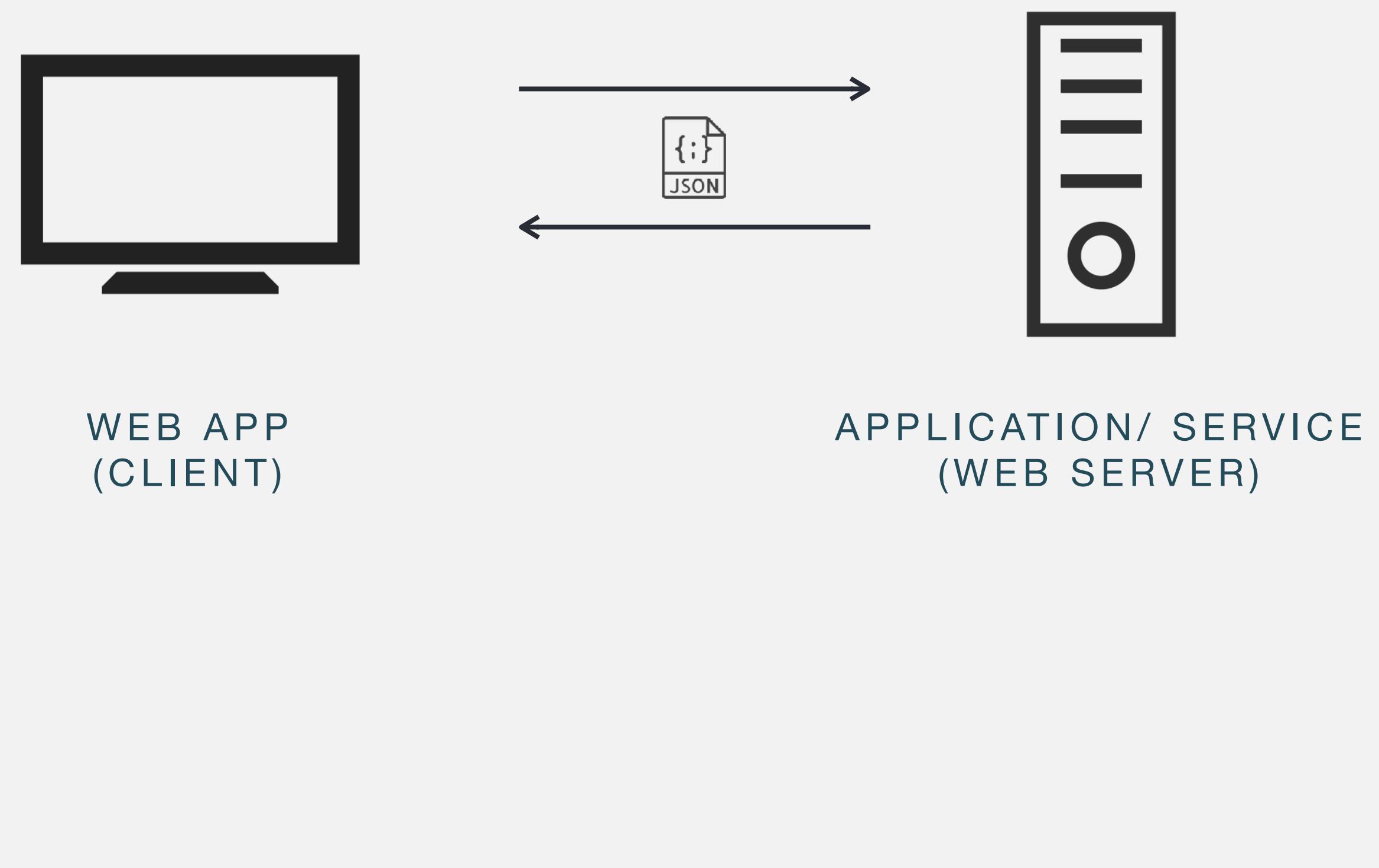
# Application Programming Interface

- **Communication Standard:** API sets a standard for how different software components should interact. It's an interface that allows two systems to communicate and exchange data in a standardised way.
- **Platform Agnostic / Platform independent:** APIs allow applications on different platforms to connect and work together. It can be used by different clients, devices, servers and users: websites, web apps, mobile apps, webshops and other clients
- **Abstraction Layer:** Serves as an abstraction layer, shielding internal complexities and exposing only necessary functionalities.
- **Scalability:** Facilitates scalability by allowing the integration of new features or services.
- **Developer Collaboration:** Enables collaboration between developers, allowing them to leverage each other's work.
- **Documentation Importance:** Well-documented APIs are crucial for developers to understand and use them effectively.
- **Security Measures:** APIs often include security measures like authentication and authorization for secure data exchange.
- **Real-time Interaction:** Supports real-time data transfer, enhancing dynamic and interactive applications.

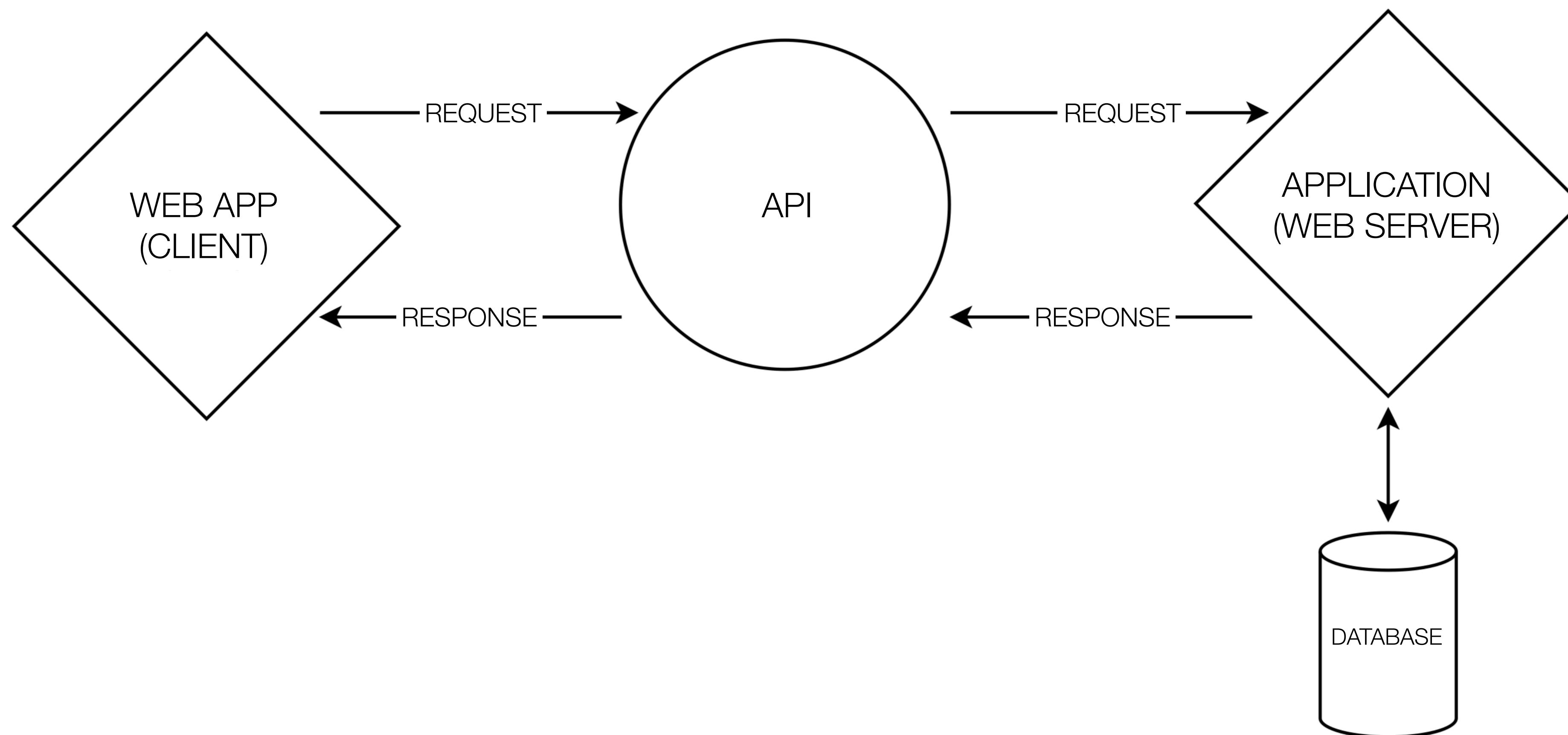


# Application Programming Interface

- **Data Formats:** APIs often use common data formats like JSON or XML for easy comprehension and parsing.
- **HTTP Methods:** Commonly employs HTTP methods such as GET, POST, PUT, and DELETE to perform different actions.
- **Versioning:** APIs may have versioning to manage changes and updates without disrupting existing integrations.
- **Rate Limiting:** Implementing rate limiting helps control the number of requests a client can make within a specified time frame.
- **Feedback Mechanism:** APIs often provide feedback in the form of status codes to indicate the success or failure of a request.
- **Testing and Debugging:** Developers can test and debug applications independently since APIs provide a clear interface.
- **Third-party Integration:** Enables third-party developers to build on top of existing services, fostering a broader ecosystem.
- **RESTful Principles:** RESTful APIs adhere to principles like statelessness and a uniform interface for consistency.



# Application Programming Interface



Course roster: WU-E22a - 1. se X +

https://eaaa.instructure.com/courses/15482/users

WU-E22a > People

60 Student view

Home Announcements Modules Assignments Discussions People BigBlueButton Grades Pages Files Syllabus Outcomes Rubrics Quizzes Collaborations Settings

Everyone Groups + Group set

Search people All roles + People

Name	Login ID	SIS ID	Section	Role	Last Activity	Total Activity
Clara Juul Birk	eaaclbi@students.eaaa.dk	WU-E22a - 1.	Student semester	Student	24 Aug at 13:16	01:04:21
Martin Rieper Boesen	eaamrbo@students.eaaa.dk	WU-E22a - 1.	Student semester	Student	24 Aug at 7:54	01:07:06
Dan Okkels Brendstrup	dob@eaaa.dk	WU-E22a - 1.	Teacher semester	Teacher	3 Aug at 8:55	
Rasmus Cederdorff	race@eaaa.dk	WU-E22a - 1.	Teacher semester	Teacher	25 Aug at 9:28	01:19:23
Jeffrey David Serio	jds@eaaa.dk	WU-E22a - 1.	Teacher semester	Teacher	17 Aug at 16:39	
Charlotte Meng Emanuel Dyrholm	eaacmed@students.eaaa.dk	WU-E22a - 1.	Student semester	Student	23 Aug at 16:59	22:24

Elements Components Network 22 Preset log Disable cache No throttling Invert Hide data URLs

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

Has blocked cookies Blocked Requests 3rd-party requests

5000 ms 10000 ms 15000 ms 20000 ms 25000 ms 30000 ms 35000 ms

Name Key Headers Payload Preview Response Initiator >

0: {id: "23974", name: "Clara Juul Birk", created\_at: "2020-08-24T13:16:21+02:00", email: "eaaclbi@students.eaaa.dk", login\_id: "eaaclbi@students.eaaa.dk", role: "Student", section: "WU-E22a - 1.", sis\_user\_id: null, short\_name: "Clara Juul Birk", sortable\_name: "Birk, Clara Juul", unreadable: false}

1: {id: "36267", name: "Martin Rieper Boesen", created\_at: "2020-08-24T07:54:06+02:00", email: "eaamrbo@students.eaaa.dk", login\_id: "eaamrbo@students.eaaa.dk", role: "Student", section: "WU-E22a - 1.", sis\_user\_id: null, short\_name: "Rieper Boesen, Martin", sortable\_name: "Boesen, Martin Rieper", unreadable: false}

2: {id: "29923", name: "Dan Okkels Brendstrup", created\_at: "2021-07-30T00:46:05+02:00", email: "dob@eaaa.dk", login\_id: "dob@eaaa.dk", role: "Teacher", section: "WU-E22a - 1.", sis\_user\_id: null, short\_name: "Brendstrup, Dan Okkels", sortable\_name: "Brendstrup, Dan Okkels", unreadable: false}

3: {id: "14427", name: "Rasmus Cederdorff", created\_at: "2020-08-25T09:28:23+02:00", email: "race@eaaa.dk", login\_id: "race@eaaa.dk", role: "Teacher", section: "WU-E22a - 1.", sis\_user\_id: null, short\_name: "Cederdorff, Rasmus", sortable\_name: "Cederdorff, Rasmus", unreadable: false}

4: {id: "41", name: "Jeffrey David Serio", created\_at: "2020-08-17T16:39:39+02:00", email: "jds@eaaa.dk", login\_id: "jds@eaaa.dk", role: "Teacher", section: "WU-E22a - 1.", sis\_user\_id: null, short\_name: "Serio, Jeffrey David", sortable\_name: "Serio, Jeffrey David", unreadable: false}

5: {id: "24043", name: "Charlotte Meng Emanuel Dyrholm", created\_at: "2020-08-23T22:24:59+02:00", email: "eaacmed@students.eaaa.dk", login\_id: "eaacmed@students.eaaa.dk", role: "Student", section: "WU-E22a - 1.", sis\_user\_id: null, short\_name: "Dyrholm, Charlotte Meng Emanuel", sortable\_name: "Dyrholm, Charlotte Meng Emanuel", unreadable: false}

6: {id: "23978", name: "Jeppe Frik", created\_at: "2020-08-03T16:59:00+02:00", email: null, login\_id: null, role: null, section: null, sis\_user\_id: null, short\_name: null, sortable\_name: null, unreadable: true}

7: {id: "23963", name: "Daniel Tjerrild Gamborg", created\_at: "2020-08-03T16:59:00+02:00", email: null, login\_id: null, role: null, section: null, sis\_user\_id: null, short\_name: null, sortable\_name: null, unreadable: true}

8: {id: "23992", name: "Casper Hedegaard Hansen", created\_at: "2020-08-03T16:59:00+02:00", email: null, login\_id: null, role: null, section: null, sis\_user\_id: null, short\_name: null, sortable\_name: null, unreadable: true}

9: {id: "36266", name: "Morten Gedsted Hansen", created\_at: "2020-08-03T16:59:00+02:00", email: null, login\_id: null, role: null, section: null, sis\_user\_id: null, short\_name: null, sortable\_name: null, unreadable: true}

10: {id: "23980", name: "Anders Husted", created\_at: "2020-08-03T16:59:00+02:00", email: null, login\_id: null, role: null, section: null, sis\_user\_id: null, short\_name: null, sortable\_name: null, unreadable: true}

11: {id: "23531", name: "Søren Bo Jørgensen", created\_at: "2020-08-03T16:59:00+02:00", email: null, login\_id: null, role: null, section: null, sis\_user\_id: null, short\_name: null, sortable\_name: null, unreadable: true}

# Canvas

Find API uses to integrate systems.

The screenshot shows the Canvas Learning Management System (LMS) interface. On the left is a sidebar with various navigation links: Account, Admin, Dashboard, Courses, Calendar, Inbox, History, Commons, Student links, BigBlueButton, Grades, Pages, Files, Syllabus, Outcomes, Rubrics, Quizzes, Collaborations, and Settings. The main area displays a 'Course roster: WU-E22a - 1. sem' with a table of student information. A 'People' section is also visible. On the right, a developer's browser window is open, showing Network traffic and a JSON response body containing user data.

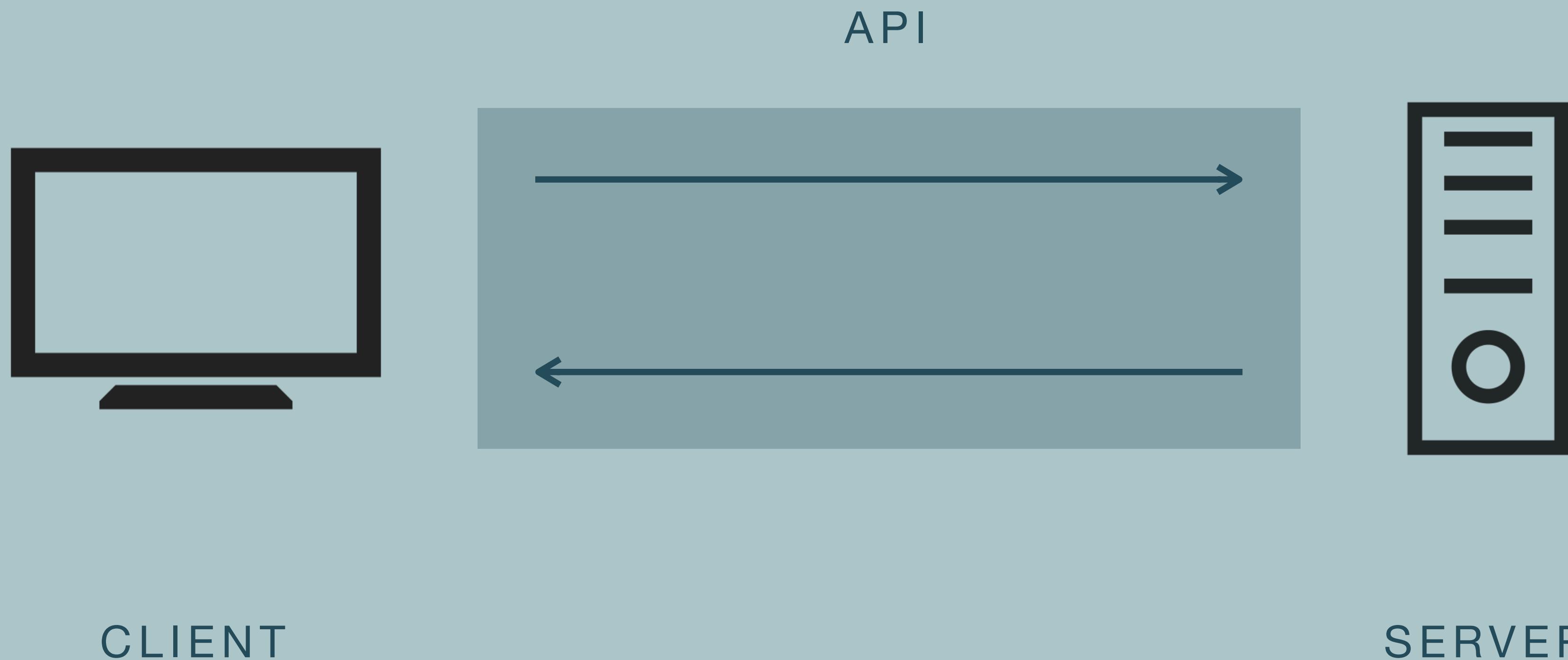
Name	Login ID	SIS ID	Section	Role	Last Activity	Total Activity
Clara Juul Birk	eaacibi@students.eaaa.dk	WU-E22a - 1.	Student	24 Aug semester	at 13:16	01:04:21
Martin Rieper Boesen	eaamrbo@students.eaaa.dk	WU-E22a - 1.	Student	24 Aug at 7:54	01:07:06	
Dan Okkels Brendstrup	dob@eaaa.dk	WU-E22a - 1.	Teacher	3 Aug semester	at 8:55	
Rasmus Cederdorff	race@eaaa.dk	WU-E22a - 1.	Teacher	25 Aug semester	at 9:28	01:19:23
Jeffrey David Serio	jds@eaaa.dk	WU-E22a - 1.	Teacher	17 Aug semester	at 16:39	
Chalotte Meng Emanuel Dyrholm	eaacmed@students.eaaa.dk	WU-E22a - 1.	Student	23 Aug semester	at 16:59	22:24

```
[{"id": "23974", "name": "Clara Juul Birk", "created_at": "2020-08-01T13:16:21Z"}, {"id": "23974", "name": "Clara Juul Birk", "created_at": "2020-08-01T13:16:21Z"}, {"id": "36267", "name": "Martin Rieper Boesen", "created_at": "2020-08-01T07:54:06Z"}, {"id": "29923", "name": "Dan Okkels Brendstrup", "created_at": "2021-07-30T00:46:05+02:00"}, {"id": "391064", "user_id": "29923", "course_id": "29923", "integration_id": null, "login_id": "dob@eaaa.dk", "name": "Dan Okkels Brendstrup", "short_name": "Dan Okkels Brendstrup (adjunkt \u2013 dob@eaaa.dk)", "sis_user_id": null, "sortable_name": "Brendstrup, Dan Okkels"}, {"id": "14427", "name": "Rasmus Cederdorff", "created_at": "2021-08-01T09:28:00Z"}, {"id": "41", "name": "Jeffrey David Serio", "created_at": "2021-08-01T16:39:00Z"}, {"id": "24043", "name": "Chalotte Meng Emanuel Dyrholm", "created_at": "2021-08-01T16:39:00Z"}, {"id": "23978", "name": "Jeppe Friki", "created_at": "2020-08-07T17:00:00Z"}, {"id": "23963", "name": "Daniel Tjerrild Gamborg", "created_at": "2021-08-07T17:00:00Z"}, {"id": "23992", "name": "Casper Hedegaard Hansen", "created_at": "2021-08-08T17:00:00Z"}, {"id": "36266", "name": "Morten Gedsted Hansen", "created_at": "2021-08-08T17:00:00Z"}, {"id": "23980", "name": "Anders Husted", "created_at": "2020-08-10T17:00:00Z"}, {"id": "23531", "name": "S\u00f8ren Bo J\u00f8rgensen", "created_at": "2021-08-11T17:00:00Z"}]
```



# Application Programming Interface

Communication between two systems



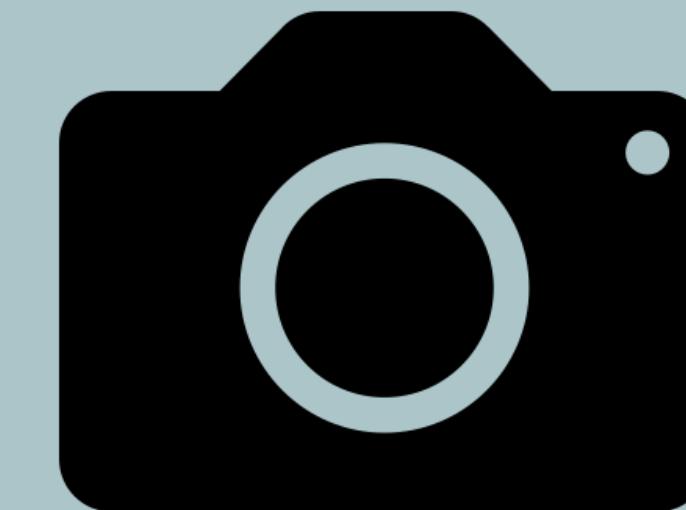
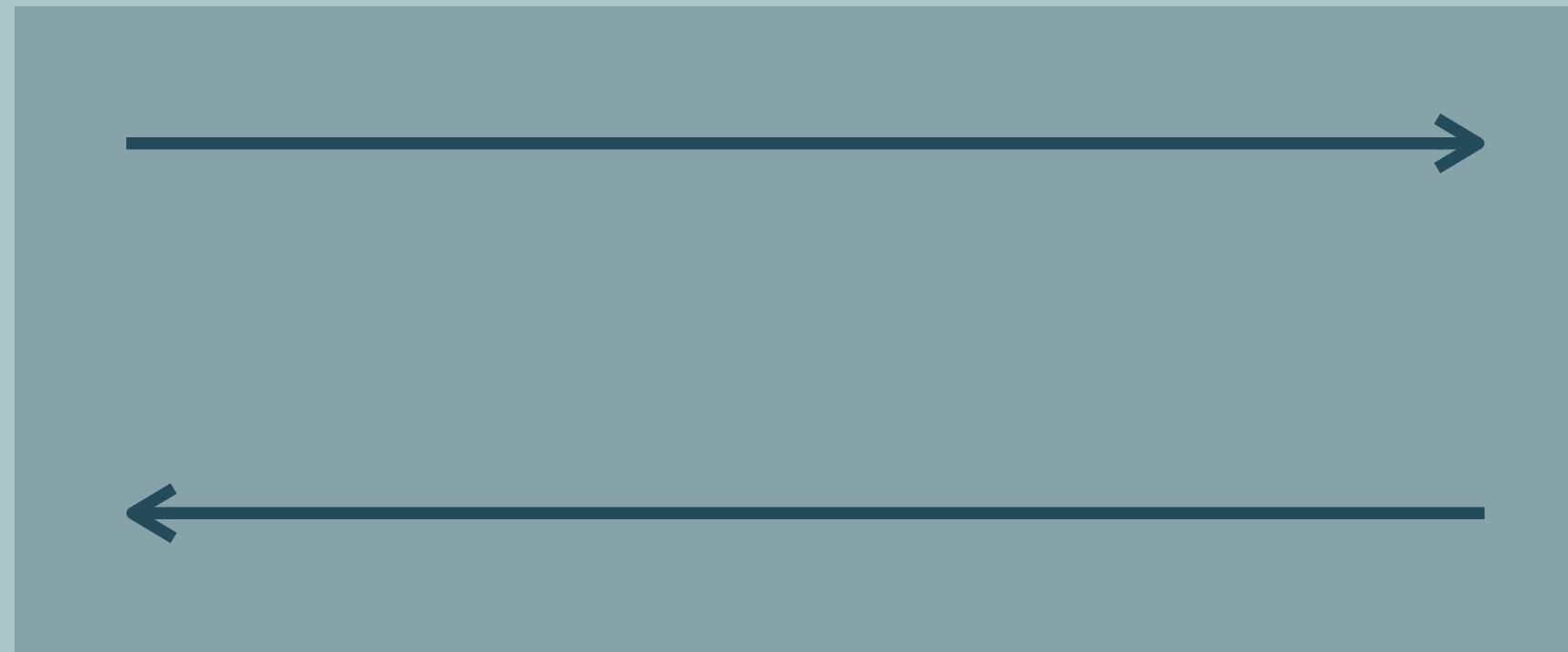
# Application Programming Interface

Communication between two systems



Web App

Camera API



Camera

The screenshot shows a browser window with the Instagram profile of user 'cederdorff'. The profile page displays basic information like posts (185), followers (5,296), and following (708). Below the bio, it says '455 accounts reached in the last 30 days' with a 'View Insights' link. A horizontal scroll bar is visible under the profile section.

The right side of the screen features the Chrome developer tools Network tab, which is active. The tab shows a timeline of network requests. Most requests are for 'graphql' and are categorized as 'Fetch/XHR'. One specific request is expanded to show its details:

```
graph JSON
{
  "data": {
    "edges": [
      {"id": 0, "cursor": "2608844783489918649_5672009939", "is_final": true, "node": {"id": "5672009939", "username": "villacph", "profile_pic_url": "https://i.instagram.com/..."}, "page_info": {"end_cursor": "2608844783489918649_5672009939", "has_next_page": true, "has_previous_page": false}, "xdt_viewer": {"id": "5672009939"}}
    ]
  }
}
```

The bottom of the developer tools shows a summary: 51 / 166 requests.

Rasmus Cederdorff (@cederdorff) | Instagram

Profile Overview: cederdorff (cederdorff) - 185 posts, 5,296 followers, 708 following. Bio: Senior Lecturer & Web App Developer, Teaching JavaScript and Web Dev. Also known as RACE. Dad to Alicia & Ida. Based in Aarhus.

Recent Post:

**cederdorff** Friday it is ✨  
;  
#happyfriday #aarhus #gayman #aarhus  
#golden #danishjewelry #staystrong #gayguy  
#aaaa  
8 w

For you

**View Insights** **Boost Post**

Liked by **margitthorgaard** and 84 others  
15 December 2023

Add a comment... **Post**

Instagram Network Tab (Screenshot from Chrome DevTools Network panel):

The Network tab in the Chrome DevTools Network panel shows a timeline of requests. The main table displays a list of requests with columns for Name, Headers, Payload, Preview, Response, Initiator, Timing, and Cookies. The 'Preview' column is currently selected, showing the JSON response for the most recent GraphQL request. The response payload includes the post's code, pk, id, accessibility\_caption, and the full caption text: "Friday it is ✨\n\n#happyfriday #aarhus #gayman #aarhus #golden #danishjewelry". Other visible requests in the list include various GraphQL queries for the user profile and feed.

# API

More Examples: <https://race.notion.site/API-Examples-48baf0ecb20f490d916edd27f70204ed>

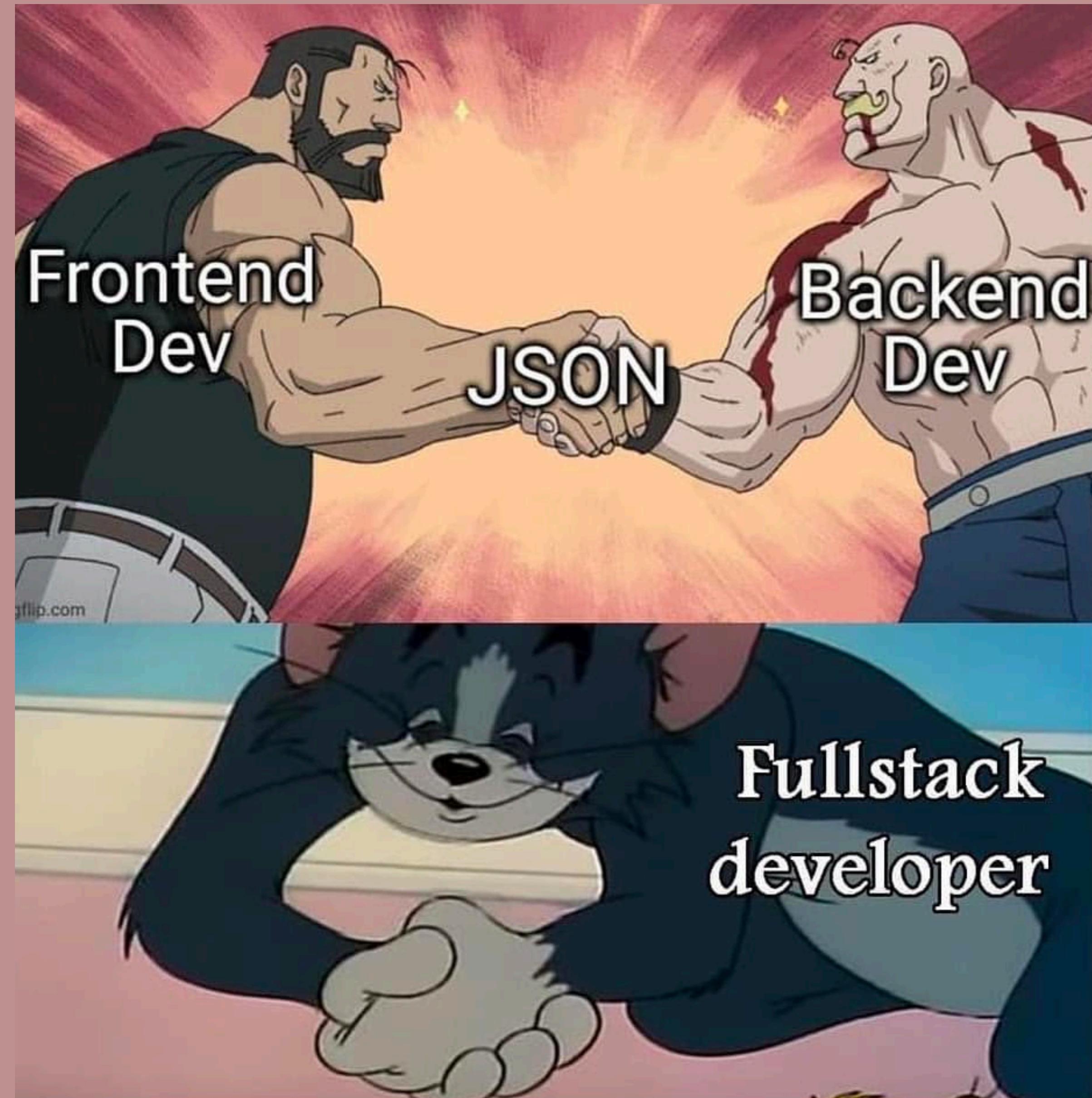
# JSON

JavaScript Object Notation

# JSON

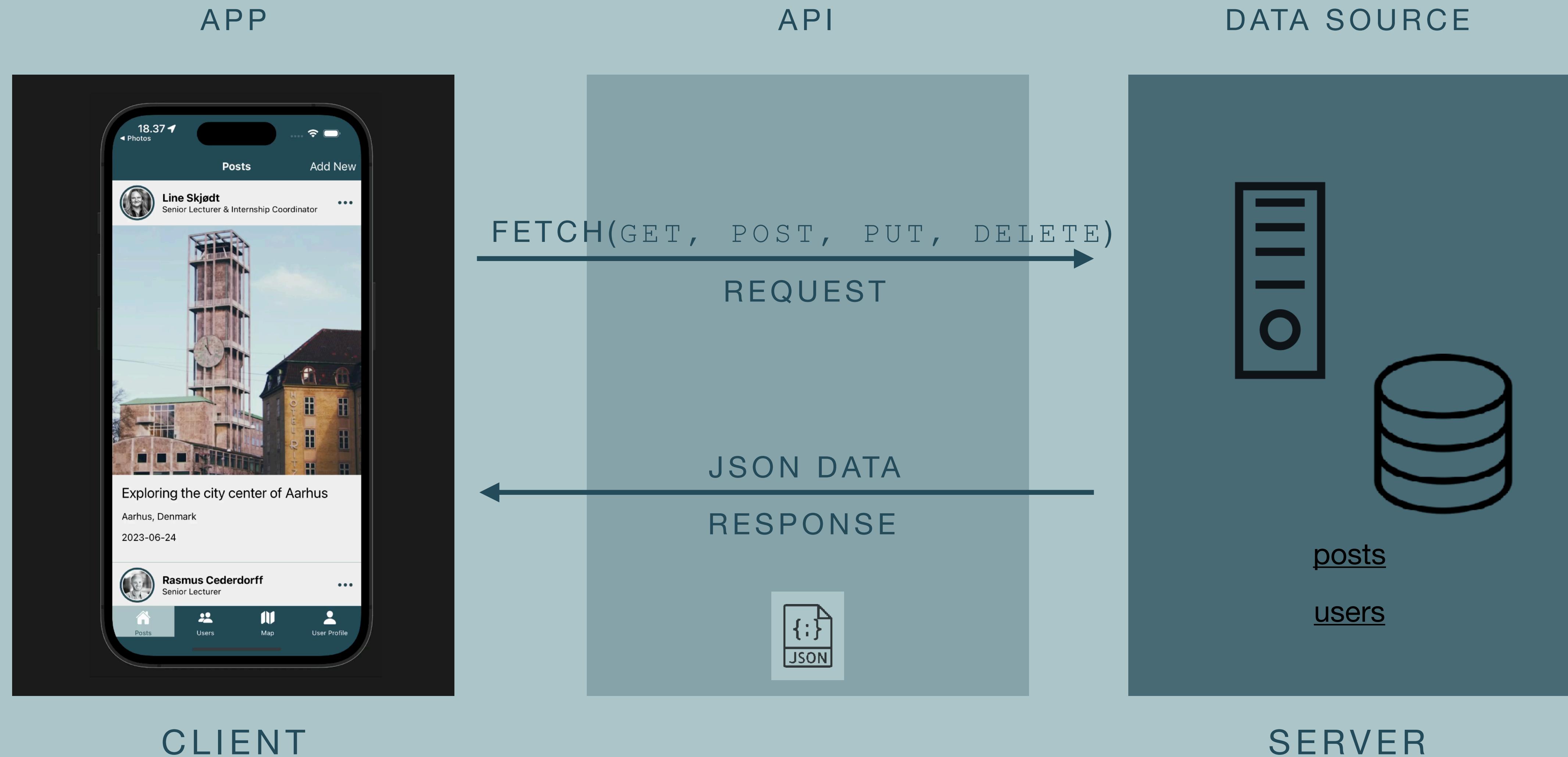
## JavaScript Object Notation

... 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

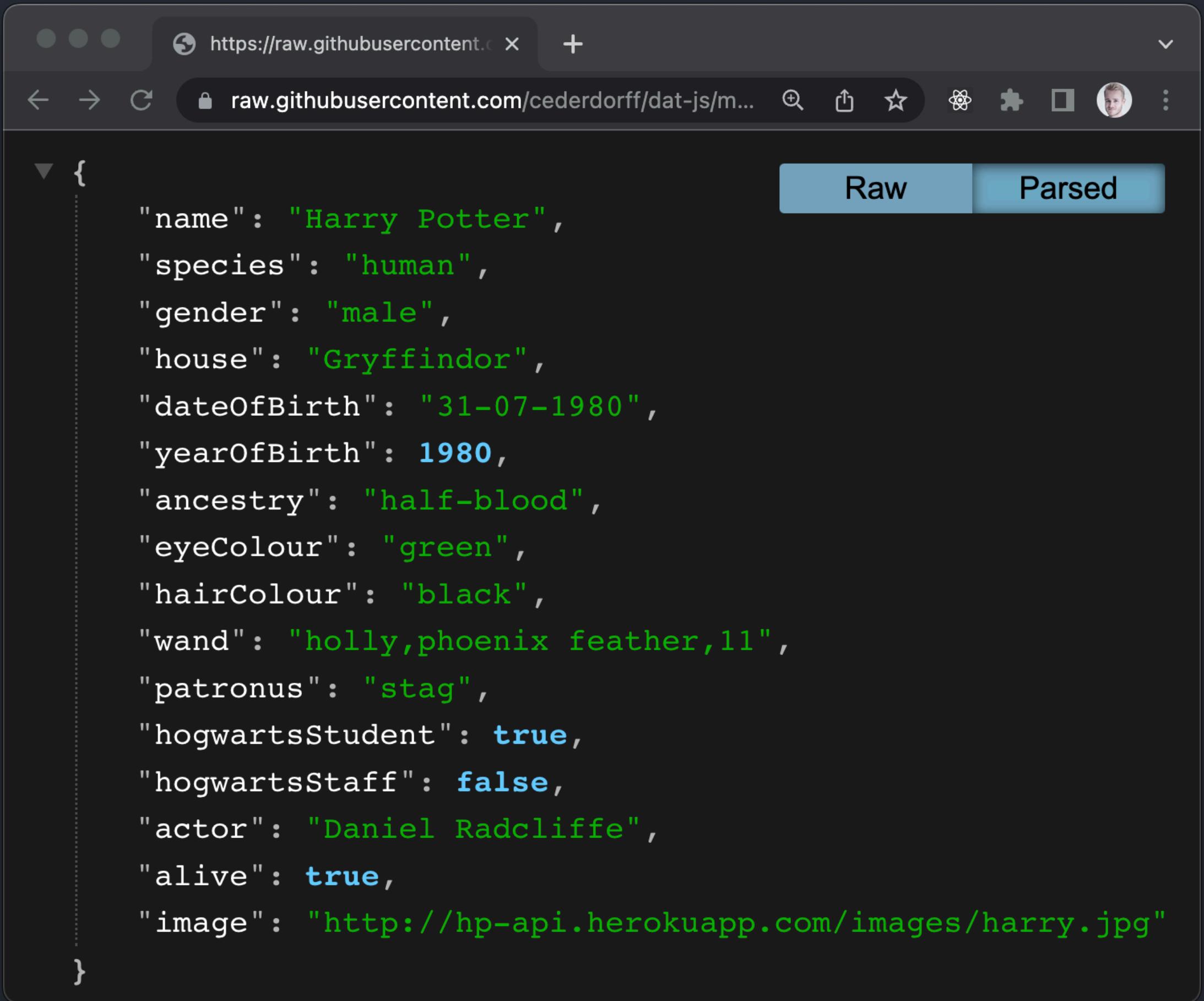
```
{  
  "name": "Alicia",  
  "age": 6  
}
```

JSON OBJECT

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

LIST OF JSON OBJECTS

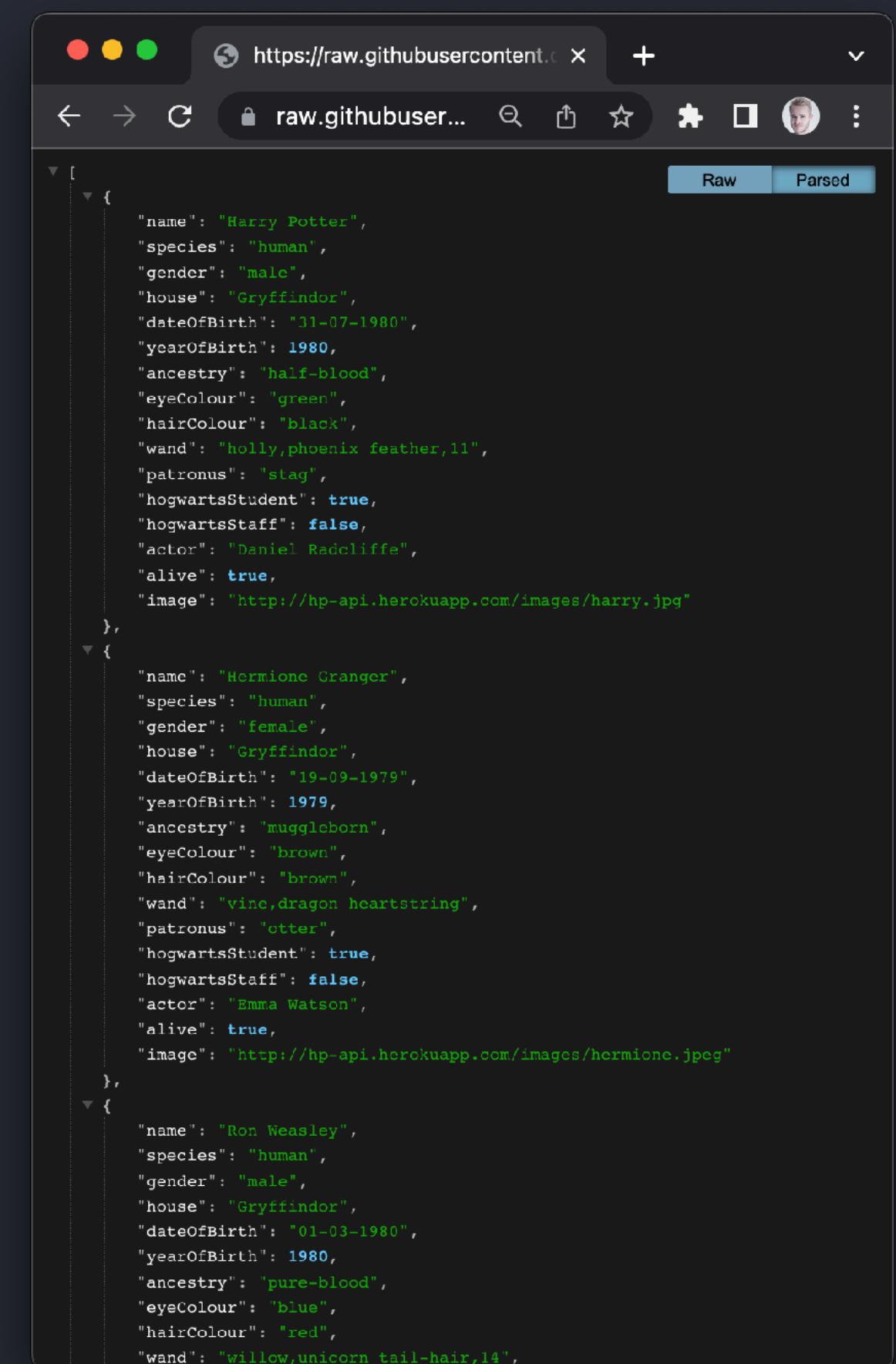
# JSON



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 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 a hierarchical 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 a list of three objects.

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

```

LIST OF JSON OBJECTS

# JSON Object

The diagram illustrates the relationship between a JSON object and its representation in a web application. On the left, a screenshot of a web browser 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 data. On the right, a screenshot of another web browser shows the raw JSON data for Harry Potter. A red arrow points from the JSON object to the Harry Potter character card, indicating that the JSON data is the source of the character information displayed in the application.

Harry Potter Characters

Harry Potter

Gryffindor

Hermione Granger

Gryffindor

Ron Weasley

Gryffindor

```
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 character's details are obtained from a JSON object passed as an argument to the function.

```
JS app.js > ⚡ showCharacterModal
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 showing the JSON object `character` from the previous code. The object contains properties such as `name`, `species`, `gender`, `house`, `dateOfBirth`, `yearOfBirth`, `ancestry`, `eyeColour`, `hairColour`, `wand`, `patronus`, `hogwartsStudent`, `hogwartsStaff`, `actor`, and `alive`. An arrow points from the `character` variable in the `showCharacter` function to the `character` object in the browser's memory.

```
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>

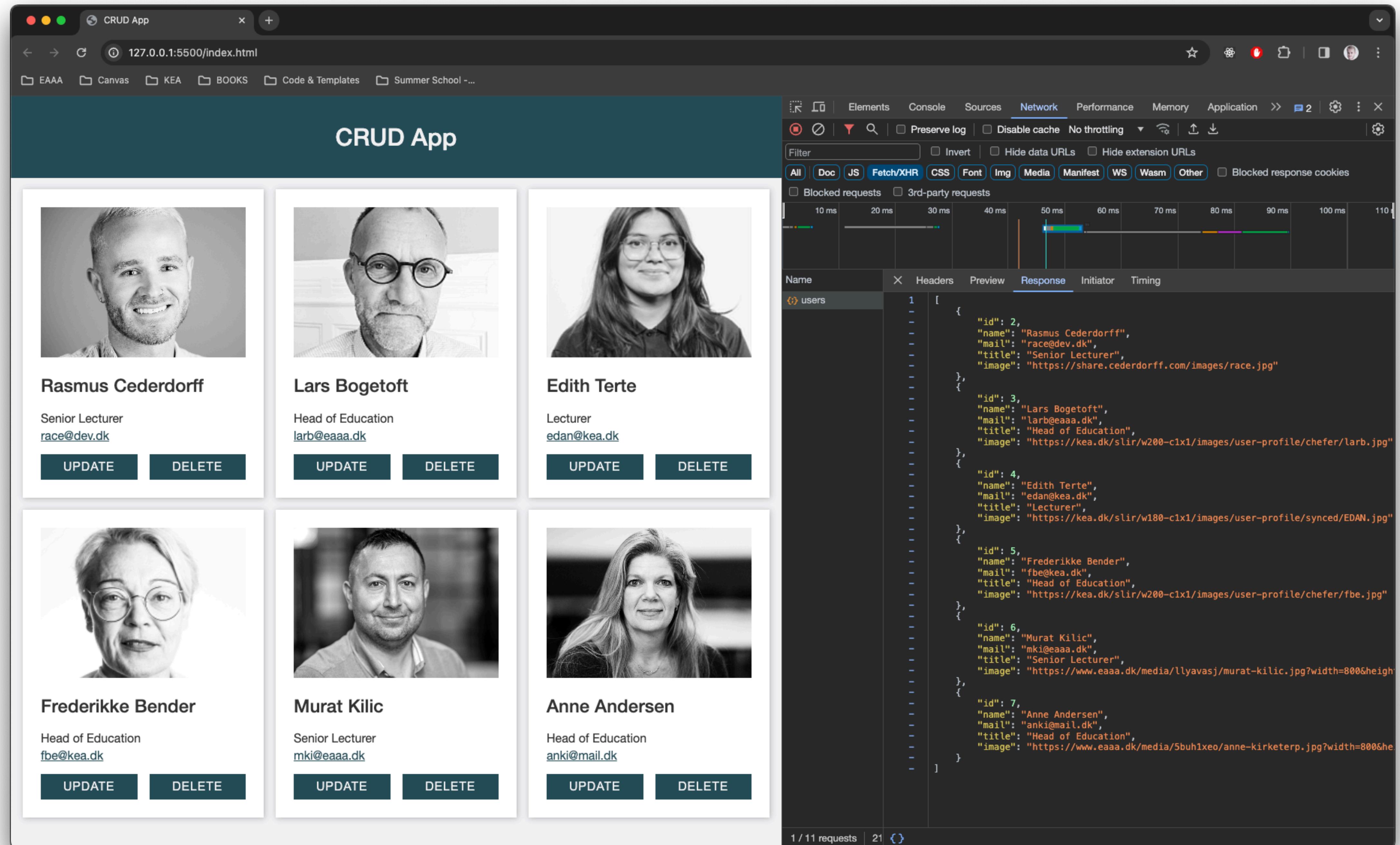
# JAVASCRIPT OBJECT NOTATION

- 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)

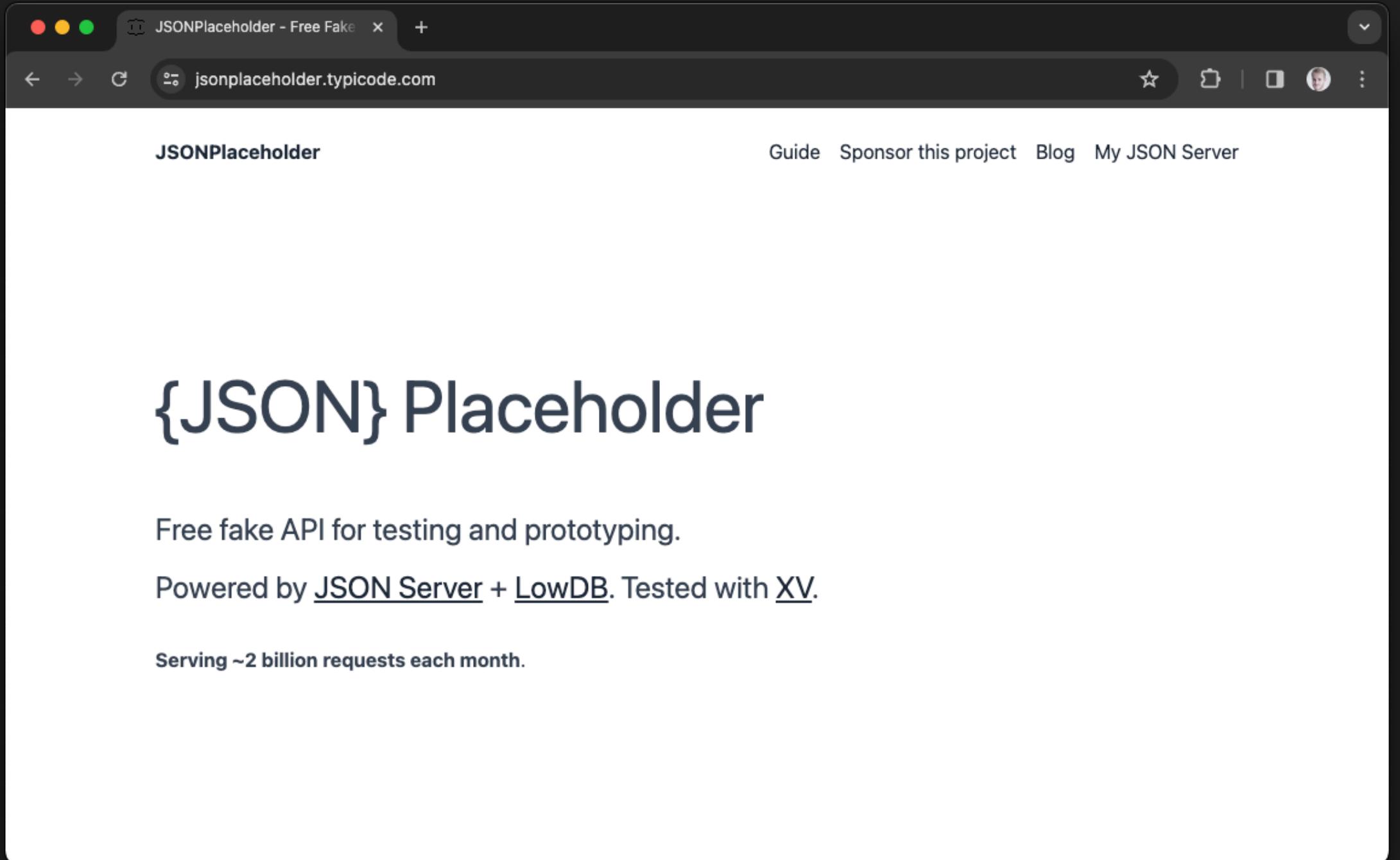
```
[  
 {  
   "id": "1",  
   "firstname": "Kasper",  
   "lastname": "Topp",  
   "age": "34",  
   "haircolor": "Dark Blonde",  
   "countryName": "Denmark",  
   "gender": "Male",  
   "lookingFor": "Female"  
 },  
 {  
   "id": "2",  
   "firstname": "Nicklas",  
   "lastname": "Andersen",  
   "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": "Norway",  
   "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
```



# 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>

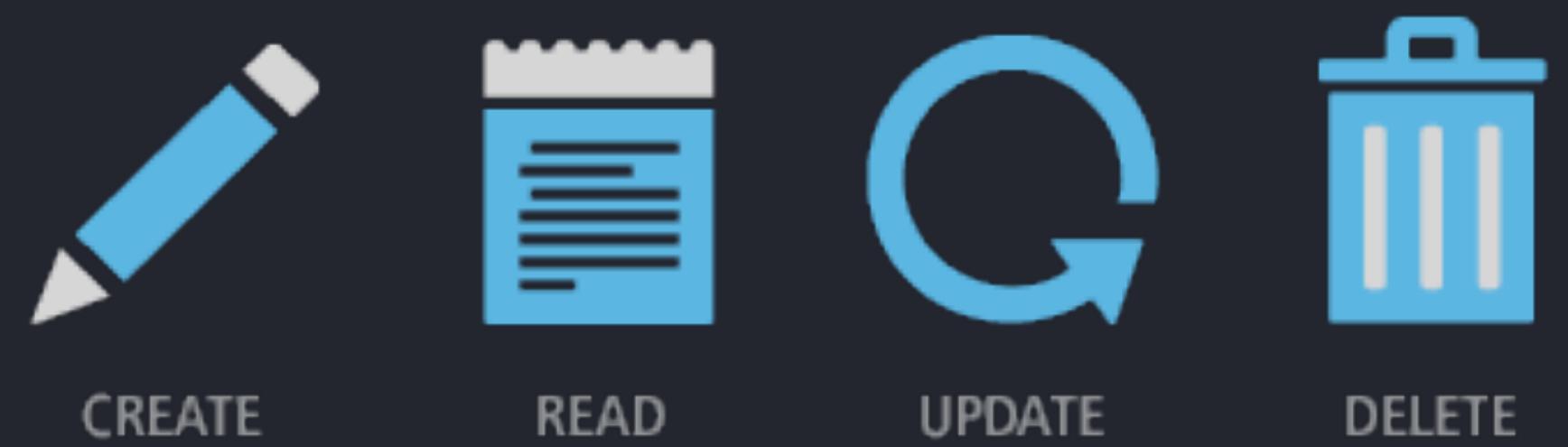
```
{  
  "name": "Rasmus Cederdorff",  
  "birthday": "1990-03-12",  
  "title": "Experienced JavaScript Developer",  
  "experienceYears": 10,  
  "education": {  
    "degree": "Master of Science in IT - Web Communication",  
    "specialization": "Web Architecture"  
  },  
  "skills": ["JavaScript", "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>

# CRUD

Create - Read - Update - Delete



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.

# CRUD

Create - Read - Update - Delete  
Find App Examples

# REST API Design & CRUD



CRUD operations

Create

/api/products **POST**

Read

/api/products **GET**

Update

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

Delete

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

# REST

## Representational State Transfer

# 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

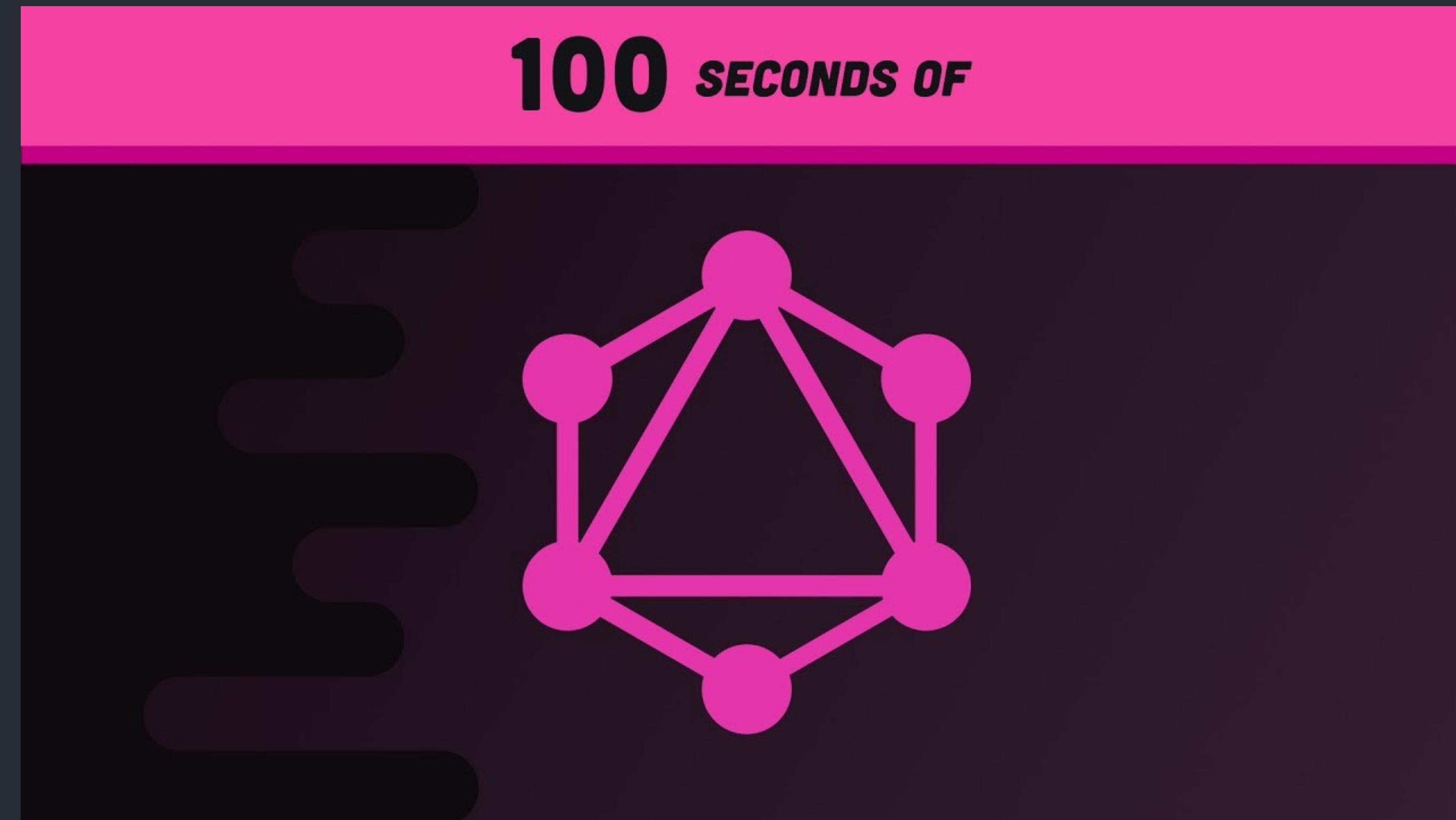
- REpresentational State Transfer
- A standard for systems (client & server) to communicate over HTTP to retrieve or modify (data) resources.
- Stateless, meaning the two systems don't need to know anything about the state.
- The client makes the requests using the 4 basic HTTP verbs to define the operation.

# 100 *SECONDS OF* node



[https://www.youtube.com/watch?v=-MTSQjw5DrM&ab\\_channel=Fireship](https://www.youtube.com/watch?v=-MTSQjw5DrM&ab_channel=Fireship)

# GraphQL - REST Alternative



[https://www.youtube.com/watch?v=elQh02xuVw4&ab\\_channel=Fireship](https://www.youtube.com/watch?v=elQh02xuVw4&ab_channel=Fireship)

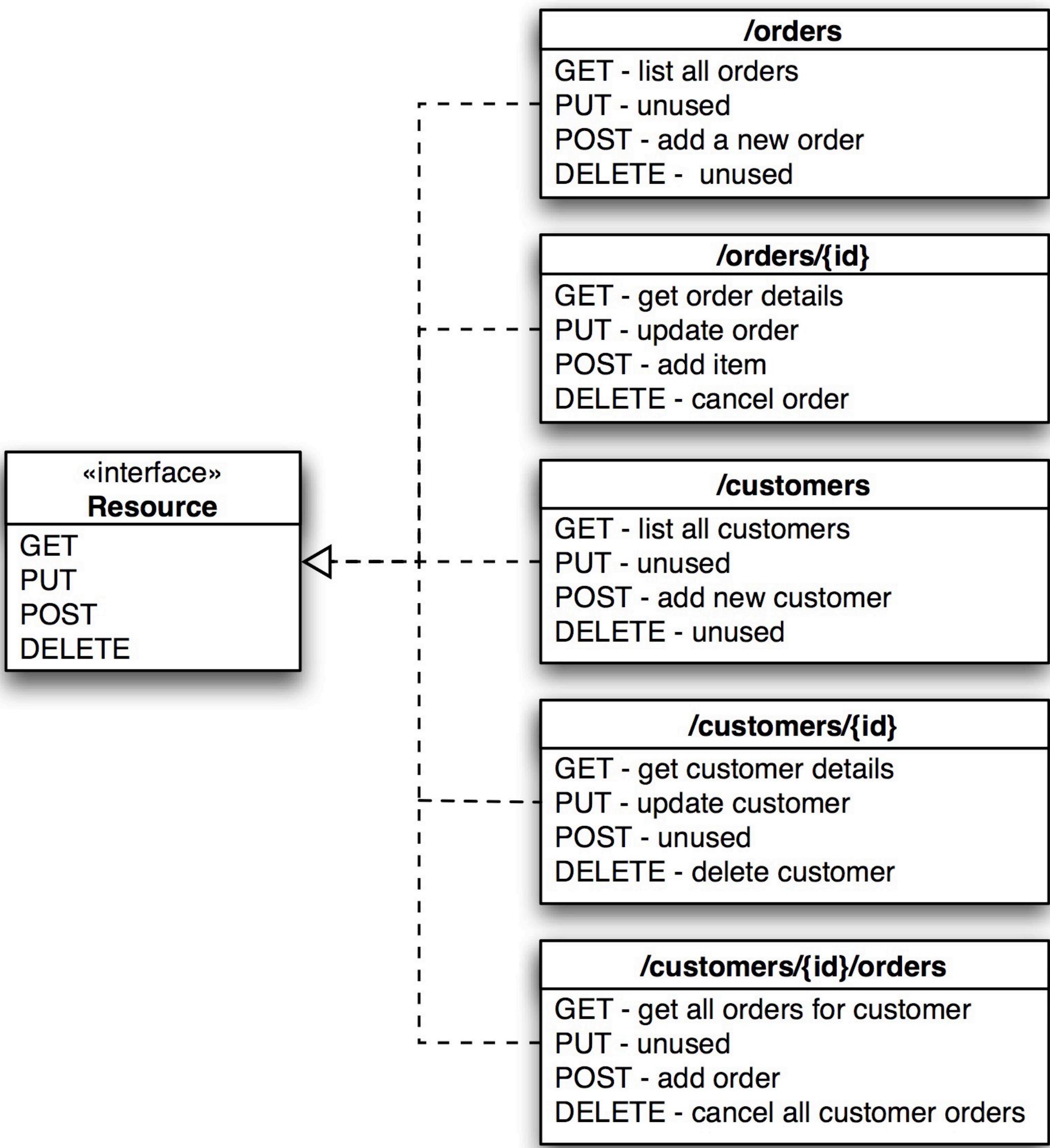
# What are REST APIs?

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

# REpresentational State Transfer

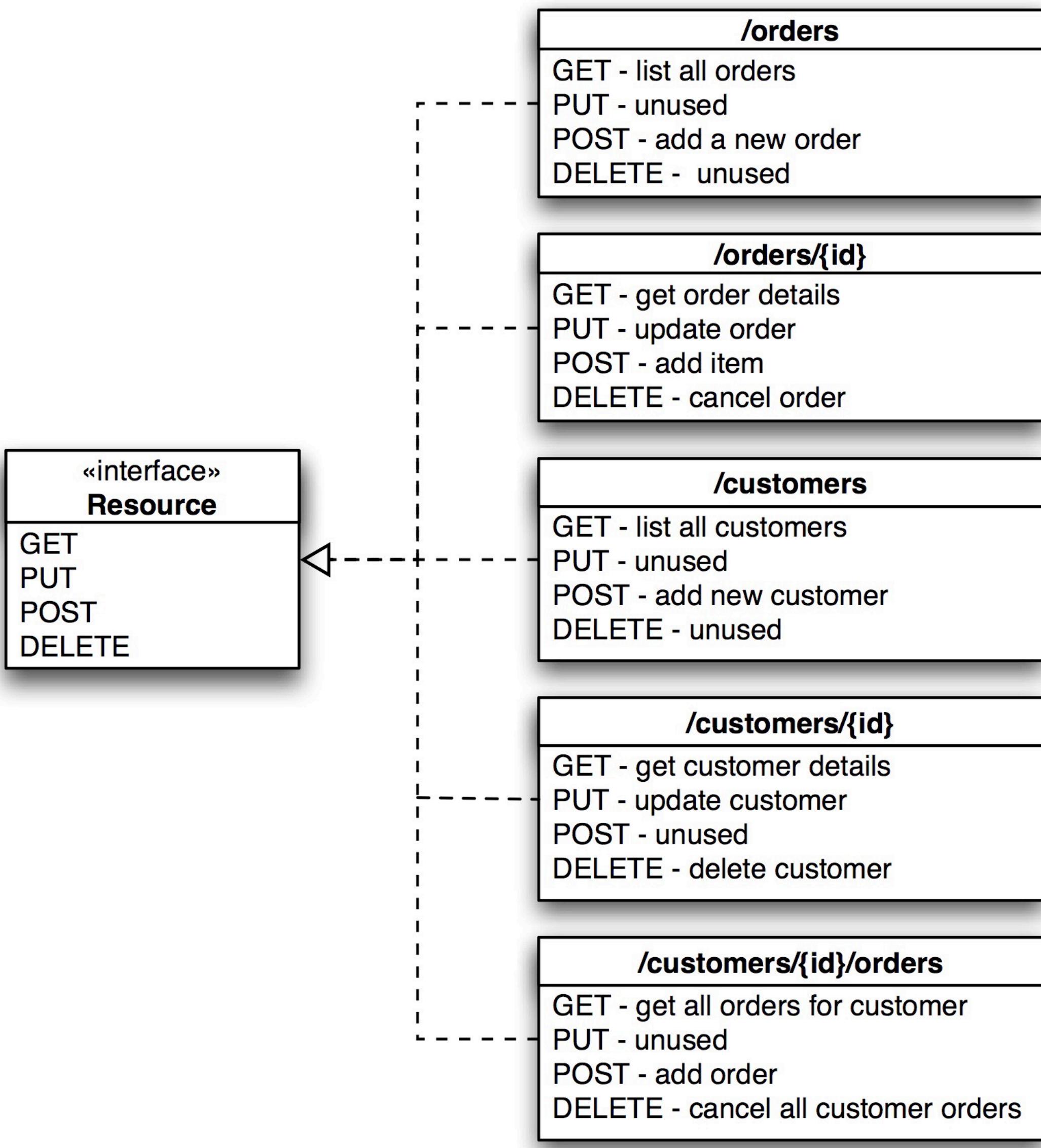
- Clean, semantic URL:
  - <http://example.com/products/2/25> instead of:
  - [http://example.com/products?  
category=2&id=25](http://example.com/products?category=2&id=25)
- Basic HTTP requests to perform create, read, update and delete with the HTTP methods GET, POST, PUT and DELETE

# Principles of REST



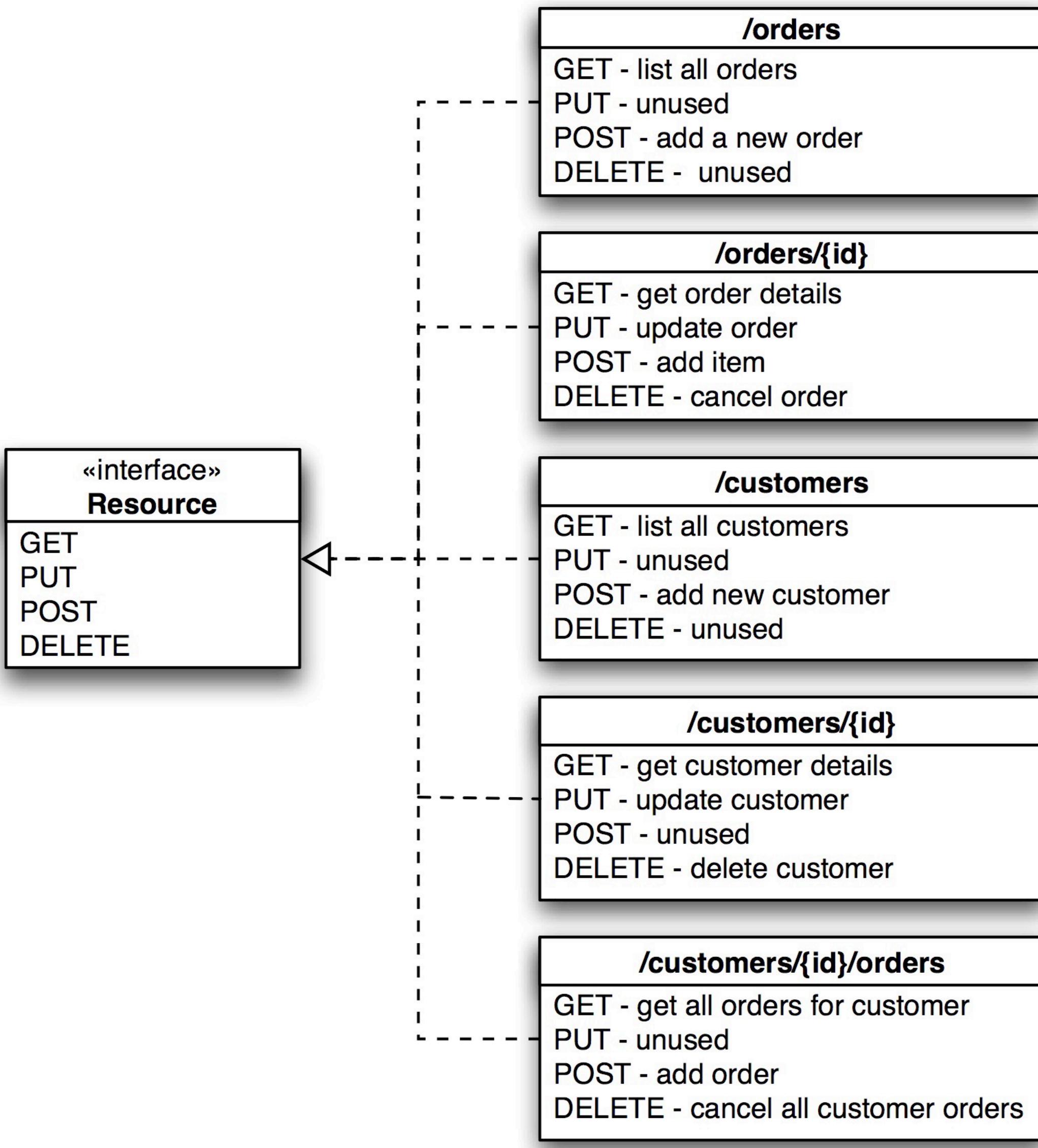
- REST defines a set of rules and guidelines on how you can interact with an API:
  - With REST, each piece of data in the database is treated as a resource. It has a unique id and URL. The URL structure represents the hierarchy of the data, allowing you to access specific data nodes.
  - You can use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

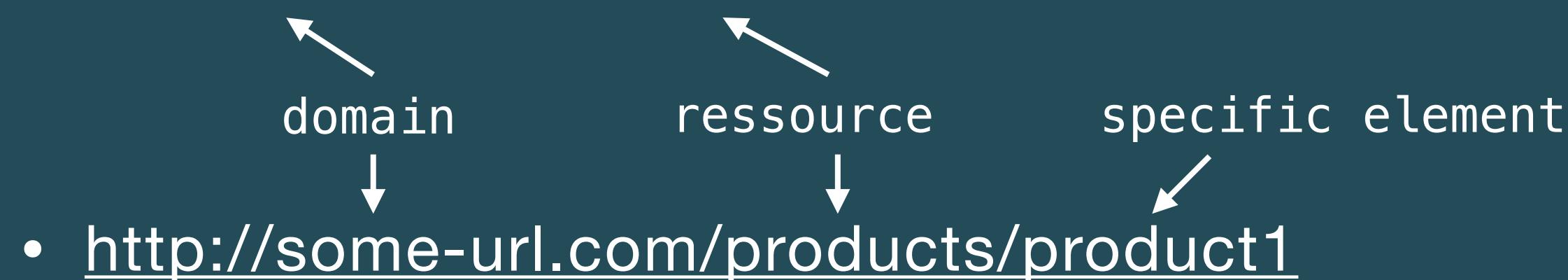
# Principles of REST



- 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

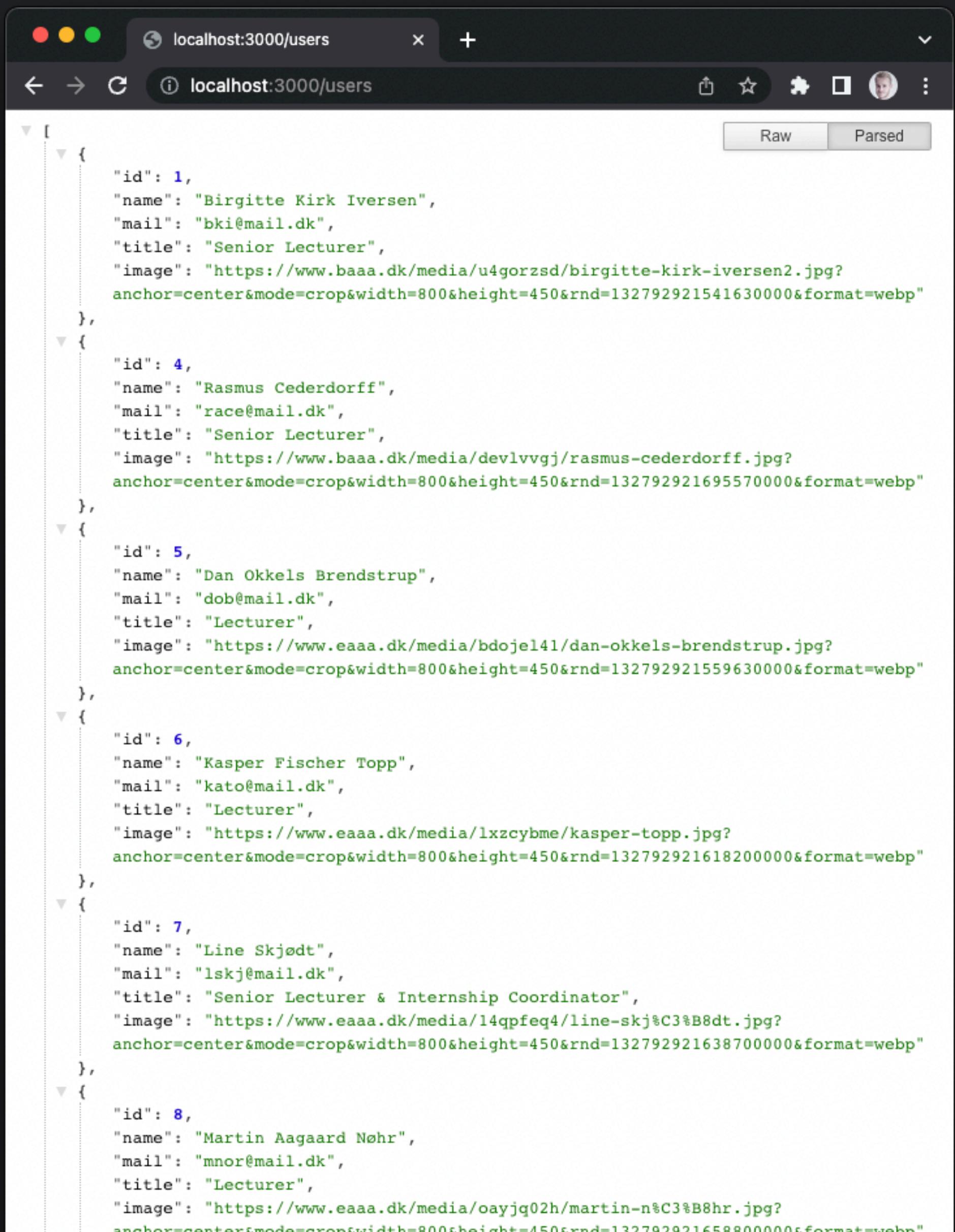
- Base URL: http://some-url.com/products



- 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

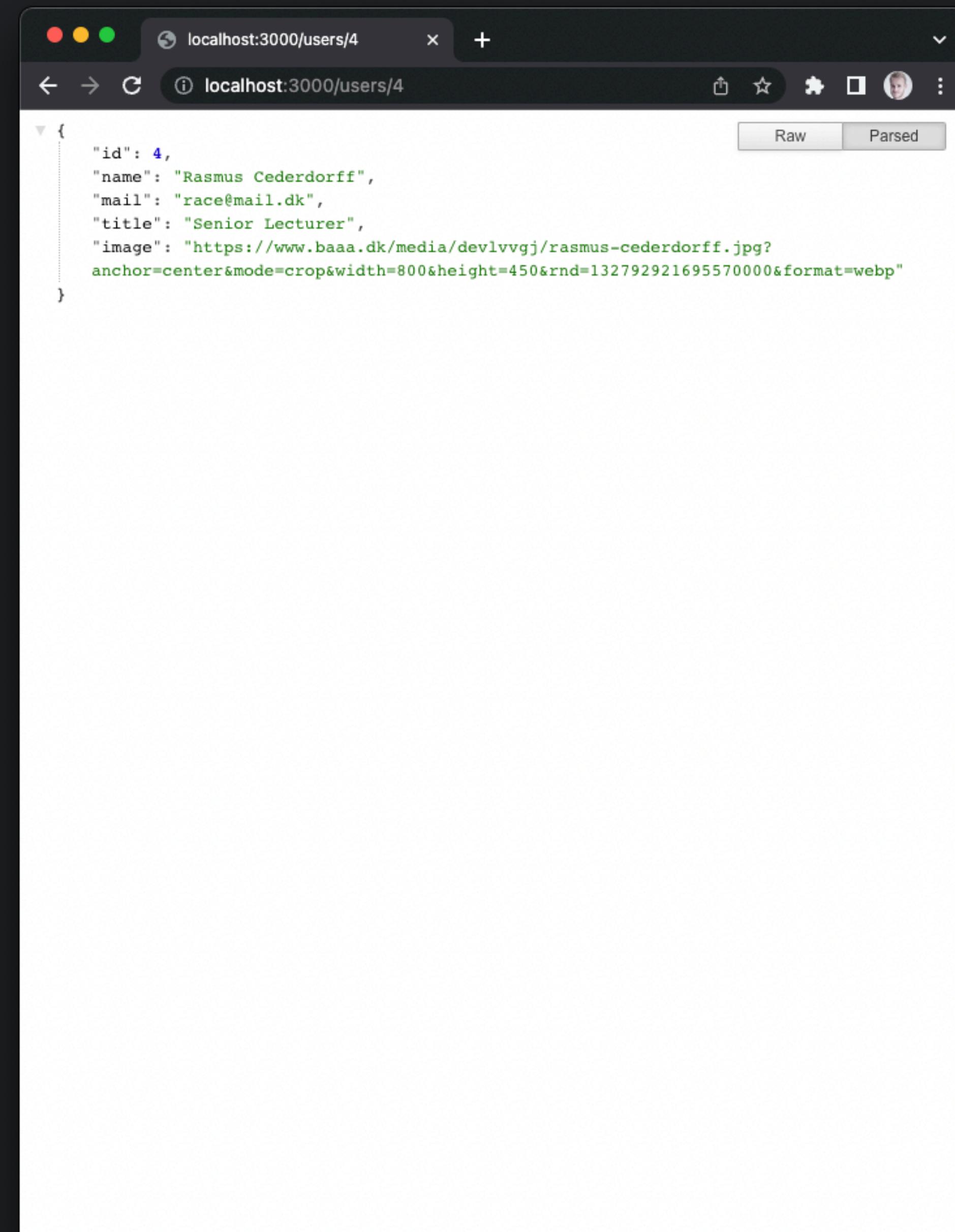
# Collections (JSON Array)



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?



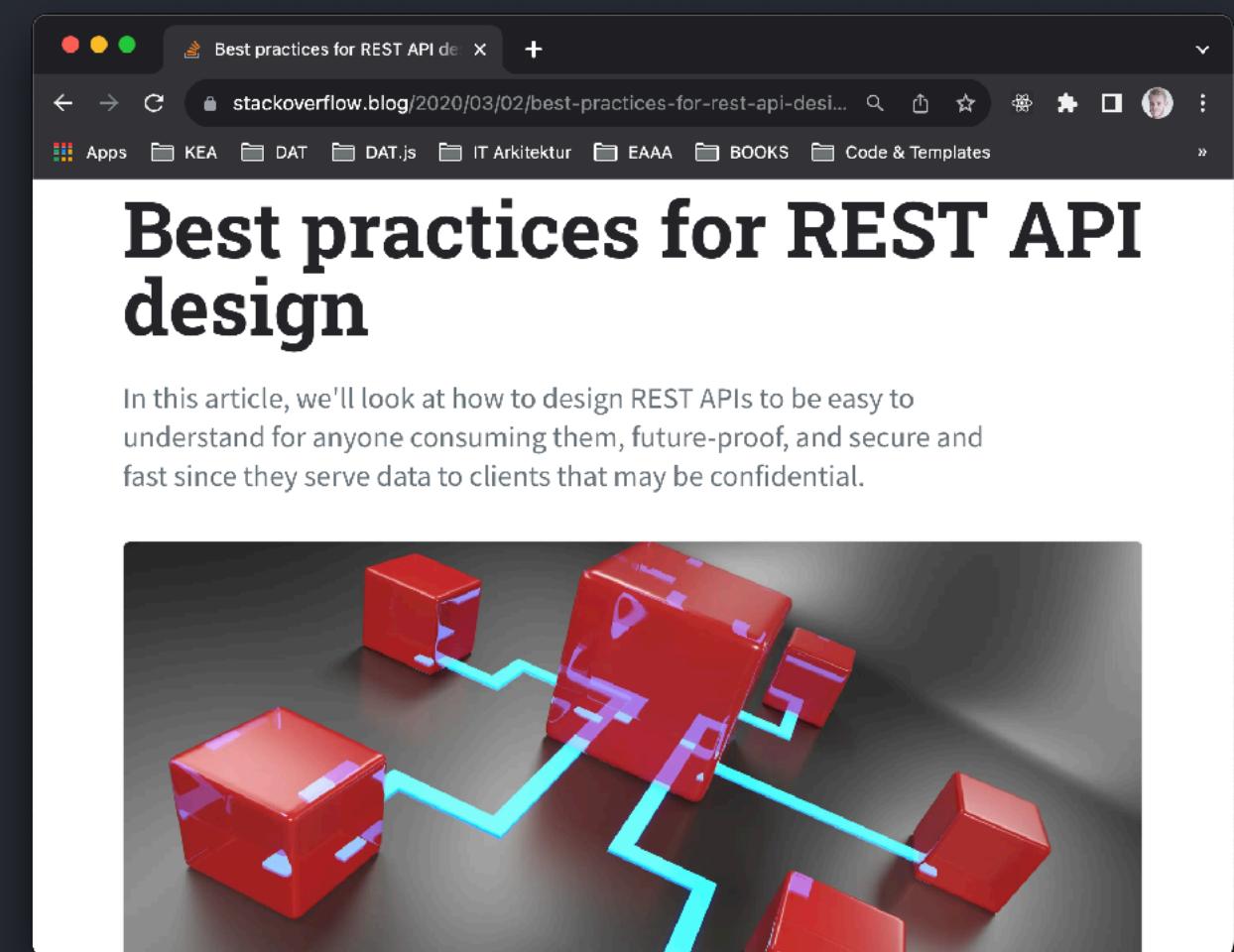
<https://dataforsyningen.dk/>

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)). The main content is divided into sections: "Eksempel på kald" (Example of call), "Regioner" (Regions), and "en region" (a region). Under "Eksempel på kald", there are examples for "Danmarks adresser og vejnavne" (Danish addresses and street names) and "Regioner". Under "Regioner", there is a link to "alle regioner" (all regions) with a GET method and the URL <https://api.dataforsyningen.dk/regioner>. Under "en region", there is a link to "Regioner" with a GET method and the URL <https://api.dataforsyningen.dk/regioner/1082>.

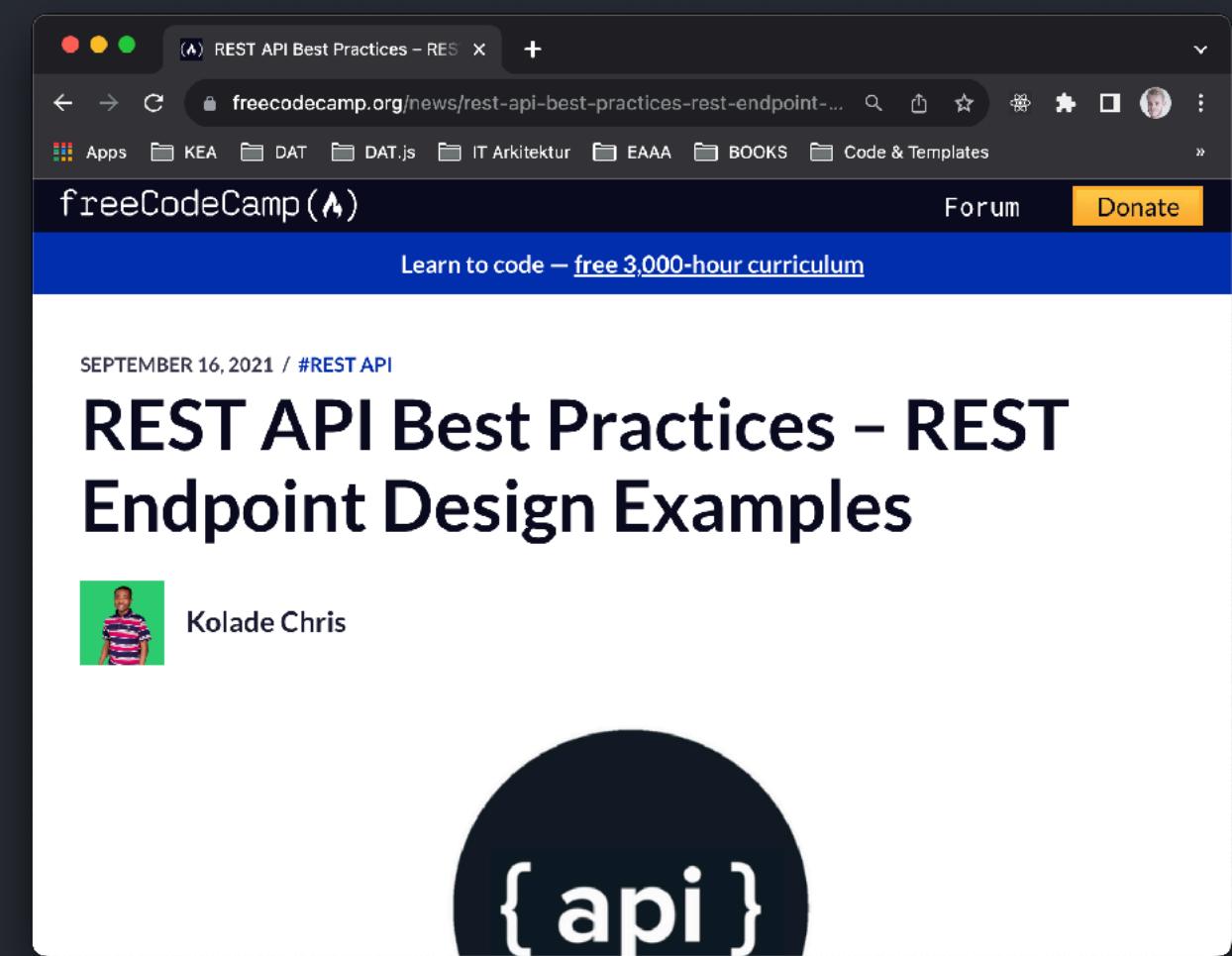
[Dataforsyningen REST API](#)

# 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



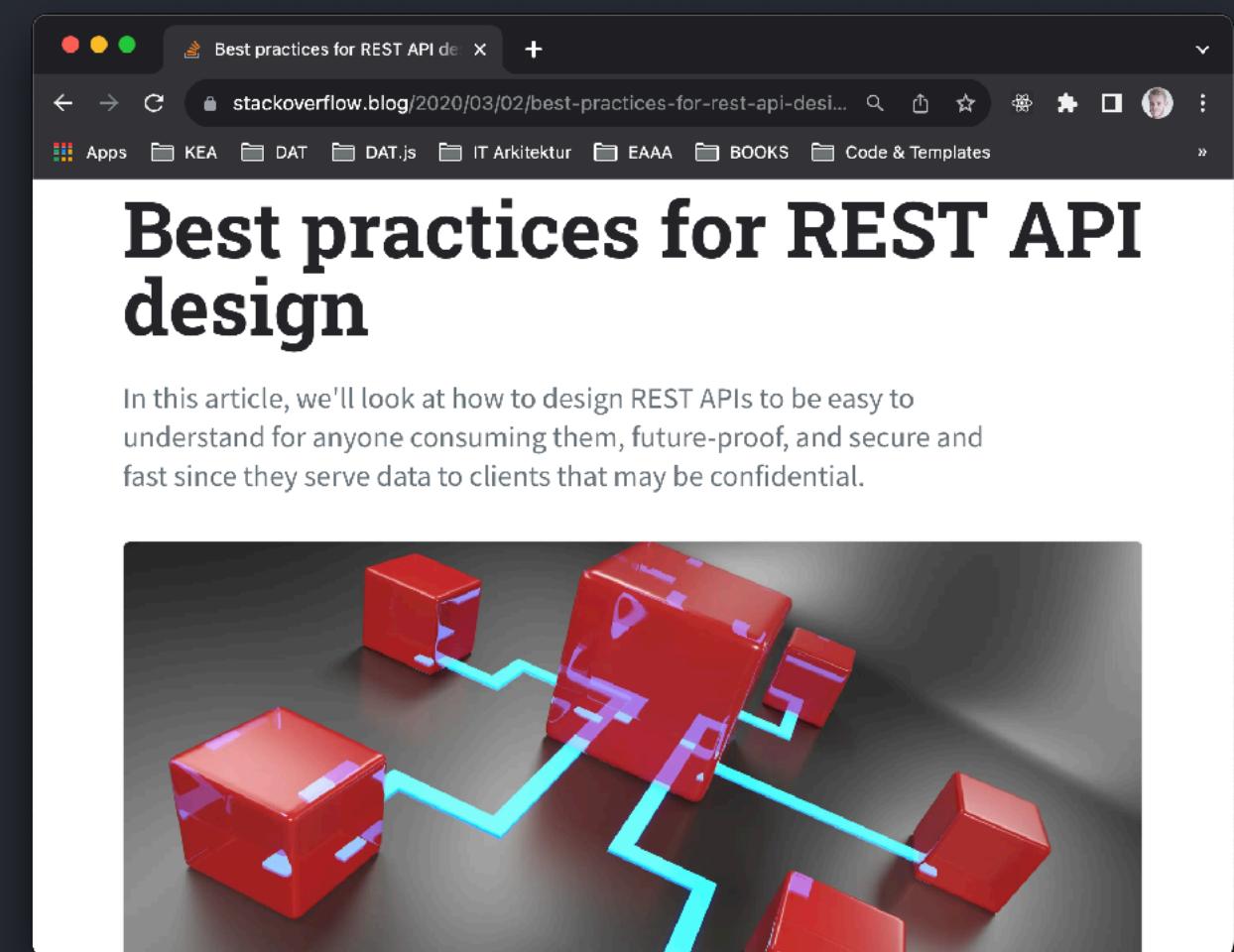
<https://stackoverflow.blog/2020/03/02/best-practices-for-rest-api-design/#h-cache-data-to-improve-performance>



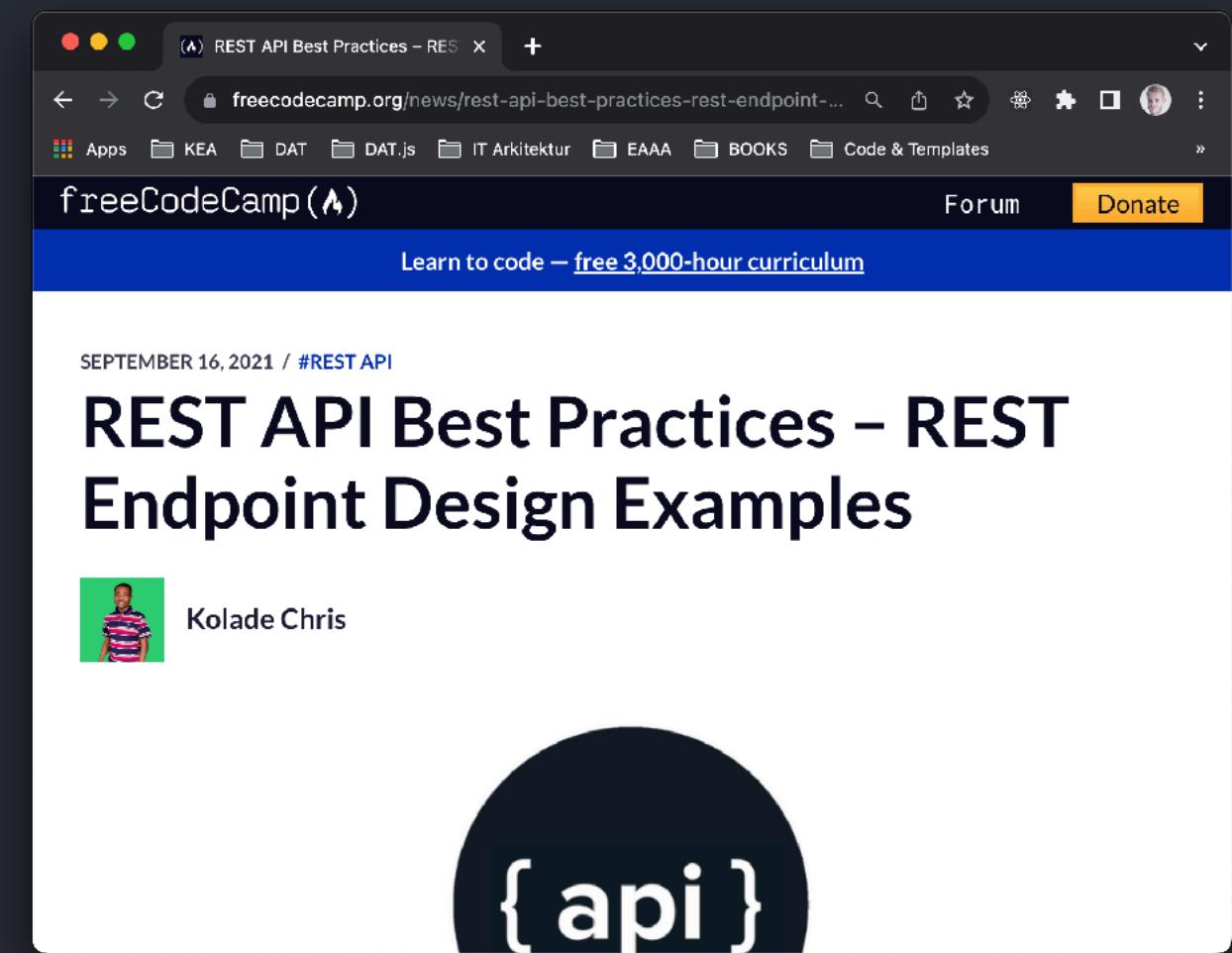
<https://www.freecodecamp.org/news/rest-api-best-practices-rest-endpoint-design-examples/>

# 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](#)

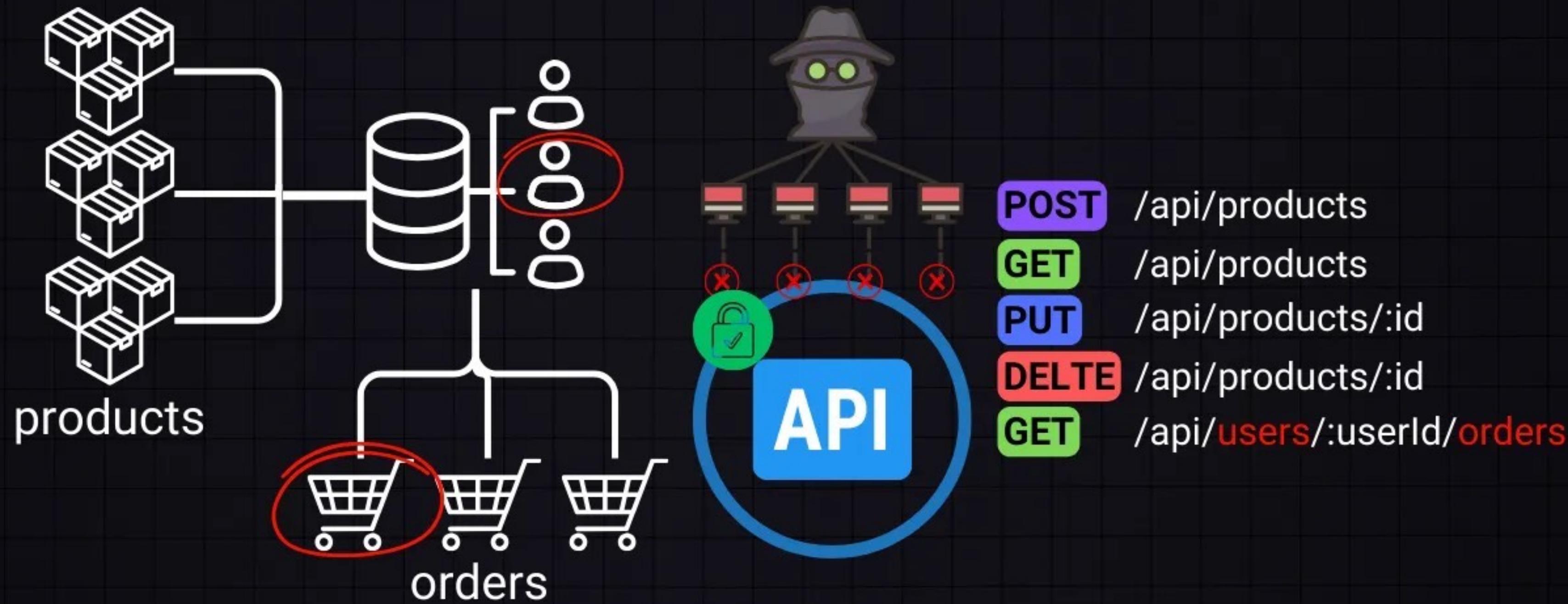


<https://stackoverflow.blog/2020/03/02/best-practices-for-rest-api-design/#h-cache-data-to-improve-performance>



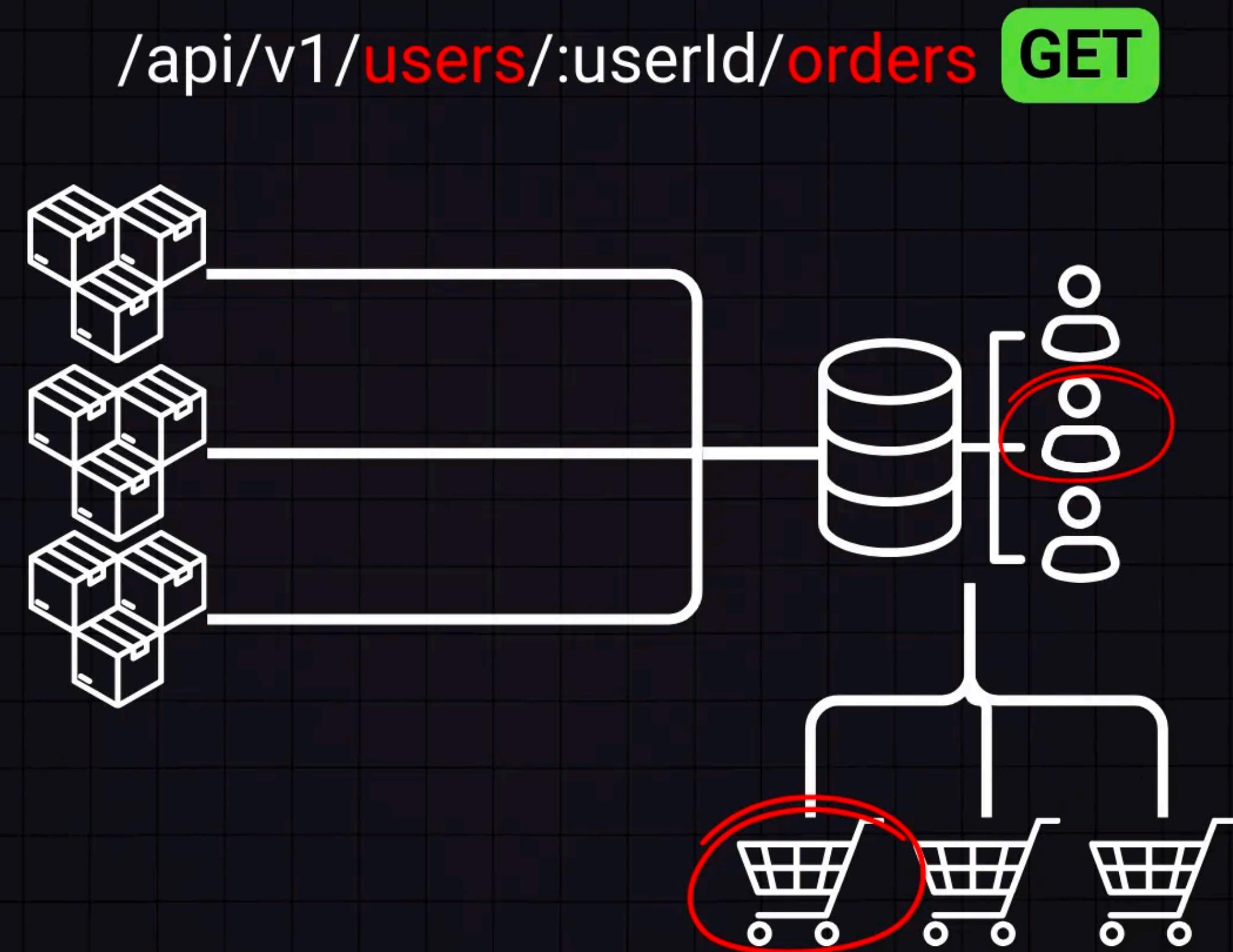
<https://www.freecodecamp.org/news/rest-api-best-practices-rest-endpoint-design-examples/>

# API DESIGN



API Design 101: From Basics to Best Practices

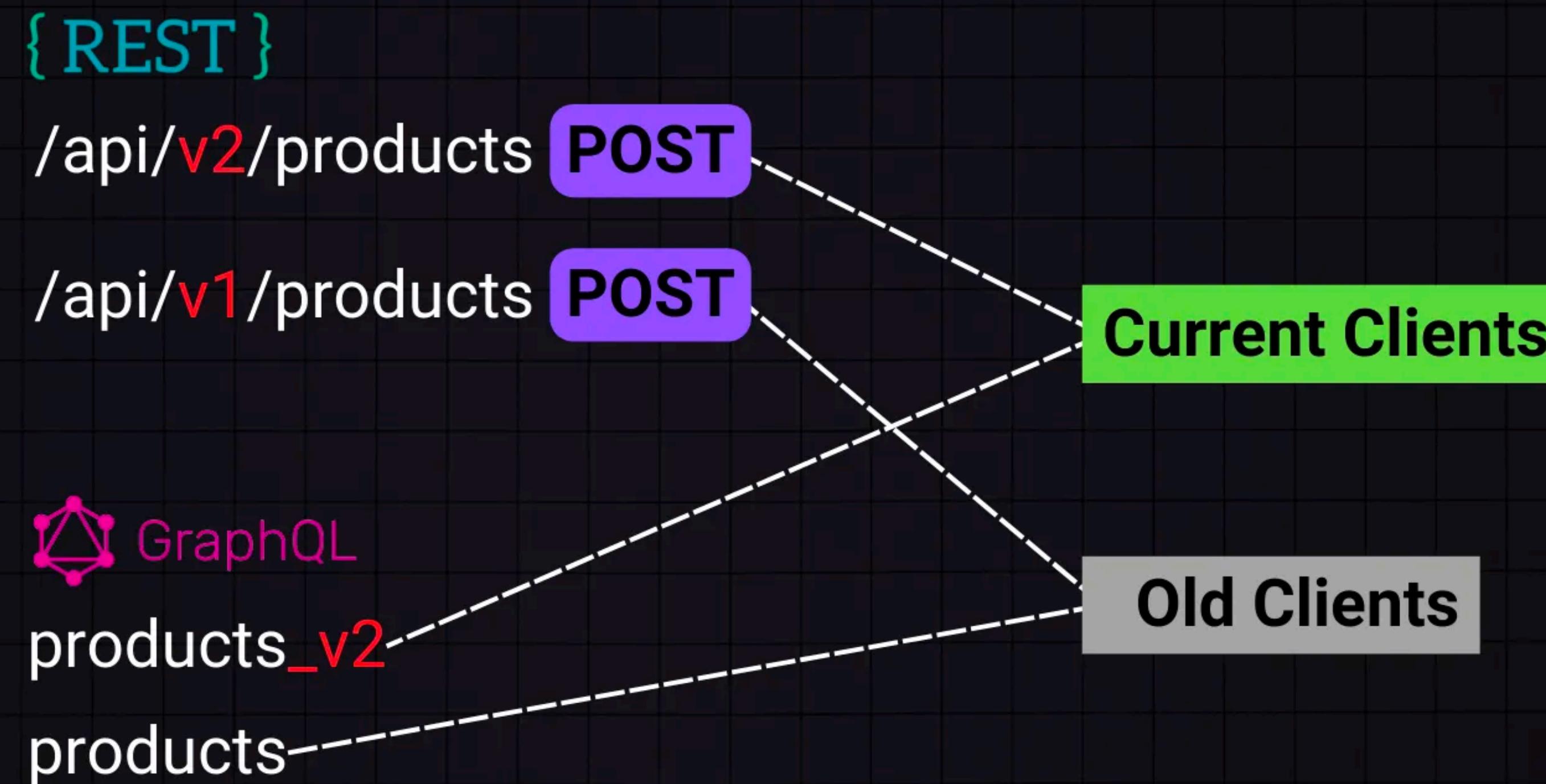
# Relationships in API Design



# 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 Requests

GET - POST - PUT - DELETE

# 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://www.w3schools.com/tags/ref_httpmethods.asp)

<https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods>

# CRUD vs REST & HTTP Verbs

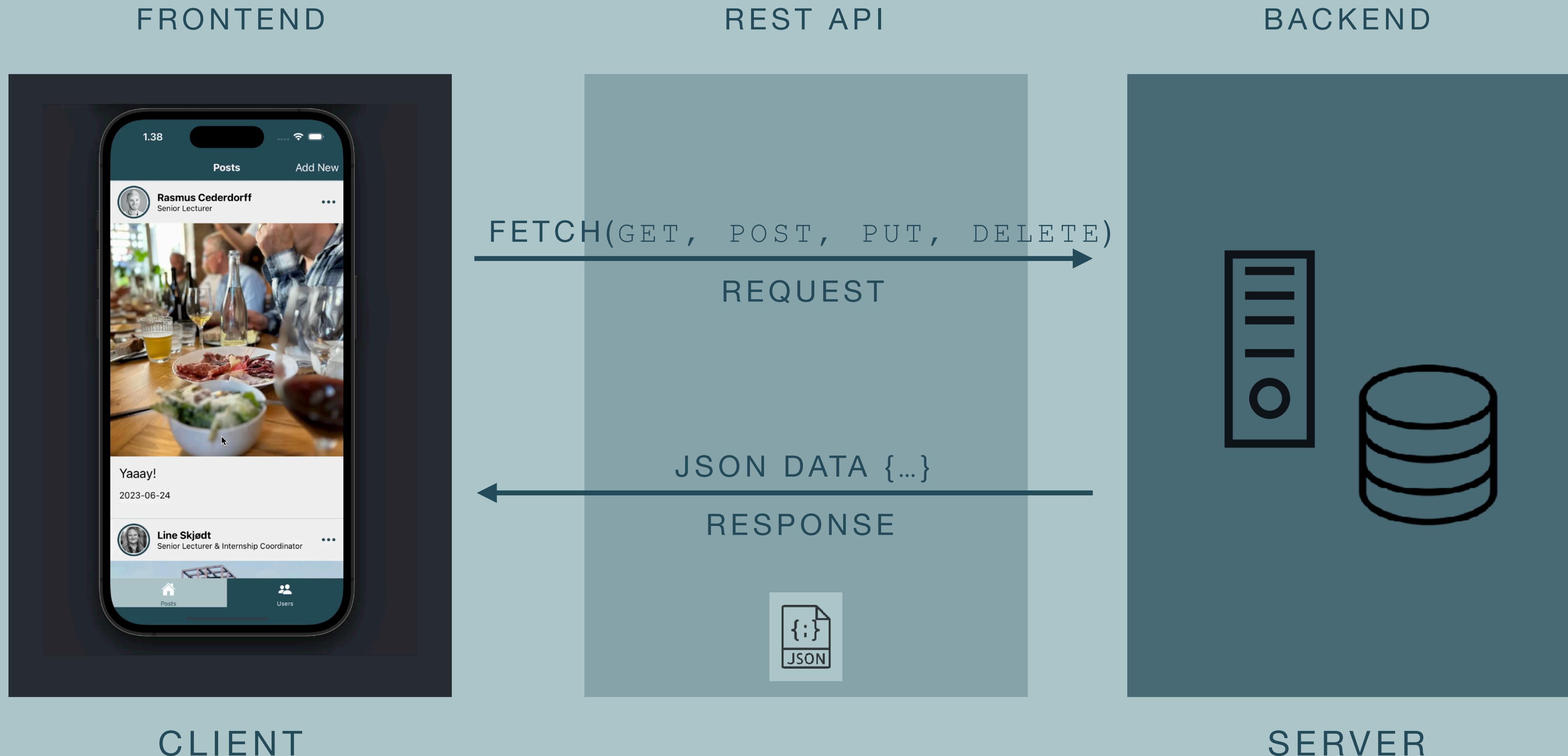
**CREATE** -> POST: create a new resource (object)

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

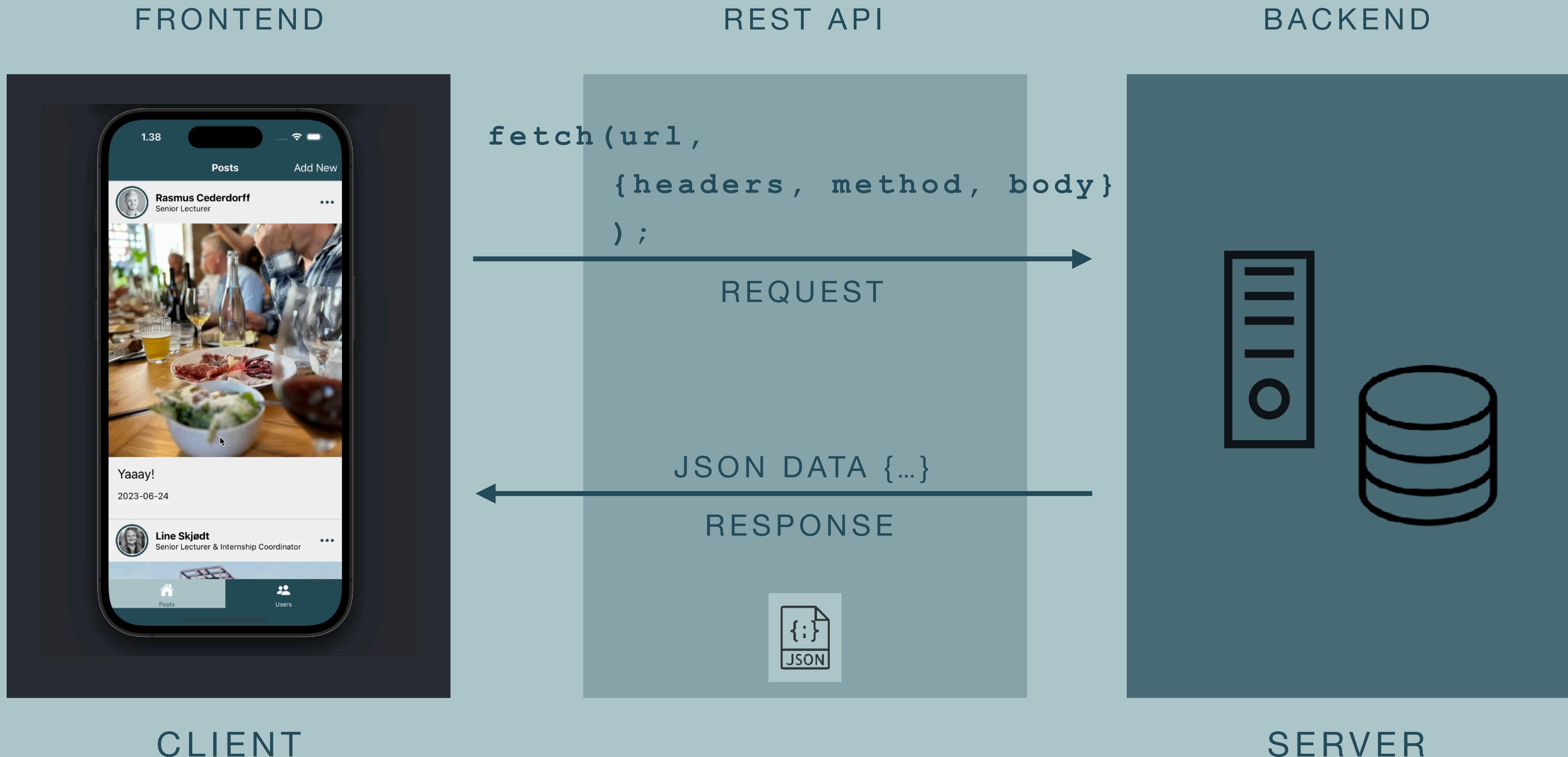
**UPDATE** -> PUT: update a specific resource (by id)

**DELETE** -> DELETE: remove a specific resource by id

# HTTP Request & Response



# HTTP Request & Response

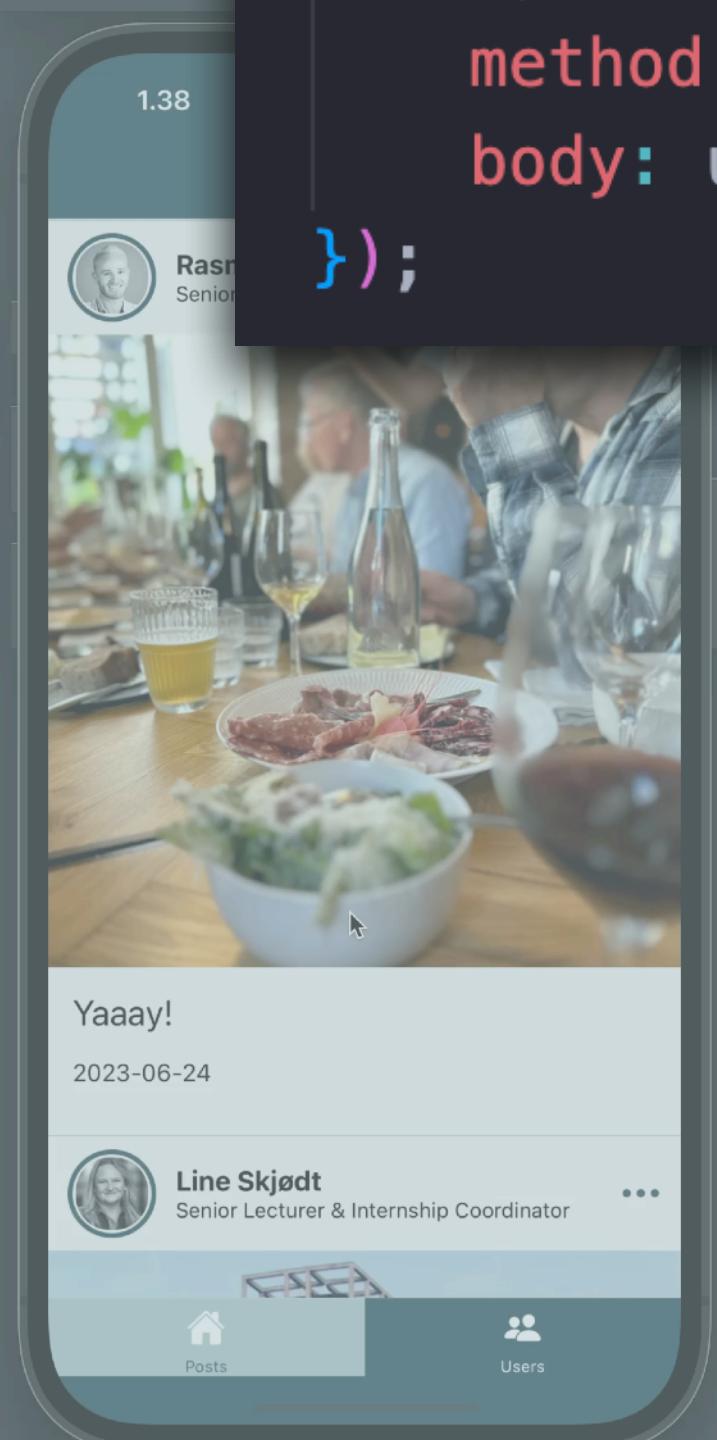


# HTTP Requests & Response

```
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
});
```

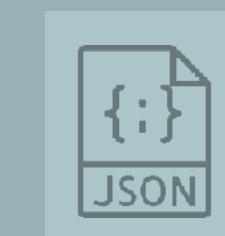
FR



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](https://developer.mozilla.org/en-US/docs/Glossary/Request_header)

# REQUEST headers

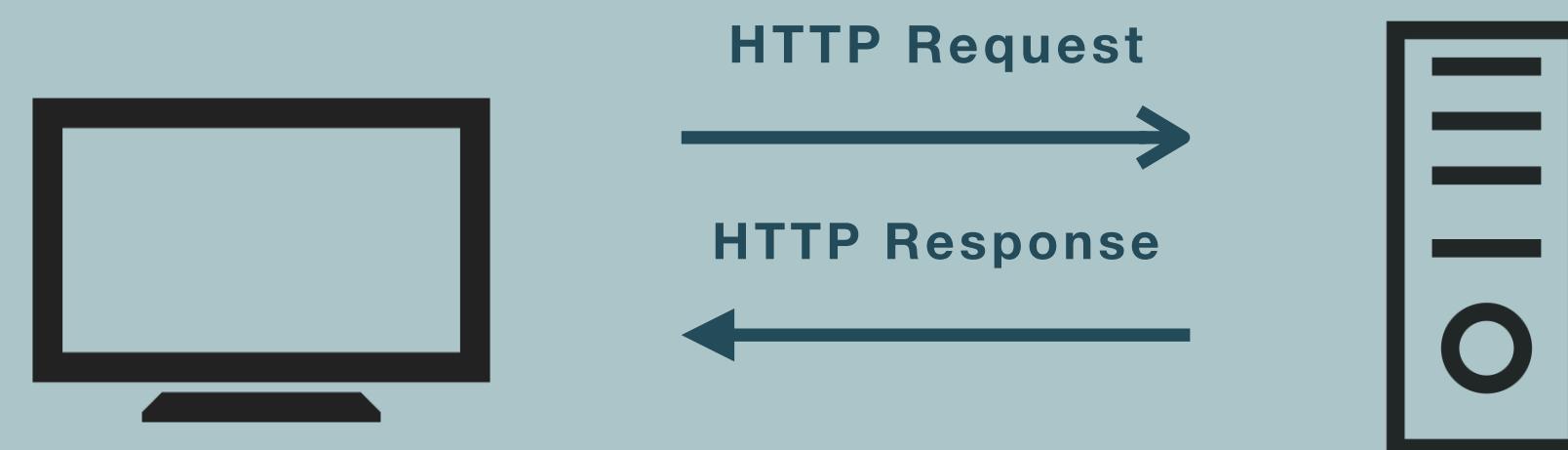
```
const response = await fetch("http://localhost:3333/users", {
  headers: {
    "Content-Type": "application/json"
  },
  method: "POST",
  body: userAsJson
});
```

```
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"
});
```

# JSON to Figma



Plugin: <https://www.figma.com/community/plugin/789839703871161985>

Video: [https://www.youtube.com/watch?v=1qk3AbTQfYQ&ab\\_channel=Figma55](https://www.youtube.com/watch?v=1qk3AbTQfYQ&ab_channel=Figma55)

# Integrate JSON data in Figma

<https://race.notion.site/Integrate-JSON-data-in-Figma-eddb262bac8148c1a46761f20759b498>

The screenshot shows the "JSON to Figma" extension interface. On the left, a JSON array is displayed:

```
[{"name": "Cecil Mccallum", "job": "Operations", "email": "cecil@mail.com"}, {"name": "Romy Brandt", "job": "Lead of Engineering", "email": "romy@mail.com"}, {"name": "Iona Cherry", "job": "Developer", "email": "iona@mail.com"}, {"name": "Robert Curran", "job": "Management", "email": "robert@mail.com"}, {"name": "Elias Bains", "job": "Software Engineering", "email": "elias@mail.com"}]
```

The middle section is a configuration panel titled "JSON to Figma" with a "JSON" tab selected. It lists fields with checkboxes:

- email (checked)
- id (checked)
- email (checked)
- address
  - street (checked)
  - suite (unchecked)
  - geo (unchecked)
  - city (unchecked)
  - username (unchecked)

At the bottom are two buttons: "Populate selected 4" and "Randomise order".

On the right, three Figma components are shown, each corresponding to one of the JSON objects:

- Cecil Mccallum**: Operations, cecil@mail.com
- Romy Brandt**: Lead of Engineering, romy@mail.com
- Iona Cherry**: Developer, iona@mail.com

Each component includes a small profile picture and the field names (name, job, email) with their respective values.