

tt()



Tl;dr explain basic API
architecture, async
coding and fetching

Schedule

1. Sync vs Async
2. Async in JavaScript
3. Data formats
4. API Architecture (public, keys, oAuth)
5. Client vs Server fetch
6. What's next?



Sync vs Async

(non-blocking)

Sync vs async

why?

Getting data from a resource (API) takes time. It needs to fetch the resource, parse it etc. But also, what if the data isn't available (no internet connection e.g.) how should errors be handled?

Sync vs async

Only one thing can happen at a time, on a single main thread, and everything else is blocked until an operation completes.

SYNCHRONOUS LOAD

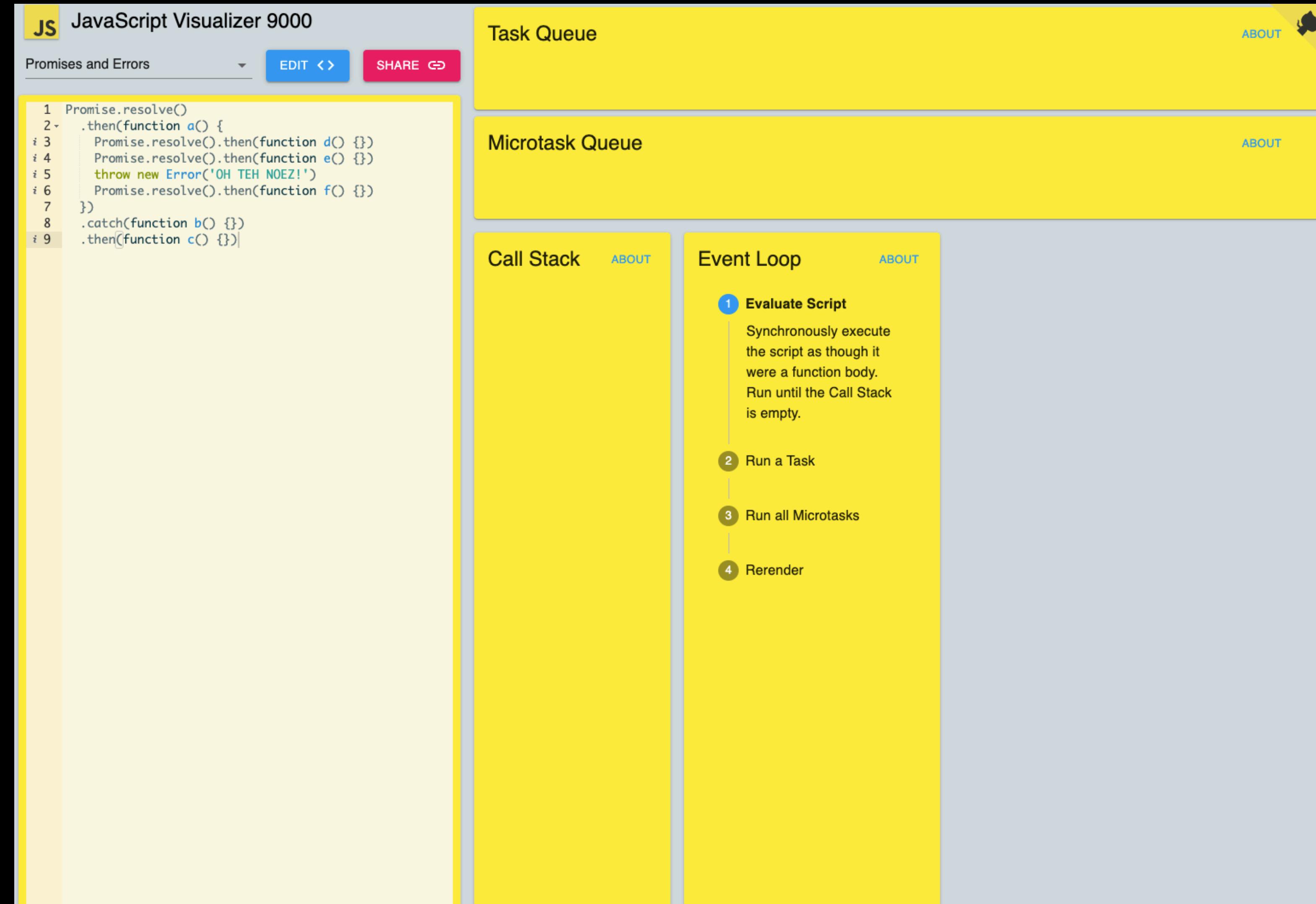


ASYNCHRONOUS LOAD



Sync vs async

Many Web API features now use **asynchronous code to run**, especially those that access or *fetch some kind of resource from an external device*, such as fetching a file from the network, accessing a database and returning data from it.



<https://www.jsv9000.app/>

Async in JavaScript

(Pasta promises)

Async in JavaScript

- ❖ **Callbacks**
- ❖ **Promises**
- ❖ **Async / Await**

Fetching in JavaScript

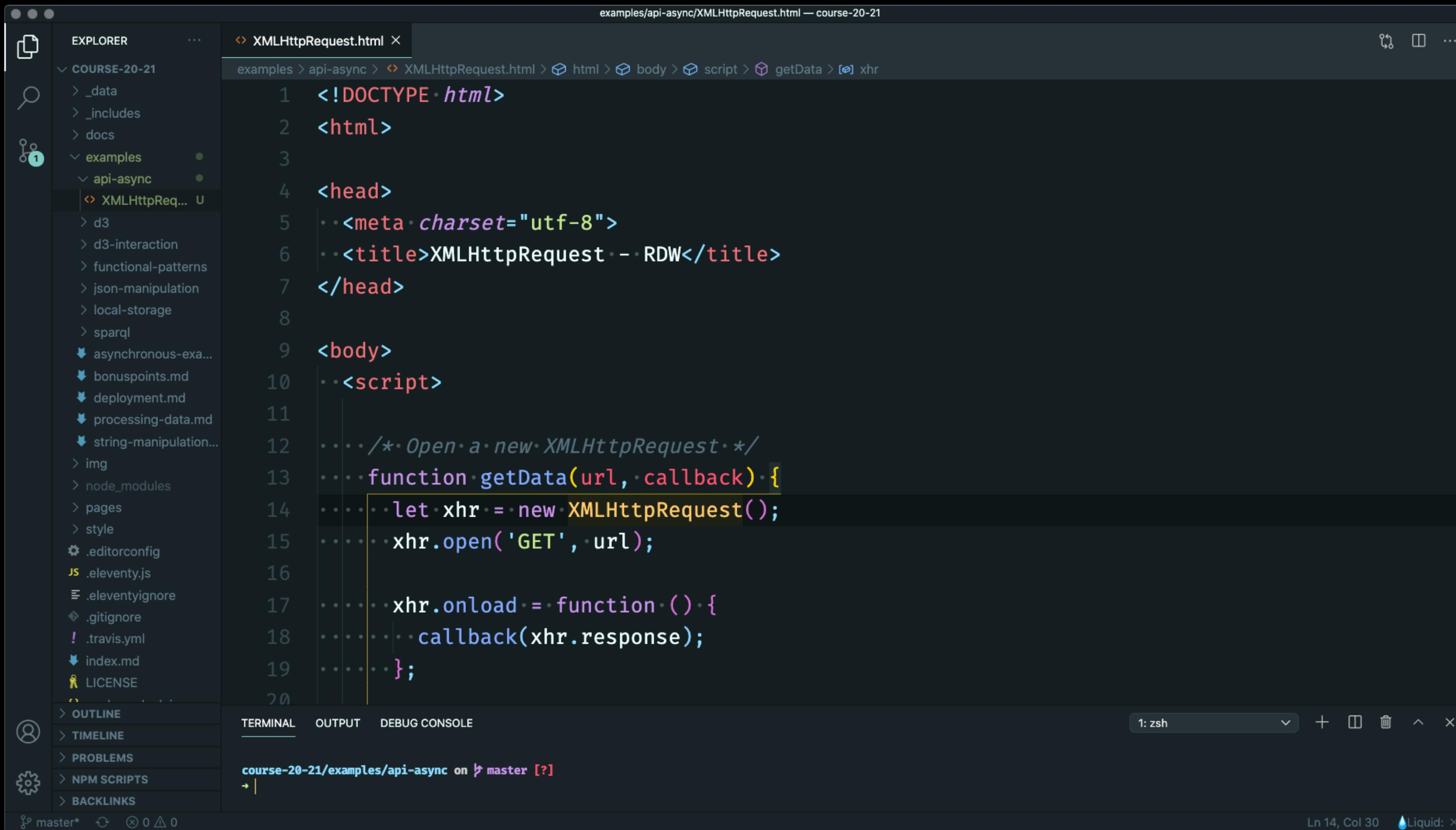
- ❖ **Callbacks (XMLHttpRequest)**
- ❖ **Promises (Fetch)**
- ❖ **Async / Await (Fetch)**

Callbacks

definition
on

Async callbacks are functions that are specified as arguments when calling a function which *will start executing code in the background. When the background code finishes running, it calls the callback function.*

Callbacks



A screenshot of a code editor (Visual Studio Code) displaying an example of XMLHttpRequest usage. The file is named XMLHttpRequest.html and is located in the examples/api-async directory of a project named course-20-21.

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>XMLHttpRequest -- RDW</title>
</head>
<body>
<script>
    /* Open a new XMLHttpRequest */
    function getData(url, callback) {
        let xhr = new XMLHttpRequest();
        xhr.open('GET', url);
        xhr.onload = function() {
            callback(xhr.response);
        };
    }
</script>

```

The code demonstrates the use of XMLHttpRequest to make a GET request to a specified URL and execute a callback function with the response data.

XMLHttpRequest Example

Promises

The Promise object represents the eventual completion (or failure) of **an asynchronous operation and its resulting value.**

Promises

A **promise can only succeed or fail once**. It cannot succeed or fail twice, neither can it switch from success to failure or vice versa.

A pending promise can either be **fulfilled with a value or rejected with a reason (error)**.

Promises

A common need is to execute two or more asynchronous operations back to back, where each subsequent operation starts when the previous operation succeeds, with the result from the previous step. We accomplish this by creating a promise chain.

Promises versus

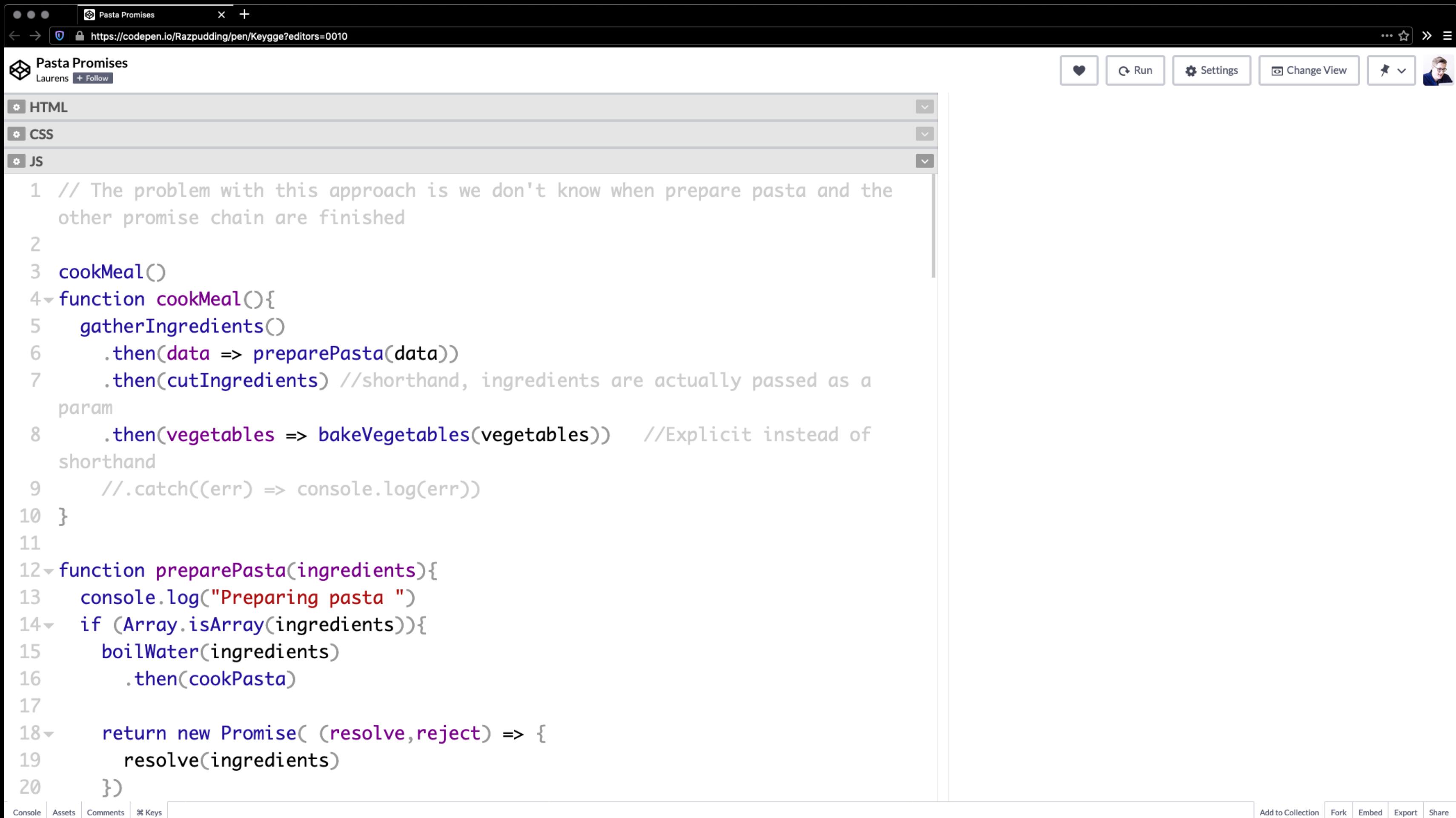
Callback 'hell'
(christmas tree)

```
1 doSomething(function(result) {
2   doSomethingElse(result, function(newResult) {
3     doThirdThing(newResult, function(finalResult) {
4       console.log('Got the final result: ' + finalResult);
5     }, failureCallback);
6   }, failureCallback);
7 }, failureCallback);
```

```
1 doSomething()
2 .then(function(result) {
3   return doSomethingElse(result);
4 })
5 .then(function(newResult) {
6   return doThirdThing(newResult);
7 })
8 .then(function(finalResult) {
9   console.log('Got the final result: ' + finalResult);
10 })
11 .catch(failureCallback);
```

Promise chain

Promises



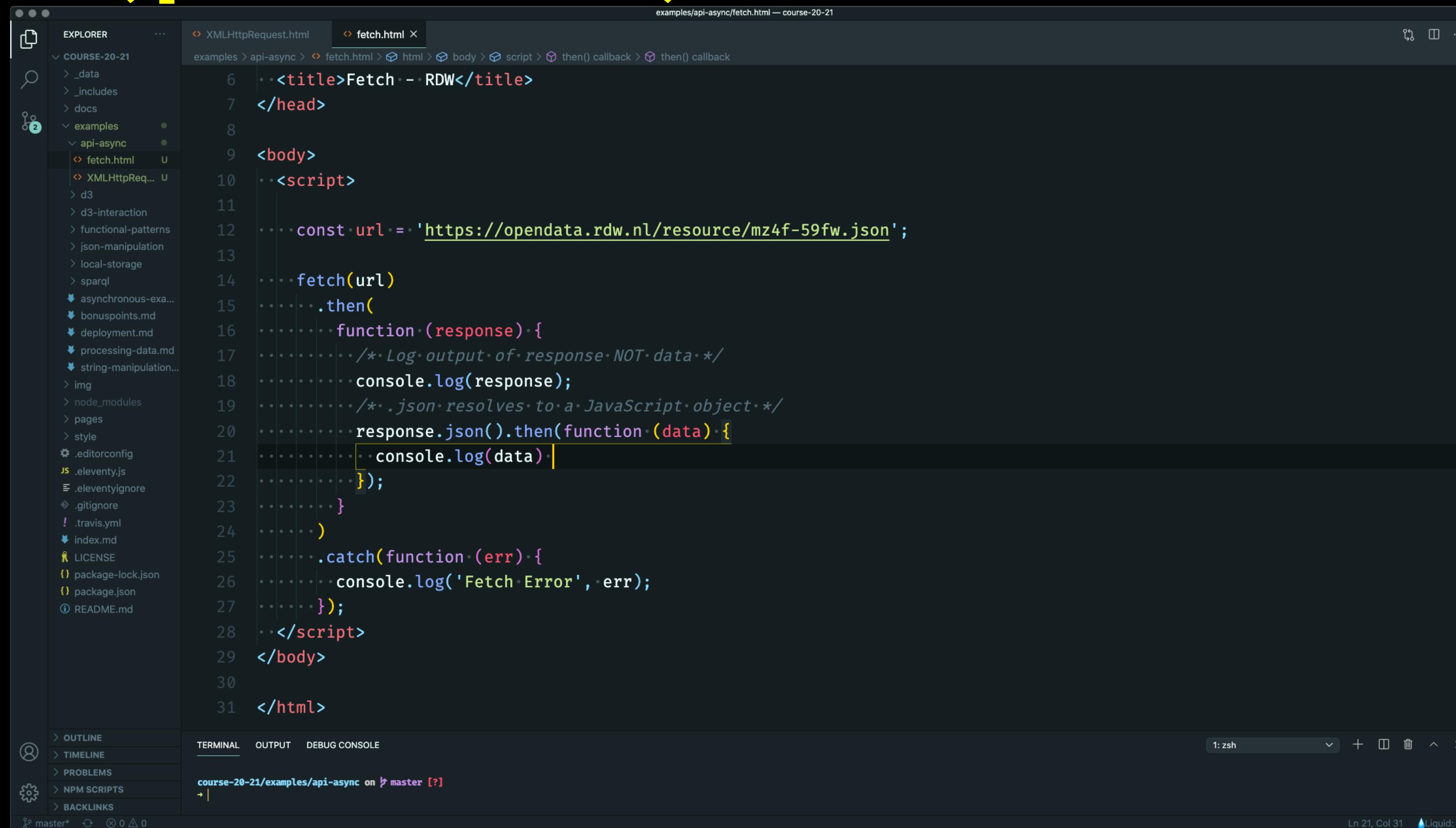
The screenshot shows a CodePen editor window with the title "Pasta Promises" by Laurens. The JS tab contains the following code:

```
1 // The problem with this approach is we don't know when prepare pasta and the
  other promise chain are finished
2
3 cookMeal()
4 function cookMeal(){
5   gatherIngredients()
6   .then(data => preparePasta(data))
7   .then(cutIngredients) //shorthand, ingredients are actually passed as a
                          param
8   .then(vegetables => bakeVegetables(vegetables)) //Explicit instead of
                          shorthand
9   //.catch((err) => console.log(err))
10 }
11
12 function preparePasta(ingredients){
13   console.log("Preparing pasta ")
14   if (Array.isArray(ingredients)){
15     boilWater(ingredients)
16     .then(cookPasta)
17
18   return new Promise( (resolve,reject) => {
19     resolve(ingredients)
20   })
21 }
```

The JS code demonstrates a promise chain for preparing a meal. It starts with a function `cookMeal` which calls `gatherIngredients`, then `preparePasta`, then `cutIngredients` (using shorthand), and finally `bakeVegetables`. It also includes a catch block for errors. The `preparePasta` function logs "Preparing pasta" and checks if the ingredients are an array before calling `boilWater` and `cookPasta`. It returns a new promise that resolves with the ingredients.

Pasta Promise Example

Fetch (promises)



A screenshot of a code editor (Visual Studio Code) displaying a file named `fetch.html`. The code example demonstrates the use of the `fetch` API with promises to make an asynchronous request to an RDW API endpoint and log the response data.

```
6 <title>Fetch -- RDW</title>
7 </head>
8
9 <body>
10 <script>
11
12 const url = 'https://opendata.rdw.nl/resource/mz4f-59fw.json';
13
14 fetch(url)
15 .then(
16   function(response){
17     /* Log output of response NOT data */
18     console.log(response);
19     /* .json resolves to a JavaScript object */
20     response.json().then(function(data){
21       console.log(data);
22     });
23   }
24 )
25 .catch(function(err){
26   console.log('Fetch Error', err);
27 });
28 </script>
29 </body>
30
31 </html>
```

The code editor interface includes:

- Explorer sidebar showing project structure and files like `XMLHttpRequest.html`, `fetch.html`, and `fetch.js`.
- Terminal tab showing the command `course-20-21/examples/api-async` and the prompt `on master [?]`.
- Status bar at the bottom showing the current branch is `master*`.

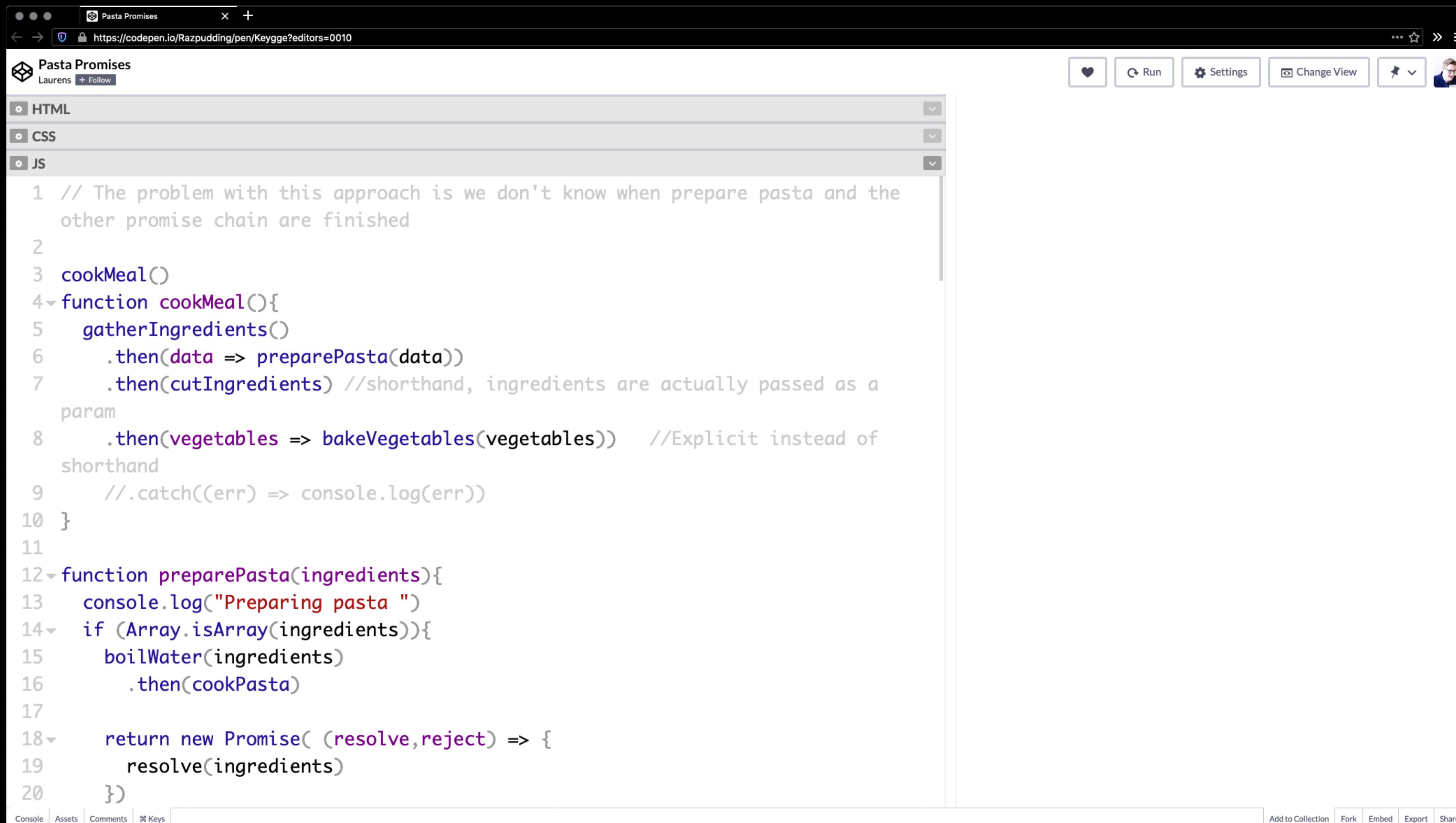
Fetch Example

Async Await

An `async` function is a function that knows how to **expect the possibility of the `await` keyword** being used to invoke asynchronous code. Act as *syntactic sugar* on top of promises, making asynchronous code easier to write and to read afterwards.

[MDN - Making asynchronous programming](#)

Async Await



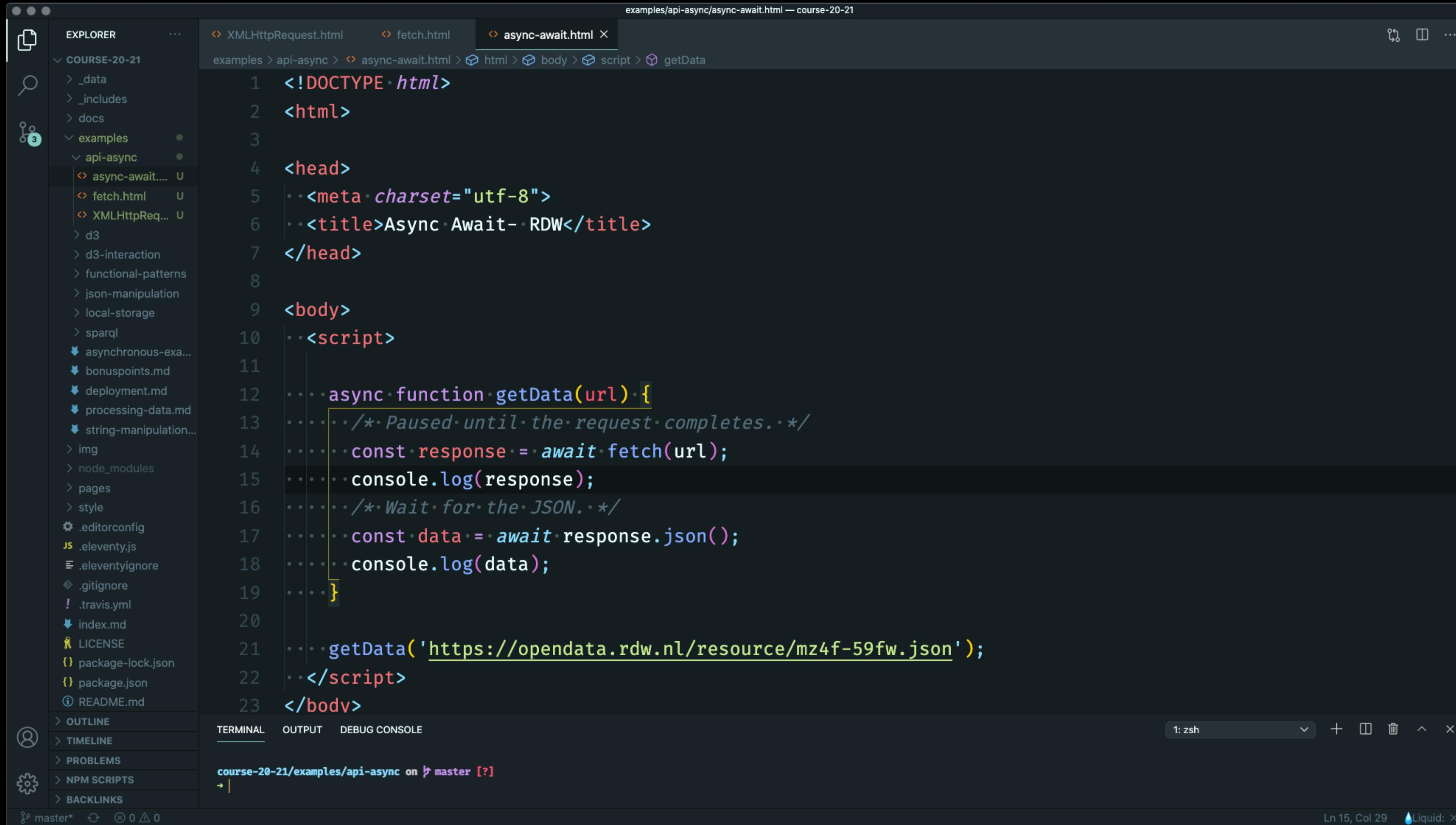
The screenshot shows a CodePen editor window titled "Pasta Promises". The URL is <https://codepen.io/Razpudding/pen/Keygge?editors=0010>. The editor interface includes tabs for HTML, CSS, and JS, and various toolbars at the top and bottom.

```
1 // The problem with this approach is we don't know when prepare pasta and the
  other promise chain are finished
2
3 cookMeal()
4 function cookMeal(){
5   gatherIngredients()
6   .then(data => preparePasta(data))
7   .then(cutIngredients) //shorthand, ingredients are actually passed as a
                          param
8   .then(vegetables => bakeVegetables(vegetables)) //Explicit instead of
                          shorthand
9   //.catch((err) => console.log(err))
10 }
11
12 function preparePasta(ingredients){
13   console.log("Preparing pasta ")
14   if (Array.isArray(ingredients)){
15     boilWater(ingredients)
16     .then(cookPasta)
17
18   return new Promise( (resolve,reject) => {
19     resolve(ingredients)
20   })
21 }
```

At the bottom of the editor, there are tabs for Console, Assets, Comments, and Keys, along with buttons for Add to Collection, Fork, Embed, Export, and Share.

Pasta Async/Await Example

Fetch (async / await)



The screenshot shows a dark-themed code editor interface with the following details:

- Explorer View:** Shows a project structure for "course-20-21" with several sub-directories and files, including "examples/api-async/async-await.html".
- Editor View:** Displays the content of "async-await.html". The code uses the Fetch API with `async/await` syntax to make a JSON request to `https://opendata.rdw.nl/resource/mz4f-59fw.json`.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>Async Await - RDW</title>
</head>
<body>
    <script>
        async function getData(url) {
            /* Paused until the request completes */
            const response = await fetch(url);
            console.log(response);
            /* Wait for the JSON */
            const data = await response.json();
            console.log(data);
        }
        getData('https://opendata.rdw.nl/resource/mz4f-59fw.json');
    </script>
</body>
```
- Terminal View:** Shows the command `course-20-21/examples/api-async` in the terminal, indicating the current working directory.
- Status Bar:** Shows the file is on the "master" branch, has 15 lines and 29 columns, and is using the "Liquid" theme.

Async Await Example

Error handling

With either method you choose, make sure you handle errors.

- `.catch`
- `if statements (status codes)`
- `throw`

Data Formats

(Transform vs clean)

Gather Data

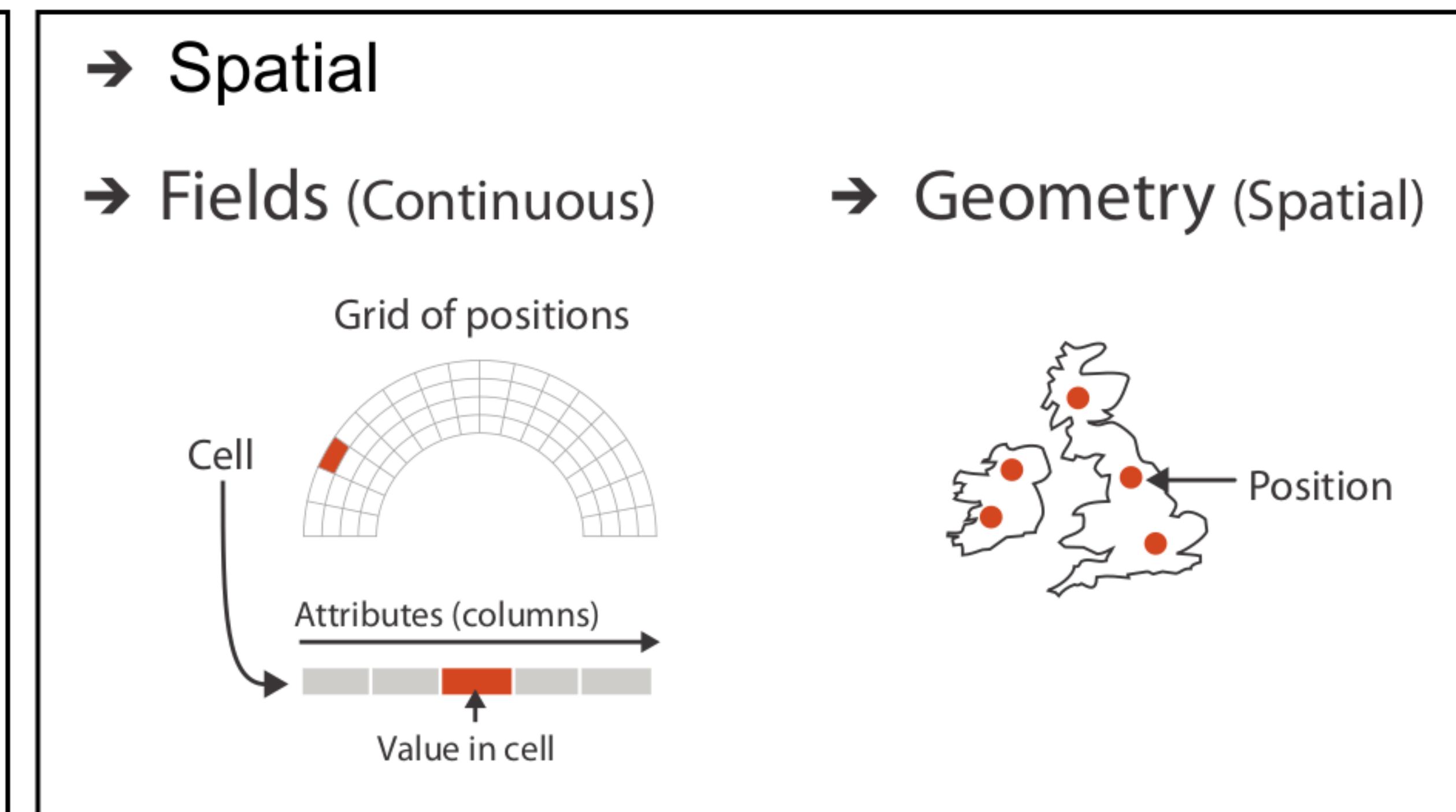
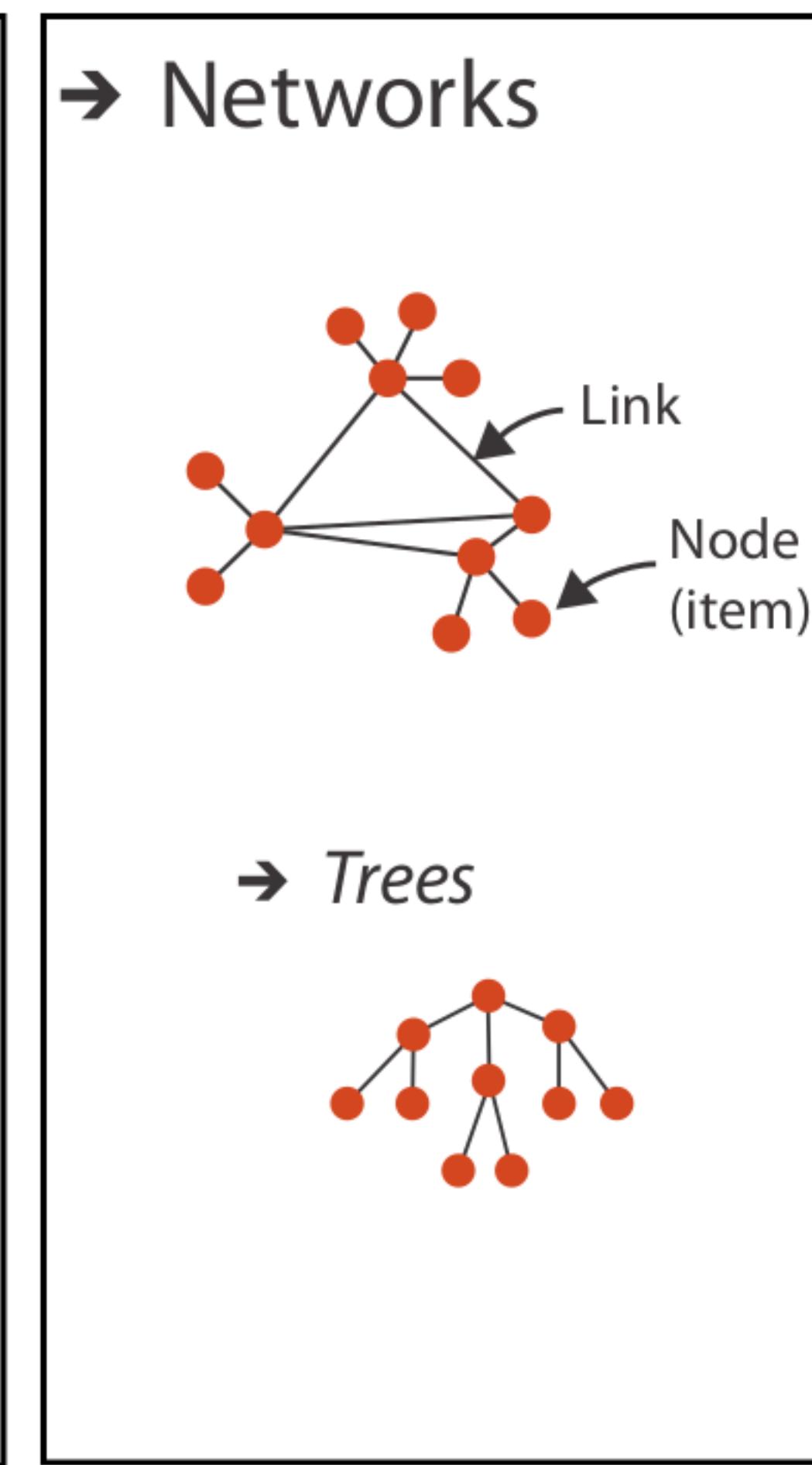
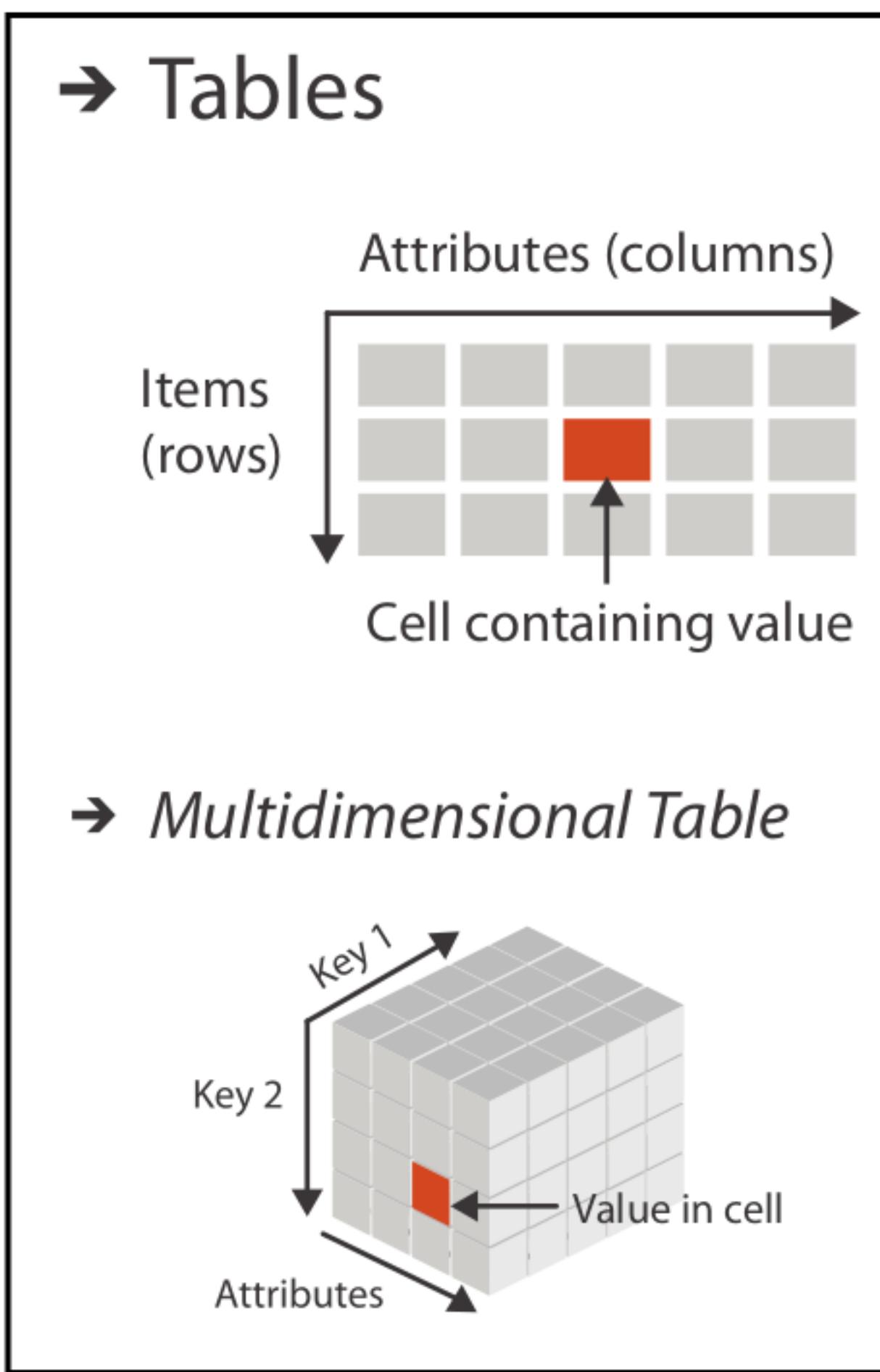
- ❖ **Get the data;** Use digital tools for automating. Put it into a spreadsheet.
- ❖ **Explore the data;** map it, count words. Use the graphic tools in spreadsheet.
- ❖ **Extend the data;** combine information from other sources. Fill in the blanks.

Gather Data

- ❖ **json** **JavaScript Object Notation**
like JavaScript, but no programming, just data
- ❖ **xml** **Extensible Markup Language**
like HTML, but for anything
- ❖ **csv** **Delimiter-separated values**
every row is a record, delimiter between fields

Three major datatypes

→ Dataset Types



➔ Attribute Types

→ Categorical

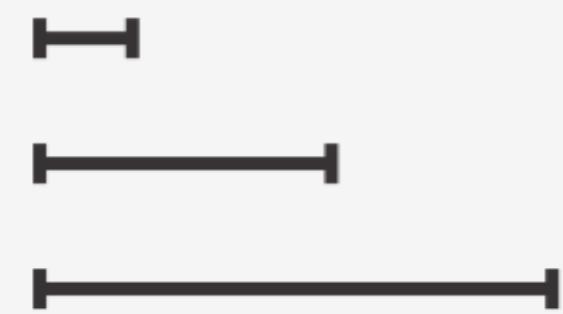


→ Ordered

→ *Ordinal*



→ *Quantitative*



	Point	Region
Space	(Latitude, Longitude) e.g. “My current location”	Geographic Identifier e.g. Countries, States, Counties
	Quantitative	Categorical
Time	Instant e.g. “Right now”	Interval e.g. “This year”
	Quantitative	Ordinal
Quantity	Value e.g. 5.2	Interval e.g. “5 - 10 years old”
	Quantitative	Ordinal

Gather Data

- ❖ **Data cleansing:**

detecting and correcting (or removing)
inaccurate records from a record set
higher order functions (map etc.)

JavaScript

- ❖ **Data transforming:**

converting data from one format or structure
to another format or structure
csv → json

D3.js

Maybe a small break?



API Architecture

(Data formats)

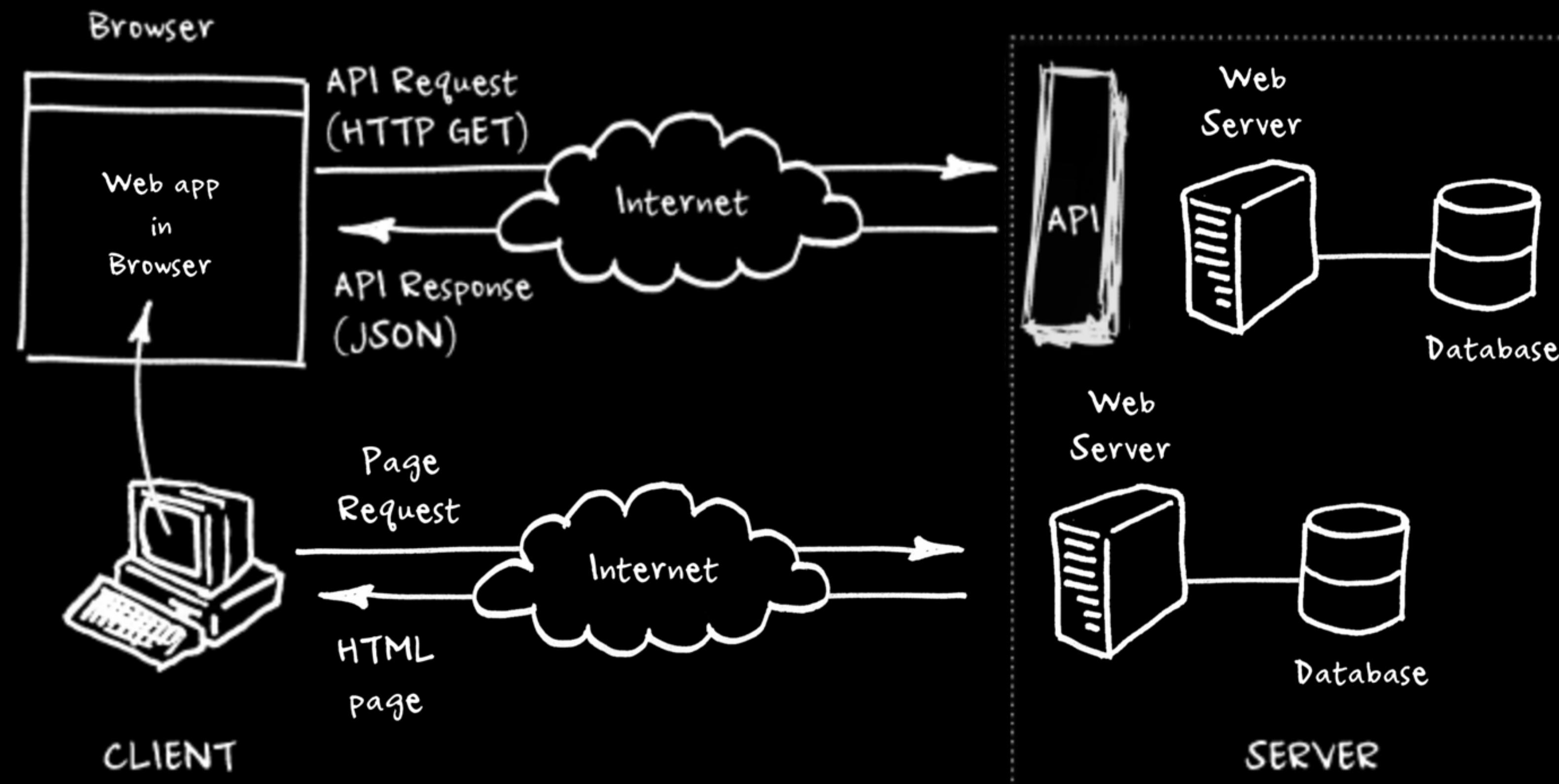
API

definition

An API is an application programming interface. It is a set of rules that allow programs to talk to each other. The developer creates the API on the server and allows the client to talk to it.

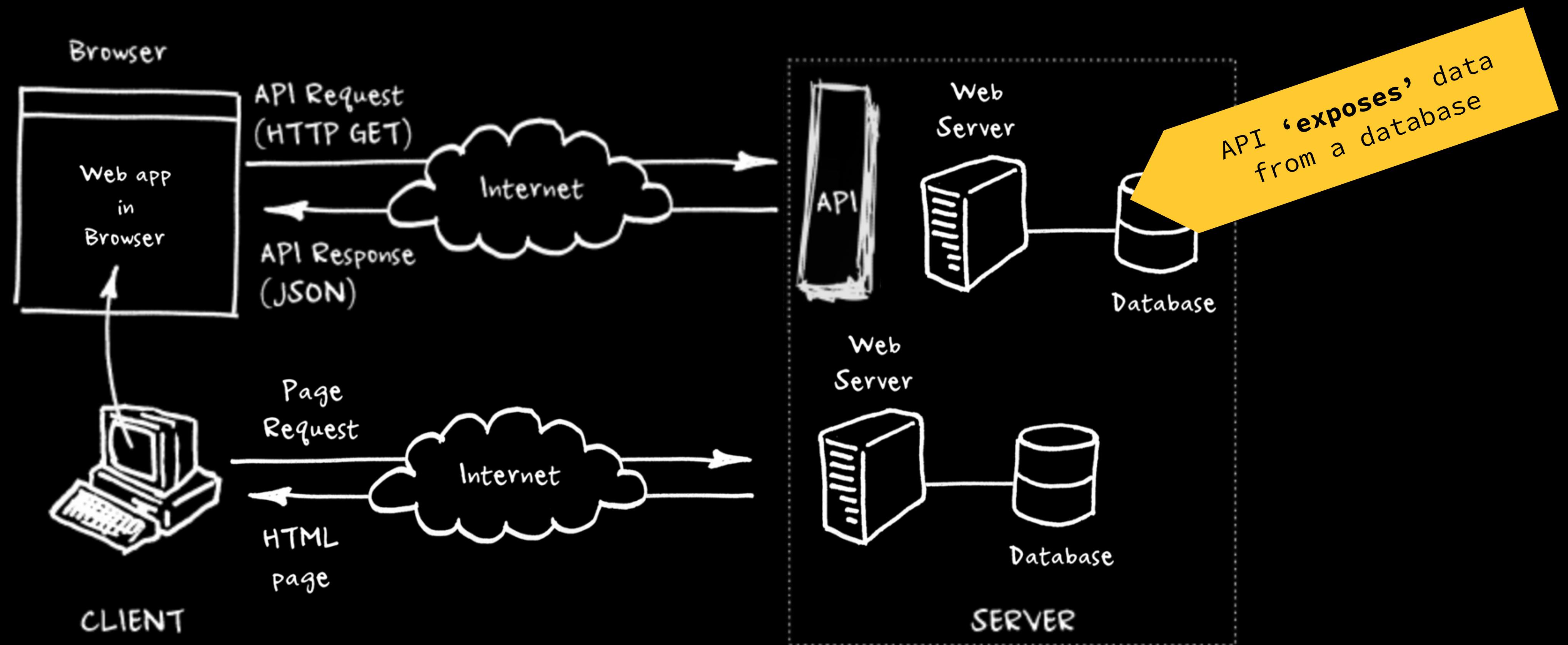
grand scheme

API



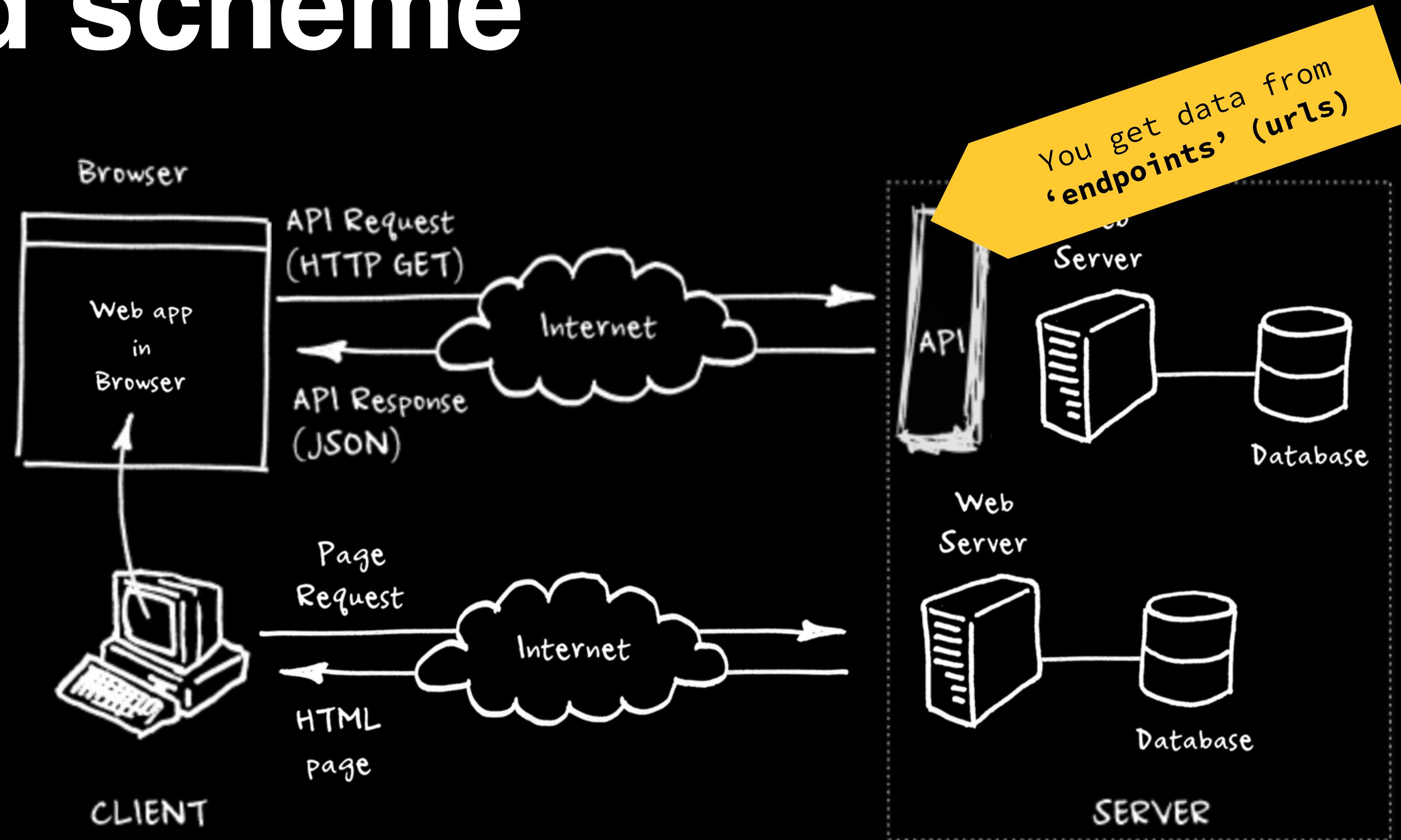
grand scheme

API



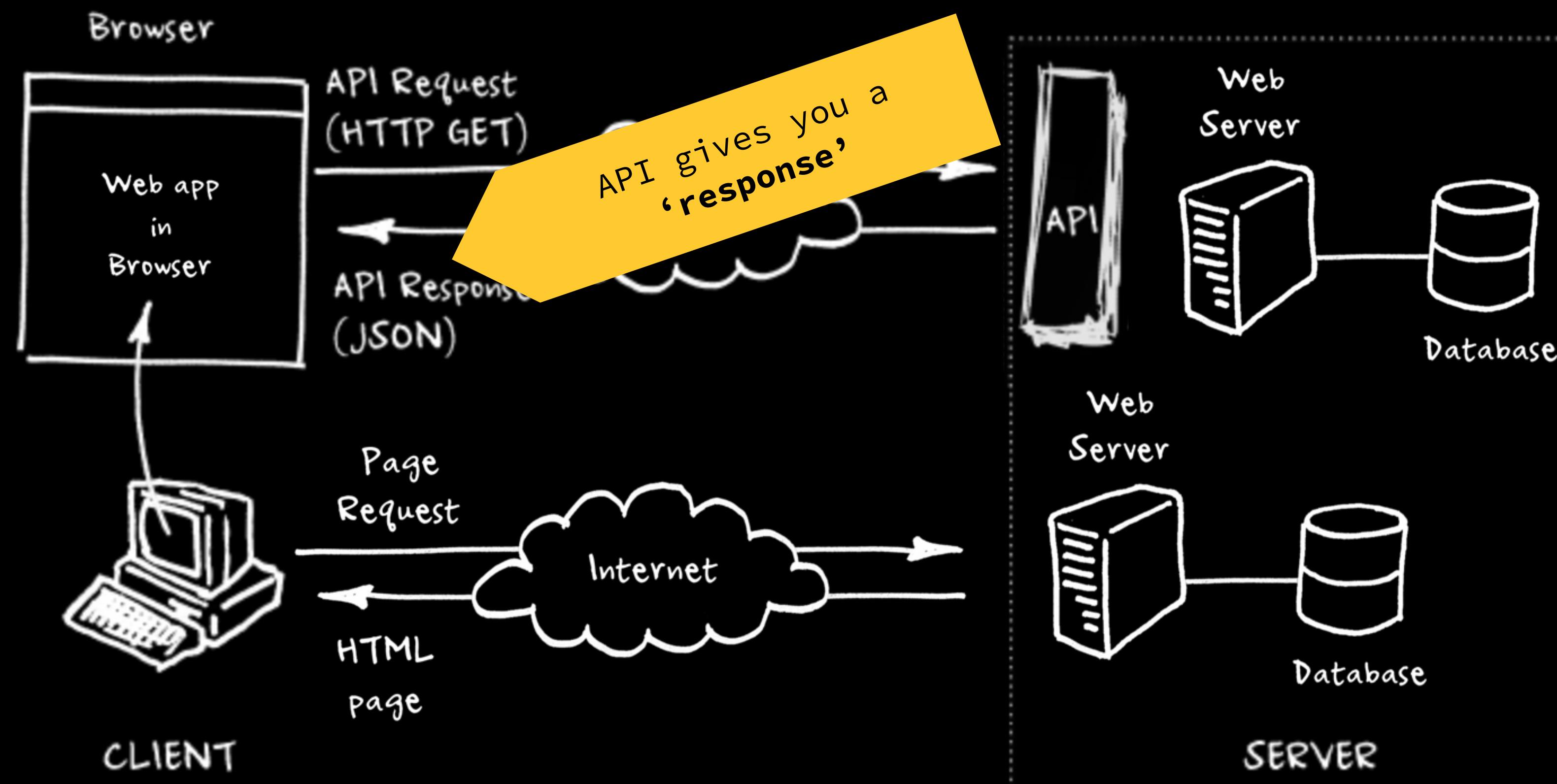
grand scheme

API



grand scheme

API



API

rest

REST determines how the API looks like. It stands for “Representational State Transfer”. It is a set of rules that developers follow when they create their API.

[Smashing - Understanding REST API's](#)

Open Data Parkeren: PARKEERGEBIED

Parkeren

Data bekijken

Visualiseren en ontdekken ▾

Exporteer

API

...

Deze tabel legt een koppeling tussen de gebieden zoals de gebieden zoals deze voor Open Data Parkeren volgens gepubliceerd worden.

Betreffende deze dataset

Bijgewerkt

25 oktober 2020

Laatst bijgewerkt op
25 oktober 2020

Metadata laatst
bijgewerkt op
25 oktober 2020

Gemaakt op
28 oktober 2014

Weergaves
4.800

Downloads
6.776

Data voorzien door
(geen) Eigenaar dataset
Open data team RDW

Krijg toegang tot deze Dataset via SODA API

De Socrata Open Data API (SODA) voorziet een programmatische toegang tot deze dataset en de mogelijkheid om gegevens te filteren, op te vragen en te combineren.

API-documenten

Ontwikkelingsportaal

API-eindpunt

<https://opendata.rdw.nl/resource/mz4f-59>

JSON

Kopiëren

Endpoint and data
format

[JSON](#)[Raw Data](#)[Headers](#)

Save Copy Collapse All Expand All



```
▼ 0:  
  areamanagerid: "344"  
  areaid: "3300"  
  uuid: "57d6f361-ffa-4186-a5a0-80a122c06fc3"  
▼ 1:  
  areamanagerid: "796"  
  areaid: "020411"  
  uuid: "b6a5905f-c79f-4669-a43a-68eccef1f3c3"  
▼ 2:  
  areamanagerid: "202"  
  areaid: "BC2"  
  uuid: "58de2c3d-2c38-4bb4-b653-30e2c9b90363"  
▼ 3:  
  areamanagerid: "344"  
  areaid: "B7200"  
  uuid: "eab21b04-bcea-4ff4-858e-eedd6a24e5b7"  
▼ 4:  
  areamanagerid: "262"  
  areaid: "1"  
  uuid: "f51fa39b-dda9-4e06-8374-1f6791c8be7c"  
▼ 5:  
  areamanagerid: "599"  
  areaid: "599_10"  
  uuid: "dca24e79-98d6-46e7-980d-b14b461cfdd8"  
▼ 6:  
  areamanagerid: "826"  
  areaid: "2"  
  uuid: "63f645d1-c2c0-4042-809e-4a217a6f2445"  
▼ 7:  
  areamanagerid: "200"  
  areaid: "PAS2"  
  uuid: "5c89ef00-b5a2-4601-82f5-38090bea1351"  
▼ 8:  
  areamanagerid: "141"
```

Get data (.json) back
(body, headers etc.)

API (auths)

- ❖ **Public**; open url endpoint which can be freely ‘fetched’ (rate-limit).



[chucknorris.io](#) is a free JSON API for hand curated Chuck Norris facts. [Read more](#)

Subscribe for new Chuck Facts

Enter your email

Subscribe

USAGE

Retrieve a random chuck joke in JSON format.

GET <https://api.chucknorris.io/jokes/random>

Example response:

```
{  
  "icon_url" : "https://assets.chucknorris.host/img/avatar/chuck-norris.png",  
  "id" : "LBpfYFjlSfSfmw4cpJS6IA",  
  "url" : "",  
  "value" : "Chuck Norris can play slide guitar with a beer bottle. Or,"}
```

API (auths)

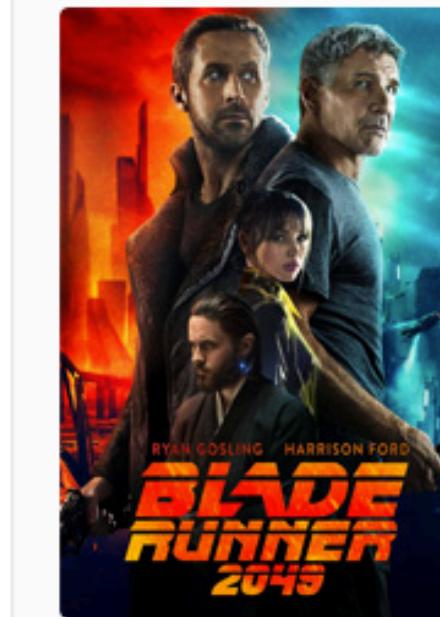
- ❖ **Public**; open url endpoint which can be freely ‘fetched’ (rate-limit).
- ❖ **apiKey**; open url endpoint which needs a ‘identifier key’ in the header or uri

OMDb API

The Open Movie Database

The OMDb API is a RESTful web service to obtain movie information, all content and images on the site are contributed and maintained by our users.

If you find this service useful, please consider making a [one-time donation](#) or [become a patron](#).



Poster API

The Poster API is only available to patrons.

Currently over 280,000 posters, updated daily with resolutions up to 2000x3000.

Attention Users

04/08/19 - Added support for eight digit IMDb IDs.

01/20/19 - Supressed adult content from search results.

01/20/19 - Added Swagger files ([YAML](#), [JSON](#)) to expose current API abilities and upcoming REST functions.

 [Become a Patron](#)

Sponsors

[Emby](#), [Trakt](#), [FileBot](#), [Reelgood](#), [Xirvik Servers](#), [Yidio](#), [mi.tv](#), [Couchpop](#), [What's on Netflix](#), [Edu Reviewer](#), [Flixboss](#), [StreamingMoviesRight](#), [Scripts on Screen](#), [Writers Per Hour](#), [Medium.com](#), [Write my paper](#), [Ramotion.com](#), [Phone Trackers](#), [Property for sale in Lake Como](#), [iStarTips](#), [What A Room](#), [Vibelovely](#), [StreamToday](#), [Property for sale in Spain](#), [Top Casinos Reviews](#), [Classics on DVD](#), [Streaming App](#), [How to make a FinTech app](#), [Vid2 - Create movie lists with AI](#)

Usage

Send all data requests to:

`http://www.omdbapi.com/?apikey=[yourkey]&`

Poster API requests:

`http://img.omdbapi.com/?apikey=[yourkey]&`

Parameter

http://www.omdbapi.com/?apikey=[yourkey]

Unique Identifier

http://www.omdbapi.com/?apikey=[yourkey]

API (auths)

- ❖ **Public**; open url endpoint which can be freely ‘fetched’ (rate-limit).
- ❖ **apiKey**; open url endpoint which needs a ‘identifier key’ in the header or uri
- ❖ **oAuth**; authentication protocol with user identification before fetching

 Web API

- Overview
- Getting started
- ▼ Concepts

Access Token

- API calls
- Apps
- Authorization
- Playlists
- Quota modes
- Rate limits
- Scopes
- Spotify URIs and IDs
- Track Relinking

Tutorials

- Authorization code
- Authorization code PKCE
- Client credentials
- Implicit grant
- Refreshing tokens

How-Tos**REFERENCE**

- ▶ Albums
- ▶ Artists
- ▶ Audiobooks
- ▶ Categories
- ▶ Chapters
- ▶ Episodes
- ▶ Genres

Access Token

The *access token* is a string which contains the credentials and permissions that can be used to access a given resource (e.g artists, albums or tracks) or user's data (e.g your profile or your playlists).

To use the *access token* you must include the following header in your API calls:

Header Parameter	Value
Authorization	Valid access token following the format: Bearer <Access Token>

Note that the *access token* is valid for 1 hour (3600 seconds). After that time, the token expires and you need to request a new one.

Examples

The following example uses cURL to retrieve information about a track using the [Get a track](#) endpoint:

```
1 curl --request GET \
2   'https://api.spotify.com/v1/tracks/2TpxZ7JUBn3uw46aR7qd6V' \
3   --header "Authorization: Bearer NgCXRK...MzYjw"
```

The following code implements the `getProfile()` function which performs the API call to the [Get Current User's Profile](#) endpoint to retrieve the user profile related information:

```
1 async function getProfile(accessToken) {
2   let accessToken = localStorage.getItem('access_token');
3
4   const response = await fetch('https://api.spotify.com/v1/me', {
5     headers: {
6       Authorization: 'Bearer ' + accessToken
7     }
8   });
9
10  const data = await response.json();
11 }
```

Multi-fetch

Often API's uses **multiple endpoints** for different types of data (collections vs individual items) or **limit the amount of items** that get returned in each request (pagination).

[Home](#)[Object metadata](#)[API](#)[Harvest](#)[Download](#)[Bibliographic data](#)[Controlled vocabularies](#)[User-generated content](#)[Object metadata](#) / API

Object metadata APIs

The object metadata APIs make the power of the award-winning [Rijksmuseum website](#) directly accessible to developers. [Searching the collection](#) through the API offers a wide range of interesting possibilities, as do the [tiled images](#) used to zoom in to close-ups of objects. The JSON-based service is so easy to use that you can create an application using the Rijksmuseum's rich and [freely accessible content](#) in no time.

Access to APIs

To start using the data and images, you first need to obtain an API key by registering for a [Rijksstudio account](#). You will be given a key instantly upon request, which you can find at the advanced settings of your Rijksstudio account.

Collection API

`GET /api/[culture]/collection` gives access to the collection with brief information about each object. The results are split up in result pages. By using the `p` and `ps` parameters you can fetch more results, up to a total of 10,000. All of the other parameters are identical to the [advanced search page](#) on the Rijksmuseum website. You can use that page to find out what's the best query to use.

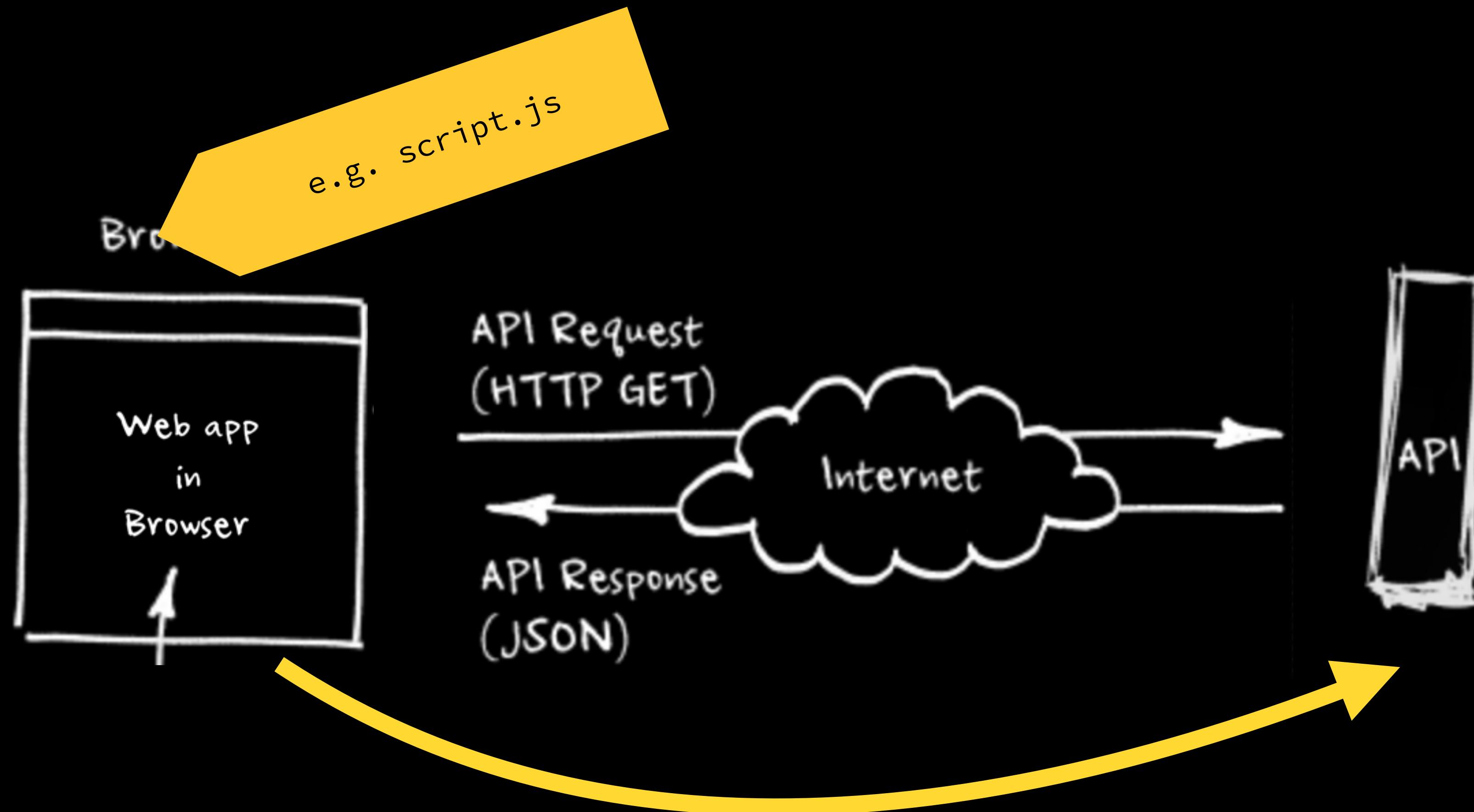
Parameter	Format	Default	Notes
<code>key</code>	<code>a-z 0-9</code>		Your API-key , mandatory for every request.
<code>format</code>	<code>json / jsonp / xml</code>	<code>json</code>	The format of the result.
<code>culture</code>	<code>nl / en</code>		The language to search in (and of the results).
<code>p</code>	<code>0-n</code>	<code>0</code>	The result page. Note that <code>p * ps</code> cannot exceed 10,000.
<code>ps</code>	<code>1-100</code>	<code>10</code>	The number of results per page.
<code>q</code>	<code>a-z</code>		The search terms that need to occur in one of the fields of the object data.
<code>involvedMaker</code>	<code>a-z</code>		Object needs to be made by this agent.
<code>type</code>	<code>a-z</code>		The type of the object.
<code>material</code>	<code>a-z</code>		The material of the object.
<code>technique</code>	<code>a-z</code>		The technique used to make the object.
<code>f.dating.period</code>	<code>0-21</code>		The century in which the object is made.

<https://data.rijksmuseum.nl/object-metadata/api/>

Client vs. Server *(Fetch)*

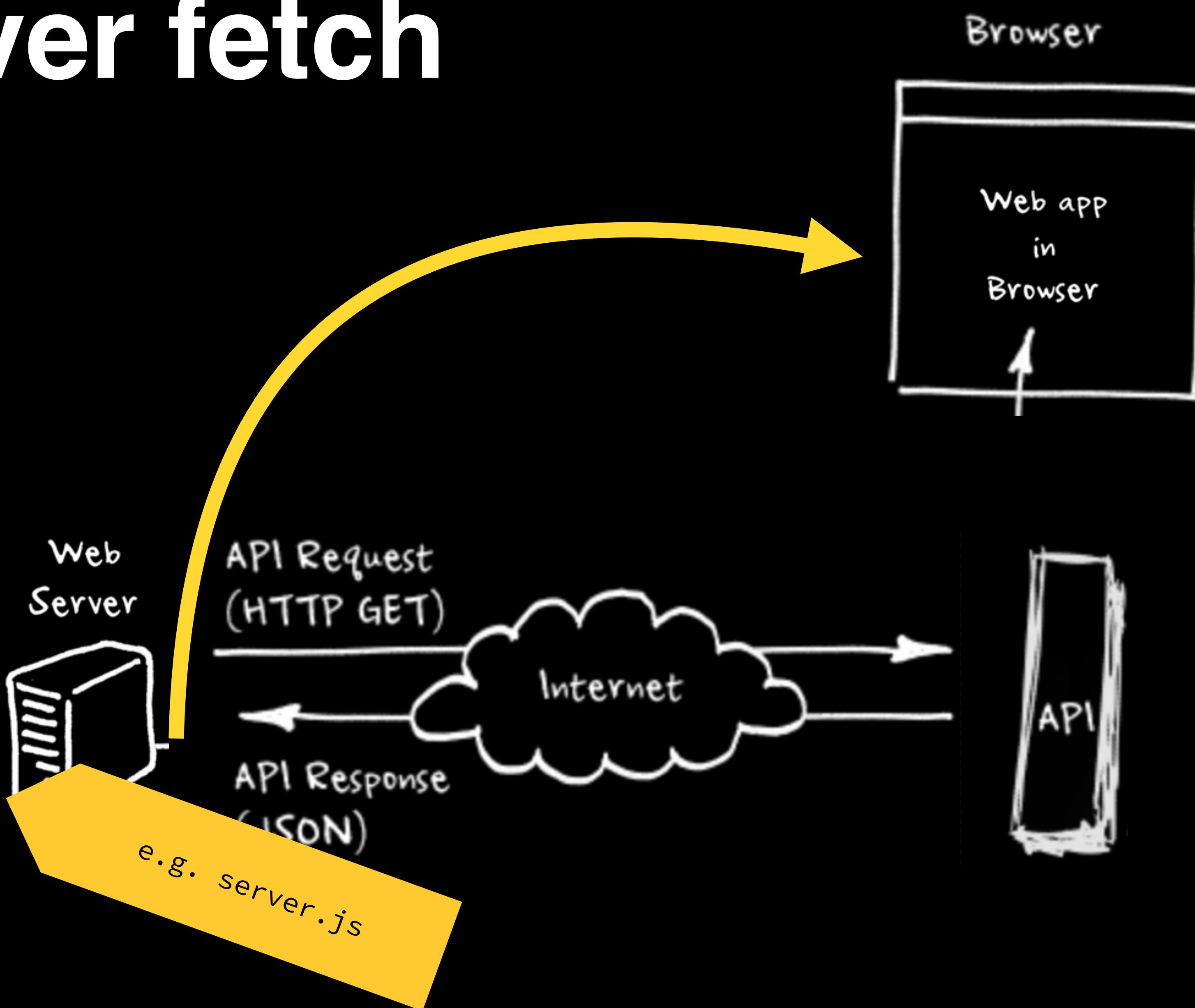
Client fetch

API



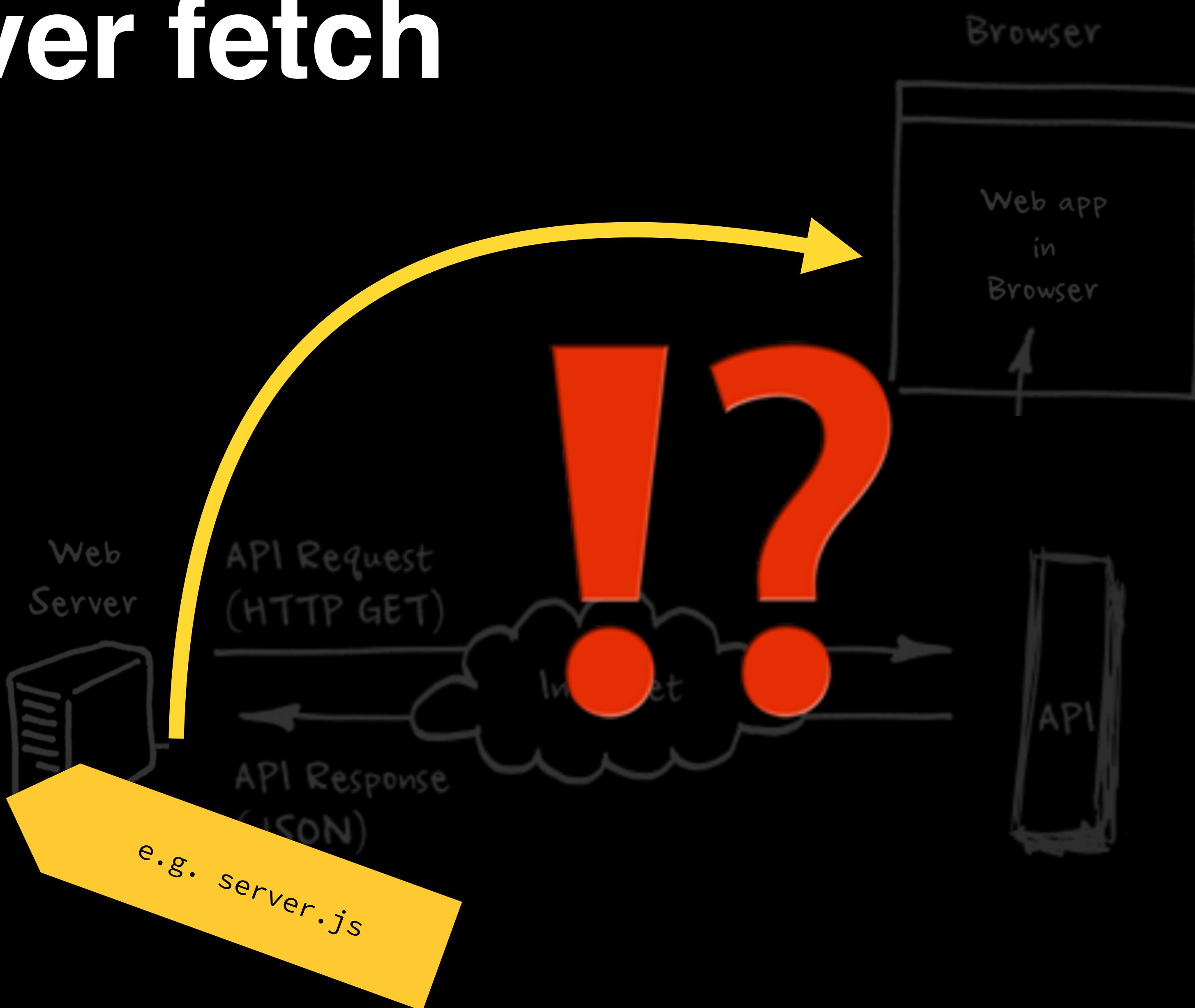
Server fetch

API



Server fetch

API



Server fetch

API keys are identifiers (unique) and acts as a password and are usually app-specific. You **don't want your password public** (either on the client in the JavaScript) or in your GitHub repository.

.env

```
# .env file
USER_ID="239482"
USER_KEY="foobar"
NODE_ENV="development"
```

server.js

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

process.env.USER_ID; // "239482"
process.env.USER_KEY; // "foobar"
process.env.NODE_ENV; // "development"
```

GETTING STARTED

Introduction
Creating a project
Project structure
Web standards

CORE CONCEPTS

Routing
Loading data
Form actions
Page options
State management

BUILD AND DEPLOY

Building your app
Adapters
Zero-config deployments
Node servers
Static site generation
Single-page apps
Cloudflare Pages
Cloudflare Workers
Netlify
Vercel
Writing adapters

ADVANCED

Advanced routing
Hooks
Errors
Link options
Service workers
Server-only modules
Asset handling
Snapshots
Packaging

ADVANCED

Server-only modules

[Edit this page on GitHub](#)

Like a good friend, SvelteKit keeps your secrets. When writing your backend and frontend in the same repository, it can be easy to accidentally import sensitive data into your front-end code (environment variables containing API keys, for example). SvelteKit provides a way to prevent this entirely: server-only modules.

Private environment variables

The `$env/static/private` and `$env/dynamic/private` modules, which are covered in the [modules](#) section, can only be imported into modules that only run on the server, such as [hooks.server.js](#) or [+page.server.js](#).

Your modules

You can make your own modules server-only in two ways:

- adding `.server` to the filename, e.g. `secrets.server.js`
- placing them in `$lib/server`, e.g. `$lib/server/secrets.js`

How it works

Any time you have public-facing code that imports server-only code (whether directly or indirectly)...

`$lib/server/secrets.js`

```
export const atlantisCoordinates = /* redacted */;
```

`src/routes/utils.js`

```
export { atlantisCoordinates } from '$lib/server/secrets.js';
```

```
export const add = (a, b) => a + b;
```

`src/routes/+page.svelte`

ON THIS PAGE

Server-only modules
Private environment variables
Your modules
How it works
Further reading

```
import fetch from 'node-fetch';
import dotenv from 'dotenv';

dotenv.config();

const url = 'https://api.schiphol.nl/public-flights/flights';
const params = {
  departure: '?flightDirection=D&route=',
  arrival: '?flightDirection=A&route='
};
const params2 = '&includedelays=false&page=0&sort=%2BscheduleTime&fromDateTime=';
const params3 = '&searchDateTimeField=scheduleDateTime';
const headers = {
  resourceversion: 'v4',
  app_id: process.env.APP_ID,
  app_key: process.env.APP_KEY,
  Accept: 'application/json'
};

const fetchData = async (iata, dateTime, direction) => {
  try {
    const res = await fetch(` ${url}${params[direction]}${iata}${params2}${dateTime}${params3}` ,
      method: 'GET',
      headers
    );
  }
}
```

<https://github.com/ninadepina/tech-track-23-24/blob/main/app/src/lib/fetchlata.js>

**Uncaught SyntaxError
Unexpected end of input**