SYNCHRONOUS CODE

BLOCKING

```
const p = document.querySelector('.p');
p.textContent = 'My name is Jonas!';
alert('Text set!');
p.style.color = 'red';
```

Part of execution context that actually executes the code in computer's CPU

SYNCHRONOUS

- 👉 Most code is **synchronous**;
- 👉 Synchronous code is **executed line by line**;
- 👉 Each line of code **waits** for previous line to finish;
- 👎 Long-running operations **block** code execution.

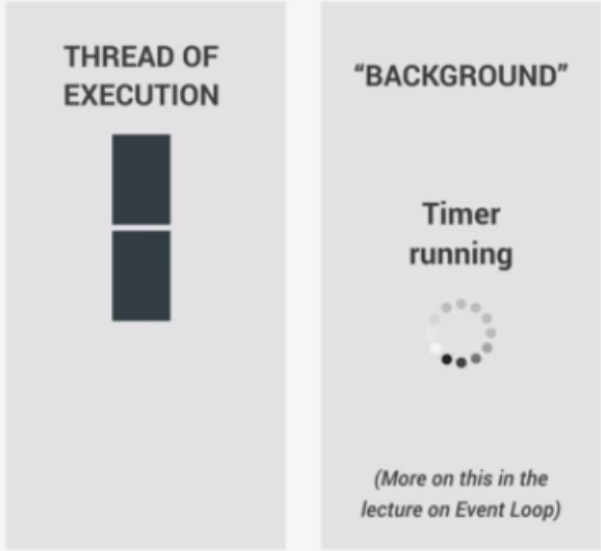
ASYNCHRONOUS CODE

Asynchronous

CALLBACK WILL RUN AFTER TIMER

```
const p = document.querySelector('.p');
setTimeout(function () {
  p.textContent = 'My name is Jonas!';
}, 5000);
p.style.color = 'red';
```

👉 **Example:** Timer with callback



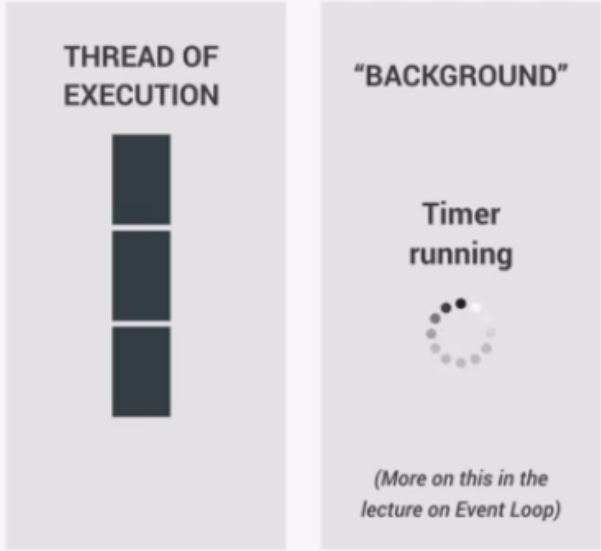
ASYNCHRONOUS CODE

Asynchronous

CALLBACK WILL RUN AFTER TIMER

```
const p = document.querySelector('.p');
setTimeout(function () {
  p.textContent = 'My name is Jonas!';
}, 5000);
p.style.color = 'red';
```

👉 **Example:** Timer with callback



ASYNCHRONOUS

- 👉 Asynchronous code is executed **after a task that runs in the "background" finishes**;
- 👍 Asynchronous code is **non-blocking**;
- 👉 Execution doesn't wait for an asynchronous task to finish its work;

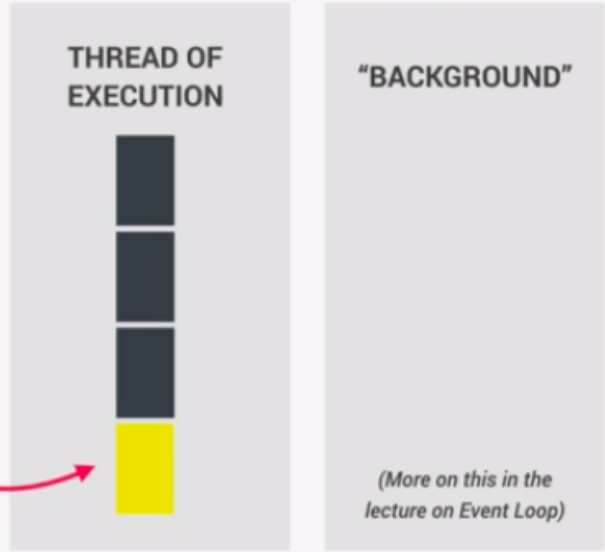
ASYNCHRONOUS CODE

Asynchronous

CALLBACK WILL RUN AFTER TIMER

```
const p = document.querySelector('.p');
setTimeout(function () {
  p.textContent = 'My name is Jonas!';
}, 5000);
p.style.color = 'red';
```

👉 **Example:** Timer with callback



ASYNCHRONOUS

- 👉 Asynchronous code is executed **after a task that runs in the "background" finishes**;
- 👍 Asynchronous code is **non-blocking**;
- 👉 Execution doesn't wait for an asynchronous task to finish its work;

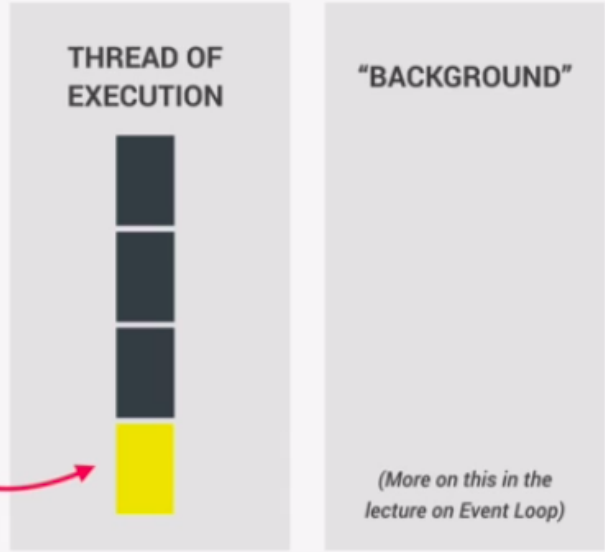
ASYNCHRONOUS CODE

Asynchronous

CALLBACK WILL RUN AFTER TIMER

```
const p = document.querySelector('.p');
setTimeout(function () {
  p.textContent = 'My name is Jonas!';
}, 5000);
p.style.color = 'red';
```

👉 **Example:** Timer with callback



ASYNCHRONOUS

Coordinating behavior of a program over a period of time

- 👉 Asynchronous code is executed **after a task that runs in the "background" finishes**;
- 👍 Asynchronous code is **non-blocking**;
- 👉 Execution doesn't wait for an asynchronous task to finish its work;

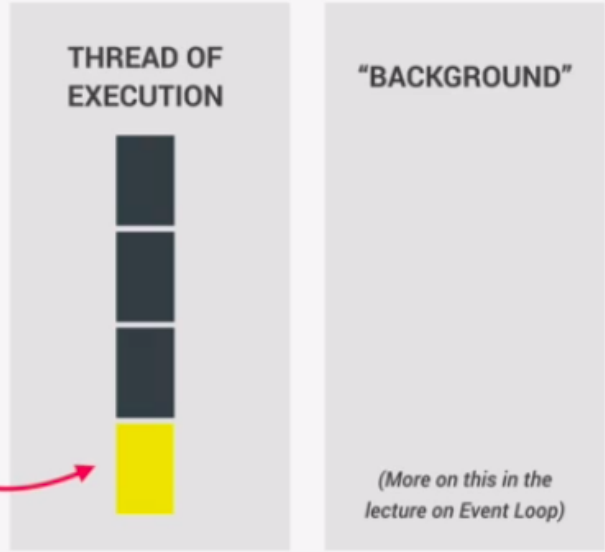
ASYNCHRONOUS CODE

Asynchronous

```
const p = document.querySelector('.p');
setTimeout(function () {
  p.textContent = 'My name is Jonas!';
}, 5000);
p.style.color = 'red';
```

CALLBACK WILL RUN AFTER TIMER

👉 **Example:** Timer with callback



ASYNCHRONOUS

Coordinating behavior of a program over a period of time

- 👉 Asynchronous code is executed **after a task that runs in the "background" finishes**;
- 👍 Asynchronous code is **non-blocking**;
- 👉 Execution doesn't wait for an asynchronous task to finish its work;
- 👉 Callback functions alone do **NOT** make code asynchronous!

ASYNCHRONOUS CODE

Asynchronous

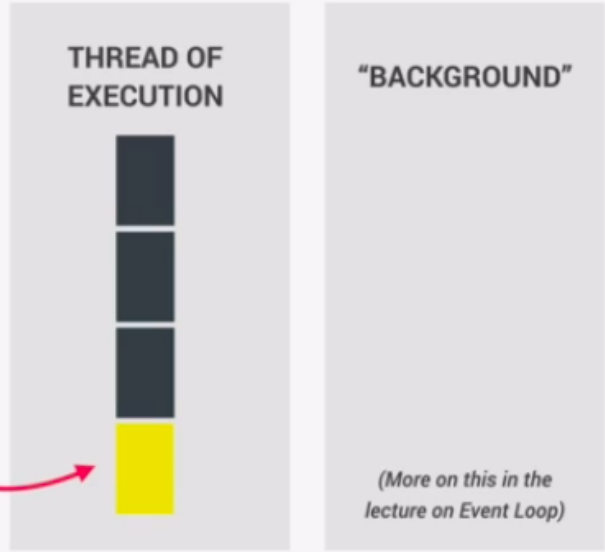
```
const p = document.querySelector('.p');
setTimeout(function () {
  p.textContent = 'My name is Jonas!';
}, 5000);
p.style.color = 'red';
```

CALLBACK WILL RUN AFTER TIMER

👉 **Example:** Timer with callback

```
[1, 2, 3].map(v => v * 2);
```

Callback does NOT automatically make code asynchronous!



Executed after all other code

ASYNCHRONOUS

Coordinating behavior of a program over a period of time

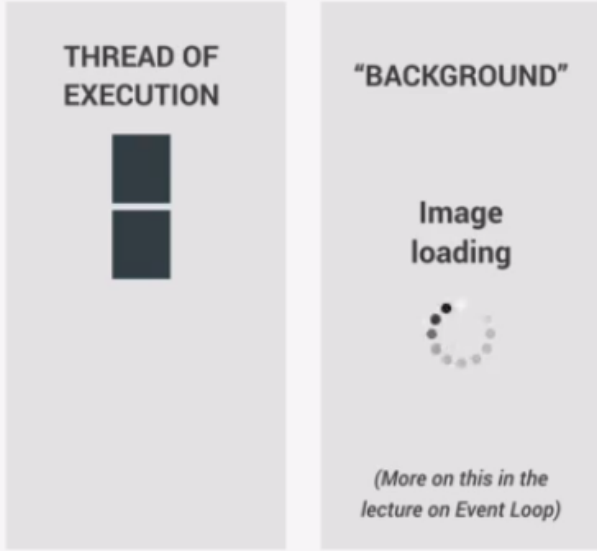
- 👉 Asynchronous code is executed **after a task that runs in the "background" finishes**;
- 👍 Asynchronous code is **non-blocking**;
- 👉 Execution doesn't wait for an asynchronous task to finish its work;
- 👉 Callback functions alone do **NOT** make code asynchronous!

ASYNCHRONOUS CODE

Asynchronous

```
const img = document.querySelector('.dog');
img.src = 'dog.jpg';
img.addEventListener('load', function () {
  img.classList.add('fadeIn');
});
p.style.width = '300px';
```

👉 **Example:** Asynchronous image loading with event and callback



ASYNCHRONOUS

Coordinating behavior of a program over a period of time

- 👉 Asynchronous code is executed **after a task that runs in the "background" finishes**;
- 👍 Asynchronous code is **non-blocking**;
- 👉 Execution doesn't wait for an asynchronous task to finish its work;
- 👉 Callback functions alone do **NOT** make code asynchronous!

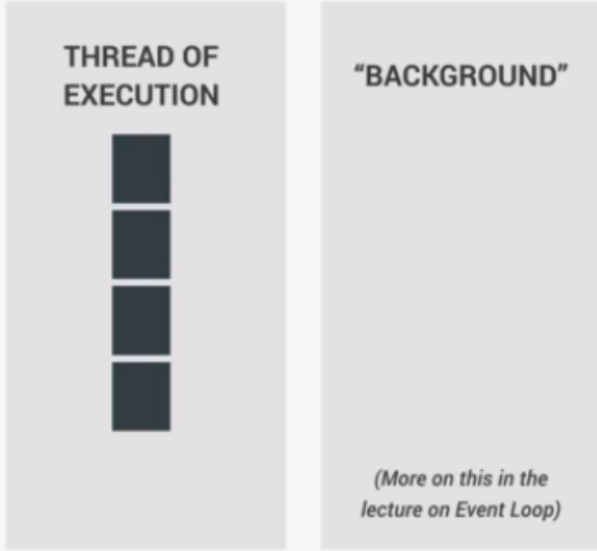
ASYNCHRONOUS CODE

Asynchronous

CALLBACK WILL RUN AFTER IMAGE LOADS

```
const img = document.querySelector('.dog');
img.src = 'dog.jpg';
img.addEventListener('load', function () {
  img.classList.add('fadeIn');
});
p.style.width = '300px';
```

👉 **Example:** Asynchronous image loading with event and callback



ASYNCHRONOUS

Coordinating behavior of a program over a period of time

- 👉 Asynchronous code is executed **after a task that runs in the "background" finishes**;
- 👍 Asynchronous code is **non-blocking**;
- 👉 Execution doesn't wait for an asynchronous task to finish its work;
- 👉 Callback functions alone do **NOT** make code asynchronous!

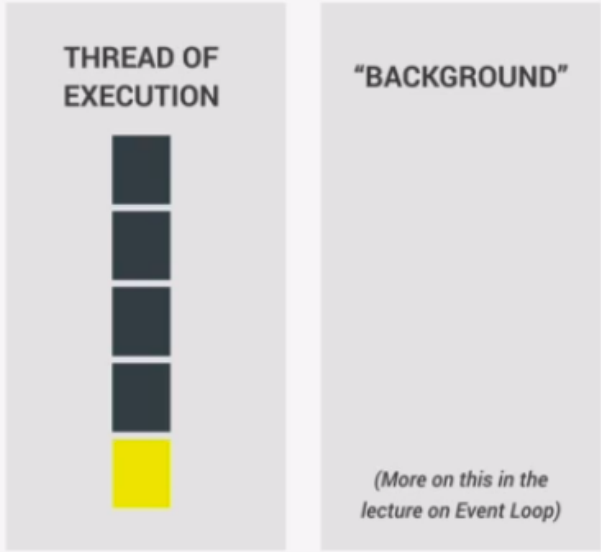
ASYNCHRONOUS CODE

Asynchronous

CALLBACK WILL RUN AFTER IMAGE LOADS

```
const img = document.querySelector('.dog');
img.src = 'dog.jpg';
img.addEventListener('load', function () {
  img.classList.add('fadeIn');
});
p.style.width = '300px';
```

👉 **Example:** Asynchronous image loading with event and callback



ASYNCHRONOUS

Coordinating behavior of a program over a period of time

- 👉 Asynchronous code is executed **after a task that runs in the "background" finishes**;
- 👍 Asynchronous code is **non-blocking**;
- 👉 Execution doesn't wait for an asynchronous task to finish its work;
- 👉 Callback functions alone do **NOT** make code asynchronous!

ASYNCHRONOUS CODE

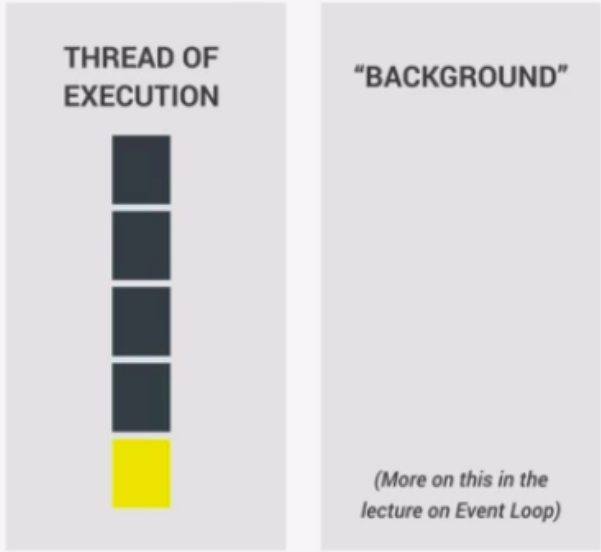
Asynchronous

```
const img = document.querySelector('.dog');
img.src = 'dog.jpg';
img.addEventListener('load', function () {
  img.classList.add('fadeIn');
});
p.style.width = '300px';
```

CALLBACK WILL RUN AFTER IMAGE LOADS

`addEventListener` does NOT automatically make code asynchronous!

👉 **Example:** Asynchronous image loading with event and callback



ASYNCHRONOUS

Coordinating behavior of a program over a period of time

- 👉 Asynchronous code is executed **after a task that runs in the "background" finishes**;
- 👍 Asynchronous code is **non-blocking**;
- 👉 Execution doesn't wait for an asynchronous task to finish its work;
- 👉 Callback functions alone do **NOT** make code asynchronous!

ASYNCHRONOUS CODE

Asynchronous

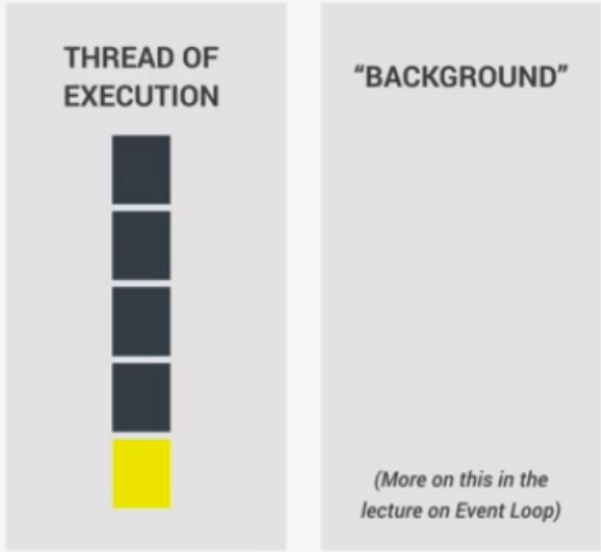
```
const img = document.querySelector('.dog');
img.src = 'dog.jpg';
img.addEventListener('load', function () {
  img.classList.add('fadeIn');
});
p.style.width = '300px';
```

CALLBACK WILL RUN AFTER IMAGE LOADS

addEventListener does NOT automatically make code asynchronous!

👉 **Example:** Asynchronous image loading with event and callback

👉 **Other examples:** Geolocation API or **AJAX calls**



ASYNCHRONOUS

Coordinating behavior of a program over a period of time

- 👉 Asynchronous code is executed **after a task that runs in the "background" finishes**;
- 👍 Asynchronous code is **non-blocking**;
- 👉 Execution doesn't wait for an asynchronous task to finish its work;
- 👉 Callback functions alone do **NOT** make code asynchronous!

WHAT ARE AJAX CALLS?

AJAX

Asynchronous **J**avaScript **A**nd **X**ML: Allows us to communicate with remote web servers in an **asynchronous way**. With AJAX calls, we can **request data** from web servers dynamically.

WHAT IS AN API?

API

- **A**pplication **P**rogramming **I**nterface: Piece of software that can be used by another piece of software, in order to allow **applications to talk to each other**;
- There are be many types of APIs in web development:

  DOM API | Geolocation API | Own Class API | **"Online" API**

  Just "API"
- **"Online" API:** Application running on a server, that receives requests for data, and sends data back as response;
- We can build **our own** web APIs (requires back-end development, e.g. with node.js) or use **3rd-party** APIs.

There is an API for everything

- Weather data
- Data about countries
- Flights data
- Currency conversion data
- APIs for sending email or SMS
- Google Maps
- *Millions of possibilities...*

WHAT IS AN API?

API

- **A**pplication **P**rogramming **I**nterface: Piece of software that can be used by another piece of software, in order to allow **applications to talk to each other**;
- There are be many types of APIs in web development:

  DOM API | Geolocation API | Own Class API | **"Online" API**

  Just "API"
- **"Online" API:** Application running on a server, that receives requests for data, and sends data back as response;
- We can build **our own** web APIs (requires back-end development, e.g. with node.js) or use **3rd-party** APIs.

There is an API for everything

- Weather data
- Data about countries
- Flights data
- Currency conversion data
- APIs for sending email or SMS
- Google Maps
- *Millions of possibilities...*

AJAX

XML

XML data format

WHAT IS AN API?

API

- **A**pplication **P**rogramming **I**nterface: Piece of software that can be used by another piece of software, in order to allow **applications to talk to each other**;
- There are be many types of APIs in web development:

  DOM API | Geolocation API | Own Class API | **"Online" API**

  Just "API"
- **"Online" API:** Application running on a server, that receives requests for data, and sends data back as response;
- We can build **our own** web APIs (requires back-end development, e.g. with node.js) or use **3rd-party** APIs.

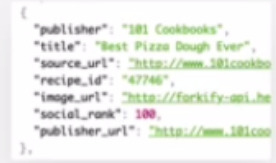
There is an API for everything

- Weather data
- Data about countries
- Flights data
- Currency conversion data
- APIs for sending email or SMS
- Google Maps
- *Millions of possibilities...*

AJAX

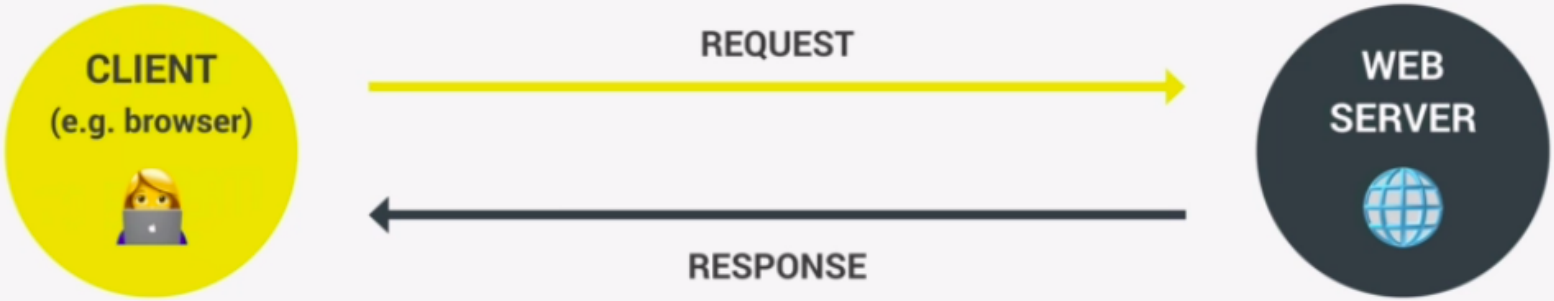
XML data format

JSON data format



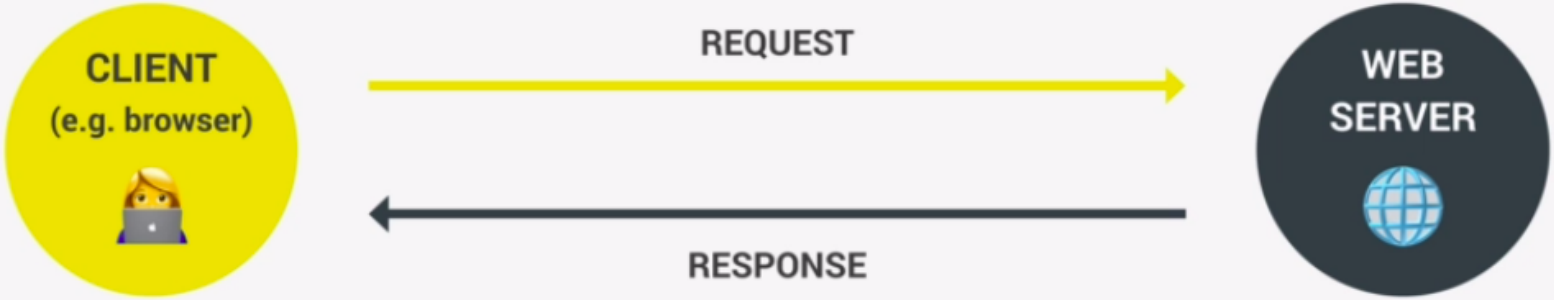
Most popular API data format

WHAT HAPPENS WHEN WE ACCESS A WEB SERVER

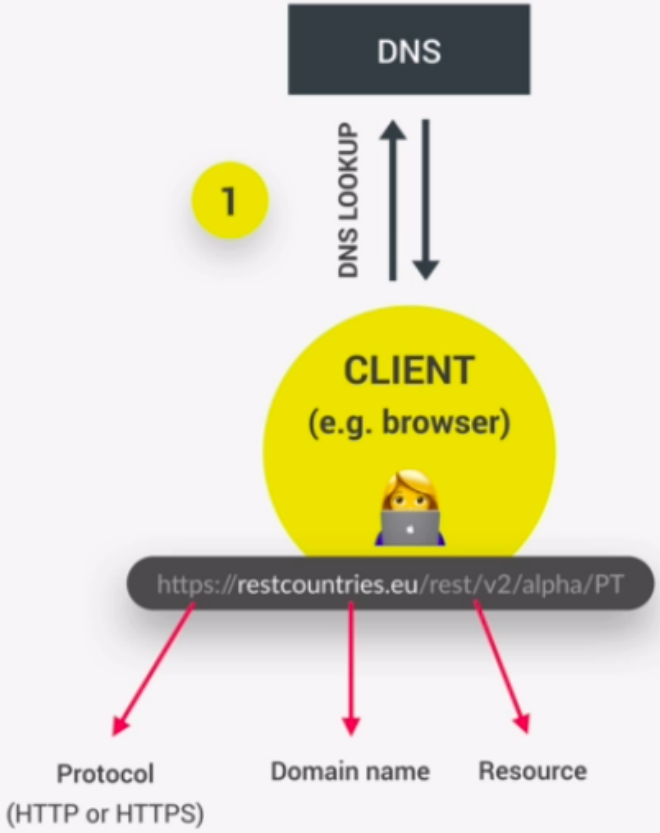


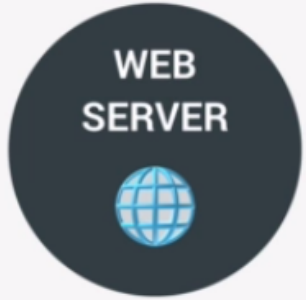
WHAT HAPPENS WHEN WE ACCESS A WEB SERVER

👉 **Request-response model** or **Client-server architecture**

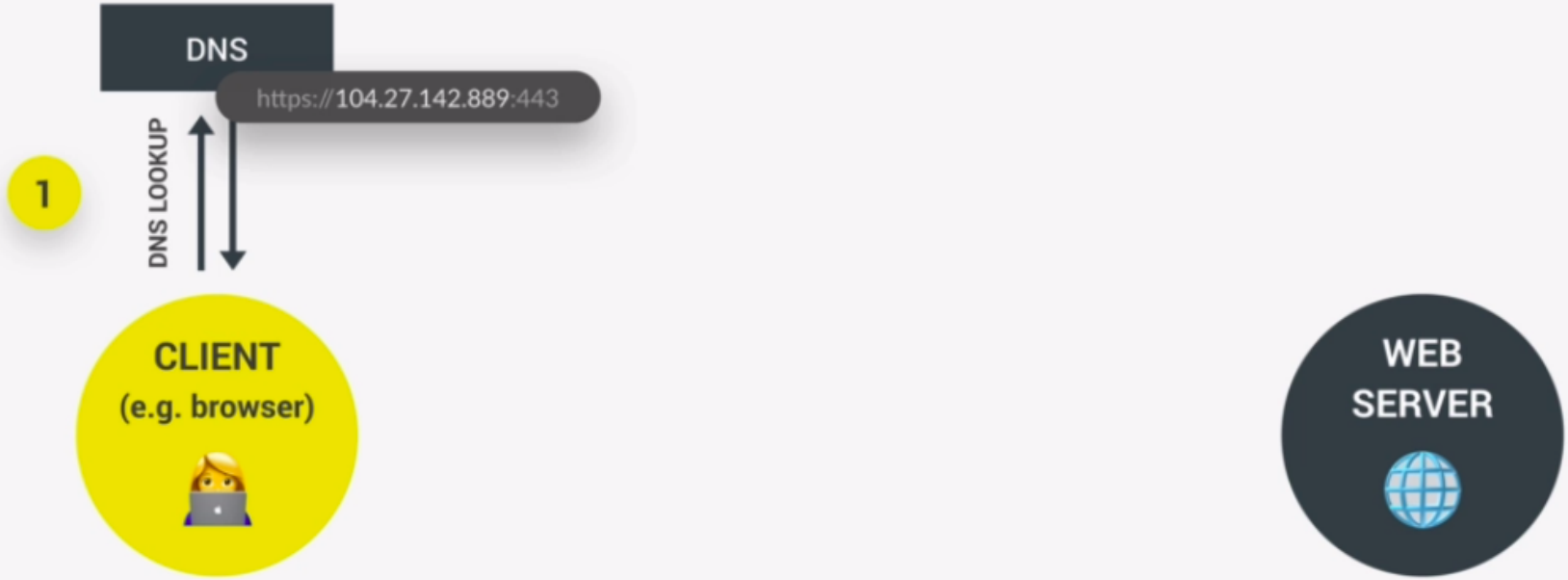


WHAT HAPPENS WHEN WE ACCESS A WEB SERVER

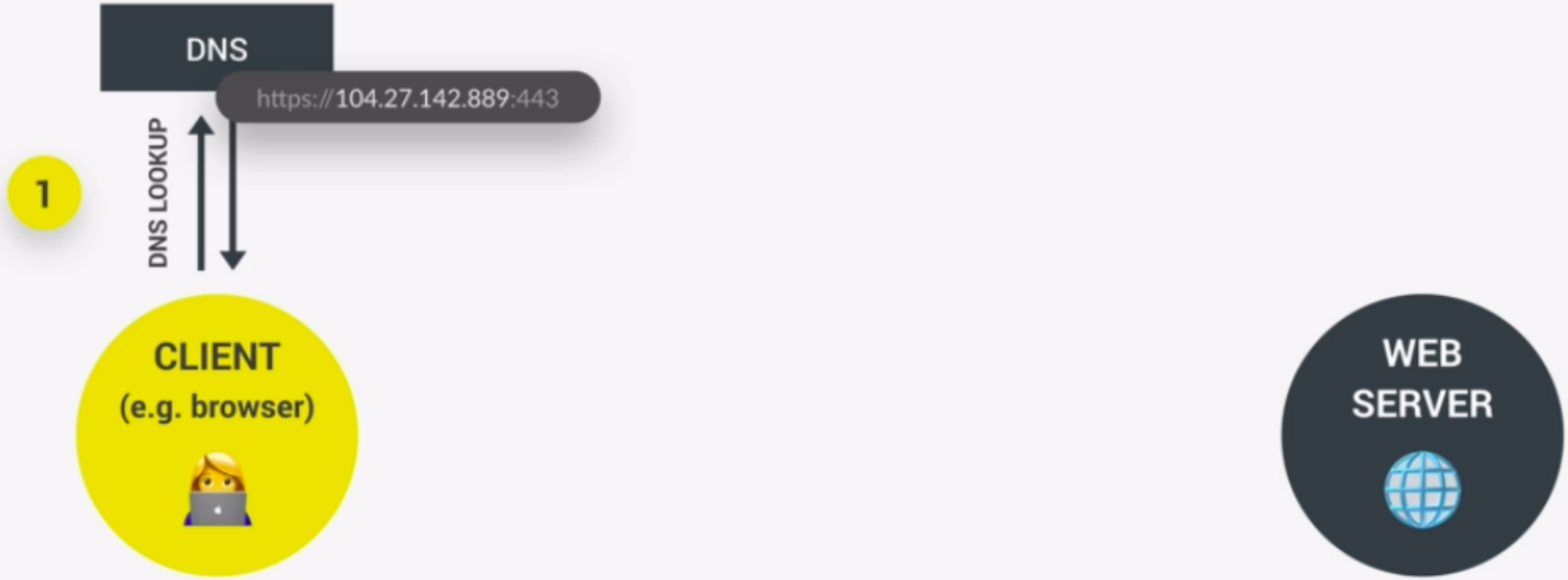




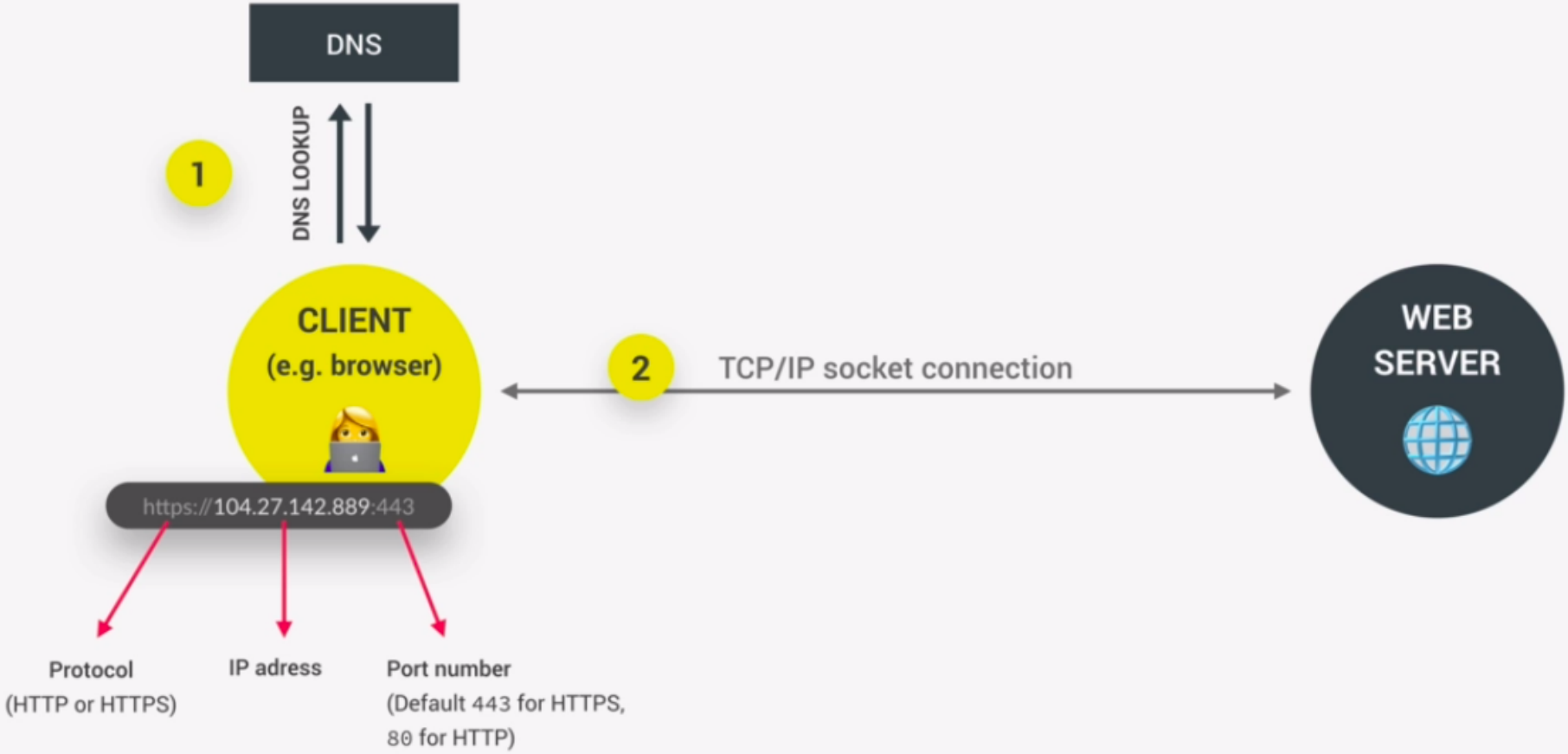
WHAT HAPPENS WHEN WE ACCESS A WEB SERVER



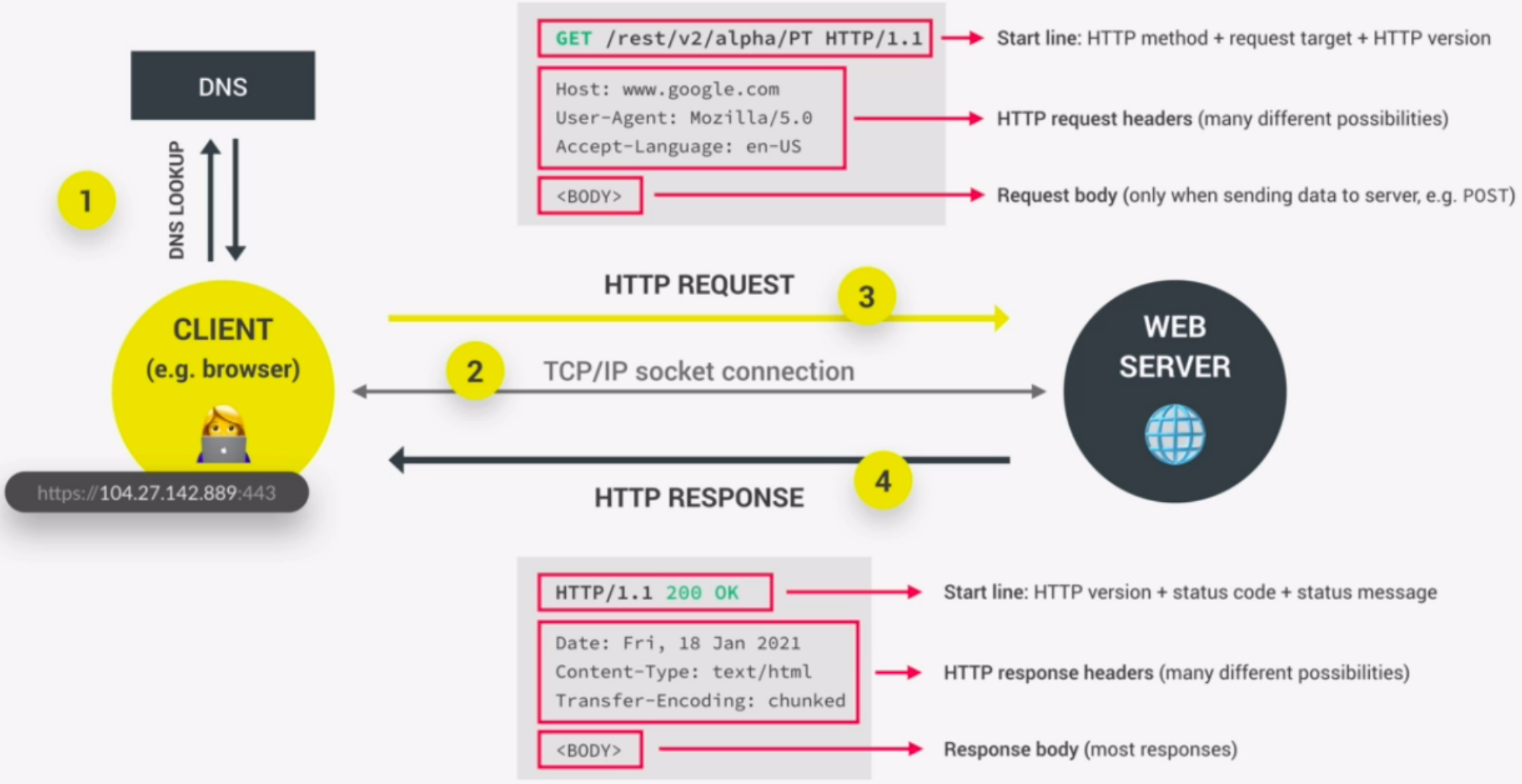
WHAT HAPPENS WHEN WE ACCESS A WEB SERVER



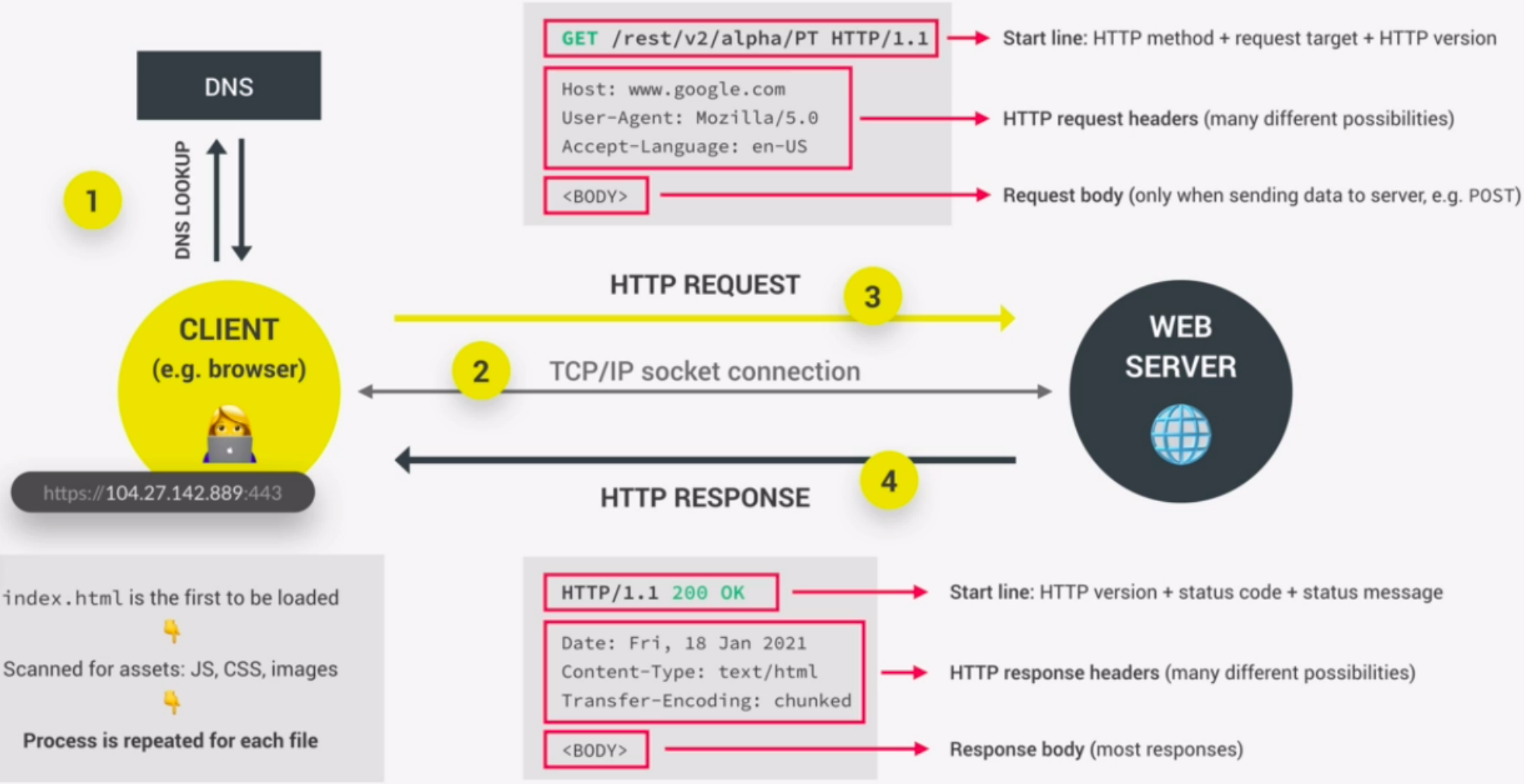
WHAT HAPPENS WHEN WE ACCESS A WEB SERVER



WHAT HAPPENS WHEN WE ACCESS A WEB SERVER



WHAT HAPPENS WHEN WE ACCESS A WEB SERVER



WHAT ARE PROMISES?

PROMISE

- 👉 **Promise:** An object that is used as a placeholder for the future result of an asynchronous operation.

  ↓ Less formal

- 👉 **Promise:** A container for an asynchronously delivered value.

  ↓ Less formal

- 👉 **Promise:** A container for a future value.

  ← **Example: Response from AJAX call**

- 👉 We no longer need to rely on events and callbacks passed into asynchronous functions to handle asynchronous results;

- 👉 Instead of nesting callbacks, we can **chain promises** for a sequence of asynchronous operations: **escaping callback hell** 🎉

Promise that I will receive money if I guess correct outcome

☝️ I buy lottery ticket (promise) right now

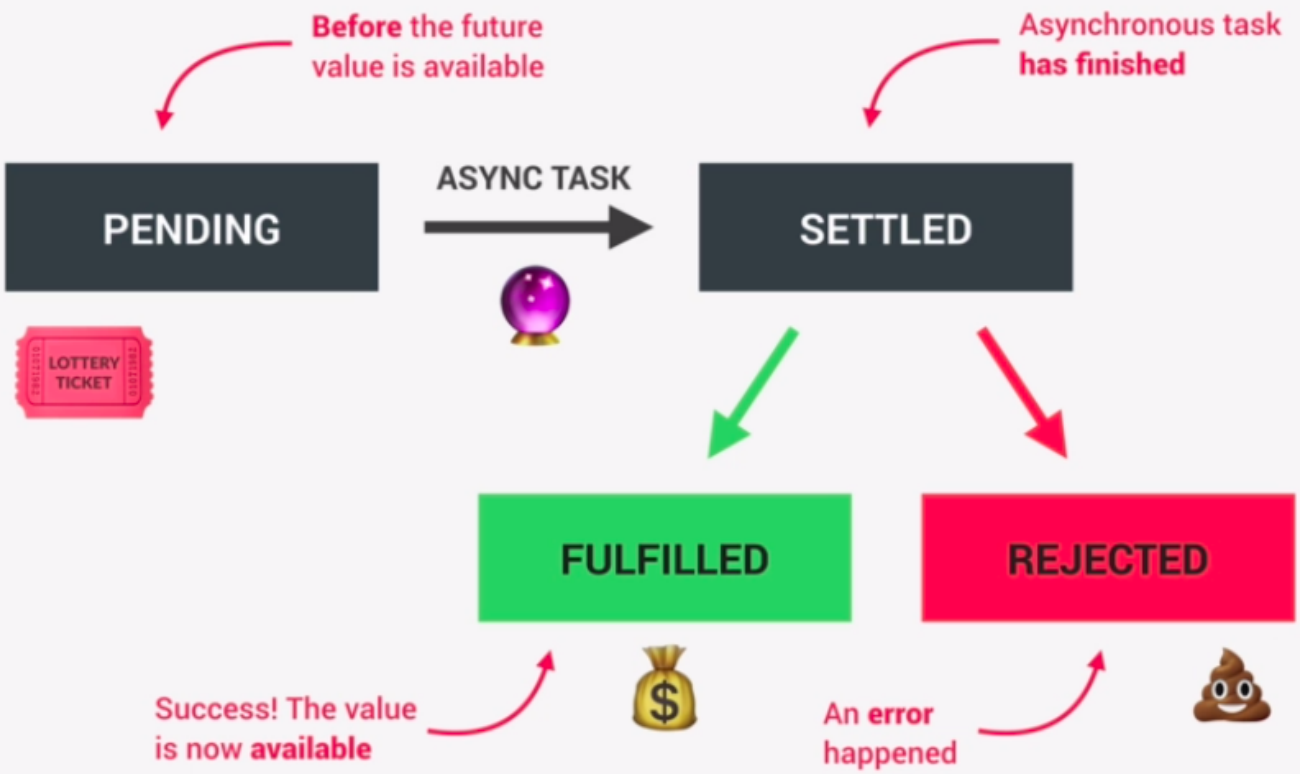
↓

🔮 Lottery draw happens asynchronously

↓

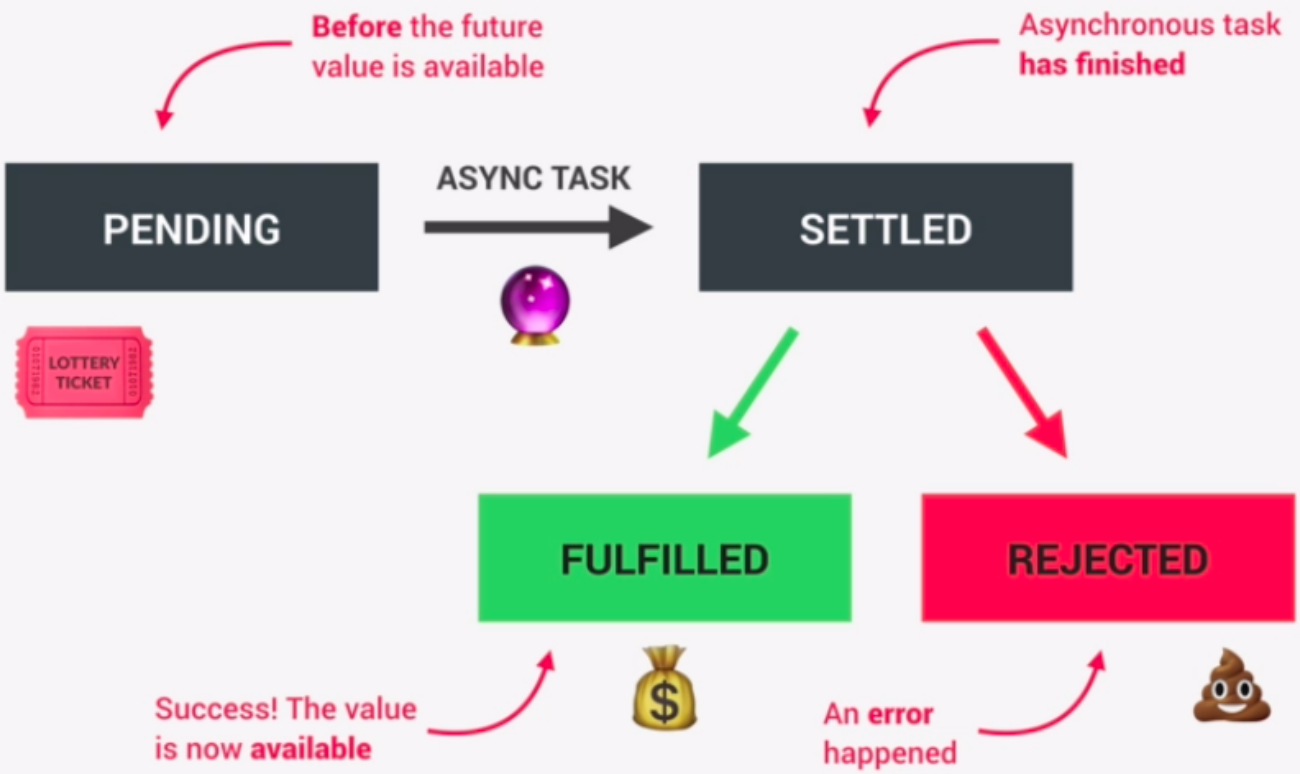
💰 If correct outcome, I receive money, because it was promised

THE PROMISE LIFECYCLE

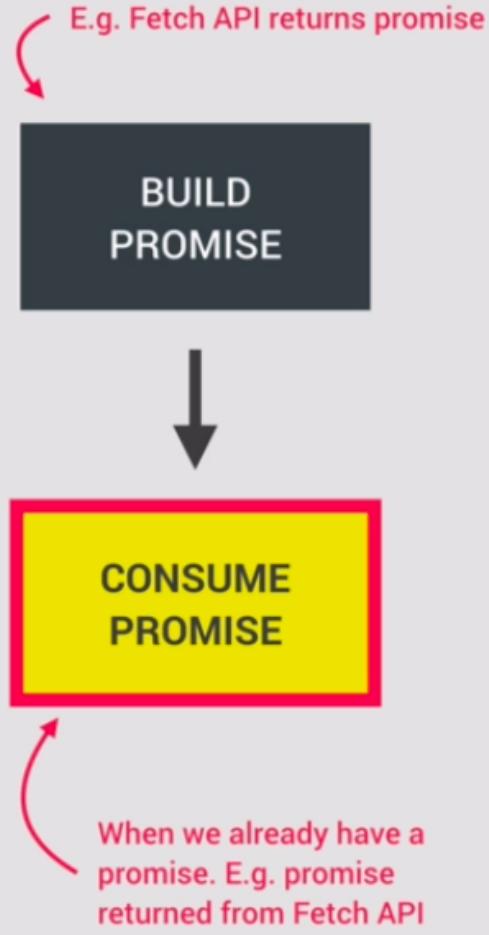


👉 We are able **handle** these different states in our code!

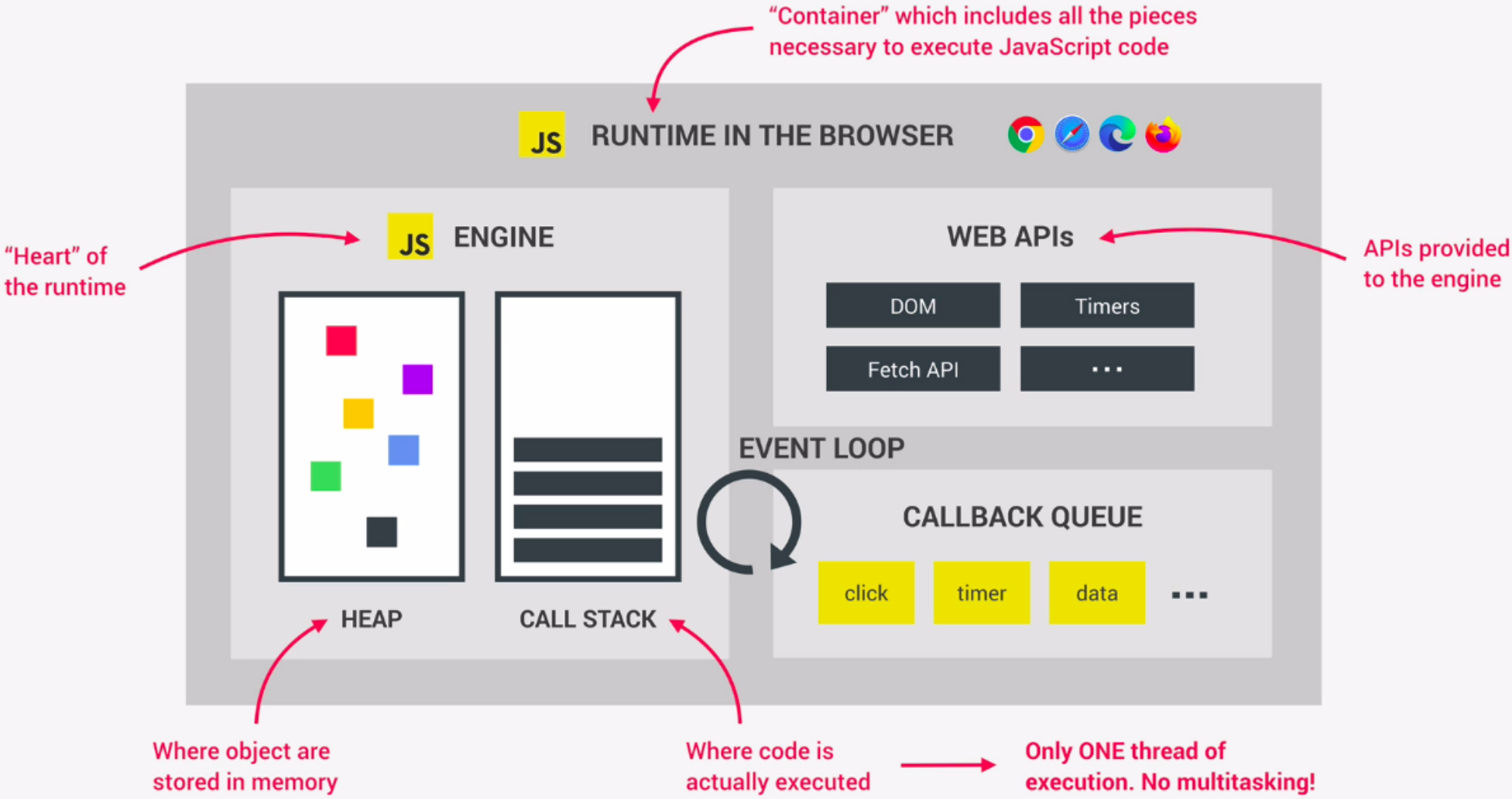
THE PROMISE LIFECYCLE



👉 We are able **handle** these different states in our code!

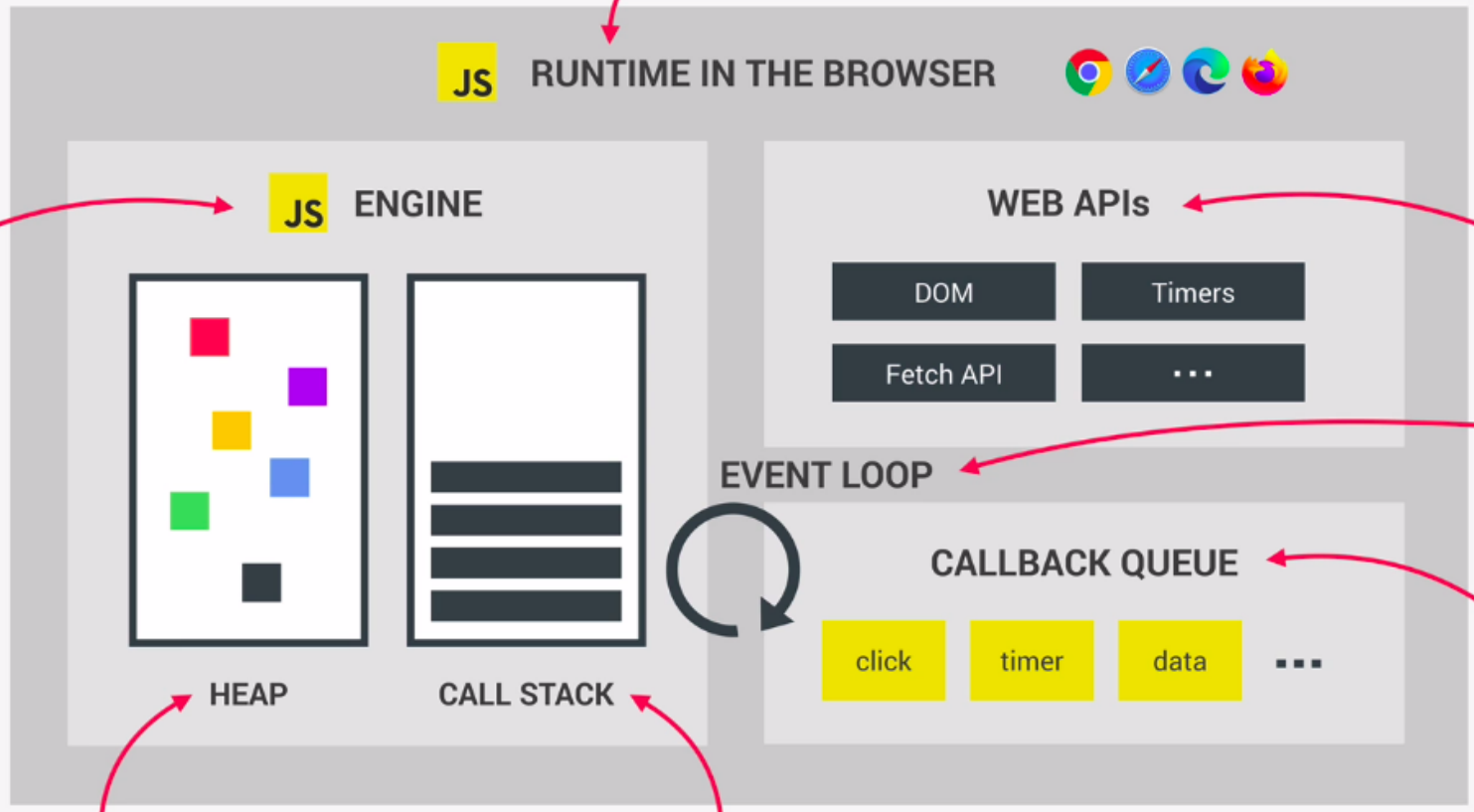


REVIEW: JAVASCRIPT RUNTIME



REVIEW: JAVASCRIPT RUNTIME

👉 **Concurrency model**: How JavaScript handles multiple tasks happening at the same time.



HOW ASYNCHRONOUS JAVASCRIPT WORKS BEHIND THE SCENES

Where asynchronous tasks run

WEB APIs

fetch()

setTimeout()

DOM()

...

Loading image

```
el = document.querySelector('img');
el.src = 'dog.jpg';
el.addEventListener('load', () => {
  el.classList.add('fadeIn');
});

fetch('https://someurl.com/api')
  .then(res => console.log(res));

// More code ...
```

Global Execution context

CALL STACK

EVENT LOOP

CALLBACK QUEUE

How can **asynchronous** code be executed in a **non-blocking way**, if there is **only one thread** of execution in the engine?

HOW ASYNCHRONOUS JAVASCRIPT WORKS BEHIND THE SCENES

Where asynchronous tasks run

WEB APIs

Loading image

```
() ⇒ {
  el.classList.add('fadeIn');
}
```

fetch()

setTimeout()

DOM()

...

```
el = document.querySelector('img');
el.src = 'dog.jpg';
el.addEventListener('load', () ⇒ {
  el.classList.add('fadeIn');
});

fetch('https://someurl.com/api')
  .then(res ⇒ console.log(res));

// More code ...
```

Execution context

addEventListener()

Global Execution context

CALL STACK

EVENT LOOP

CALLBACK QUEUE

🤔 How can **asynchronous** code be executed in a **non-blocking way**, if there is **only one thread** of execution in the engine?

HOW ASYNCHRONOUS JAVASCRIPT WORKS BEHIND THE SCENES

Where asynchronous tasks run

WEB APIs

Loading image

```
() ⇒ {
  el.classList.add('fadeIn');
}
```

fetch()

setTimeout()

DOM()

...

```
el = document.querySelector('img');
el.src = 'dog.jpg';
el.addEventListener('load', () ⇒ {
  el.classList.add('fadeIn');
});

fetch('https://someurl.com/api')
  .then(res ⇒ console.log(res));

// More code ...
```

Execution context

fetch()

Global Execution context

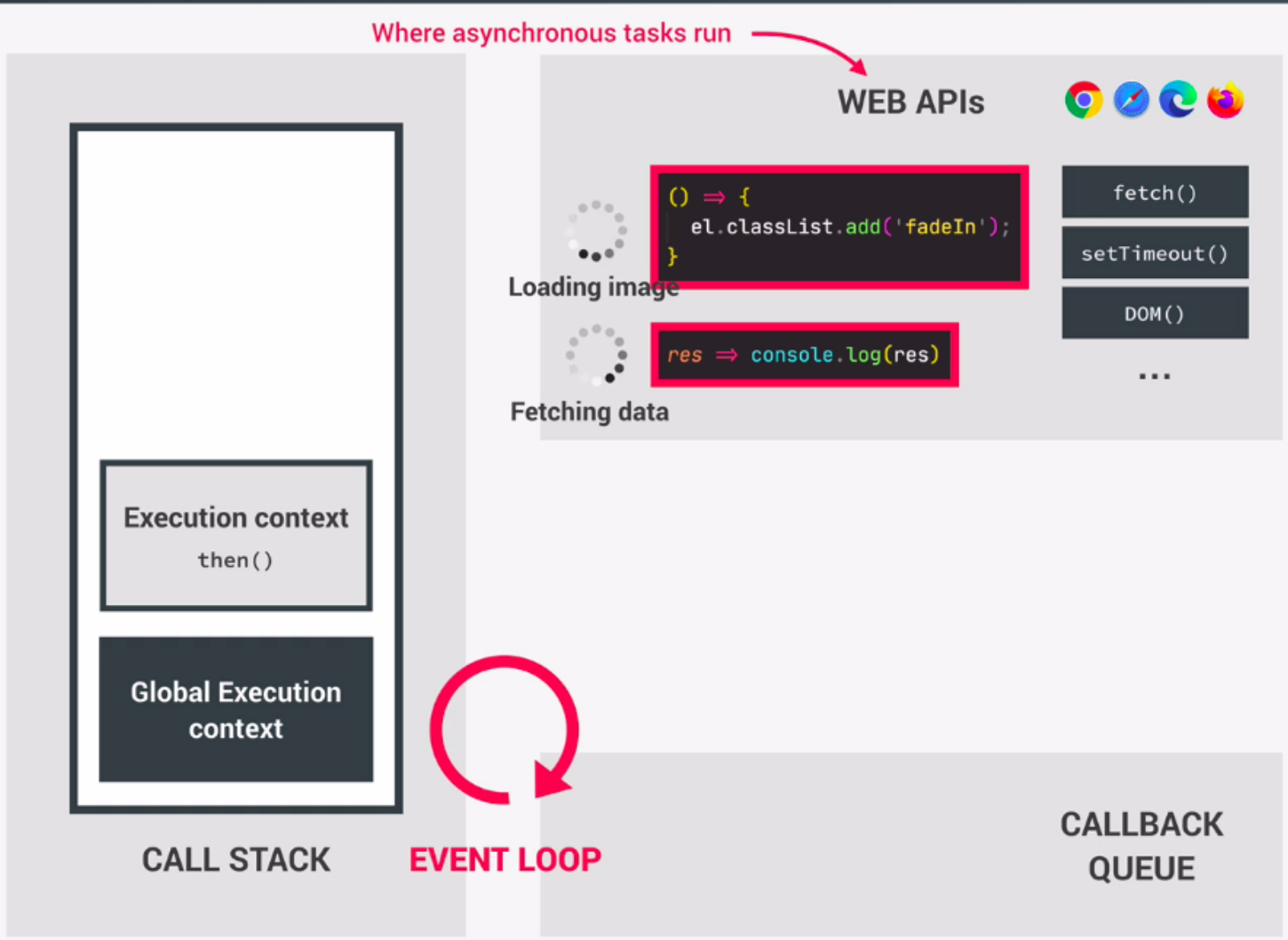
CALL STACK

EVENT LOOP

CALLBACK QUEUE

How can **asynchronous** code be executed in a **non-blocking way**, if there is **only one thread** of execution in the engine?

HOW ASYNCHRONOUS JAVASCRIPT WORKS BEHIND THE SCENES




```
el = document.querySelector('img');
el.src = 'dog.jpg';
el.addEventListener('load', () ⇒ {
  el.classList.add('fadeIn');
});

fetch('https://someurl.com/api')
  .then(res ⇒ console.log(res));

// More code ...
```

🤔 How can **asynchronous** code be executed in a **non-blocking way**, if there is **only one thread** of execution in the engine?

HOW ASYNCHRONOUS JAVASCRIPT WORKS BEHIND THE SCENES

Where asynchronous tasks run

WEB APIs

Loading image

```
() ⇒ {
  el.classList.add('fadeIn');
}
```

Fetching data

```
res ⇒ console.log(res)
```

fetch()

setTimeout()

DOM()

...

```
el = document.querySelector('img');
el.src = 'dog.jpg';
el.addEventListener('load', () ⇒ {
  el.classList.add('fadeIn');
});

fetch('https://someurl.com/api')
  .then(res ⇒ console.log(res));

// More code ...
```

Global Execution context

CALL STACK

EVENT LOOP

CALLBACK QUEUE

How can **asynchronous** code be executed in a **non-blocking way**, if there is **only one thread** of execution in the engine?

HOW ASYNCHRONOUS JAVASCRIPT WORKS BEHIND THE SCENES

Where asynchronous tasks run

WEB APIs

fetch()

setTimeout()

DOM()

...

Fetching data

```
res ⇒ console.log(res)
```

```
el = document.querySelector('img');
el.src = 'dog.jpg';
el.addEventListener('load', () ⇒ {
  el.classList.add('fadeIn');
});

fetch('https://someurl.com/api')
  .then(res ⇒ console.log(res));

// More code ...
```

Global Execution context

CALL STACK

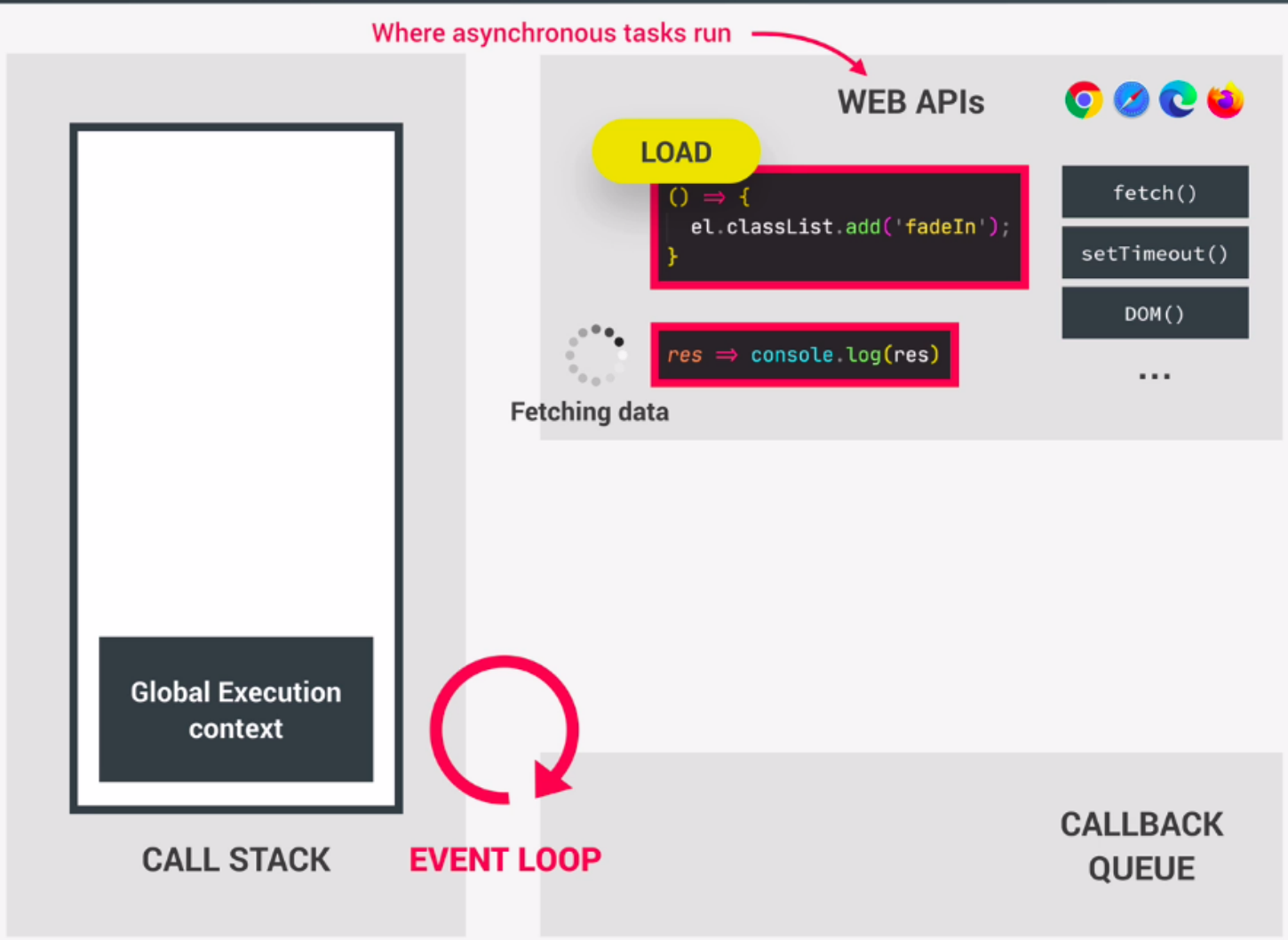
EVENT LOOP

```
() ⇒ {
  el.classList.add('fadeIn');
}
```

CALLBACK QUEUE

🤔 How can **asynchronous** code be executed in a **non-blocking way**, if there is **only one thread** of execution in the engine?

HOW ASYNCHRONOUS JAVASCRIPT WORKS BEHIND THE SCENES




```
el = document.querySelector('img');
el.src = 'dog.jpg';
el.addEventListener('load', () ⇒ {
  el.classList.add('fadeIn');
});

fetch('https://someurl.com/api')
  .then(res ⇒ console.log(res));

// More code ...
```

🤔 How can **asynchronous** code be executed in a **non-blocking way**, if there is **only one thread** of execution in the engine?

HOW ASYNCHRONOUS JAVASCRIPT WORKS BEHIND THE SCENES

Where asynchronous tasks run

WEB APIs

fetch()

setTimeout()

DOM()

...

Fetching data

```
res ⇒ console.log(res)
```

```
el = document.querySelector('img');
el.src = 'dog.jpg';
el.addEventListener('load', () ⇒ {
  el.classList.add('fadeIn');
});

fetch('https://someurl.com/api')
  .then(res ⇒ console.log(res));

// More code ...
```

Global Execution context

CALL STACK

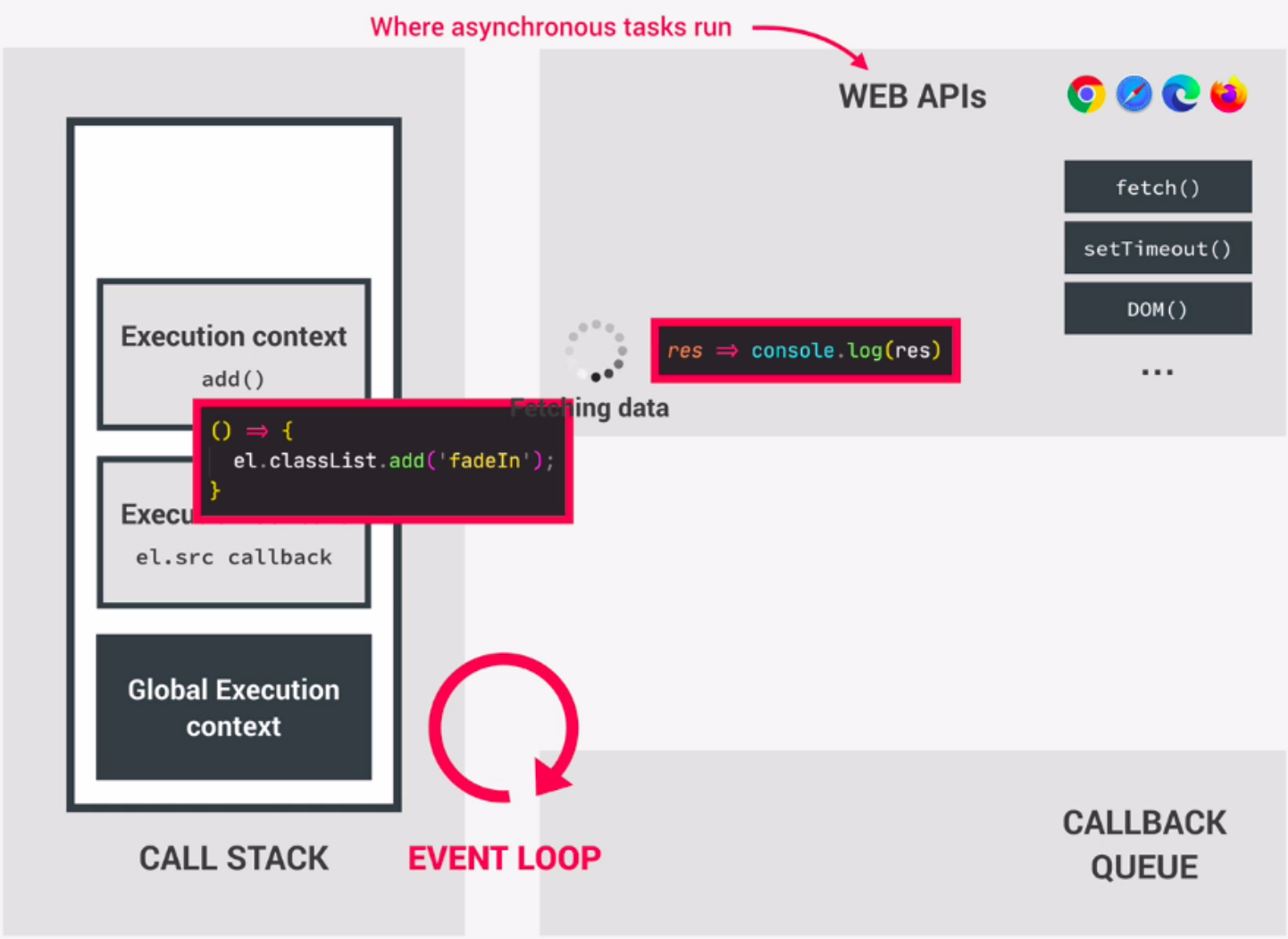
EVENT LOOP

```
() ⇒ {
  el.classList.add('fadeIn');
}
```

CALLBACK QUEUE

How can **asynchronous** code be executed in a **non-blocking way**, if there is **only one thread** of execution in the engine?

HOW ASYNCHRONOUS JAVASCRIPT WORKS BEHIND THE SCENES



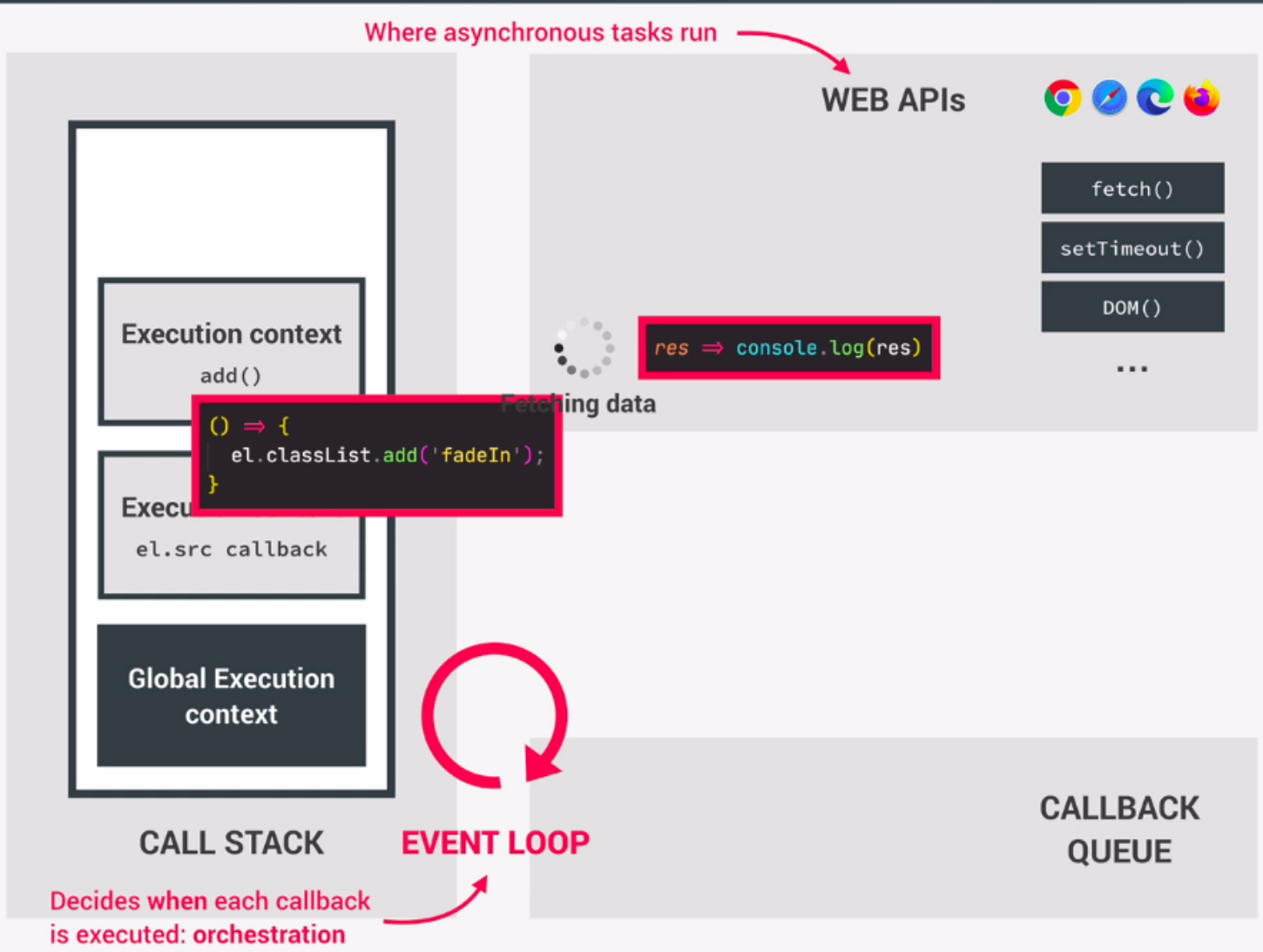

```
el = document.querySelector('img');
el.src = 'dog.jpg';
el.addEventListener('load', () ⇒ {
  el.classList.add('fadeIn');
});

fetch('https://someurl.com/api')
  .then(res ⇒ console.log(res));

// More code ...
```

🤔 How can **asynchronous** code be executed in a **non-blocking way**, if there is **only one thread** of execution in the engine?

HOW ASYNCHRONOUS JAVASCRIPT WORKS BEHIND THE SCENES




```
el = document.querySelector('img');
el.src = 'dog.jpg';
el.addEventListener('load', () ⇒ {
  el.classList.add('fadeIn');
});

fetch('https://someurl.com/api')
  .then(res ⇒ console.log(res));

// More code ...
```

🤔 How can **asynchronous** code be executed in a **non-blocking way**, if there is **only one thread** of execution in the engine?

HOW ASYNCHRONOUS JAVASCRIPT WORKS BEHIND THE SCENES

Where asynchronous tasks run

WEB APIs

fetch()

setTimeout()

DOM()

...

```
res ⇒ console.log(res)
```

```
el = document.querySelector('img');
el.src = 'dog.jpg';
el.addEventListener('load', () ⇒ {
  el.classList.add('fadeIn');
});

fetch('https://someurl.com/api')
  .then(res ⇒ console.log(res));

// More code ...
```

Global Execution context

CALL STACK

EVENT LOOP

Decides when each callback is executed: **orchestration**

CALLBACK QUEUE

🤔 How can **asynchronous** code be executed in a **non-blocking way**, if there is **only one thread** of execution in the engine?

HOW ASYNCHRONOUS JAVASCRIPT WORKS BEHIND THE SCENES

Where asynchronous tasks run

WEB APIs

fetch()

setTimeout()

DOM()

...

```
el = document.querySelector('img');
el.src = 'dog.jpg';
el.addEventListener('load', () => {
  el.classList.add('fadeIn');
});

fetch('https://someurl.com/api')
  .then(res => console.log(res));

// More code ...
```

Global Execution context

CALL STACK

EVENT LOOP

Decides when each callback is executed: orchestration

```
res => console.log(res)
```

MICROTASKS QUEUE

CALLBACK QUEUE

How can **asynchronous** code be executed in a **non-blocking way**, if there is **only one thread** of execution in the engine?

HOW ASYNCHRONOUS JAVASCRIPT WORKS BEHIND THE SCENES

Where asynchronous tasks run

WEB APIs

fetch()

setTimeout()

DOM()

...

```
el = document.querySelector('img');
el.src = 'dog.jpg';
el.addEventListener('load', () => {
  el.classList.add('fadeIn');
});

fetch('https://someurl.com/api')
  .then(res => console.log(res));

// More code ...
```

Like callback queue, but for callbacks related to **promises**. Has **priority** over callback queue!

```
res => console.log(res)
```

MICROTASKS QUEUE

Global Execution context

CALL STACK

EVENT LOOP

CALLBACK QUEUE

Decides when each callback is executed: **orchestration**

🤔 How can **asynchronous** code be executed in a **non-blocking way**, if there is **only one thread** of execution in the engine?

HOW ASYNCHRONOUS JAVASCRIPT WORKS BEHIND THE SCENES

Where asynchronous tasks run

WEB APIs

fetch()

setTimeout()

DOM()

...

```
el = document.querySelector('img');
el.src = 'dog.jpg';
el.addEventListener('load', () => {
  el.classList.add('fadeIn');
});

fetch('https://someurl.com/api')
  .then(res => console.log(res));

// More code ...
```

Execution context
log()

Execution context
fetch callback

`res => console.log(res)`

Global Execution context

CALL STACK

EVENT LOOP

Decides when each callback is executed: **orchestration**

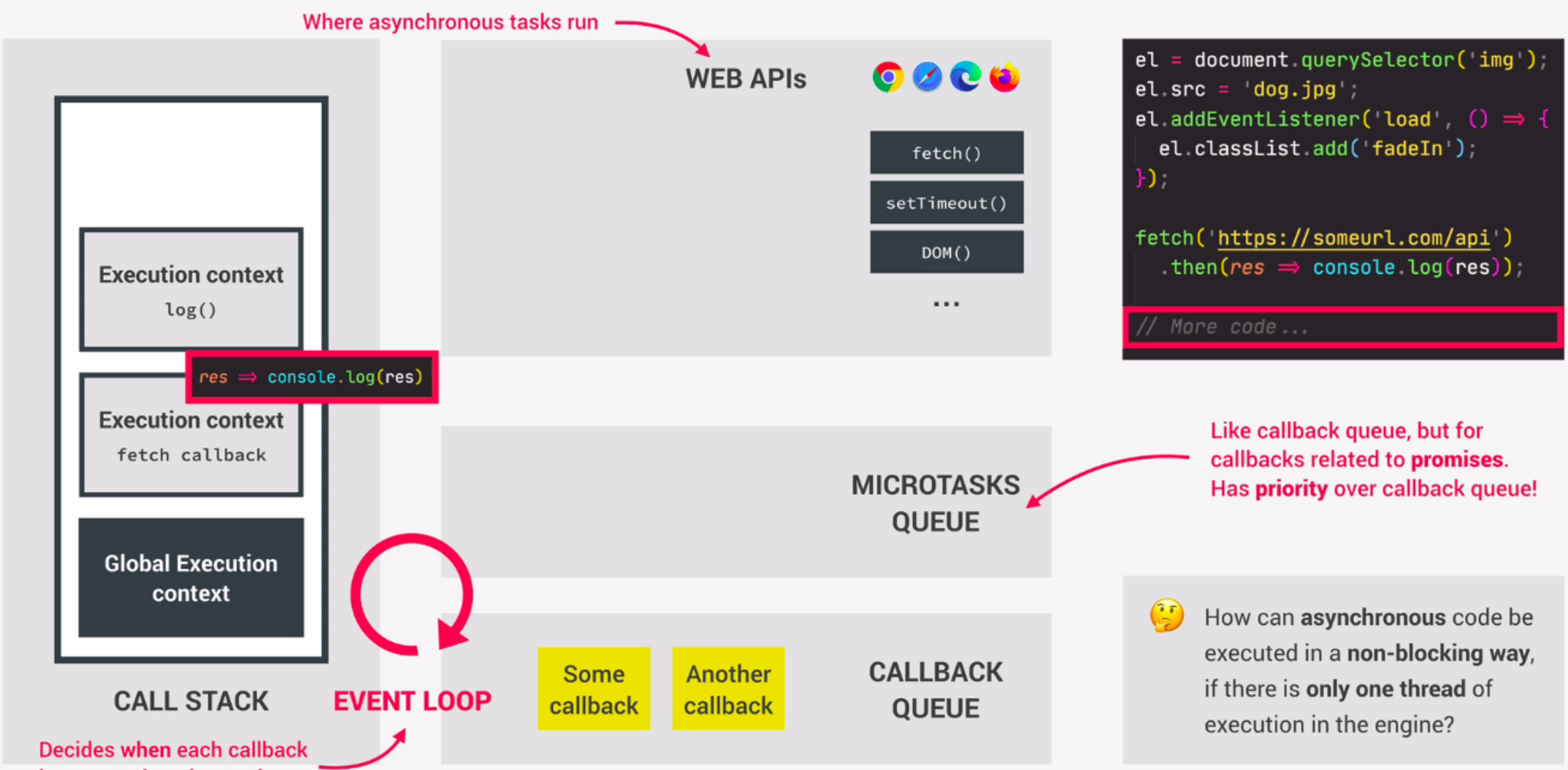
MICROTASKS QUEUE

Like callback queue, but for callbacks related to **promises**. Has **priority** over callback queue!

CALLBACK QUEUE

🤔 How can **asynchronous** code be executed in a **non-blocking way**, if there is **only one thread** of execution in the engine?

HOW ASYNCHRONOUS JAVASCRIPT WORKS BEHIND THE SCENES




```
el = document.querySelector('img');
el.src = 'dog.jpg';
el.addEventListener('load', () ⇒ {
  el.classList.add('fadeIn');
});

fetch('https://someurl.com/api')
  .then(res ⇒ console.log(res));

// More code ...
```

HOW ASYNCHRONOUS JAVASCRIPT WORKS BEHIND THE SCENES

Where asynchronous tasks run

WEB APIs

fetch()

setTimeout()

DOM()

...

```
el = document.querySelector('img');
el.src = 'dog.jpg';
el.addEventListener('load', () => {
  el.classList.add('fadeIn');
});

fetch('https://someurl.com/api')
  .then(res => console.log(res));

// More code ...
```

Global Execution context

CALL STACK

EVENT LOOP

Decides when each callback is executed: **orchestration**

MICROTASKS QUEUE

Like callback queue, but for callbacks related to **promises**. Has **priority** over callback queue!

Some callback

Another callback

CALLBACK QUEUE

How can **asynchronous** code be executed in a **non-blocking way**, if there is **only one thread** of execution in the engine?