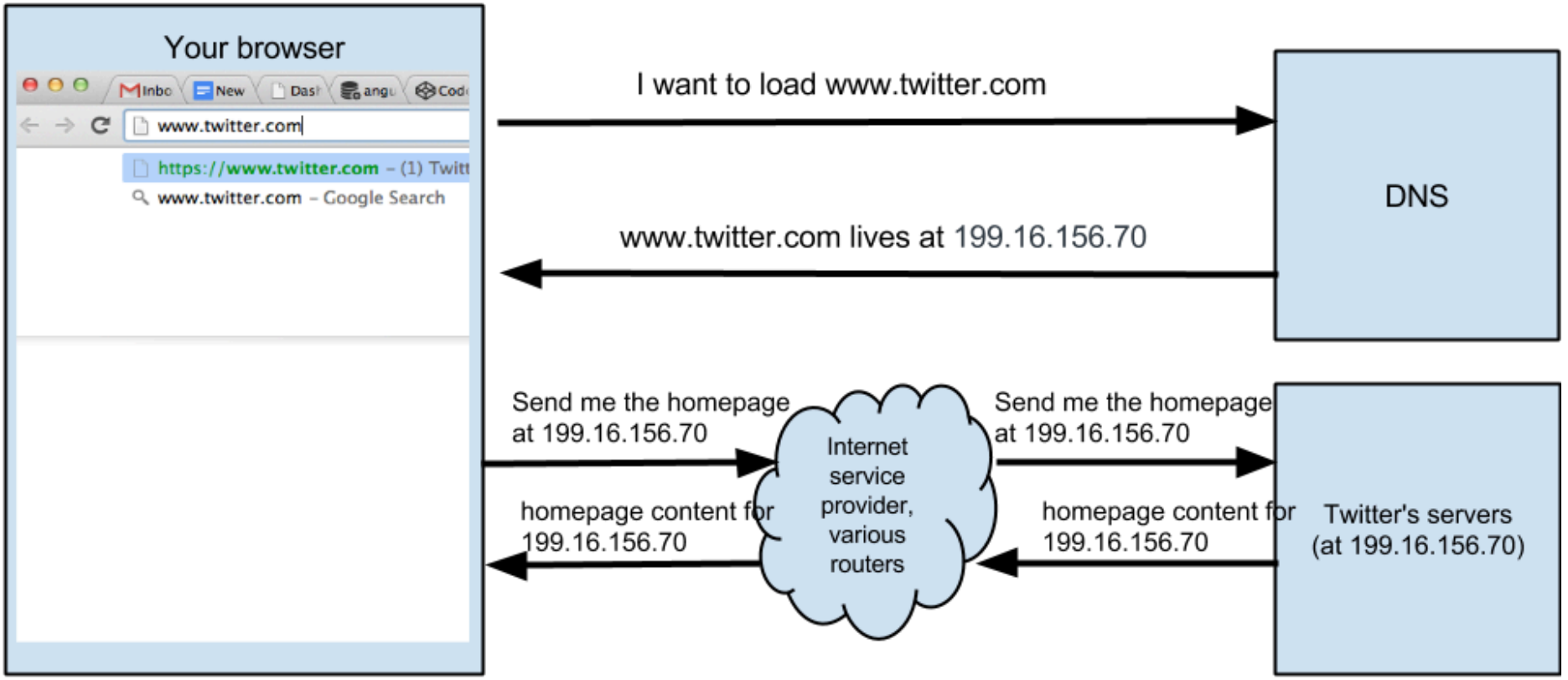
{POWER.CODERS}

AJAX and JSON

Contents

HTTP/HTTPS


Your browser
www.twitter.com
https://www.twitter.com – (1) Twitt
www.twitter.com – Google Search
I want to load www.twitter.com
DNS
www.twitter.com lives at 199.16.156.70
Send me the homepage at 199.16.156.70
Internet service provider, various routers
Send me the homepage at 199.16.156.70
homepage content for 199.16.156.70
homepage content for 199.16.156.70
Twitter's servers (at 199.16.156.70)

Communication between computers

- Computers communicate with each other with **HTTP/HTTPS**.
- Computers can be **clients** or **servers**.
- The client sends a **request** to the server.
- The server sends a **response** back to the client.

What is HTTP?

- **h**yper**t**ext **t**ransfer **p**rotocol
- Protocol used for websites to transfer HTML, CSS, JS, images and text

HTTP REQUESTS

- **GET**: I receive the Twitter feed with all tweets from today
- **POST**: I create a new user which is added to the server
- **PUT**: I edit a tweet I made before
- **DELETE**: I delete a tweet or my user account

HTTP RESPONSES

- **Status Code**, e.g. 200 (OK), 404 (Not found), 500 (Server error)
- **Data**, e.g. HTML, CSS, JS, images and more

HTTPS

- **h**yper**t**ext **t**ransfer **p**rotocol **s**ecure
- The data is encrypted between client and server using a secret key.
- The technology used is today the transport layer security (TLS) and before was the secure sockets layer (SSL).

JSON

Data formats

Standard formats to send data over the internet and receive it

- Plain text
- HTML
- XML
- JSON

XML

e**X**tensible **M**arkup **L**anguage

`XML` uses tags to define its structure similar to `HTML` and `SVG`.

Tags are not predefined, you can specify your own tags.

Nested tags are possible.

XML EXAMPLE

```xml
<person>
  <name>Susanne</name>
  <surname>Koenig</surname>
  <nationality>German</nationality>
  <languages>
    <language>German</language>
    <language>English</language>
  </languages>
</person>
```

JSON

Java**S**cript **O**bject **N**otation

`JSON` is a lightweight, readable format for structuring data.

It is a modern alternative to **XML** to transmit data from server to client.

JSON is a string whose format very much resembles JavaScript objects.

JSON EXAMPLE

```
{
  "name"        : "Susanne",
  "surname"     : "Koenig",
  "nationality" : "German",
  "languages"   : ["German", "English"]
}
```

Like a JS object JSON maps **keys** to **values**.

Always use **double quotes** `""` in JSON.

Why JSON?

- JSON is easier to parse and use with JS
- JSON saves bandwidth
- JSON improves the response time
- JSON is the standard today

JSON in JavaScript

```
const obj = JSON.parse('{"name": "Susanne","surname": "Koenig"}');

let myJSON = JSON.stringify(obj);
```

`JSON.parse()` is a built-in JavaScript method to turn **JSON** into a **JavaScript object**.

`JSON.stringify()` is a built-in JavaScript method to turn a **JavaScript object** into a **JSON**.

AJAX

AJAX

HTTP can also fetch only one part of documents to update web pages on demand. Without reloading.

Google came up with that in 2006.

AJAX

Asynchronous **J**avaScript **a**nd **X**ML

It combines a group of existing technologies: HTML, CSS, JavaScript, XML, JSON, etc.

Together this group can build modern applications.

JavaScript

- initiates the AJAX request
- parses the AJAX response
- updates the DOM

The old way

`XMLHttpRequest`

Also known as **XHR API** is used to make a request to a server.

API: Application Programming Interface is a set of methods which specify the rules of communication between two interested parties.

Note: The incoming data does not need to be in `XML`.

XHR EXAMPLE

```
var request = new XMLHttpRequest();

request.open('GET', '/path/to/api', true);
request.setRequestHeader('Content-type', 'application/json'); // Not for text + HTML

request.onload = function() {
  if (request.status >= 200 && request.status < 400) {
    console.log(JSON.parse(request.responseText));
  } else {
    // We reached our target server, but it returned an error
  }
});

request.onerror = function() {
  // There was a connection error of some sort
```

The new way

FETCH API

```
fetch("https://jsonplaceholder.typicode.com/todos/1")
  .then(response => response.json())
  .then(data => console.log(data));
```

`fetch`

1. returns a **promise**: I promise to let you know when the response of my request is returned
2. then it returns a **response**: with its own method `json()` to parse the response
3. then it returns a **object**: with the data of your AJAX call

Promises

new in ES6

A **promise** is an **object** that may produce a single value some time **in the future**. Either a **resolve** value, or a reason why it's not resolved (**rejected**).

```
const promise = new Promise((resolve, reject) => {
            let number = 6 + 4;
            if(number === 10) {
              resolve("Promise was fulfilled");
            } else {
              reject();
            }
          });

          promise
            .then(result => console.log(result))
            .catch(() => alert("Promise was rejected"));
```

Callbacks

```
button.addEventListener("click", submitForm);
```

Once the button is clicked, the function submitForm will be called.

Nested callbacks

```
movePlayer(100, "left", function(){
  movePlayer(200, "right", function(){
    movePlayer(400, "left", function(){

    });
  });
});
```

Once the player moved 100 steps to the left and that is done, the player should move 200 steps to the right and then this move is done, the player moves 400 steps to the left again.

= pyramid of doom (repetition of code, difficult to read)

MOVEPLAYER WITH PROMISES

```
movePlayer(100, "left")
  .then(() => movePlayer(200, "right"))
  .then(() => movePlayer(400, "left"));
```

Better, but still hard to read

Different syntax

```
const promise = new Promise((resolve, reject) => {
  let number = 6 + 4;
  if(number === 10) {
    resolve("Promise was fulfilled");
  } else {
    reject("Promise was rejected");
  }
});

promise.then(result => console.log(result))
```

Error handling

```
promise
  .then(result => result + "!")
  .then(result2 => result2 + "?")
  .then(result3 => {
    throw Error;
    console.log(result3)
  })
  .catch(() => console.log("error!"));
```

Promises are used for **asynchronous** JavaScript.

The dot syntax here is called **chaining**

Asynchronous JavaScript?

To understand **asynchronous** JavaScript, we need to understand **synchronous** JavaScript first.

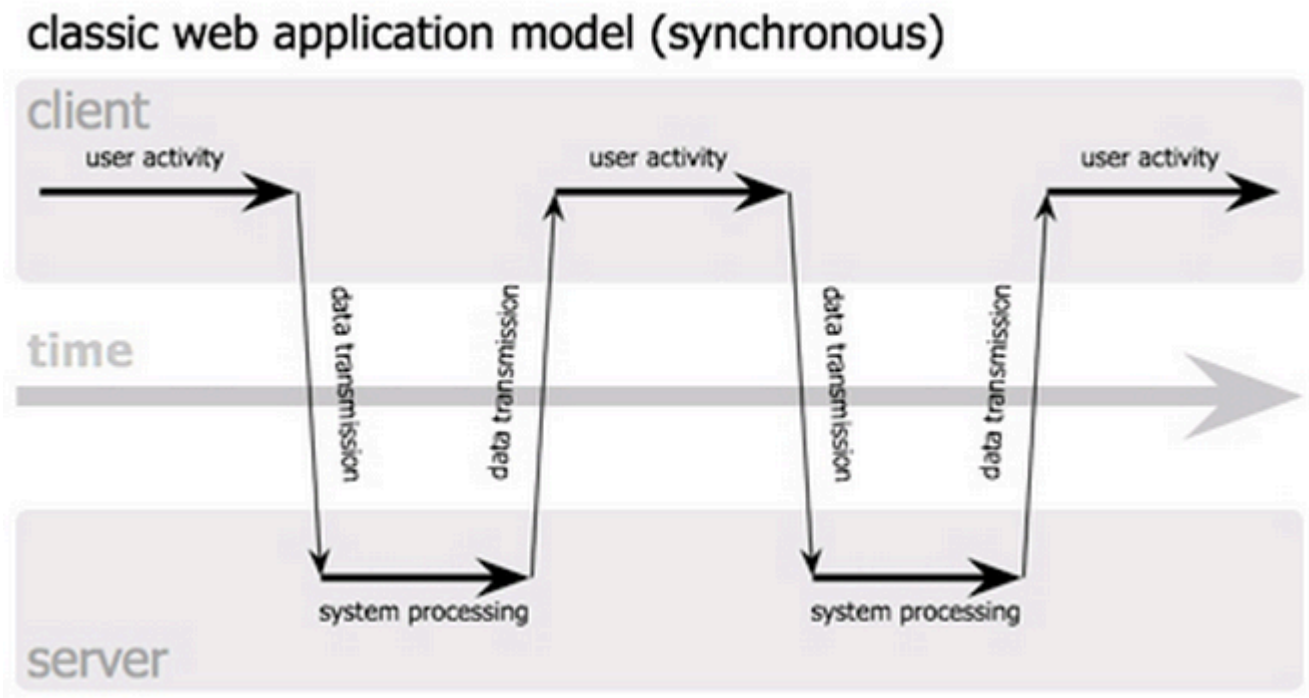
Until now all our JS was synchronous: you run some code and the result will be returned as soon as the browser can do it.

Synchronous JavaScript

While an operation is processed (calling a function for example), nothing else can happen - remember `alert`?

Reason for that is that JS is **single-threaded**. All tasks run on the main thread like pearls on a necklace.

Synchronous workflow

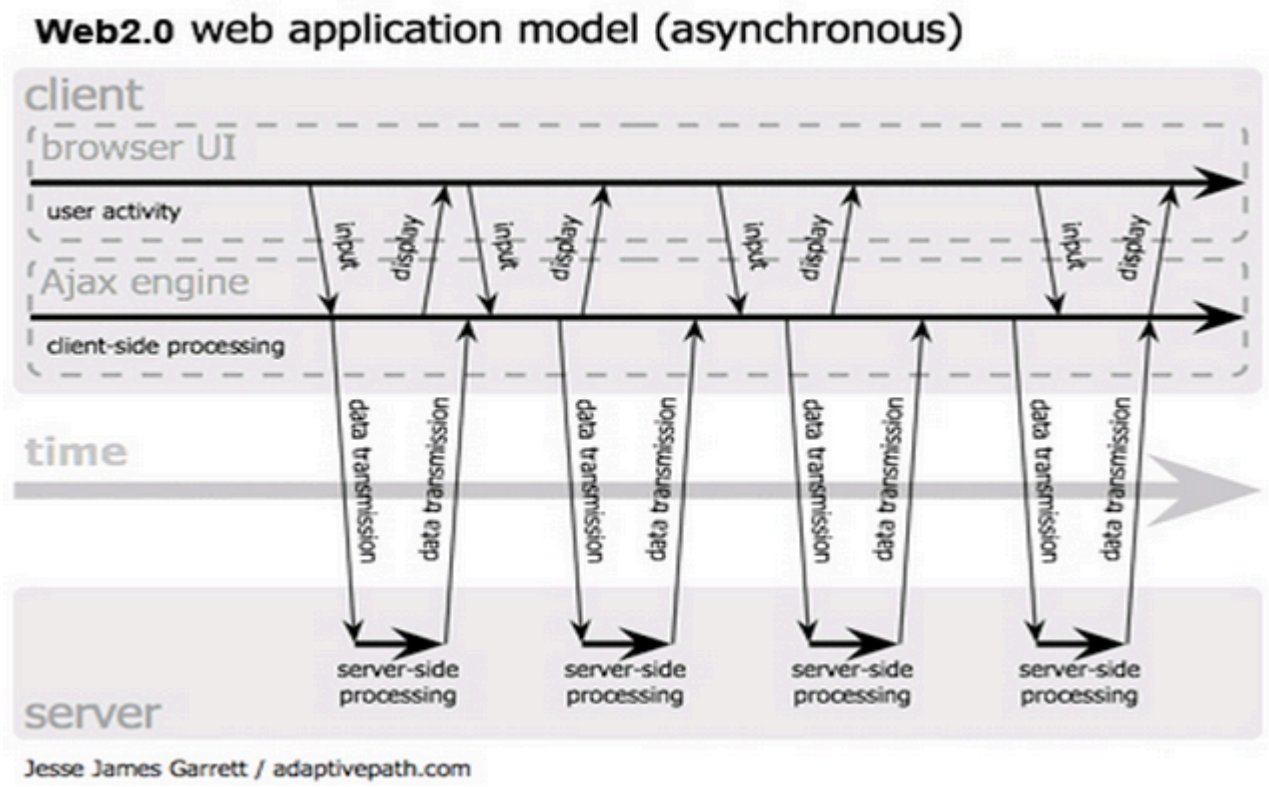


Asynchronous JavaScript

Often you need to wait inside a function for something to happen, for a **response**, before you can move on.

If you `fetch` an image from a server via JS for example, you cannot use that image right away - it has not been downloaded yet.

Asynchronous workflow



Online resources

- HTTP messages
- JSON placeholders
- JSON View extension
- Using fetch
- Fetch API and promises
- All you need to know about Promise.all()
- Async await making icecream
- Async await (ES8)
- Why Async await (ES8)
- Web Workers API
- Web workers vs Promises

What is an API?

What is an API?

Application Programme Interface

- Allows strangers to **communicate** with each other
- Allows to share **data** across machines and systems

Remember fetch API?

```
fetch("https://jsonplaceholder.typicode.com/todos/1")
  .then(response => response.json())
  .then(data => console.log(data));
```

`fetch`

1. returns a **promise**: I promise to let you know when the response of my request is returned
2. then it returns a **response**: with its own method `json()` to parse the response
3. then it returns a **object**: with the data of your AJAX call

Live coding

Remember our image slider?

Github image slider

Resources

for our live coding example

- Photos API
- Pexels API
- unsplash API (with key)
- Swiper Plugin

Free APIs

For fun and testing

- Star wars API
- Robohash API
- Fake JSON API
- Food API
- Connect Form to Email - API 1
- Connect Form to Email - API 2

Business APIs

Usually charge for use

- Google APIs, e.g. Maps
- Twilio API for messaging
- Mailchimp API for newsletters
- Stripe API for payment

Other APIs

There are so many...

- Speech recognition
- Google Text to Speech
- Clarifai API
- Aggregator to find public APIs

Code refactoring

What is code refactoring?

Rewriting code to improve it and make it cleaner is refactoring.

It is rare to design a program correctly on first go.

- Requirements can change
- Knowledge can change

Don't repeat yourself

One basic principle of code refactoring.

You can achieve that by

- Using variables to prevent duplication
- Improving event handling
- Improving class handling

Be careful when refactoring

Refactoring code always involves several steps to ensure that the working code does not break accidentally.

Break the change down in to small, independent steps that should each be safe. At each stage you **test the change** and make sure that things still work.

If they don't work then it's very easy to go back and undo the last small change you made, or review it to see what went wrong.

Online resources

- Authentication
- 8 amazing Google APIs
- W3schools Google Maps tutorial
- Official Google Maps documentation
- Cheaper alternatives