

JavaScript for Web

Course Material • Hamburg Coding School • 7.9. - 23.9.2020

Course Goal

The goal of this course is to be able to read, understand and write modern JavaScript code to work on web applications.

Outline

- Modern JavaScript
 - Arrow Functions
 - Template Literals
 - Array Destructuring
 - Object Destructuring
 - Default Values
 - Object Spreading
 - Array Spreading
 - Array.forEach()
 - Array.map()
 - Array.filter()
 - Array.reduce()
 - Constants
 - Ternary Operator
- Front-end Architectures
- JavaScript for Web Applications
 - Parts of a Web Application
 - Data as JSON
 - JSON.stringify()
- The DOM
 - Query Selectors
 - Nodes and NodeList
 - Special Accessors
 - Manipulating the DOM
 - Working with Forms
- Accessing Data from APIs
 - Load Data with fetch()

-
- Local Storage
 - Get current location with the Geolocation API
 - Display a Map with Google Maps API
 - Showing a marker
 - Handling bounds of the map
 - Handling events on a map
 - Drawing a line on a map
 - Reverse Geocoding
 - Firebase
 - Cloud Firestore
 - Cloud Storage
 - Firebase Authentication
 - More Firebase Services
 - Glossary
 - Useful Links

Modern JavaScript

In this course we will only use modern JavaScript. (In a work environment we might have to maintain code written 10 years ago, but this will not be part of the course.)

Arrow Functions

Arrow functions were introduced in ES6. They allow us to write shorter function syntax.

Before:

```
function sumOfApples(bucket1, bucket2) {  
    const sum = bucket1 + bucket2;  
    return sum;  
}
```

With arrow function:

```
const sumOfApples = (bucket1, bucket2) => {  
    const sum = bucket1 + bucket2;  
    return sum;  
}  
  
// or with implicit return:  
const sumOfApples = (bucket1, bucket2) => bucket1 + bucket2;
```

The arrow comes after the list of parameters and is followed by the function's body. It expresses something like "this input (the parameters) produces this result (the body)".

Special case:

If the arrow function only receives a single parameter you can omit the brackets:

```
const log = value => console.log('Value:', value);
```

Special case:

If the arrow function receives no parameters you use empty brackets:

```
const sayHello = () => console.log('Hello');
```

Template Literals

String concatenation is cumbersome:

```
const person = {
  name: 'Henning',
  age: 28,
  role: 'Engineer'
}
console.log(person.name + ' is ' + person.age + 'years old.')
// logs: Henning is 28 years old.
```

Instead you can use *template literals*:

```
console.log(`${person.name} is ${person.age} years old.`)
// logs: Henning is 28 years old.
```

Template literals are enclosed by the back-tick (``) (*grave accent*) character instead of double or single quotes. Template literals can contain placeholders. These are indicated by the dollar sign and curly braces: **`${expression}`**. The expressions in the placeholders and the text between the back-ticks gets passed to a function. The default function just concatenates the parts into a single string.

Array Destructuring

The ***destructuring assignment*** syntax is a JavaScript expression that makes it possible to unpack values from arrays, or properties from objects, into distinct variables.

To extract values from an array into single variables we would normally write:

```
const values = ['Henning', 28, 'Engineer'];
const name = values[0];
const profession = values[2];
```

Instead of repeatedly creating variables and assigning them we can destructure the array in a single line:

```
const [name, age, profession] = values;
```

Object Destructuring

Similarly we can destructure objects.

Instead of:

```
const person = {  
  name: 'Henning',  
  age: 28,  
  profession: 'Engineer',  
};  
const name = person.name;  
const profession = person.profession;
```

we can write:

```
const {name, profession} = person;
```

Default Values

With destructuring we can set default (or fallback) values for unknown properties, so that if an item is not in the object (or array) it will fall back to what you have set as the default.

```
const person = {  
  age: 28,  
  profession: 'Engineer',  
};  
const {name = 'Henning', age = 20} = person;  
console.log(name); // Henning - comes from the default value  
console.log(age); // 28 - comes from the object 'person'
```

Object Spreading

When we want to make a copy of an object we run into a problem if we simply use the assignment operator `=`.

```
const person = { name: 'Henning', age: 28 };  
const personCopy = person;  
copy.name = 'Nina';  
  
console.log(`${person.name} is ${person.age} years old.`);  
// prints: Nina is 28 years old.
```

By assigning **person** to a new variable we don't make a copy, we just assign the object a new reference (**personCopy**) that will now be addressed by both identifiers.

Instead we can use the **spread operator** (...).

This will allow to *spread* the object and its value into a new object:

```
const person = { name: 'Henning', age: 28 };
const person2 = { ...person }
person2.name = 'Nina'

console.log(`${person.name} is ${person.age} years old.`)
// prints: Henning is 28 years old.

console.log(`${person2.name} is ${person2.age} years old.`)
// prints: Nina is 28 years old.
```

Array Spreading

The spread operator works on arrays just as well and makes it very easy to merge to arrays into a new one:

```
const values1 = [1, 2, 3]
const values2 = [4, 5, 6]
const allValues = [...values1, ...values2]
// [1, 2, 3, 4, 5, 6]
```

This is used less often as there is also the **concat()** method that will do the same:

```
const allValues = values1.concat(values2)
```

Array.forEach()

The **forEach()** method calls a function (a **callback function**) once for each array element.

```
const colors = ['blue', 'green', 'white'];

function iterate(item) {
  console.log(item);
}

colors.forEach(iterate);
```

```
// logs "blue"
// logs "green"
// logs "white"
```

Optional knowledge:

forEach() executes the callback function with up to 3 parameters: the current array item, the index of the iterated item and the array instance itself.

Have a look at a callback function with the first two parameters:

```
const colors = ['blue', 'green',, 'white'];

function iterate(item, index) {
  console.log(`${item} has index ${index}`);
}

colors.forEach(iterate);
// logs "blue has index 0"
// logs "green has index 1"
// logs "white has index 3"
// comment: the missing value at colors[2] doesn't invoke the
// callback function.
```

Array.map()

The **map()** method loops over each element in an array, executes a function on it and returns a new array. The original array will not be changed.

It takes a single parameter in its brackets which is the function it uses on the array.

```
const numbers = [1, 2, 3, 4]
const doubles = numbers.map(number => number * 2)
// [2, 4, 6, 8]
```

Array.filter()

The **filter()** method creates an array filled with all array elements that pass a test (provided as a function).

The original array stays unchanged.

```
const numbers = [1, 2, 3, 4, 5, 6, 7, 8]
const evens = numbers.filter(number => number % 2 === 0)
// [2, 4, 6, 8]
```

Array.reduce()

The **reduce()** method executes a function for each element in an array and returns a single reduced result.

The intermediate result is always passed as a parameter, **result** in this example. The other parameter, **number**, is the current element of the iteration.

The intermediate result is also called **accumulator** (result/total).

The last accumulator is the return value. The original array stays unchanged.

```
const numbers = [1, 2, 3, 4]
const initialValue = 0
const sum = numbers.reduce((result, number) => {
  return result + number
}, initialValue)

// sum is 10
```

Constants

Constants are like variables that cannot change. In modern JavaScript, constants are defined with the **const** keyword.

```
const x = 4;
```

While the value of a variable can change, the value of a constant cannot.

```
const x = 4;
x = 5; // error!
```

If you have a constant that holds an **array** or an **object**, however, the values of the array or object can change.

```
const myArray = [5, 4, 2];
myArray[1] = 3; // no error
```


💡 *Optional knowledge: Constant References*

If you use the keyword **const**, it does not actually define a constant *value*. Instead, it defines a constant *reference to a value*. Because of this we cannot *reassign* variables, arrays or objects set with **const**, but we can still change the values in an array or the properties in an object.

Ternary Operator

Ternary operators are not a new feature of JavaScript but experience a revival. The logic is the same as in our **if...then** blocks, just written a little different:

```
const number = 3
if (number > 5) {
  console.log('Larger than 5')
} else {
  console.log('Lesser or equal 5')
}
// Lesser or equal 5

number > 5
  ? console.log('Larger than 5')
  : console.log('Lesser or equal 5')
// prints: Lesser or equal 5
```

It is as simple as that:

```
[condition we want to test]
  ? [code we want executed if true]
  : [code we want executed if false]
```

Then what about our **else if** condition?

```
function canYouHaveBeer(age) {
  if (age >= 18) {
    alert("Sure, we are in Europe, have a cold one!")
  } else if (age === 17) {
    alert("So close! But unfortunately, not yet.")
  } else {
    alert("Sorry young gun. How about a soda?")
  }
}
```

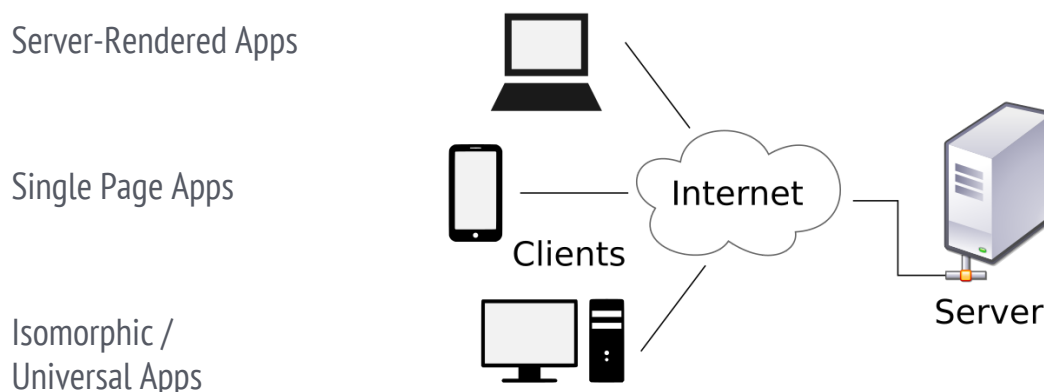
This is easily translated by nesting ternary operators. We just put another condition to be tested after the colon (else):

```
function canYouHaveBeer(age) {  
  age >= 18  
    ? alert("Sure, we are in Europe, have a cold one!")  
    : age === 17 alert("So close! But unfortunately, not yet.")  
    : alert("Sorry young gun. How about a soda?")  
}
```

Front-end Architectures

In modern web development, we always deal with a client-server setup. There is a server that hosts our HTML, CSS and JavaScript files, and a client that loads them from the server and then displays them. JavaScript is usually executed on the client, in the web browser.

There are multiple architectures that you can choose from. They all create HTML code in the end, but how this HTML is created is different.



Server-Rendered App:

A web application where the server hosts and delivers all HTML and CSS. For every page of the app, the browser makes a request to a backend server which returns an HTML document with some optional JavaScript for interactivity. It is also called: **thin client**.

Single Page App (SPA):

A web application where most HTML is created by JavaScript running on the client side. The server delivers only a minimal HTML skeleton and a JS file. The browser on the client then executes the JS, and the JS creates all HTML elements in the skeleton dynamically. All page changes are handled on the client side. This is called: **fat client**.

Isomorphic / Universal App:

This is a mix-type of server-rendered app and single page app. The server usually returns a pre-rendered page of the app, and all subsequent page changes are handled by the client.

In this course we build a server-rendered app.

JavaScript for Web Applications

Parts of a Web Application

In general, a web application contains three technologies that interact with each other:

- HTML
- CSS
- JavaScript

They come together in the HTML file:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Web Application</title>
    <link rel="stylesheet" href="styles.css" />
    <script type="javascript">
      /* ... */
    </script>
  </head>
  <body>
    <h1>Hello World</h1>
  </body>
</html>
```

Data as JSON

In web applications, we use JSON (JavaScript Object Notation) to store data.

```
const data = {
  keyString: 'Text value',
  keyNumber: 123,
  keyObject: {
    key: 'value'
  },
  keyArray: [1, 'a', { key: 'value' }]
}
```

JSON.stringify()

A common use of JSON is to exchange data with a server. But for sending data over the web, the data has to be a string. In our JavaScript code, JSON objects are usually objects. We can convert an array or object into a string with **JSON.stringify()**:

```
var data = { name: "John", age: 30, city: "New York" };
var dataString = JSON.stringify(obj);
console.log(dataString)
// logs: {"name":"John","age":30,"city":"New York"}
// dataString is now a string and ready to be sent to the server
```

JSON.parse()

The reverse function is **JSON.parse()**: it takes a string and converts it into a JSON.

```
var dataString = '{"name":"John","age":30,"city":"New York"}';
var data = JSON.parse(dataString);
```

The variable **data** is now a JSON.

The DOM

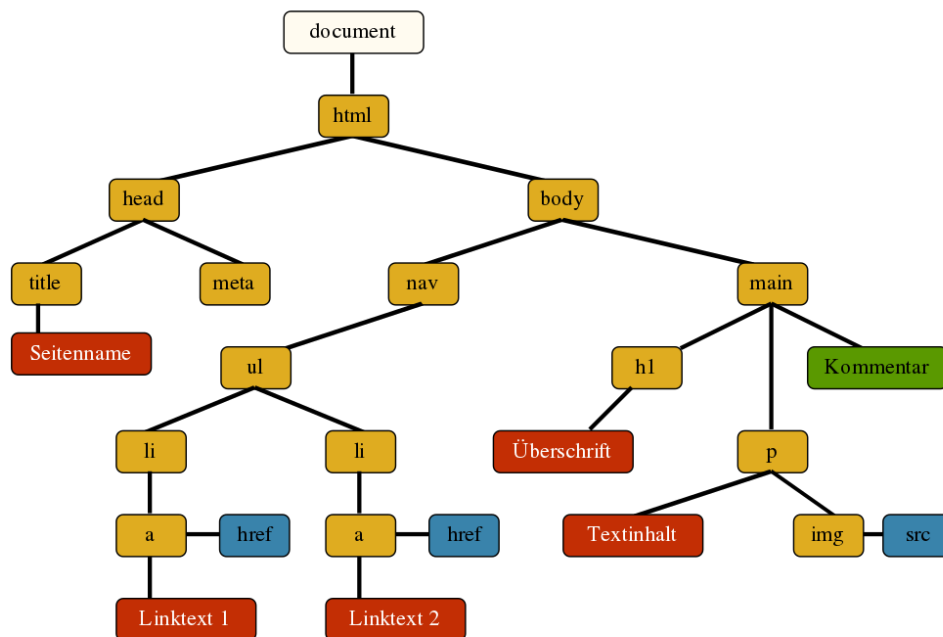
HTML elements are all nested into another.

```
<html>
  <head>
    <title>My Homepage</title>
  </head>
  <body>
    <h1>Welcome to My Homepage!</h1>
    <p>This is the content of my website.</p>
  </body>
</html>
```

Starting with the HTML tag, you can build a tree from all the text. In this example, the tag **<html>** has two children: **<head>** and **<body>**. Then, **<head>** has one child: **<title>**, and **<body>** has two children: **<h1>** and **<p>**. You can draw this as a tree.

In this tree, every tag is called a **node**.

JavaScript automatically creates objects for us that are structured exactly the same way. They are nested into each other and build a tree. This is called the **DOM**: the **Document Object Model**.



Source: <https://wiki.selfhtml.org/wiki/JavaScript/DOM>

DOM - Document Object Model The tree of all tags in an HTML document

Node A JavaScript objects for a HTML tag with all necessary information (e.g. attributes)

node.children All nested HTML tags (children) of a node

Query Selectors

The **querySelector()** method returns the first element in the DOM that matches a specified CSS selector.

```
<h1>Captivating Headline</h1>
```

```
document.querySelector('h1');
```

Inside the brackets the same rules apply as with any CSS selectors:

- # for IDs,
- .(dot) for classes,
- [] for attributes, etc.

Select by ID

```
<p id="first-text">Lorem Ipsum ...</p>  
document.querySelector("#first-text");
```

Select by class

```
<p class="text">Lorem Ipsum ...</p>  
document.querySelector(".text");
```

Select by tag

```
<p>Lorem Ipsum ...</p>  
document.querySelector("p");
```

Select by attribute

```
<p aria-hidden="true">Lorem Ipsum ...</p>  
document.querySelector("[aria-hidden]");  
document.querySelector('[aria-hidden="true"]');
```

Select by order

```
<ul>  
  <li>Item 1</li>  
  <li>Item 2</li>  
  <li>Item 3</li>  
</ul>  
  
// Select Item 2 and Item 3  
document.querySelector('li + li');
```

Multiple selectors

```
<h1>Header</h1>
<h2>Sub Header</h2>

document.querySelector('h1, h2');
```

Usage

You can apply this method on the whole DOM or on a selected part of the DOM.

```
// Get the first match:
document.querySelector()

// Get all matches:
document.querySelectorAll()

// Match on children:
const navigation = document.querySelector("#nav")
navigation.querySelector('li')
```

It returns a `NodeList` object, representing the first element that matches the specified CSS selector(s). If no matches are found, `null` is returned.

The `querySelector()` method only returns the first element that matches the specified selectors. To return all the matches, use the **`querySelectorAll()`** method instead.

Shorthands

Instead of using the prefix that selects tag, class, id etc. you can also use these functions as shorthands:

```
document.getElementById()
document.getElementsByTagName()

document.getElementsByClassName()
document.getElementsByName()
```

Nodes and NodeList

Every element in the DOM is a **node**. This is represented by a **Node** object.

The **NodeList** object represents a collection of nodes, which can be accessed by their index numbers. A NodeList object is a list or collection of nodes extracted from a document. For example, the following code selects all <p> nodes in a document:

```
var allPNodes = document.querySelectorAll('p');
```

The NodeList object is an array-like collection of objects. It has a **length** property defining the number of items in the list, and an **index** (0, 1, 2, 3, ...) to access each item, very similar to an array.


However a node list is not an array. Hence, you cannot use Array methods, like `map()`, `filter()`, `reduce()`, `push()`, `pop()`, or `join()` on a node list. With one exception: `forEach()` does work though.

If you want to use an array method on a NodeList transform it into an array first:

```
const divs = document.querySelectorAll('div')
divs.forEach(div => {
  /* ... */
}) // this works

divs.map(div => { /* ... */ }) // throws exception (an error)

// turn NodeList into array first:
[...divs].map(div => { /* ... */ })
```

 **Optional Knowledge:** There are 12 node types.

In practice we usually work with 4 of them:

- **document** - the whole document is formally a DOM node as well
- **element nodes** - HTML-tags, the tree building blocks
- **text nodes** - represents textual content in an element or attribute
- **comments** - won't be shown, but JS can still read it

Special Accessors

Every Node object has special accessors. We can, for example, access all children of a node:

```
var c = document.getElementById("myDIV").childNodes;
```

The **childNodes** property returns a collection of a node's child nodes, as a NodeList object.

Similarly, we can access the first and last child of an element like this:

```
element.firstChild
```

It is also possible to access the parent node of an element:

```
element.parentNode
```

You can also search for siblings or other “relatives” in a tree, with the **closest()** method. This will go to the parent and from there traverse through all children and their children until it finds a match that is closest to the element.

```
// Traverses all children of the parent in search for a match  
element.closest('p')
```

These are properties of DOM elements and *read-only*.

These are all the accessors:

```
element.childNodes  
element.firstChild  
element.lastChild  
element.parentNode  
element.closest('p')
```

Manipulating the DOM

Creating new elements

This creates a new **div** with text inside:

```
const div = document.createElement('div')

div.innerHTML = '<p>Text</p>'
// or
div.innerText = 'Text'

document.createTextNode('Text')
```

Another way is to create a text node (just the text that goes inside the tag), and then attach it as a child to another node.

```
var h = document.createElement("h1")
var t = document.createTextNode("Hello World");
h.appendChild(t);
```

Adding, removing and replacing nodes

JavaScript gives us functions to attach nodes as children, insert at a certain position, to replace or to remove a node.

This nests a new element inside the body tag:

```
document.body.appendChild(div)
```

This nests a new element inside another element:

```
element.appendChild(div)
```

This puts a new element just before the other element:

```
document.insertBefore(newElement, referenceElement)
```

This replaces one element with another:

```
document.replaceChild(newElement, oldElement)
```

This removes an element:

```
document.removeChild(element)
```

This empties an element:

```
element.innerHTML = ''
```

This deletes all children as well as all listeners.

Working with CSS classes

Similarly, you can also add, remove and check for classes:

```
element.classList.add('cool-class', 'second-class')
element.classList.remove('bad-class')
element.classList.contains('another-class')
```

Adding event listeners

We can add click listeners to elements like this:

```
element.addEventListener('click', (event) => {
  /* ... */
})
```

For certain elements, for example form input fields, it can make sense to add change listeners. A change listener listens for changes in the text content of an element.

```
element.addEventListener('change', (event) => {
  /* ... */
})
```

Working with Forms

For accessing user input in a form field, it is not enough to address the **input** element and its **innerText** or **innerHTML**.

The reason is that **innerText** and **innerHTML** refer to the content between tags, and the tag is usually empty. Like in this example:

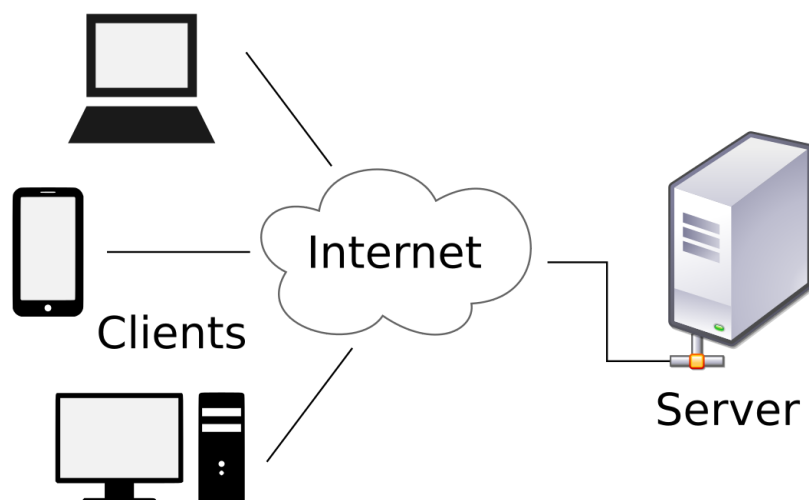
```
<form id="search-form">
  <input id="search-query" type="text">
  <button>Search</button>
</form>
```

Instead, we can use the **value** property of the input element. This contains the value that the user typed in.

```
const inputElement = document.getElementById('search-query');
const searchQuery = inputElement.value;
console.log('You searched for:', searchQuery);
```

Accessing Data from APIs

In web development it is common to access data from other servers. This is usually done by connecting to an **API** - an Application Programming Interface.



There is a certain type of APIs for the web: so-called **REST** APIs.

This is a certain format for the **HTTP** protocol, an agreement for client and server about how to communicate and exchange data with each other.



Example APIs:

Star Wars people: <https://swapi.co/api/people/>

Beers: <https://api.punkapi.com/v2/beers>

Load data with fetch()

In JavaScript, the modern way of loading data from an HTTP API from a server is using **fetch()**:

```
fetch("https://api.punkapi.com/v2/beers")
  .then(response => response.json())
  .then(beers => {
    // Do something with the loaded data
  })
  .catch(error => {
    // Handle the error
  })
```

In this example, we are fetching the content from <https://api.punkapi.com/v2/beers>. You can try out what the response is by typing it into your browser.

You will see that this is not returning an HTML website, but some content in JSON format. This is meant for being used by applications, e.g. by your JavaScript code.

In the second line, we take the response and call `.json()` on it. This takes the text from the response and creates a JSON object from it. Now we can use it in our JavaScript code.

Local Storage

When we load data from an API, we need to keep it somewhere so that we don't lose it when we route to another page.

For that, we can use **localStorage**.

This is a JavaScript feature, that we can access via the window element:

[window.localStorage](#)

It can be used for temporarily storing data, like a cache mechanism.

It is a synchronous **key-value** storage.

```
localStorage.setItem('key', value);
```

These operations are possible:

```
// Save data
localStorage.setItem('myData', data);

// Get saved data
localStorage.getItem('myData');

// Remove data item
localStorage.removeItem('myData');

// Remove all data
localStorage.clear();
```

⚠ **Note:** localStorage can only save **Strings** as values!

For more complex data objects, we can save them as JSON.

```
localStorage.setItem('randomBeer', JSON.stringify(data))
```

For example:

```
const data = [{ id: 1, text: 'Milk' }, { id: 2, text: 'Bread' }]

// Create JSON string from data and save it
localStorage.setItem('data', JSON.stringify(data))

// Get data string from local storage and create JSON object
JSON.parse(localStorage.getItem('data'))
```



You can see and manipulate the local storage in the Chrome Dev Tools at
Application > Storage

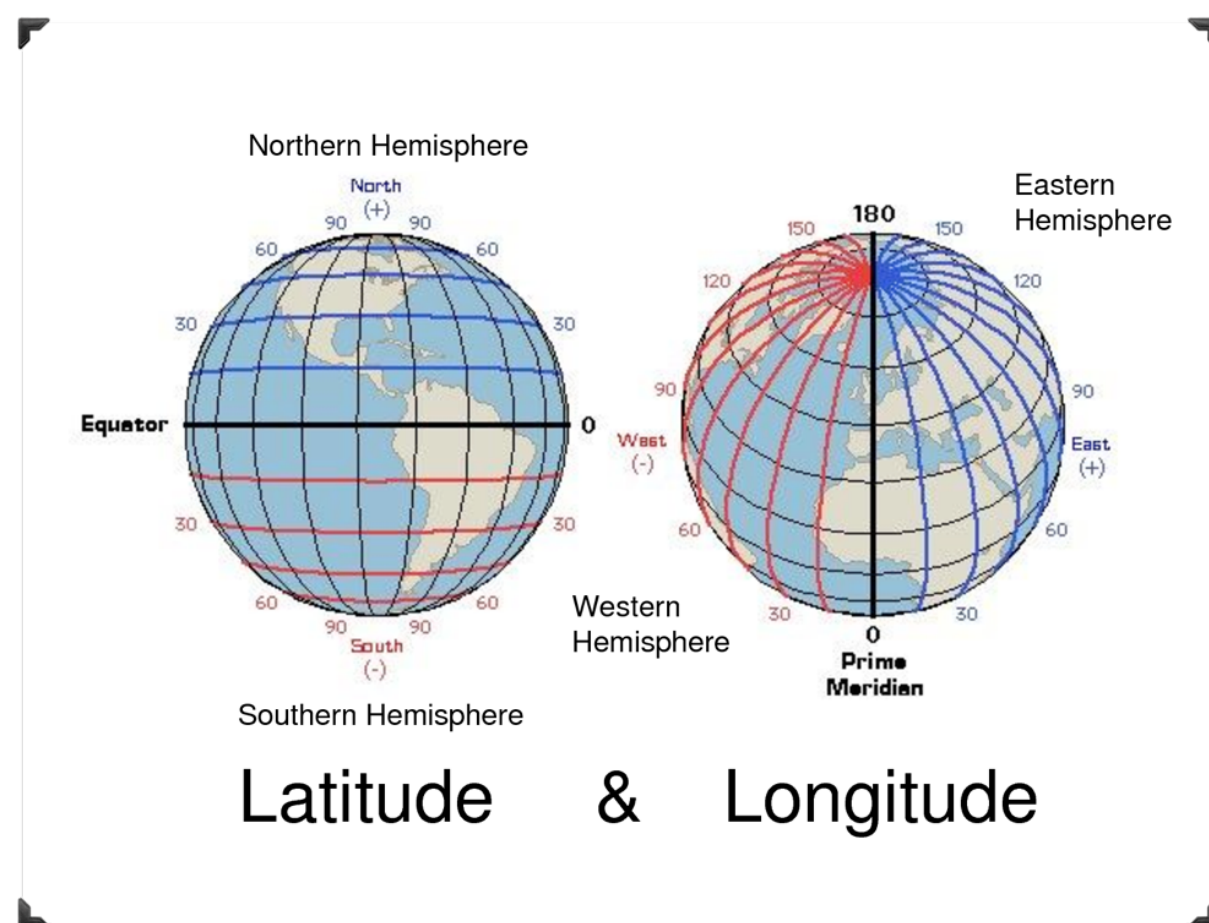
Get current location with the Geolocation API

A location on our globe is usually specified by **longitude** and **latitude**.

Latitude describes the place between north and south pole.

Longitude describes the place to the west or east.

With these two coordinates we can specify any position / geolocation on the globe.



The browser is already giving us an interface for JavaScript (an API) where we can find out our current location: with [navigator.geolocation](#).

The method `.getCurrentPosition()` tells us the geolocation of the device.

```

if (navigator.geolocation) {
  navigator.geolocation.getCurrentPosition(
    (position) => {
      const { latitude, longitude } = position.coords
      console.log('Location:', latitude, longitude)
    },
    (error) => {
      console.log('Oh no:', error)
    }
  )
}

```

This requires **user permission**. A pop-up will ask the user to grant this permission.

If the user doesn't grant the permission, we will get an error, so we need to handle that in our callback.

Display a Map with Google Maps API

We can use Google Maps to display a map on our website.

Overview: <https://cloud.google.com/maps-platform/>

Documentation: <https://developers.google.com/maps/documentation/javascript/tutorial>

API Key: <https://developers.google.com/maps/documentation/javascript/get-api-key>

```

<html lang="en">
  <head>
    <script src="googleapis.com?callback=initMap" async
defer></script>
    <script>
      let map

      function initMap() {
        const mapElement = document.querySelector('#map')
        map = new google.maps.Map(mapElement, {
          center: { lat: -33.9249, lng: 18.4241 },
          zoom: 12
        })
      }
    </script>
  </head>
  <body>
    <div id="map"></div>
  </body>
</html>

```


Showing a marker

With the Google Maps API we can put markers on the map. This works like that:

```
const marker = new google.maps.Marker({  
  position: { lat: -33.0249, lng: 18.4241 },  
  map: map  
})
```

We can later read out the position of a marker:

```
const position = marker.getPosition()
```

Handling bounds of the map

Sometimes a marker is not visible, because the map view shows a different window of the map. We can programmatically set the bounds, i.e. set the window on the map.

This, for example, sets the window so that the marker is visible inside:

```
const bounds = new google.maps.LatLngBounds()  
cities.forEach(city => {  
  const marker = new google.maps.Marker(/* ... */)  
  bounds.extend(marker.getPosition())  
})  
map.fitBounds(bounds)
```

Handling events on a map

Similar to our standard DOM elements, we can add an event listener to the map.

```
map.addListener('click', event => {  
  /* ... */  
})
```

Possible events include:

- click, dblclick, rightclick, mouseout, dragend
- center_changed, bounds_changed

Drawing a line on a map

In addition to markers, you can also draw polylines on a map. A polyline is a line that can have multiple segments.

```
const line = new google.maps.Polyline({
  map: map,
  path: [
    { lat: -33.0249, lng: 18.4241 },
    { lat: -32.0249, lng: 19.4241 }
  ]
})
```

Reverse Geocoding

Geocoding means: taking an address and getting the coordinates for that.

Reverse geocoding means: taking a coordinate and getting the closest address for that.

Google provides another API that can do that for us: **Geocoding API**.

To use it, we need to set up this API in the Google Cloud Console.

<https://developers.google.com/maps/documentation/geocoding/start>

```
const geocoder = new google.maps.Geocoder()
geocoder.geocode(
  { location: { lat: -33, lng: 18 } },
  (results, status) => {
    if (status === 'OK') {
      if (results) {
        console.log(results)
      } else {
        console.error('No results found')
      }
    } else {
      console.error('Geocoder failed due to: ' + status)
    }
  })
```

There can be no address at the given geo coordinates, so we need to implement the error callback.

Firebase

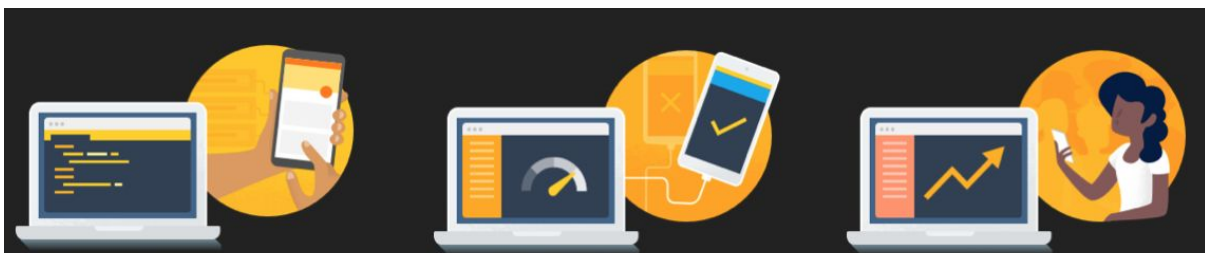
Firebase is an app development platform.

It offers all sorts of services so that developers can develop their apps without the need to set up a backend server. Therefore, this way of cloud hosting is also called **serverless**.

You can use it for services like:

- Database
- Storage
- Authentication
- Tracking
- Notifications
- A/B Testing

You can set all this up on Firebase, and then access Firebase's APIs to connect your app to the services.



The image contains three icons at the top, each with a laptop and a smartphone. The first icon shows a hand holding a smartphone next to a laptop displaying code. The second icon shows a smartphone with a checkmark next to a laptop displaying a gauge. The third icon shows a person holding a smartphone next to a laptop displaying a line graph.

Build better apps	Improve app quality	Grow your business
▶ Databases	▶ Crash / Error reporting	▶ Analytics
▶ Authentication	▶ Performance monitoring	▶ A/B Testing
▶ Machine Learning	▶ App distribution	▶ Notifications
▶ Backends		
▶ Hosting		

<https://firebase.google.com/>

Cloud Firestore

Firebase provides a cloud database called **Firestore**.

It is a **NoSQL** database, this means it is based on documents and saves data in JSON format. This comes in handy if we access it with the JavaScript API, because we can easily handle JSON data with JavaScript.

<https://firebase.google.com/docs/firestore/>

Firestore provides methods for **CRUD** operations:

CRUD:

- Create
- Read
- Update
- Delete

Create

```
<script src="gstatic.com/firebasejs/7/firebase-app.js"></script>
<script
src="gstatic.com/firebasejs/7/firebase-firestore.js"></script>
<script>
  const firebaseConfig = { /* ... */ }
  // Initialize Firebase
  firebase.initializeApp(firebaseConfig)
  const database = firebase.firestore()

  database
    .collection('posts')
    .add({
      title: 'Title of the post',
      description: 'Description of the post',
      postedAt: new Date()
    })
    .then(docRef => console.log('Created data:', docRef.id))
    .catch(error => console.error('Failed to create data',
error))
</script>
```

Read

```
database
  .collection('posts') // Collection to access (like FROM in SQL)
  .get()
  .then(snapshot => {
    snapshot.forEach(doc => {
      const data = doc.data()
      console.log("Data", data)
    })
  })
})
```

Cloud Storage

Cloud Storage is a service by Firebase for hosting files like images or videos.

Uploading a file:

```
const storage = firebase.storage()

// read a file from an input element
const inputElement = document.getElementById('file-input')
const file = inputElement.files[0]

const imageRef = storage.ref(`images/${file.name}`)
imageRef
  .put(file)
  .then(snapshot => console.log('Uploaded file!'))
  .catch(error => console.error('Failed to upload', error))
```

Getting the URL:

```
const imageRef = storage.ref('images/portrait1.jpg')

imageRef.getDownloadURL().then(url => {
  const img = document.createElement('img')
  img.src = url
  document.body.appendChild(img)
})
```

Firestore Authentication

Firestore Authentication is a service that you can use to provide a login on your website.

It provides options to use

- Email + password
- Google
- Facebook
- GitHub
- Microsoft

It even comes with UI elements for fast implementation.

<https://firebase.google.com/docs/auth>

First, you need to add the JavaScript and CSS files to your HTML.

```
<script src="firebase.com/firebaseui/firebaseui.js"></script>
<link href="firebase.com/firebaseui/firebaseui.css" />
```

Then you need to set up **auth** and **ui** like this:

```
const auth = firebase.auth()
const ui = new firebaseui.auth.AuthUI(auth)
```

To show a login form, use **ui** like this:

```
ui.start('#auth-container', {
  signInOptions: [firebase.auth.EmailAuthProvider.PROVIDER_ID],
  callbacks: { signInSuccess: () => false }
})
```

To listen to the login state (is the user signed in or signed out?), use **auth** like this:

```
auth.onAuthStateChanged(function(user) {
  if (user) {
    // signed in
  } else {
    // signed out
  }
})
```

To programmatically log out the user, use **auth** like this:


```
auth.signOut()
```

More Firebase Services

Firebase provides a lot more services, such as:

- Analytics
- Realtime Database
- Hosting
- Cloud Function
- Machine Learning
- Push Notifications
- Remote Config
- A/B Testing

And many more: <https://firebase.google.com/products>

 **Best Practice:** You can use Firebase for prototyping and testing your app ideas. It is much quicker than setting up your own backend-server.

Glossary

ES6

EcmaScript 6: A new version of JavaScript. EcmaScript is the standard for JavaScript. Version 6 contains features like the **let** and **const** keywords.

Read more:

https://www.w3schools.com/js/js_es6.asp

Arrow functions

A shorter way to write JavaScript functions. Looks like:

```
const sumOfApples = (bucket1, bucket2) =>
  bucket1 + bucket2;
```

Parameter

The variable that you put in to a function. In this example, the parameters are bold:

```
const sumOfApples = (bucket1, bucket2) =>
  bucket1 + bucket2;
```

Template literals

A way to put placeholders into strings that are later filled by the content of variables.

Example:

```
`${person.name} is ${person.age}
  years old.`
```

Grave accent

The “backtick” accent: `

Destructuring

Assigning parts of an array or object to multiple new variables.

Example:

```
const values = ['Henning', 28,
  'Engineer'];
const [name, profession] = values;
```

Spreading

Copying all fields of an object to a new object, or copying all elements of an array to a new array.

Example:

	<pre>const person = { name: 'Henning', age: 28 } const person2 = { ...person }</pre>
Callback	<p>A function that is passed into a function as a parameter and then executed inside when a certain condition holds true.</p> <p>For example, you can pass an error callback that is called when an error occurs.</p>
array.forEach()	Calls a function once for each array element.
array.map()	Changes each element, returns an array of the same size.
array.filter()	Filters out elements from an array. It lets only those elements pass for which a filter condition holds true. Returns a new array.
array.reduce()	Creates a single value from an array.
Accumulator	The variable that memorizes the last state of each iteration in the <code>reduce()</code> function.
Constant	A variable the content of which cannot be changed.
const	<p>Exception: values of an object and elements of an array can still be changed.</p>
Ternary Operator	A way of writing an if-else statement in a shorter form.
? :	<p>Example:</p> <pre>number > 5 ? console.log('> 5') : console.log('<= 5')</pre>
Architecture	A way of structuring an application.
Server-rendered App	A web application where the server hosts and delivers all HTML and CSS. For every

	<p>page of the app, the browser makes a request to a backend server which returns an HTML document.</p>
Single Page App (SPA)	<p>A web application where most HTML is created by JavaScript running on the client side. The server delivers only a minimal HTML skeleton and a JS file, which then takes care of generating all HTML pages.</p>
Isomorphic / Universal App	<p>A mix-type of server-rendered app and single page app.</p>
JSON	<p>JavaScript Object Notation, a way of declaring objects for JavaScript in the form of:</p> <pre>{ field: "value", number: 42, data: [1, 2, 3] }</pre>
JSON.stringify()	<p>Converts a JSON to a string.</p>
JSON.parse()	<p>Converts a string into a JSON.</p>
DOM	<p>Document Object Model, a representation of the tree of all HTML elements of a document in JavaScript / JSON.</p>
node	<p>A node is an HTML element in the DOM.</p>
nodeList	<p>A collection of nodes in the DOM.</p>
node.children	<p>A property that contains a nodeList of all nested HTML elements.</p>
querySelector()	<p>Returns the first element that matches the specified CSS selector in the DOM.</p>

querySelectorAll()	Returns all elements that match the specified CSS selector(s) in the DOM. Returns a <code>nodeList</code> .
getElementById()	Returns the first element that matches the specified ID.
getElementsByTagName()	Returns the first element that matches the specified tag name.
getElementsByClassName()	Returns a <code>nodeList</code> of elements that match the specified class name.
getElementsByTagName()	Returns a <code>nodeList</code> of elements that match the specified tag name.
Accessor	A property of an object. For example: <code>node.children</code> is an accessor for all the node's children.
node.childNodes	Accessor for all children of the node. Returns a <code>nodeList</code> . In contrast to <code>node.children</code> it will also return text content between tags.
node.firstChild	Accessor for the first child node of the node. Returns a node object.
node.lastChild	Accessor for the last child node of the node. Returns a node object.
node.parentNode	Accessor for the parent node of the node. Returns a node object.
node.closest()	Traverses all children of the parent and returns the closest node that matches the selector. Returns a node object.
document.createElement('...')	Creates a new node with the specified tag name.

document.createTextNode('...')	Creates text content that can then be nested into another node.
node.appendChild(div)	Appends a node or text content to the node, i.e. nests it into the node.
node.innerHTML = '...'	Assigns new nested HTML or text to a node.
node.innerText = '...'	Assigns new nested text to a node.
document.insertBefore(new, reference)	Takes the new node and inserts it at the same level as the reference node, just before the reference node.
document.replaceChild(new, old)	Takes the new node and replaces the old node with it.
document.removeChild(element)	Searches for the specified element and removes it from the DOM.
node.classList.add()	Adds a new class to a node.
node.classList.remove()	Removes a class from a node.
node.classList.contains()	Checks for the existence of a class, returns true or false.
API	Application Programming Interface - An interface for programming languages. Usually used to talk to a server.
HTTP	The protocol we use for websites (see the <code>http://</code> or <code>https://</code> prefix).
REST	A certain kind of API, where we talk to a server over the HTTP protocol.
fetch()	Loads data from an API.

Local Storage	Storage in the browser that we can write to and read from with JavaScript.
<code>localStorage.setItem('key', value)</code>	Writes a new key-value pair into local storage.
<code>localStorage.getItem('key')</code>	Reads from local storage.
<code>localStorage.removeItem('key')</code>	Removes a key-value pair from local storage.
<code>localStorage.clear()</code>	Deletes all data in local storage.
Latitude	North/south location on the globe.
Longitude	East/west location on the globe.
navigator.geolocation	An interface from the browser to access the user's geolocation information.
Permission	A pop-up that asks the user to grant the JavaScript a certain right, e.g. accessing the geolocation of the user.
Marker	A pin on a map.
Bounds	The window of the visible map.
Polyline	A line consisting of multiple segments.
Geocoding	Finding the coordinates for a given address.
Reverse geocoding	Finding an address for given coordinates.
Serverless	A term for a variety of services that make app development possible without a server, by offering hosting, databases and other server-functionality in the cloud. An example is Firebase.

**Firestore /
Cloud Firestore**

A service from Firebase that offers a NoSQL database in the cloud.

NoSQL

A document-based database.
In contrast to relational databases (e.g. MySQL) where data is saved in tables, NoSQL databases save data in documents, usually in a JSON format.

CRUD

Four common operations for databases and APIs:

C - Create
R - Read
U - Update
D - Delete

Cloud Storage

A service by Firebase for hosting files like images or videos.

Useful Links

MDN Web Docs: <https://developer.mozilla.org/en-US/docs/Web>

You Don't Know JS: <https://github.com/getify/You-Dont-Know-JS/tree/1st-ed>

About ES6: https://www.w3schools.com/js/js_es6.asp

Understanding var, let and const:

<https://www.freecodecamp.org/news/var-let-and-const-whats-the-difference/>

Google Maps Documentation:

<https://developers.google.com/maps/documentation/javascript/tutorial>

Geocoding API: <https://developers.google.com/maps/documentation/geocoding/start>

Code Examples for Google Maps:

<https://developers-dot-devsite-v2-prod.appspot.com/maps/documentation/javascript/examples/>

Firebase: <https://firebase.google.com/>

Firebase Firestore: <https://firebase.google.com/docs/firestore/manage-data/add-data>

Firebase Storage: <https://firebase.google.com/docs/storage/web/start>

Firebase Auth: <https://firebase.google.com/docs/auth>

Firebase UI: <https://firebase.google.com/docs/auth/web/firebaseui>