



Workshop

JavaScript Introduction



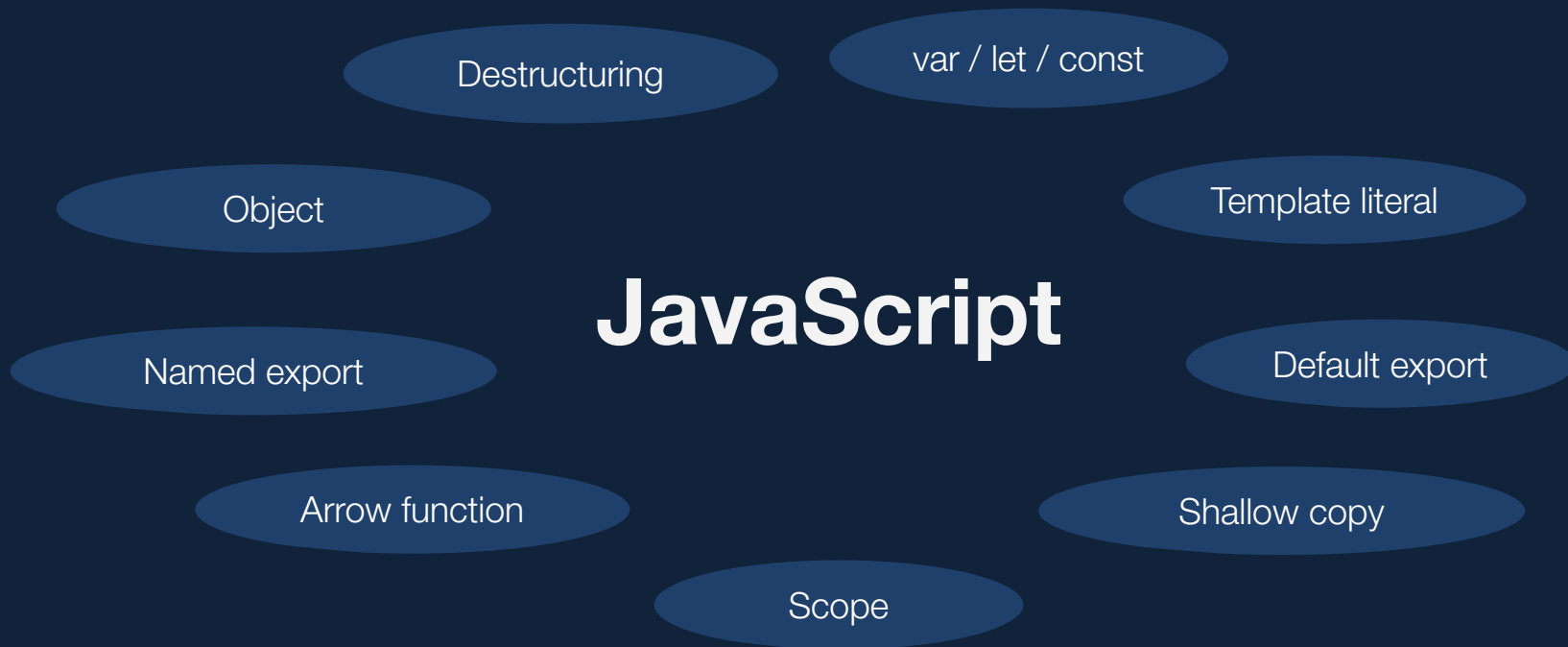
Workshop

JavaScript Introduction

Hint for trainers

- Report each change or addition to the **trainers'** Discord-Channel.
- Tell which Slide is affected, why the change is important and what benefit your change provides.
- Use the [code-highlighting-app](#) if you work with code-snippets.
- Use the following slide if you want to repeat certain topics of the workshop.

Task: Test your knowledge





JavaScript

The language of the web

Primitive Data Types

Primitive Data Types

<code>

Data Initialization

```
// Number
const a1 = 8; const a2 = 4.3; const a3 = 0x16;
// BigInt
const a4 = 2n ** 53n
// String
const b = 'Sophie';
// Boolean
const c = false;

const d = null;
const e = undefined;
```

Objects

Objects



An object is an **unordered** collection of **key-value pairs**.

Objects

<code>

Object creation

```
const a = {};
```

```
const car = {  
  make: 'Ford'  
};
```

Objects

<code>

Object property **access** and **assignment**

```
const car = {  
  make: 'Ford',  
};
```

```
car.model = 'Mustang';  
car['full year'] = 1969;
```

```
// {  
//   make: 'Ford',  
//   model: 'Mustang',  
//   'full year': 1969  
// }
```

Dot property accessor

*Square brackets
property accessor*

Computed Property Names

<code>

Allows you to put an expression in brackets, that will be computed and used as the property name.

```
const param = 'size';
const config = {
  [param]: 12,
  ['mobile' + param.charAt(0).toUpperCase() + param.slice(1)]: 4
};
```

```
console.log(config); // {size: 12, mobileSize: 4}
```

Arrays

Arrays

<code>

Arrays are **ordered** - objects are not!

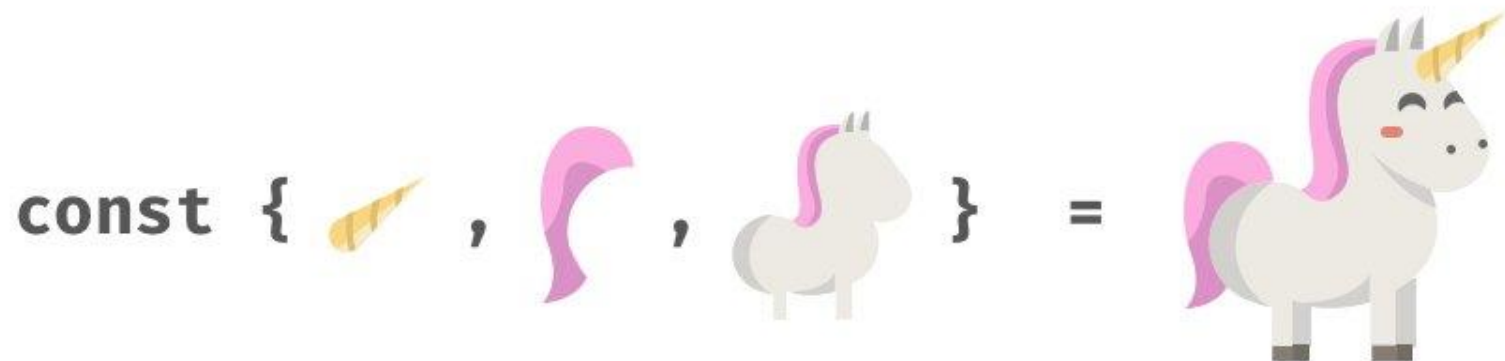
```
const myArray = ['a', 'b'];
```

```
console.log(myArray[0]); // a
```

```
console.log(myArray[1]); // b
```

Destructuring

Destructuring



Destructuring - Objects

<code>

Get multiple local variables from an object with destructuring.

```
const circle = {radius: 10, x: 140, y: 70};
```

```
circle.x = 2480234
```

```
const {x, y} = circle;
```

```
// const x = circle.x;
```

```
// const y = circle.y;
```

```
console.log(x, y)
```

```
// => 140, 70
```

```
circle.x
```

Destructuring - Objects

<code>

Renaming keys

```
const circle = {radius: 10, x: 140, y: 70};  
const circle2 = {radius: 10, x: 140, y: 70};
```

```
const {x, y} = circle2;  
// const newX = circle.x;
```

```
const {x: x2, y} = circle;  
// const newX = circle.x;  
// const newY = circle.y;
```

```
console.log(newX, newY)  
// => 140, 70
```

Destructuring - Arrays

<code>

Get multiple local variables from an object with destructuring.

```
const coords = [51, 6];  
const [, second] = coords;  
const [lat, lng] = coords;  
// const lat = coords[0];  
// const lng = coords[1];  
  
console.log(lat, lng)  
// => 51, 6
```

Destructuring - Arrays

<code>

Get specific local variables from an object with destructuring.

```
const coords = [51, 6];
```

```
// Extract second argument only  
const [, lng, third] = coords;
```

```
console.log(lng)  
// => 6
```

Spread Syntax

Spread Syntax

<code>

Create shallow copies of arrays and objects.

```
const numbersObject = {  
  one: 1,  
  two: 2,  
  three: 3,  
};
```

```
const extendedNumbersObject = Object.assign(numbersObject);
```

```
// { one: 1, two: 2, three: 3, four: 4 }
```

Functions

Function expression

<code>

Function expression (aka function declaration)

```
const showAlert = function() {  
    alert('Hello!');  
};
```


Functions - JavaScript

<code>

JavaScript Functions are “First-Class Citizens”

```
// Provide anonymous functions as arguments
http.get(url, function() { alert('Hello!'); });

// Provide named functions as arguments
const showAlert = function () { alert('Hello!'); };

http.get(url, showAlert);
```

Function statement

<code>

Function statement (hoisted to top of file)

```
// Can call sayHello before it's defined (thanks to hoisting)
alert(sayHello());
```

```
// function statement
function sayHello() {
  return "Hello, world!";
};
```

Arrow Function Expression (aka lambda function)

=>

Functions - Arrow-Function

<code>

Concept

```
// function keyword  
const add = function add(one, two) { return one + two };
```

```
// Fat-arrow function  
const add = (one, two) => { return one + two };
```

```
// Implicit return  
const add = (one, two) => one + two;
```

Functions - Arrow-Function (Benefits)

<code>

No need for braces around single parameters.

```
const square = n => n * n;
```

```
// const square = function (n) { return n * n; };
```

Functions - Arrow-Function

<code>

Use *curly braces* and *return* if you have multiple lines

```
const even = n => {  
  const rest = n % 2;  
  
  return rest === 0;  
};
```

```
// const even = function(n) {  
//   const rest = n % 2;  
//  
//   return rest === 0;  
// };
```

Functions - Arrow-Function

<code>

Use round *braces* and *curly braces* if you wanna return an object.

```
const person = () => ({  
  firstName: 'John',  
  lastName: 'Doe',  
});
```

```
// const person = function() {  
//   return {  
//     firstName: 'John',  
//     lastName: 'Doe',  
//   };  
// };
```

Arrays methods

Arrays

<code>

Arrays are **ordered** - objects are not!

```
const myArray = ['a', 'b'];  
  
console.log(myArray[0]); // a  
  
{name: "Laurenz"}["name"]
```

Arrays - Iterators

<code>

With a `for` and a `for...of` loop you have the opportunities to `break` or `continue` the loop and exit the surrounding function with `return`.

```
const names = ['Hanni', 'Nanni'];

for (let i = 0; i < names.length; i++) {
  const name = names[i];
  console.log(name);
}

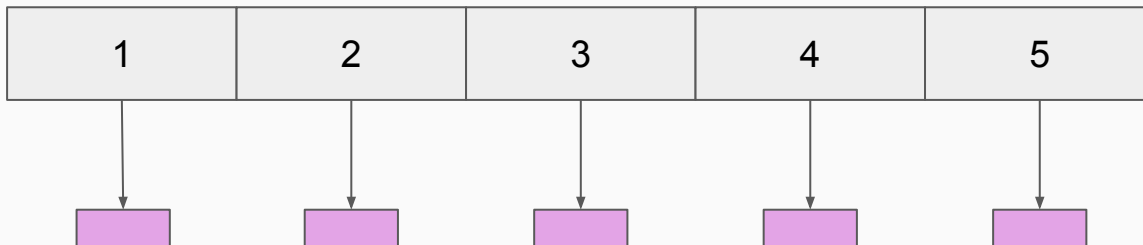
for (let name of names) {
  console.log(name)
}
```

Arrays - Iterators

Array.forEach()

```
const myArray = [1,2,3,4,5];  
const foo = (x) => {console.log(x)}  
myArray.forEach(foo);
```

numbers

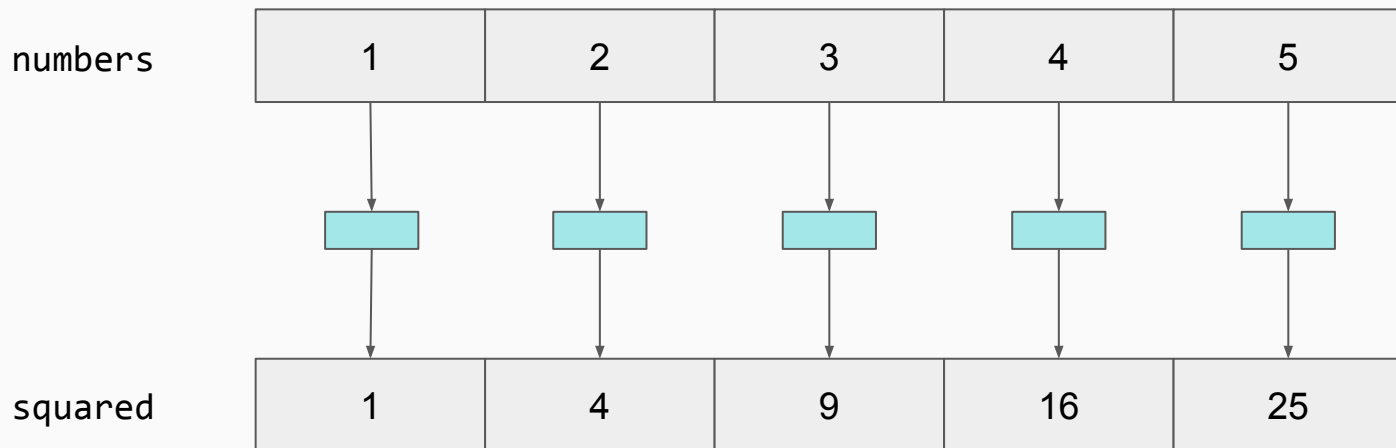


Arrays - Transformations

Array.map()

```
const numbers = [1, 2, 3, 4, 5];  
const squared = numbers.map(num => num * num);  
// squared is [1, 4, 9, 16, 25]
```

Transforming an array

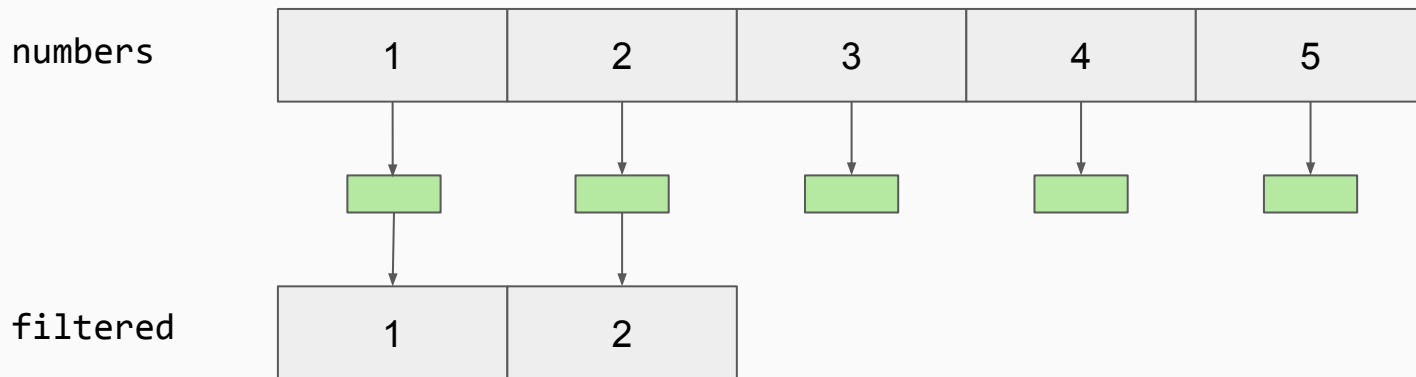


Arrays - Transformations

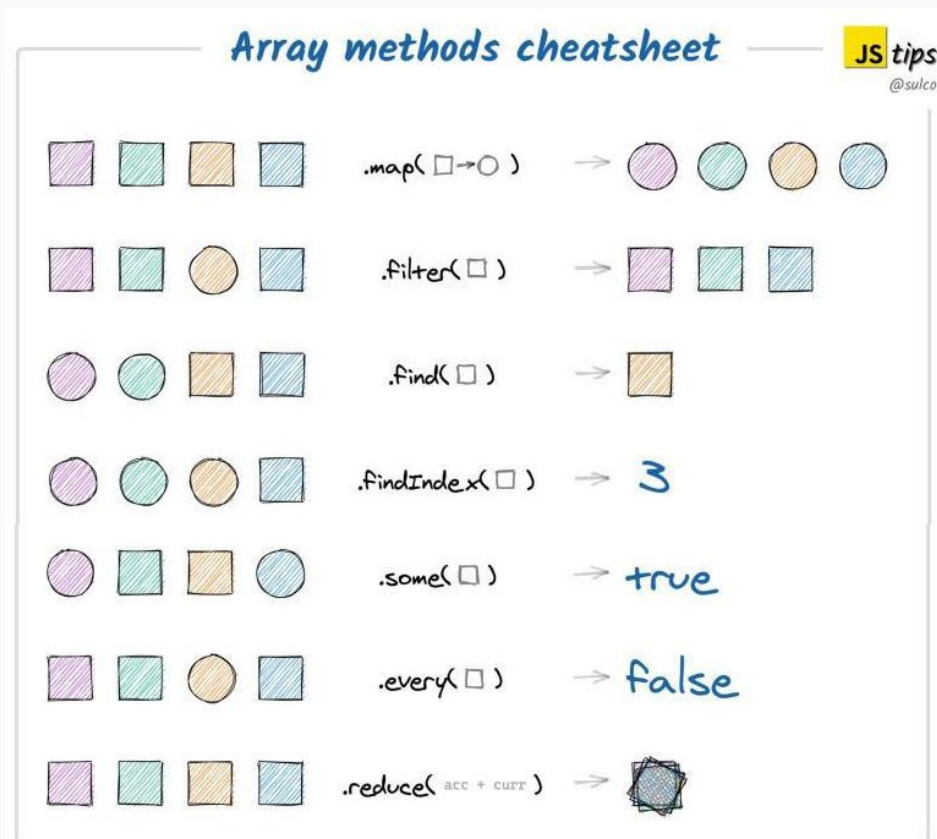
Array.filter()

```
const numbers = [1, 2, 3, 4, 5];  
const filtered = numbers.filter(num => num < 3);  
// filtered is [1, 2]
```

Filtering an array



Arrays - Transformations



Variables

declaration and usage

Variables - Declaration

<code>

Declared with the keyword **var**, **let** or **const**

```
var value = 23;
```

```
let anotherValue = 42;  
anotherValue=2;
```

```
const PI = {x: 3.1416};  
PI.x = 4
```


Variables - let, const, var

	let	const	var
scope	block	block	function
reassignable	✓	✗	✓
mutable	✓	✓	✓
Standard	ES2015 / TS	ES2015 / TS	since ever
Cases to use	~5%	~95%	nearly never

let is block scoped

```
let example1 = 1;

if (true) {
  let example = 2;
  console.log('Inside: ' + example);
}

console.log('Outside: ' + example);
// => Inside: 2
// => Outside: 1
```

var is function scoped

```
var example = 1;

if (true) {
  var example = 2;
  console.log('Inside: ' + example);
}

console.log('Outside: ' + example);
// => Inside: 2
// => Outside: 2
```

Variables - Fun fact

<code>

UTF-8 characters are also allowed!

```
const  $\pi$  = Math.PI;
```

```
const ღ_ბჰლ_ლ = 42;
```

Variables

<code>

Hold the result of an expression

```
const helloWorld = 'Hello World';
```

```
const helloFunction = function() {};
```

```
const helloFunction = () => {};
```

```
const returnValue = getCurrentTime();
```

```
const returnValue = getCurrentTime;
```

```
returnValue()
```

Variables - Primitive types

<code>

Call by value

```
let a = 'Hello World';  
const b = a; // Only value is copied  
a = 4;  
  
console.log(b);  
// => 'Hello World'
```

Variables - Object types (Array)

<code>

Call by reference

```
const a = [1, 2, 3];  
const b = a; // Copy the reference  
a[0] = 99;   // Modify the array using the reference  
  
console.log(b);  
// => [99, 2, 3]
```

Variables - Object types

<code>

Call by reference

```
const person = { firstName: 'John', lastName: 'Doe' };  
const secondPerson = person;      // Copy the reference  
  
secondPerson.firstName = 'Jane'; // Modify the object using the reference  
  
console.log(secondPerson);  
// => { firstName: 'Jane', lastName: 'Doe' }  
console.log(person);  
// => { firstName: 'Jane', lastName: 'Doe' }
```


Template Literals (template strings)

Template Strings

<code>

Simple template strings

```
const singleLine = `My first template string`; // single line
const multiLine = `
  My first multiline template string!
</span>`; // multiline
```

Template Strings

<code>

Variables in template strings

```
const lastName = 'Eich';  
const name = `Brendan ${lastName}`; // single line  
// Brendan Eich
```

“Brendan ”+ lastName

```
const multilineString = `<span>  
  My name is ${name}!  
</span>`; // multiline
```

```
// <span>  
//   My name is Brendan Eich!  
// </span>
```

Template Strings

<code>

Expressions in template strings

```
const active = true;
const name = `Is active: ${active ? 'Yes!' : 'No!'}!`;
// Is active: Yes!
```

```
function isActive() {
  return 'No!';
}
const name = `Is active: ${isActive()}!`;
// Is active: No!
```

Modules in JavaScript

Modules - General

- organize code
- split the application into multiple files
- solve a specific problem/deal with a specific topic
- share functionalities between modules

Named Export

<code>

You can export functions, objects, or primitive values from the module so they can be used by other programs with the import statement.

```
// foo.js
export const myFunction = () => {};
export const foo = Math.sqrt(2);
export const MY_CONSTANT = 'MY_CONSTANT';
```

```
// bar.js
import { myFunction, foo } from './foo'; // names must match!
```

Default Export

<code>

You can have multiple named exports per module but **only one default export**.

```
// WelcomeScreen.js
export default (props) => { /* ... */ };
```

```
// HomeScreen.js
const HomeScreen = (props) => { /* ... */ };
export default HomeScreen;
export { HomeScreen as default };
```

```
// App.js
import WelcomeScreen from './WelcomeScreen';
import Home from './HomeScreen';
```


Renaming

<code>

You can rename an export when importing it.

```
// foo.js
```

```
export const myValue = 123;
```

```
// bar.js
```

```
import {myValue as differentName} from './foo';
```

```
import {myValue as diffeentName} from './fuz';
```

Import entire module's contents

<code>

This inserts myModule into the current scope, containing all the exports from the module in the file foo.js

```
// foo.js
export myFunction;
export const foo = Math.sqrt(2);
export const MY_CONSTANT = 'MY_CONSTANT';
export default MY_CONSTANT = 'MY_CONSTANT';
```

```
// bar.js
import * as myModule from './foo';

myModule.myFunction();
console.log(myModule.MY_CONSTANT);
```

Import entire module's contents

<code>

You can import default and named exports in one statement.

```
// Example: General
```

```
import defaultVar, { namedVar1, namedVar2 } from './foo';
```

```
// Example: React
```

```
import React, { ReactDOM, useState } from 'react';
```

Equality operator

Check for value (in)equality

<code>

```
const a = 'test';  
const b = 'test';  
const c = 'test2';
```

```
a == b  
a != c
```

```
const num1 = 1;  
const num2 = 1;  
const num3 = 2;
```

```
num1 == num2  
num1 != num3
```

Check for value AND type (in)equality

<code>

```
const a = '1';  
const b = '1';  
const c = '2';
```

```
a === b  
a !== c
```

```
const num1 = '2';  
const num2 = '2';  
const num3 = 2; ⚡
```

```
num1 === num2  
num1 !== num3 ⚡
```

Always prefer `===` and `!==`

- Forces you to write better code (typesafe)
- Clearer intentions
- Especially when working with user input

Object (in)equality

<code>

```
const car = {  
  brand: 'Ford'  
};
```

```
const car2 = {  
  brand: 'Ford'  
};
```

```
car == car2 ⚡  
car === car2 ⚡
```

The objects are *visually* equal, but unequal when comparing like that

Object (in)equality

<code>

```
const car = {  
  brand: 'Ford'  
};
```

```
const car2 = car;
```

```
car == car2  
car === car2
```

Array (in)equality

<code>

```
const a = ['a', 'b'];
```

```
const b = ['a', 'b'];
```

```
a == b ⚡
```

```
a === b ⚡
```

Beware of Objects & Arrays (in)equality

- Even if the values are kind of equal to us humans, equality behaves differently
- Different storing technique in memory
- We will come back to this later 🖐️

Falsy / Truthy Values

Falsy and Truthy Values

<code>

```
const framework = 'vue';
```



true aka boolean

```
if (framework === 'vue') {...}
```

Falsy and Truthy Values

<code>

```
const framework = 'vue';
```



value is a string

```
if (framework) {...}
```

Falsy and Truthy Values

<code>

```
const framework = 'vue';
```



value is a string

```
if (framework) {...}
```

JavaScript tries to coerce values to a Boolean value if required

Type coercion to boolean values

	Coerced value	
0	false	falsy
Any other number (incl. negative) except for NaN	true	truthy
' ' (empty string)	false	falsy
Any other non-empty string	true	truthy
{}, [] & all other objects and arrays (even empty)	true	truthy
null, undefined and NaN	false	falsy

Falsy and Truthy Values

- JavaScript conditions work with booleans → `true` / `false`
- If an operation needs a boolean, JavaScript coerces the value into a boolean
- “Converting without really converting”
- The values which are coerced to true are called truthy, the other ones are called falsy

Nullish Values

- Nullish is a more specific falsy value
- `null` and `undefined` are considered nullish

Nullish coalescing

Nullish coalescing

- `??` is the fourth logical operator in JavaScript (next to `&&`, `||` and `!`).
- New since ES2020 (June 2020).
- Checks if a value is **nullish** instead of **falsy** (like the AND- and OR-Operator).
 - If so, returns the value on the right hand side.
 - Otherwise returns the value on the left hand side.

Nullish coalescing

<code>

```
1      ?? "Workshops.de";    // Result is: 1
42     ?? "Workshops.de";    // Result is: 42
true   ?? "Workshops.de";    // Result is: true
false  ?? "Workshops.de";    // Result is: false
0      ?? "Workshops.de";    // Result is: 0
""     ?? "Workshops.de";    // Result is: ""

null   ?? "Workshops.de";    // Result is: Workshops.de
undefined ?? "Workshops.de"; // Result is: Workshops.de
```

Nullish coalescing vs logical OR ||

<code>

1	??	"Default";	// 1	✓
42	??	"Default";	// 42	✓
true	??	"Default";	// true	✓
""	??	"Default";	// ""	✓
null	??	"Default";	// Default	✓
undefined	??	"Default";	// Default	✓



1		"Default";	// 1	✓
42		"Default";	// 42	✓
true		"Default";	// true	✓
""		"Default";	// Default	✓
null		"Default";	// Default	✓
undefined		"Default";	// Default	✓



Nullish coalescing vs logical OR ||

<code>

This new operator should be **the first choice** to check for *nullish* values. Do not longer use the OR-Operator for this kind of checks.

```
0      ?? "Default";    // 0      ✓  
false  ?? "Default";    // false  ✓
```



```
0      || "Default";    // Default ✗  
false  || "Default";    // Default ✗
```



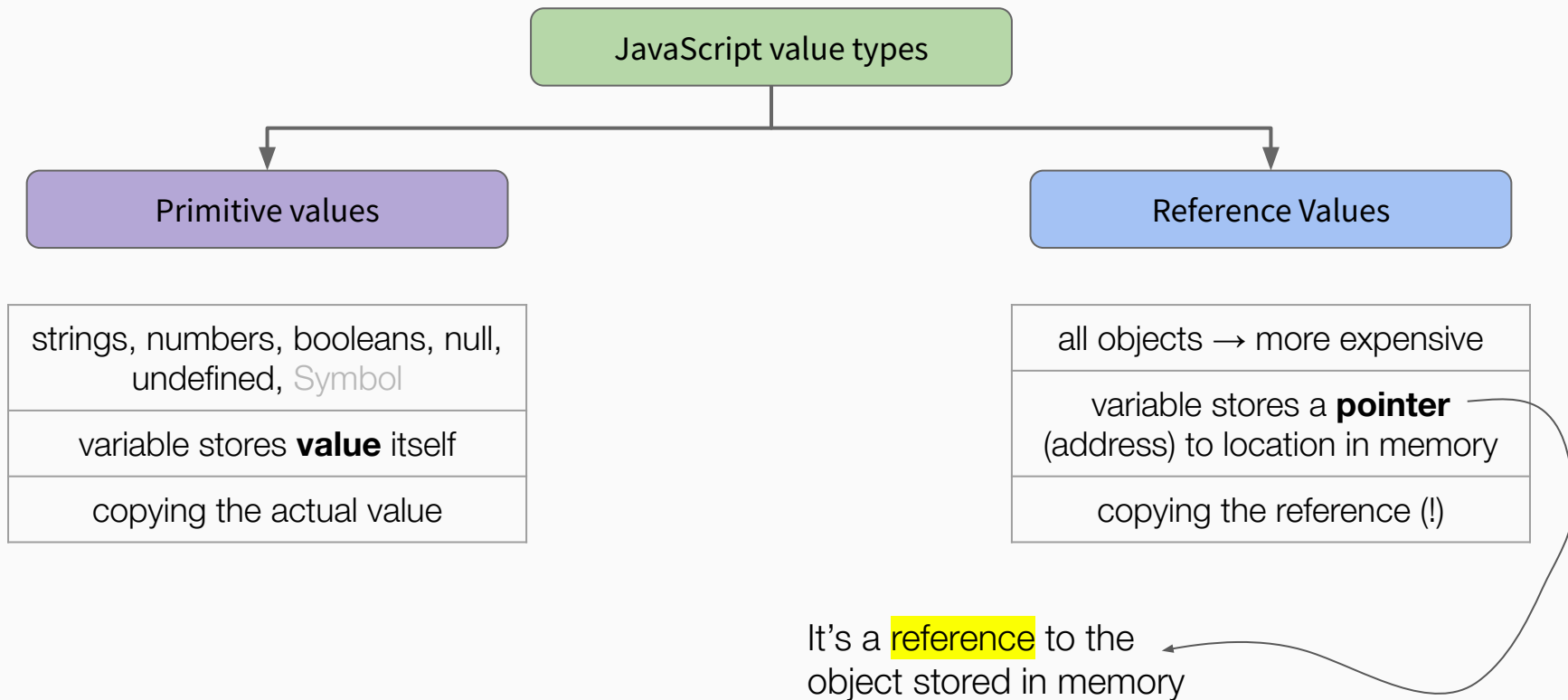
Task

JavaScript Playground 01



Passing variables in JavaScript

Primitive vs Reference values



Primitive values - copy by value

<code>

```
let framework = 'vue';  
  
let anotherFramework = framework; // 'vue'  
  
framework = 'react';  
  
console.log(anotherFramework); // 'vue'
```

Reference values - copy by reference

<code>

Be careful here, we are only copying the **reference**, not the actual value

```
let car = {  
  make: 'Ford'  
};
```

```
let car2 = car; // { make: 'Ford' }
```

```
car.model = 'Mustang';
```

```
console.log(car2); // { make: 'Ford', model: 'Mustang' }
```

Object (in)equality

<code>

```
const car = {  
  make: 'Ford'  
};
```

```
const car2 = {  
  make: 'Ford'  
};
```

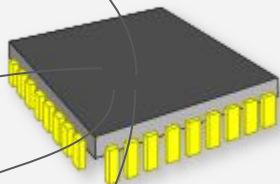
```
car == car2 ⚡  
car === car2 ⚡
```

// value of car is 0ddaf4...

// value of car2 is 5f9f98...

// 0ddaf4... != 5f9f98...

// 0ddaf4... !== 5f9f98...



Objects & Arrays deep (in)equality

- Use recursive equality algorithm
- Objects have to be compared by their properties
- Write your own logic or use third party libraries
- i.e. Lodashs [isEqual](#) function

Passing variables in JavaScript

- All function arguments are always passed by value
- JavaScript copies values of the variables into local variables
- Changes to local variables will not be reflected outside the function
- But primitive and reference values behave different (somehow)

Primitive values - pass by value

<code>

```
function addOne(x) {  
    x = x + 1;  
}
```

```
const value = 10;
```

```
addOne(value);
```

```
console.log(value);
```


Primitive values - pass by value

<code>

```
function addOne(x) {  
  x = x + 1;  
}
```

```
const value = 10;
```

```
addOne(value);
```

```
console.log(value);
```



Primitive values - pass by value

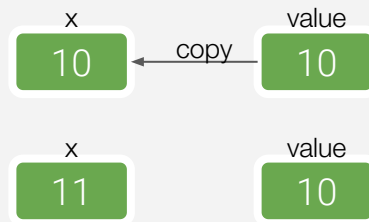
<code>

```
function addOne(x) {  
  x = x + 1;  
}
```

```
const value = 10;
```

```
addOne(value);
```

```
console.log(value);
```



Primitive values - pass by value

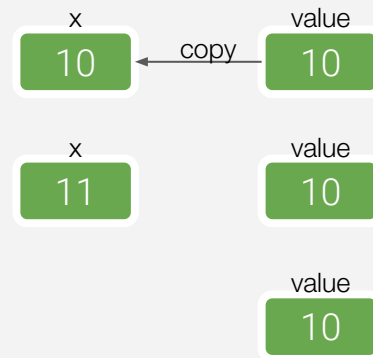
<code>

```
function addOne(x) {  
  x = x + 1;  
}
```

```
const value = 10;
```

```
addOne(value);
```

```
console.log(value);
```



Reference values - pass by reference

<code>

```
function setActive(obj) {  
    obj.isActive = true;  
}
```

```
const state = {  
    isActive: false  
};
```

```
setActive(state);
```

Reference values - pass by reference

<code>

```
function setActive(obj) {  
  obj.isActive = true;  
}
```

```
const state = {  
  isActive: false  
};
```

```
setActive(state);
```



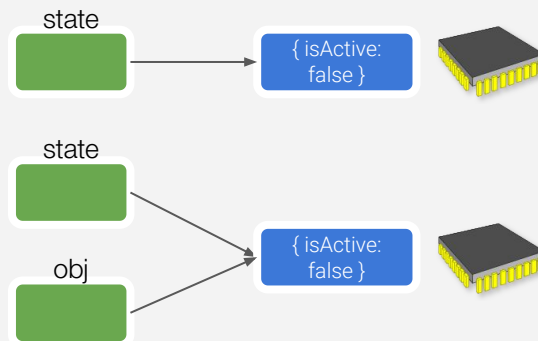
Reference values - pass by reference

<code>

```
function setActive(obj) {  
  obj.isActive = true;  
}
```

```
const state = {  
  isActive: false  
};
```

```
setActive(state);
```



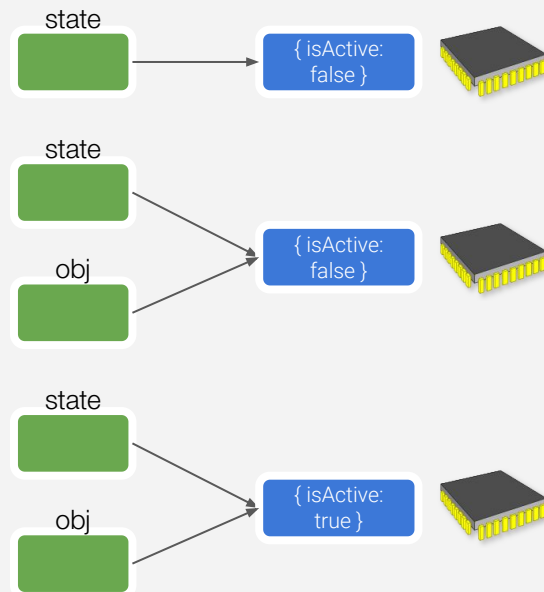
Reference values - pass by reference

<code>

```
function setActive(obj) {  
  obj.isActive = true;  
}
```

```
const state = {  
  isActive: false  
};
```

```
setActive(state);
```



Promises

The **Promise** object represents the potential completion of an asynchronous operation, and its resulting value.

**Why not simple
callbacks?**

Callbacks

<code>

A callback can be called asynchronously.


```
http(url, response => {  
  user = response.data.user;  
}); // http() is async / non-blocking!
```

```
user; // undefined
```

Callback problems

<code>

1. Callback nesting → The Pyramid of Doom



```
http(url, response => {  
  performOperation(response.data.user, result => {  
    calc(result, calcResult => {  
      doSomething(calcResult, () => {  
        // ...  
      });  
    });  
  });  
});
```

Callback problems

<code>

2. Error handling → Catch errors in every block?

```
http(url, (err, response) => {  
  if (err) ...  
  performOperation(response.data.user, (err, result) => {  
    if (err) ...  
    calc(result, (err, calcResult) => {  
      if (err) ...  
      doSomething(calcResult, () => {  
        // ...  
      });  
    });  
  });  
});  
});
```

Callback problems

<code>

2. Error handling → try-catch error handling doesn't work!

```
const asyncFn = () => {  
  setTimeout(() => {  
    console.log('doSomethingAsync() called!');  
    throw new Error('Something went wrong!');  
  });  
};  
  
try {  
  asyncFn();  
} catch(e) {  
  // This won't catch the error thrown in asyncFn()!  
}
```

Callback problems

<code>

3. Synchronizing multiple callbacks

```
const api1 = http.get(url, () => {  
  // ...  
})
```

```
const api2 = http.get(url, () => {  
  // ...  
})
```

```
// How to wait for both?
```

Promises to the rescue

Promises

<code>

Success and error semantics via `.then()` and `.catch()`

```
fetch('http://example.com/books')
```

```
promise.then(successFn).catch(errorFn)
```

Promises

- Read-only view to a **single** future value
- **Not lazy**: As soon as the JavaScript Interpreter sees a Promise declaration it immediately executes its implementation synchronously. Even though it will get settled eventually (resolved or rejected).
- Immutable and **uncancellable**. Your promise will be resolved **or** rejected once.

Promises

<code>

Simple example: async operation with callbacks

```
const setTimer = (duration, callbackFn) => {  
  setTimeout(() => {  
    callbackFn('Done!');  
  }, duration * 1000);  
};  
  
setTimer(14, (message) => {  
  console.log(message); // Done!  
});
```

Promises

<code>

Simple example: Async operation with Promise (transformed)

```
const setTimer = (duration) => {  
  return new Promise((resolve, reject) => {  
    setTimeout(() => {  
      resolve('Done!');  
    }, duration * 1000);  
  });  
};
```

```
setTimer(14).then((message) => {  
  console.log(message); // Done!  
});
```

asdfsdf

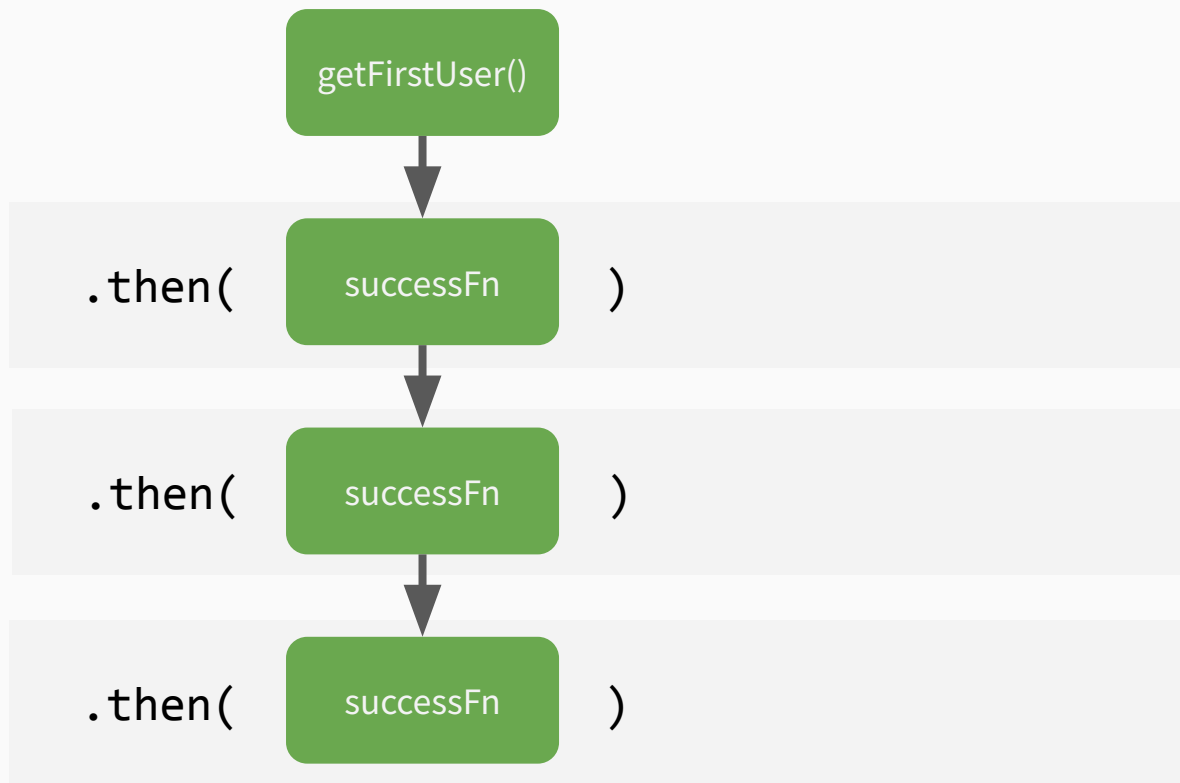
Promises

<code>

Avoid pyramid of doom

```
api.getFirstUser() // returns a promise
  .then(response => {
    return response.data.user;
  }) // a new resolved promise with value
  .then(user => {
    return api.getUserBooks(user.id) // returns a new promise
  })
  .then(book => { ... })
```

Promises - Success flow



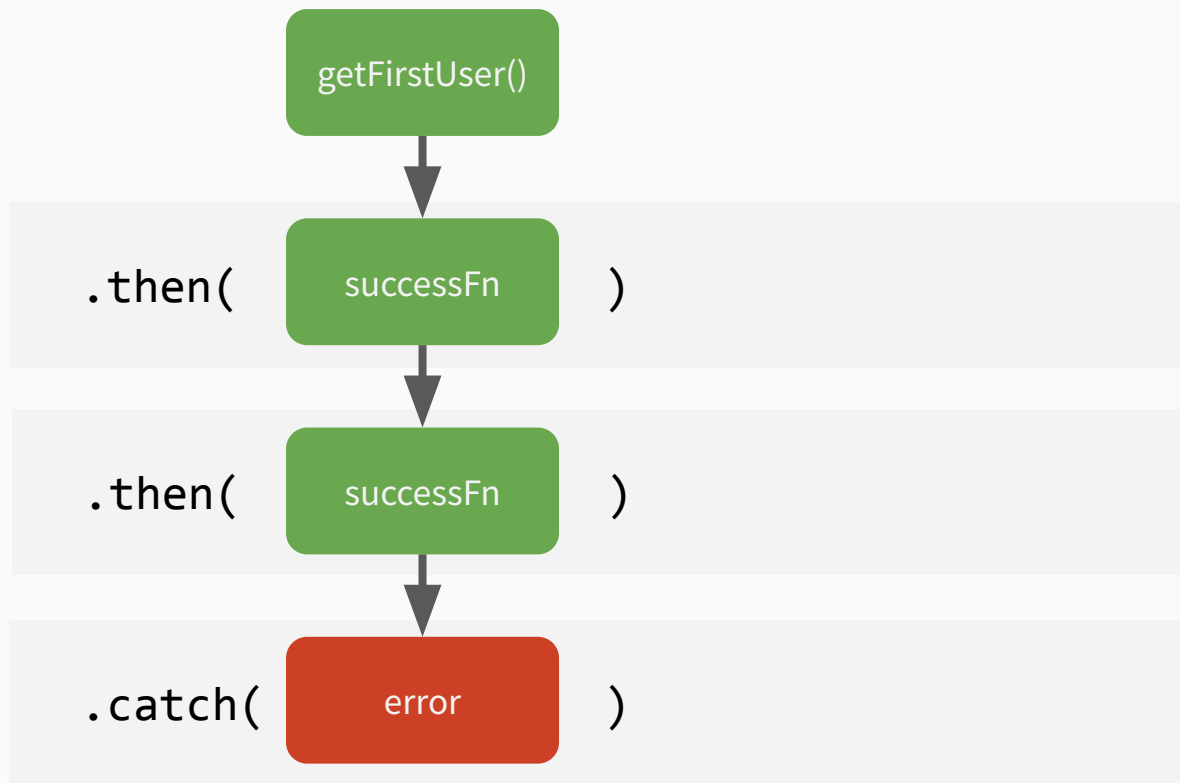
Promises

<code>

Downstream **value** and **catch** errors in a single handler

```
api.getFirstUser(url)
  .then(response => {
    return response.data.user;
  })
  .then(() => throw new Error('An error!!!'))
  .then(calc) // doesn't handle an error
  .then(doSomething)
  .catch(error => { // catches all previously occurred errors and Promise
    rejections
    console.log('An error occurred!', error);
  });
```

Promises - Success flow



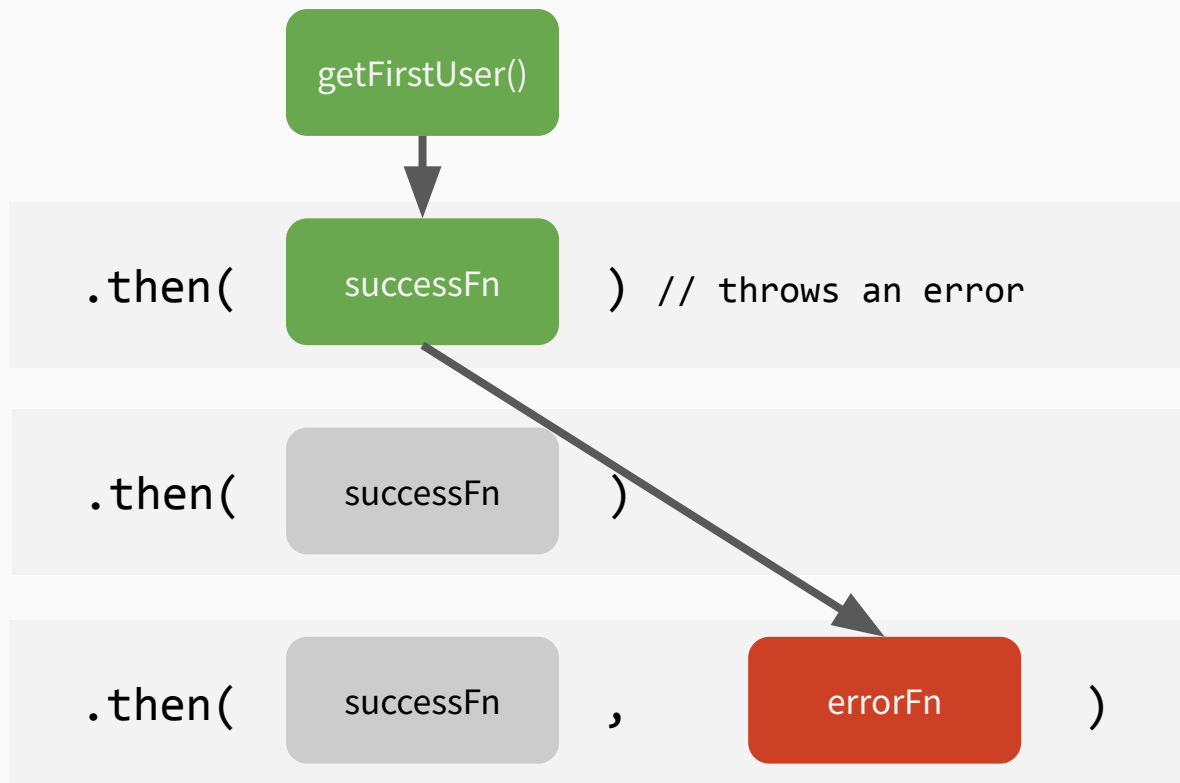
Promises

<code>

Downstream **value** and **error** propagation

```
api.getFirstUser(url)
  .then(response => {
    return response.data.user;
  })
  .then((user) => throw new Error('An error!!!'))
  .then((data) => { ... }) // doesn't handle an error
  .then((doSomething, error) => {
    console.log('An error occurred!', error);
  });
```

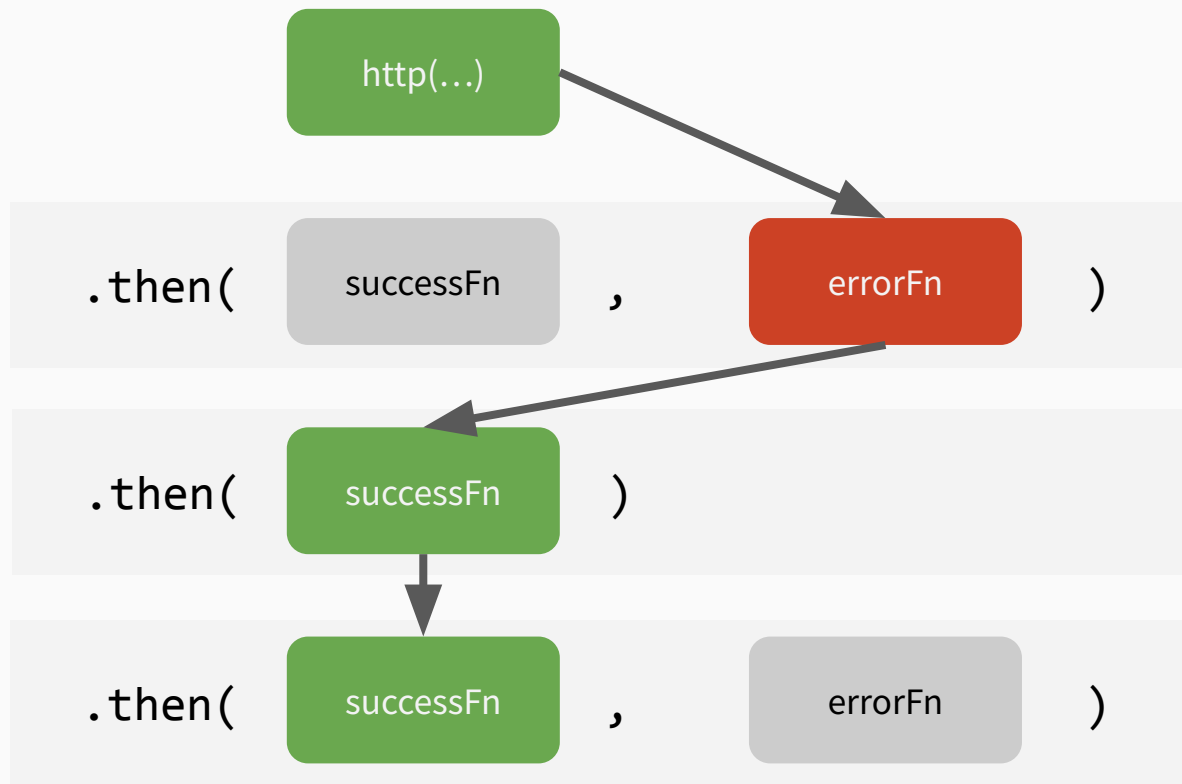
Promises - Error flow



Promises

- Errors and rejections can be handled
- E.g. an API that has a mirror
- You can recover from errors

Promises - Recovery flow



Promises

<code>

You can synchronize multiple promises easily - waits until all promises are resolved.

```
const api1 = http.get('/api1'); // returns a promise
const api2 = http.get('/api2'); // returns a promise
```

```
Promise.all([api1, api2])
  .then([result1, result2] => {
    return ...
  })
  .then(anotherTransform)
```

Async data fetching

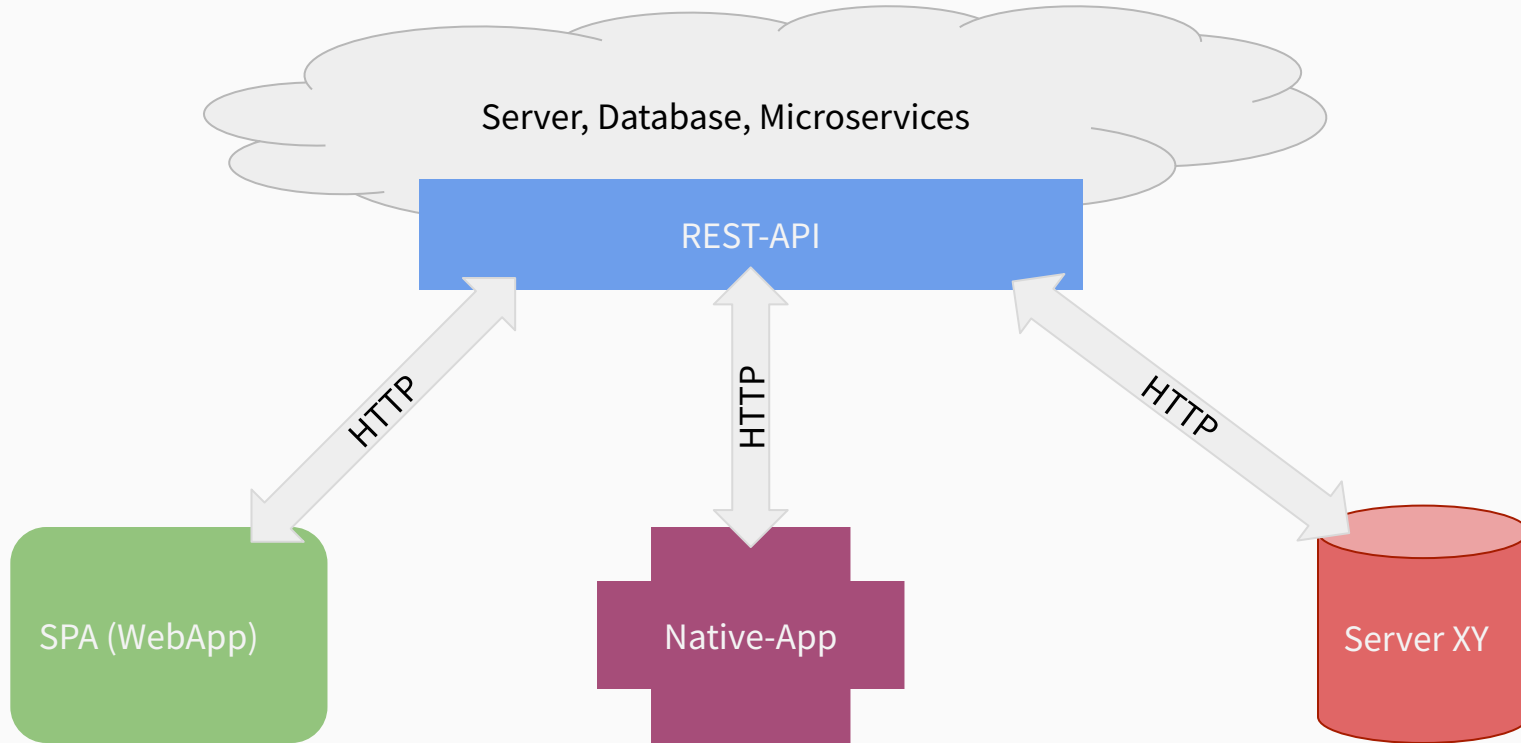
Load data from an API via HTTP

Why / What you'll learn



- Your data isn't stored locally
 - Multiple clients
 - Huge amount of data
- Communication via HTTP (REST/CRUD)
- Different ways to get data from an API

Using a Rest API



Basic CRUD Service

`http://localhost:4730`

POST	<code>/books</code>	// Create a new book
GET	<code>/books</code>	// Read all books
PUT	<code>/books/:isbn</code>	// Update a book by ISBN
DELETE	<code>/books/:isbn</code>	// Delete a book by ISBN
GET	<code>/books/:isbn</code>	// Read a specific book by ISBN

Using Fetch

- Fetch is a modern concept equivalent to XMLHttpRequest.
- The API is completely Promise-based.

Request an API via Fetch

<code>

One argument as a string results in a GET request to this URL

```
return fetch(URL)
  .then(response => response.json())
  .then(result => console.log(result.status))
```

Request an API via Fetch

<code>

Request interface allows more detailed control of a resource request

```
const request = new Request(URL, {  
  headers: {  
    'Accept': 'application/json',  
    'Content-Type': 'application/json'  
  },  
  method : 'PUT',  
  body    : JSON.stringify(aJavaScriptObject)  
});  
  
return fetch(request)  
  .then(response => response.json());
```

Using async and await

<code>

Instead of chaining then you also could use async/wait

```
const fetchData = async () => {  
  try {  
    const response = await fetch('http://localhost:4730/books')  
  } catch (  
    const result = await response.json();  
    setBooks(result); // React useState setter function  
  }  
  
  fetchData().then(...)
```

Task

JavaScript Playground 02





We teach.

workshops.de