

# Asynchronous Programming and Promises

Fetch API, Promises, async/await



SoftUni Team  
Technical Trainers



**SoftUni**



Software University

<https://softuni.bg>

# Table of Contents

1. AJAX
2. Asynchronous Programming
3. Promises Deep Dive
4. Async / Await



sli.do

**#js-advanced**



**AJAX**

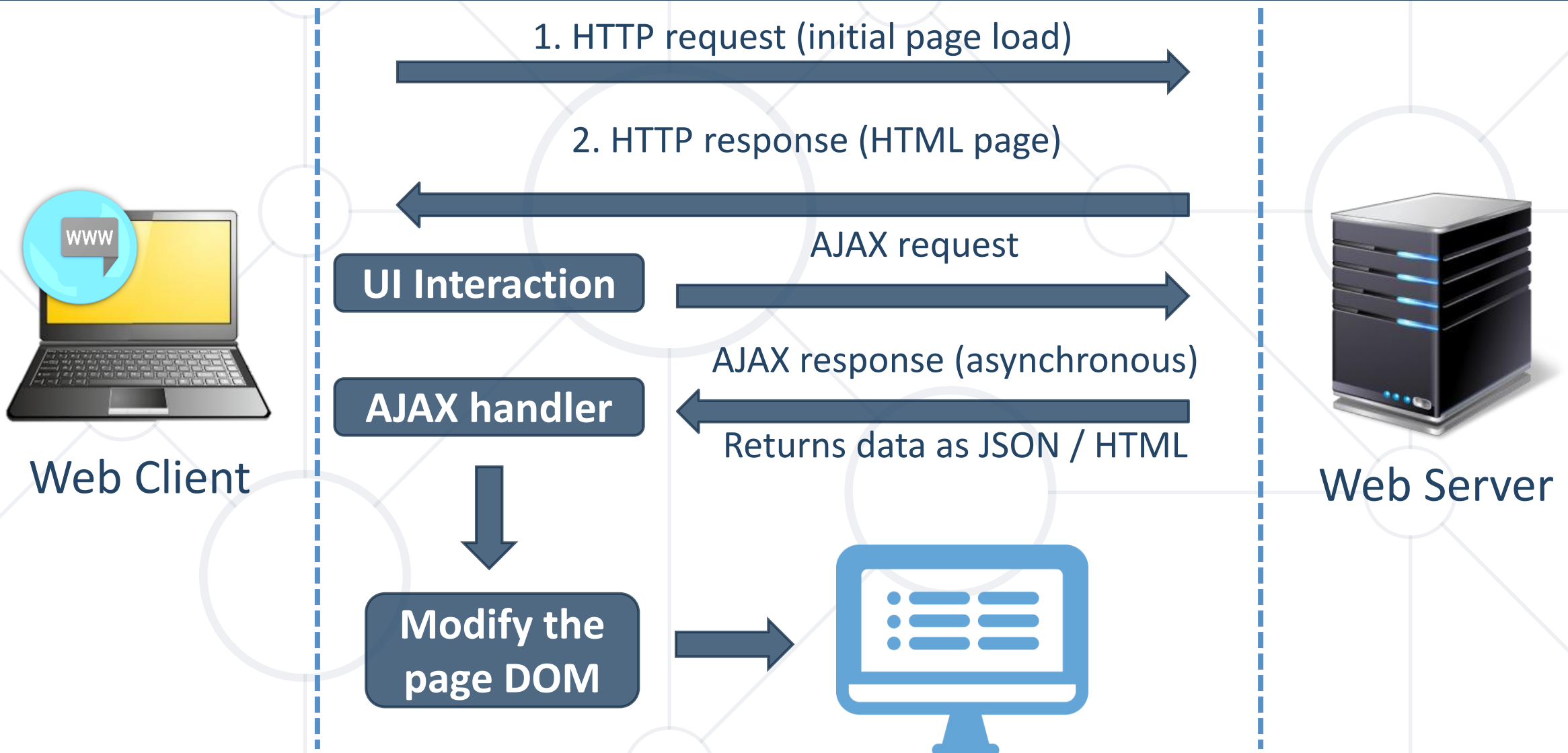
Asynchronous JavaScript and XML

# What is AJAX?



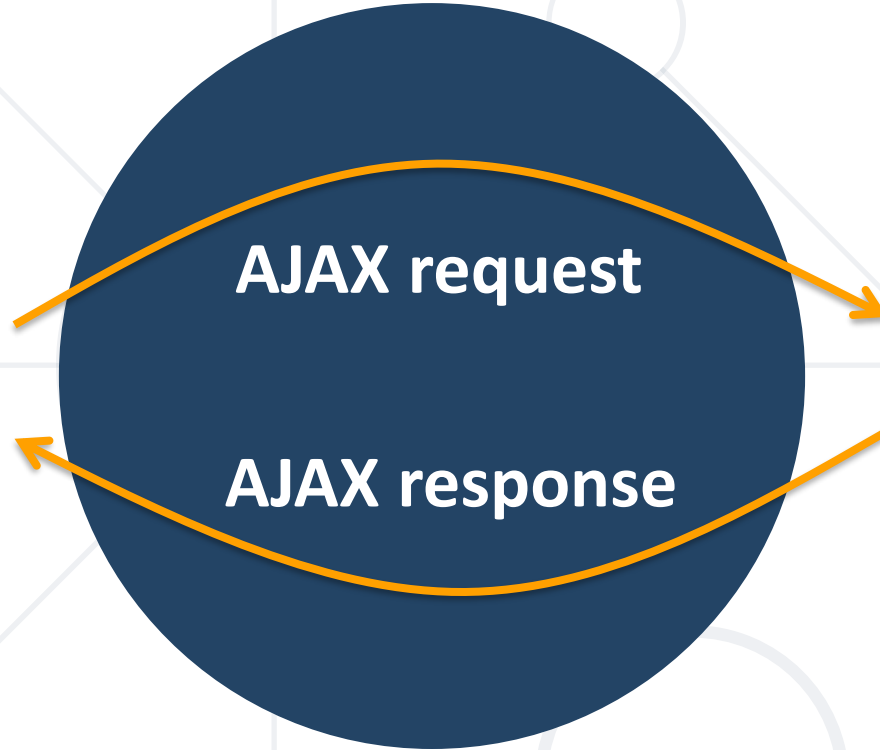
- **Asynchronous JavaScript And XML**
  - Background loading of **dynamic content/data**
  - Load data from the Web server and **render** it
- Two types of AJAX
  - **Partial page rendering**
    - Load HTML fragment + show it in a **<div>**
  - **JSON service**
    - Loads JSON object and displays it

# AJAX: Workflow





Web Client



Web Server

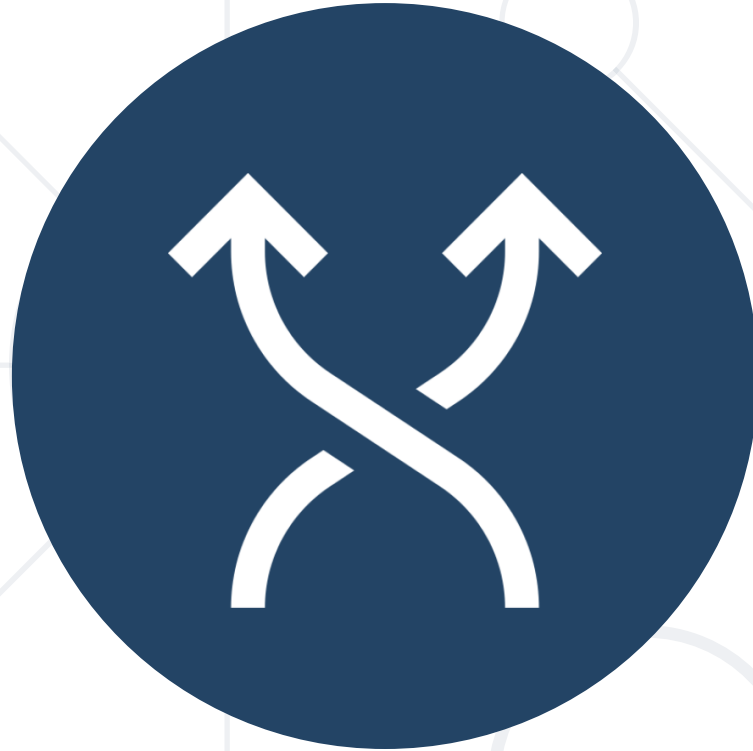
# Using the XMLHttpRequest Object

# XMLHttpRequest – Standard API for AJAX

```
<button id = "load">Load Repos</button>
<div id="res"></div>
```

```
let button = document.querySelector("#load");
button.addEventListener('click', function loadRepos() {
  let url = 'https://api.github.com/users/testnakov/repos';
  const httpRequest = new XMLHttpRequest();
  httpRequest.addEventListener('readystatechange', function () {
    if (httpRequest.readyState == 4 && httpRequest.status == 200) {
      document.getElementById("res").textContent = httpRequest.responseText;
    }
  });
  httpRequest.open("GET", url);
  httpRequest.send();
});
```





# **Synchronous vs Asynchronous**

Asynchronous Programming

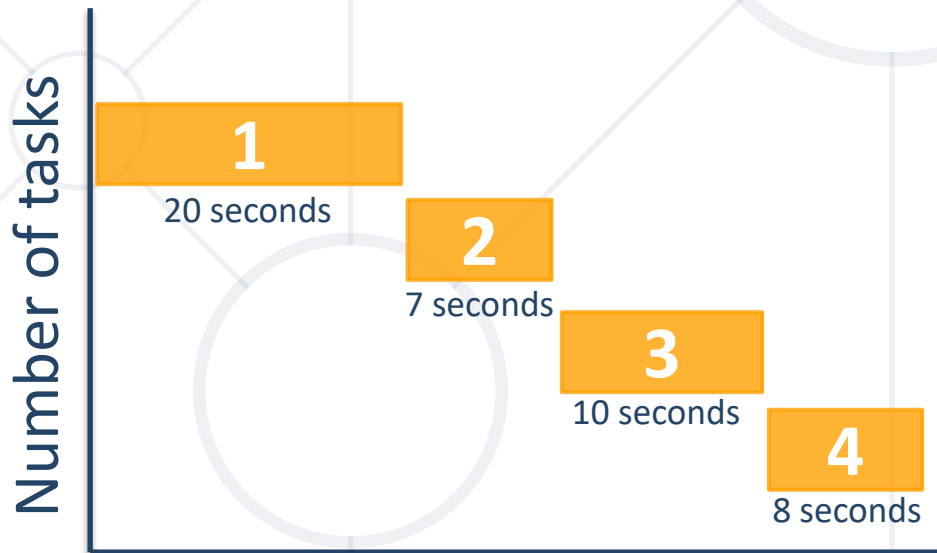
# Asynchronous Programming in JS



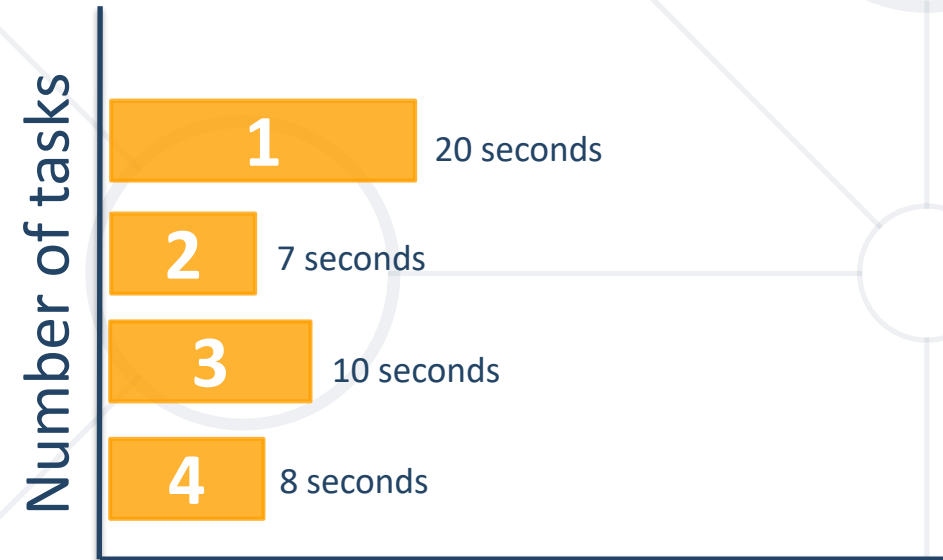
- Not the same thing as **concurrent** or **multi-threaded**
- There can be **asynchronous code**, but it is **generally single-threaded**
- Structured using **callback functions**
- In current versions of JS there are:
  - **Callbacks**
  - **Promises**
  - **Async Functions**

- Runs several tasks (pieces of code) in parallel, **at the same time**

## Synchronous



## Asynchronous



# Asynchronous Programming – Example

- The following commands will be executed as follows:



```
console.log("Hello.");  
setTimeout(function() {  
    console.log("Goodbye!");  
}, 2000);  
console.log("Hello again!");
```


```
// Hello.
```

```
// Hello again!
```

```
// Goodbye!
```

# Callbacks

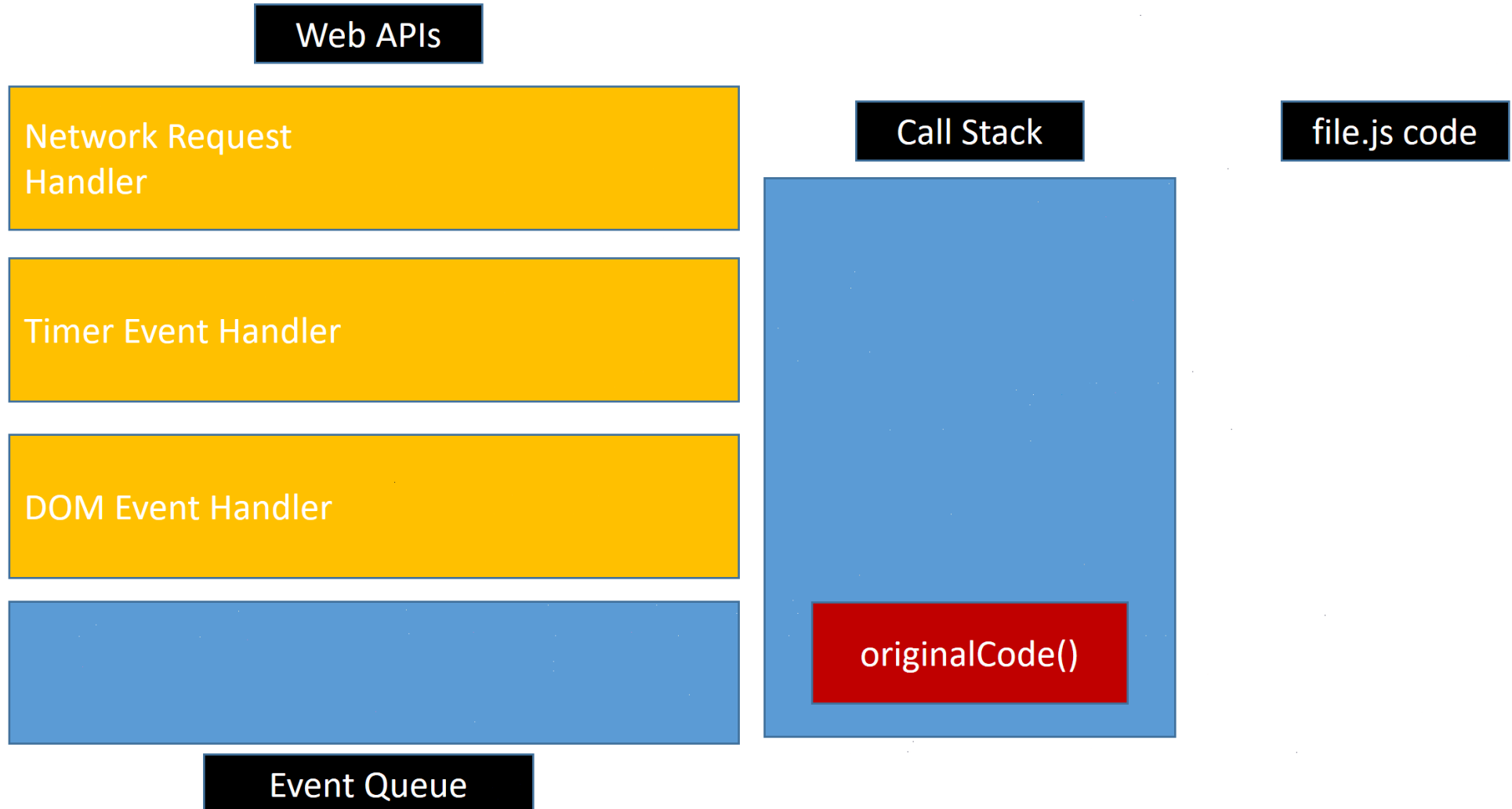
- Function **passed** into another function as an **argument**
- Then **invoked** inside the outer function to complete some kind of routine or action



```
function running() {  
    return "Running";  
}  
function category(run, type) {  
    console.log(run() + " " + type);  
}  
category(running, "sprint"); //Running sprint
```

Callback function

# Event Loop






# Promises

Objects Holding Asynchronous Operations

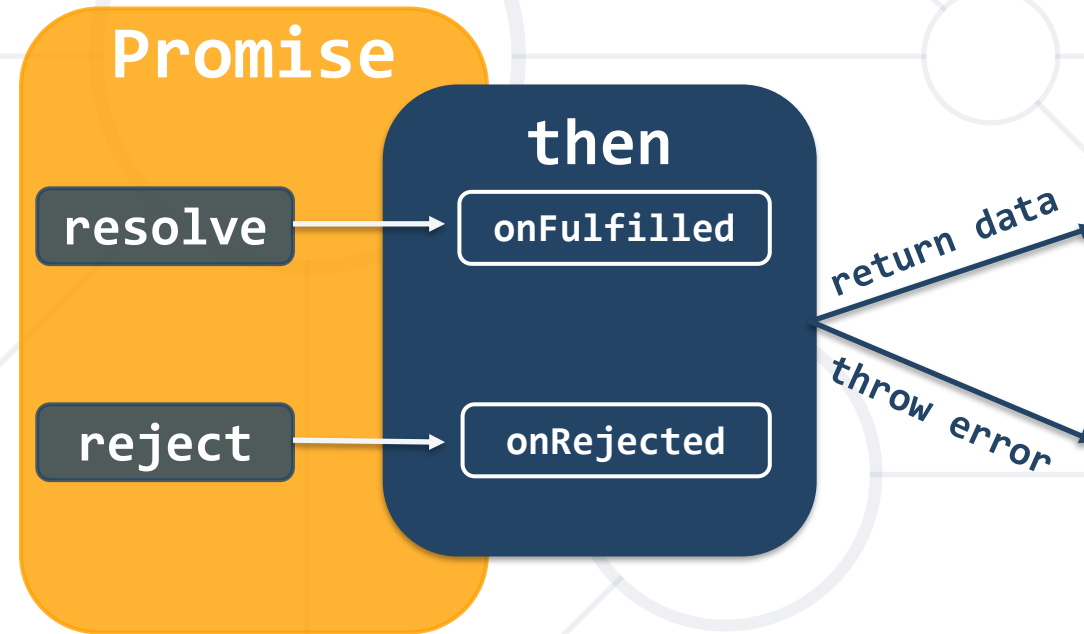
# What is a Promise?

- 
- A promise is an **asynchronous action** that **may complete** at some point and **produce a value**
  - States:
    - **Pending** - operation still running (unfinished)
    - **Fulfilled** - operation finished (the result is available)
    - **Failed** - operation failed (an error is present)
  - Promises use the **Promise** object

```
new Promise(executor);
```



# Promise Flowchart



# Promise.then() – Example

```
console.log('Before promise');
```

```
new Promise(function(resolve, reject) {  
  setTimeout(function() {  
    resolve('done');  
  }, 500);  
})  
.then(function(res) {  
  console.log('Then returned: ' + res);  
});
```

Resolved after 500 ms

```
console.log('After promise');
```

```
// Before promise
```


```
// After promise
```

```
// Then returned: done
```

# Promise.catch() – Example

```
console.log('Before promise');
```

```
new Promise(function (resolve, reject) {  
  setTimeout(function () {  
    reject('fail');  
  }, 500);  
})  
  .then (function (result) { console.log(result); })  
  .catch (function(error) { console.log(error); });
```




Rejected after 500 ms

```
console.log('After promise');
```

- **Promise.reject**(reason)
  - Returns an **object** that is **rejected** with the given **reason**
- **Promise.resolve**(value)
  - Returns an object that is **resolved** with the given **value**
- **Promise.all**(iterable)
  - Returns a **promise**
    - Fulfills when **all** of the promises **have fulfilled**
    - Rejects as soon as **one** of them **rejects**

- **Promise.allSettled**(iterable)
  - Wait until all promises have settled
- **Promise.race**(iterable)
  - Returns a promise that fulfills or rejects as soon as one of the promises in an iterable is settled
- **Promise.prototype.finally**()
  - The handler is called when the promise is settled

# What is Fetch?

- 
- The **fetch()** method allows making network requests
  - It is similar to **XMLHttpRequest** (XHR). The main **difference** is that the **Fetch API**:
    - Uses **Promises**
    - Enables a **simpler** and **cleaner** API
    - Makes code more readable and maintainable

```
fetch('./api/some.json')  
  .then(function(response) {...})  
  .catch(function(err) {...})
```

- The response of a **fetch()** request is a **Stream** object
- The **reading** of the stream happens **asynchronously**
- When the **json()** method is called, a **Promise** is **returned**
  - The **response status** is checked (should be **200**) **before parsing** the response as **JSON**

```
if (response.status !== 200) {  
    // handle error  
}  
response.json()  
    .then(function(data) { console.log(data)})
```

# GET Request

- **Fetch API** uses the **GET** method so that a direct call would be like this

```
fetch('https://api.github.com/users/testnakov/repos')  
  .then((response) => response.json())  
  .then((data) => console.log (data))  
  .catch((error) => console.error(error))
```





# POST Request

- To make a **POST** request, we can set the **method** and **body** parameters in the **fetch()** options

```
fetch('/url', {  
  method: 'post',  
  headers: { 'Content-type': 'application/json' },  
  body: JSON.stringify(data),  
})
```




- **clone()** create a clone of the response
- **json()** resolves the promise with JSON
- **redirect()** create new promise but with different URL
- **text()** resolves the promise with string
- **arrayBuffer()** resolve body with ArrayBuffer
- **blob()** resolve body with Blob (file, image, etc.)
- **formData()** resolve body with FormData

- **basic** - normal, same origin response
- **cors** - response was received from a valid cross-origin request
- **error** - error network
- **opaque** - Response for "no-cors" request to cross-origin resource
- **opaqueredirect** - the fetch request was made with **redirect: "manual"**

# Chaining Promises

- When working with a JSON API, you can:
  - Define the **status** and **JSON parsing** in **separate functions**
  - The functions **return promises** which can be **chained**



```
fetch('users.json')  
  .then(status)  
  .then(json)  
  .then(function(data) {...})  
  .catch(function(error) {...});
```

# Problem: Load GitHub Commits

GitHub username:

```
<input type="text" id="username" value="nakov" /> <br>
```

Repo: `<input type="text" id="repo" value="nakov.io.cin" />`

```
<button onclick="loadCommits()">Load Commits</button>
```

```
<ul id="commits"></ul>
```

```
<script>
```

```
function loadCommits() {
```

```
    // Use Fetch API
```

```
}
```

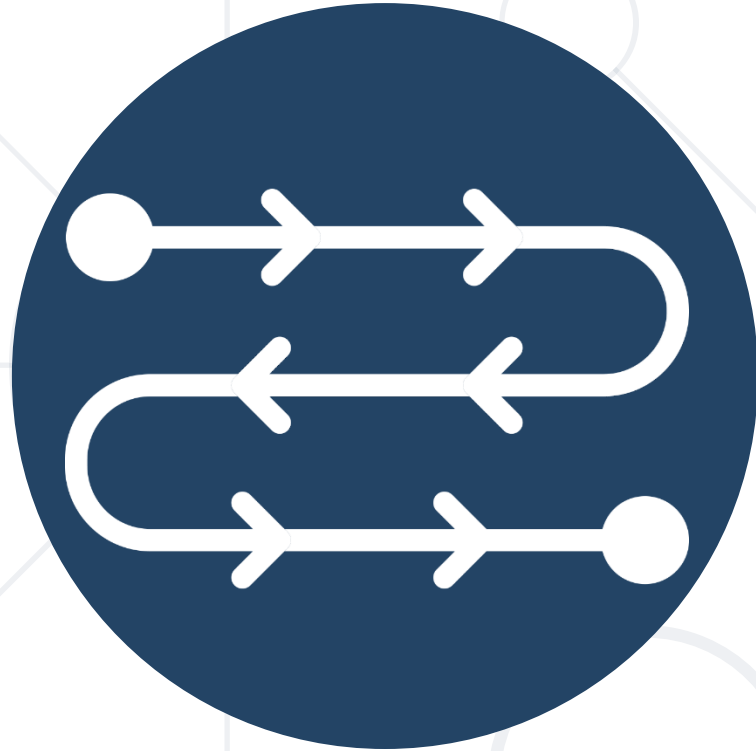
```
</script>
```

GitHub username:

Repo:

**Load Commits**

- Svetlin Nakov: Delete Console.Cin.v11.suo
- Svetlin Nakov: Create LICENSE
- Svetlin Nakov: Update README.md
- Svetlin Nakov: Added better documentation



**Async / Await**

Simplified Promises

# Async Functions

- Returns a **promise**, that can await other promises in a way that **looks synchronous**
- Operate **asynchronously** via the event loop
- Contains an **await** expression that:
  - Is **only valid** inside **async functions**
  - **Pauses** the execution of that function
  - Waits for the Promise's **resolution**



# Async Functions (2)



```
function resolveAfter2Seconds() {  
  return new Promise(resolve => {  
    setTimeout(() => {  
      resolve('resolved');  
    }, 2000);  
  });  
}
```

Expected output:

```
// calling  
// resolved
```

```
async function asyncCall() {  
  console.log('calling');  
  let result = await resolveAfter2Seconds();  
  console.log(result);  
}
```



- Do not confuse **await** with **Promise.then()**
  - **await** is always used for a **single promise**
  - To **await two or more** promises in **parallel**, use **Promise.all()**
- If a promise resolves normally, then **await** promise **returns the result**
- In case of a rejection, it **throws an error**

# Async/Await vs Promise.then

## ■ Promise.then

```
function logFetch(url) {  
  return fetch(url)  
    .then(response => {  
      return response.text()  
    })  
    .then(text => {  
      console.log(text);  
    })  
    .catch(err => {  
      console.error(err);  
    });  
}
```

## ■ Async/Await

```
async function logFetch(url) {  
  try {  
    const response =  
      await fetch(url);  
    console.log(  
      await response.text()  
    );  
  }  
  catch (err) {  
    console.log(err);  
  }  
}
```



# Error Handling



```
async function f() {  
  try {  
    let response = await fetch();  
    let user = await response.json();  
  } catch (err) {  
    // catches errors both in fetch and response.json  
    alert(err);  
  }}  

```

```
async function f() {  
  let response = await fetch();  
}  
// f() becomes a rejected promise  
f().catch(alert);  

```

- To execute different promise methods **one by one**, use **Async /Await**

```
function execute(x,sec) {  
  return new Promise(resolve => {  
    console.log('Start: ' + x);  
    setTimeout(() => {  
      console.log('End: ' + x);  
      resolve(x);  
    }, sec *1000);  
  });  
}
```

```
async function serialFlow() {  
  let result1 = await execute(1, 1);  
  let result2 = await execute(2, 2);  
  let result3 = await execute(3, 3);  
  let finalResult = result1 + result2 + result3;  
  console.log(finalResult);  
}
```

```
// Start: 1  
// End: 1  
// Start: 2  
// End: 2  
// Start: 3  
// End: 3  
// 6
```

# Concurrent Execution



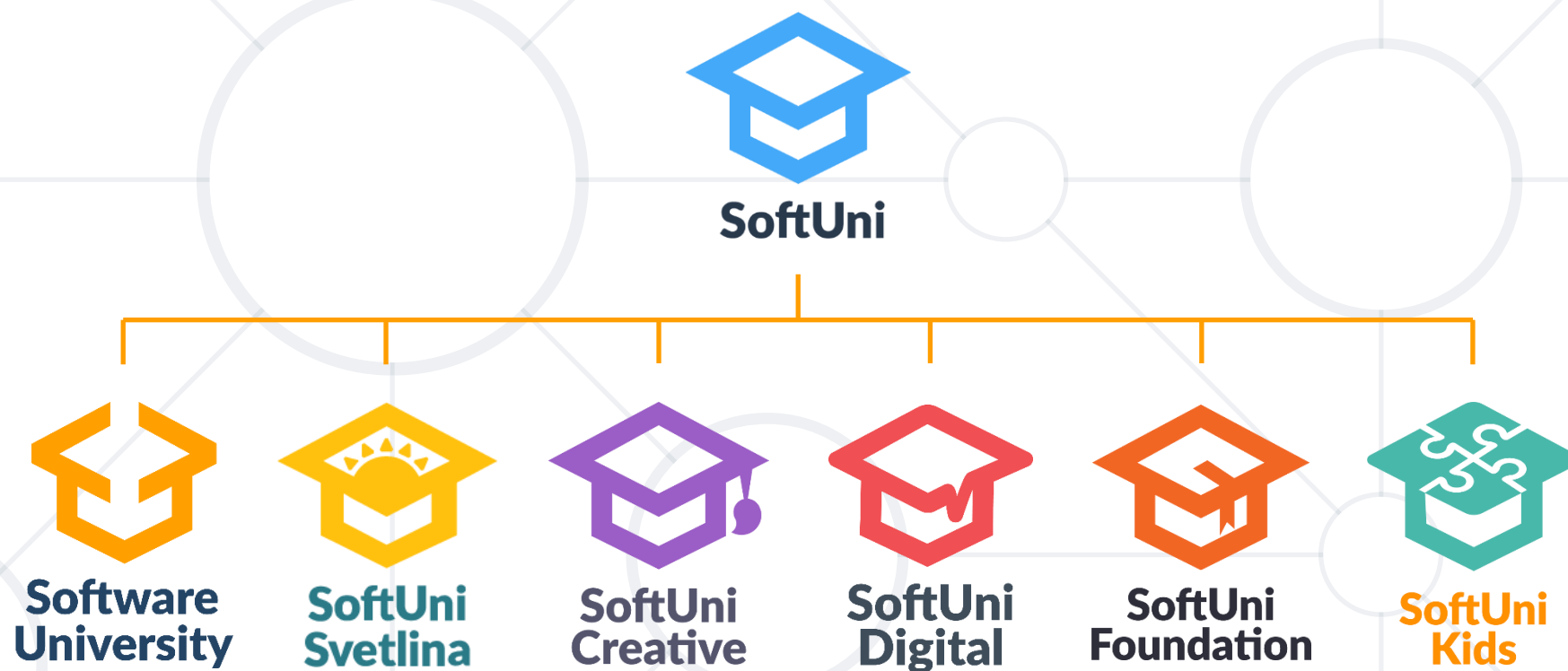
```
async function parallelFlow() {  
  let result1 = execute(1,1);  
  let result2 = execute(2,2);  
  let result3 = execute(3,3);  
  let finalResult = await result1 +  
                    await result2 +  
                    await result3;  
  console.log(finalResult);  
}
```

```
// Expected output:  
// Start: 1  
// Start: 2  
// Start: 3  
// End: 1  
// End: 2  
// End: 3  
// 6
```

- Asynchronous programming
  - Runs **several tasks** in **parallel**, at the **same time**
- Promises hold **operations**
  - Can be **resolved** or **rejected**
- **Async** functions contain an **await** expression
  - **Yields** the **execution**
  - **Waits** for the **Promise's resolution**



# Questions?



# SoftUni Diamond Partners



**Coca-Cola HBC**  
Bulgaria



**INFRAGISTICS®**



**SmartIT**



**SOFTWARE  
GROUP**

**INDEAVR**

Serving the high achievers



**Postbank**

Решения за твоето утре



**MOTION SOFTWARE**



**SUPER  
HOSTING  
.BG**



# Educational Partners



- Software University – High-Quality Education, Profession and Job for Software Developers

- [softuni.bg](http://softuni.bg), [softuni.org](http://softuni.org)

- Software University Foundation

- [softuni.foundation](http://softuni.foundation)

- Software University @ Facebook

- [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)

- Software University Forums

- [forum.softuni.bg](http://forum.softuni.bg)



- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://softuni.org>
- © Software University – <https://softuni.bg>

