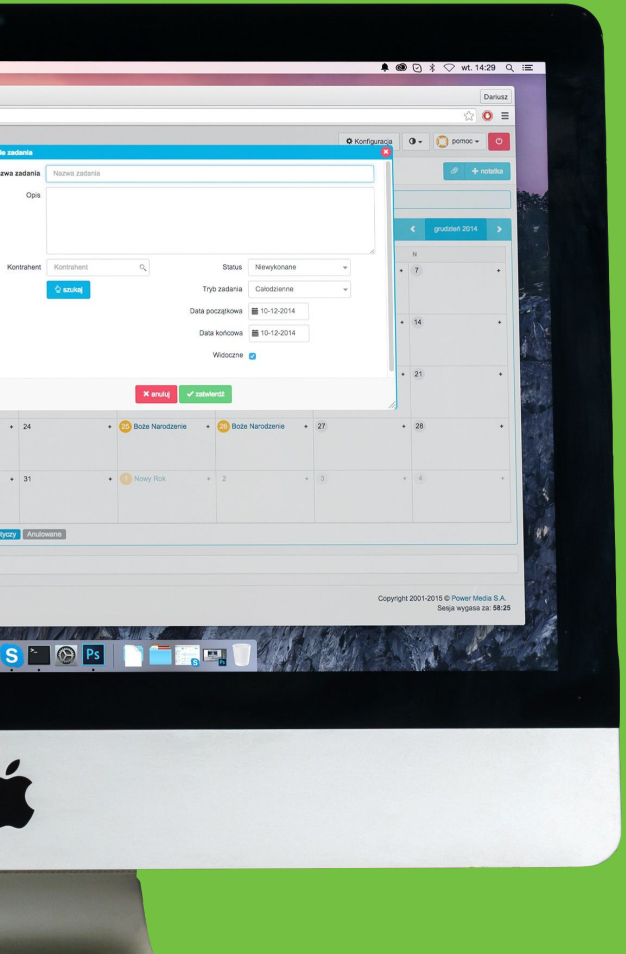


4CS017

Internet Software Architecture

ISA

Lecture Week 4

Asynchronous Javascript

Previously covered Area

- Basics of JavaScript(Previous semester)
- Functions
- DOM
- Events

This week's agenda

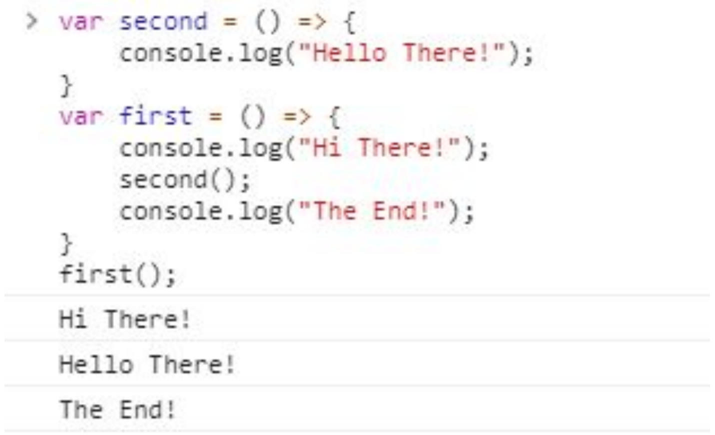
- Asynchronous JavaScript
- Call Back functions
- Promises
- Async/Await
- Fetch API
- ES6 and Beyond

Asynchronous JavaScript...

- JavaScript is a single-threaded programming language, which means only one thing can happen at a time.
- JavaScript engine can only process one statement at a time in a single thread.
- While single-threaded languages simplify writing codes because you don't have to worry about the concurrency issues(Race Condition, DeadLocks, Starvation), this also means you cannot perform long operations such as network access without blocking the main thread.
- A good example of this happening is the window alert function **alert("Hello World")**

Asynchronous JavaScript...

- Using Asynchronous JavaScript (AJAX, Callback, Promises, async-await), you can perform long network requests without blocking the main thread.
- Let's take an example of a synchronous programming in JavaScript:



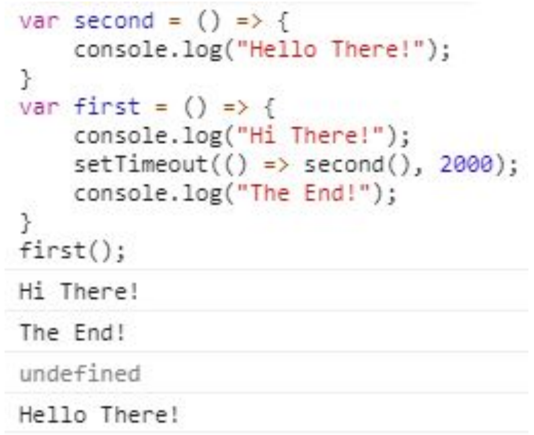
```
> var second = () => {
      console.log("Hello There!");
  }
  var first = () => {
      console.log("Hi There!");
      second();
      console.log("The End!");
  }
  first();
```

```
Hi There!
Hello There!
The End!
```

Asynchronous JavaScript...

- From the above example, the console will log, “Hi There!”, “Hello There!” and “The End!” sequentially.
- As you can see, JavaScript logs the later statements only after the above statement is completed.
- Now, let's take a look at how a basic asynchronous programming works.
- We will use a **setTimeout()** function to achieve that. Remember this function is not a JavaScript function but rather a Web API (browser-function).

Asynchronous JavaScript...



```
var second = () => {
    console.log("Hello There!");
}
var first = () => {
    console.log("Hi There!");
    setTimeout(() => second(), 2000);
    console.log("The End!");
}
first();
```

```
Hi There!
The End!
undefined
Hello There!
```

- Now, if you try this code in your browser, this will log “Hi There!”, “The End!” and then wait 2 seconds to log “Hello There!”.
- This way the code worked on two different thread without blocking the main thread.

Asynchronous JavaScript...

What will be the output of the following code ??

```
function fetchData() {

 console.log("Fetching data...");

 setTimeout(() => {

  console.log("Data fetched!");

  const data = { name: "John", age: 30 };

  console.log(data);

 }, 2000);

}

fetchData();

console.log("Fetching data, please wait...");
```

Asynchronous JavaScript...

- Although **setTimeout()** is not the best way to achieve all asynchronous programming, but is enough to help you understand how asynchronous JavaScript works.
- We will now move more into the asynchronous JavaScript. Lets have a look at **Ajax**, **Callback functions** and **Promises**.

AJAX

- Ajax (Asynchronous JavaScript and XML) is a technique used to send and receive data from a server without refreshing the entire page.

- It allows you to update specific parts of a web page asynchronously, which can improve the user experience and reduce network traffic.

- Ajax requests can be made using the XMLHttpRequest object in JavaScript, which provides a way to send and receive data from a server using HTTP or HTTPS protocols.

AJAX..

- Here are the basic steps to get started with AJAX in JS:

1. Create an XMLHttpRequest object:

```
const xhr = new XMLHttpRequest();
```

2. Open a connection to the server:

```
xhr.open('GET', 'https://api.example.com/data');
```

3. Send the request:

```
xhr.send();
```

AJAX..

4. Handle the response:

```
xhr.onreadystatechange = function() {
  if (xhr.readyState === XMLHttpRequest.DONE) {
    if (xhr.status === 200) {
      console.log(xhr.responseText);
    } else {
      console.log('Error!');
    }
  }
};
```

Any
Questions

Callback Functions...

- A callback function is a function that is passed as an argument to another function, to be “called back” at a later time.
- A callback function can either be a general function or an anonymous function.
- Callback functions are great way to **handle something** after **something else** has been completed. By something here we mean a **function execution**.
- If we want to execute a function right after the return of some other function, then callback can be used.

Callback Functions

- For JavaScript to know when an asynchronous operation has a result, it points to a function that will be executed once the result is ready. That function is called a callback function.
- Here the result can either be the **data** or an **error** that occurred during the process.

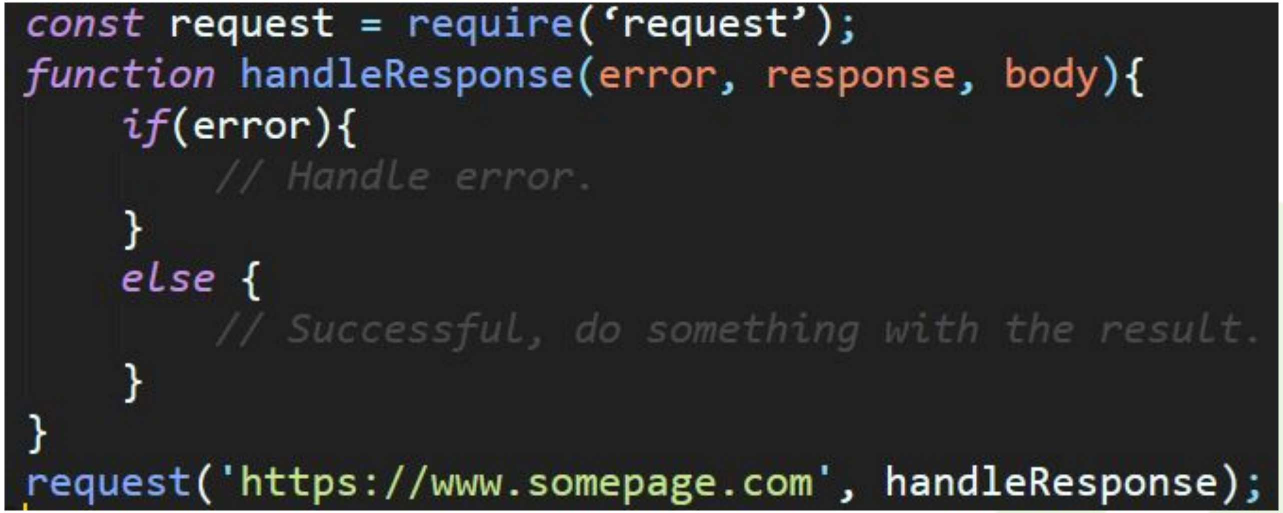
Callback Functions Example:

Here is a simple example of callback function:

```
function greet(name, callback) {

  console.log(`Hello, ${name}!`);
  callback();
}

function sayGoodbye() {

  console.log("Goodbye!");
}

greet("John", sayGoodbye);
```

Callback Functions...

- Here is a another example of callback function.



```
const request = require(‘request’);
function handleResponse(error, response, body){
    if(error){
        // Handle error.
    }
    else {
        // Successful, do something with the result.
    }
}
request('https://www.somepage.com', handleResponse);
```

Callback Functions: Callback Hell...

- Callbacks are a good way to declare what will happen once an I/O operation has a result, but what if you want to use that data in order to make another request?
- You can only handle the result of the request within the callback function provided.

Callback Functions: Callback Hell...

- "Callback hell" is a term used to describe a situation where a large number of nested callback functions are used in JavaScript code.
- This can happen when multiple asynchronous tasks need to be performed one after the other, and each task requires a callback function to handle its completion.
- The result can be code that is difficult to read, understand, and maintain.

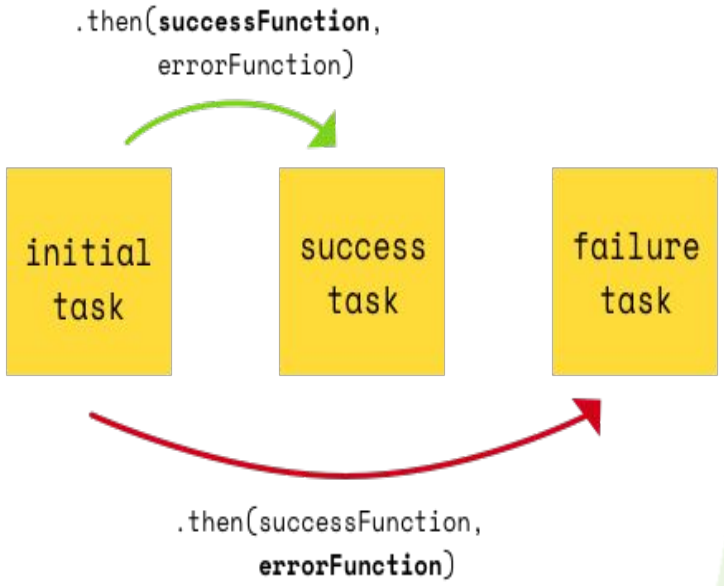
Callback Functions: Callback Hell...

- This will make our code messy to read and very difficult to understand.
- With that pattern, it means we need to check for errors in every callback. Which just adds to the mess when dealing with multiple requests and callbacks.
- This anti-pattern is what we call **Callback Hell**.

Callback Functions: Callback Hell Example

```
function step1(data, callback) {
  console.log("Step 1 with " + data + " is complete");
  callback();
}
function step2(data, callback) {
  console.log("Step 2 with " + data + " is complete");
  callback();
}
function step3(data, callback) {
  console.log("Step 3 with " + data + " is complete");
  callback();
}
step1("data", function() {
  step2("more data", function() {
    step3("even more data", function() {
      console.log("All steps are complete");
    });
  });
});
```

Any Questions


.then(successFunction,
errorFunction)
initial task
success task
failure task
.then(successFunction,
errorFunction)

JavaScript Promises...

- A promise is an object that wraps an asynchronous operation and notifies when its done.
- They are easy to manage when dealing with multiple asynchronous operations where callbacks can create callback hell leading to unmanageable codes.
- Promises are the ideal choice for handling asynchronous operations in the simplest manner. They can handle multiple asynchronous operations easily and provide better error handling than callbacks.
- Promise always returns a response.

JavaScript Promises...

A promise represents a value that may not be available yet, but will be at some point in the future.

Promises have three states:

1. **Pending:**

   The initial state. The promise is neither fulfilled nor rejected.

2. **Fulfilled:**

   The operation completed successfully and the promise now has a resulting value.

3. **Rejected:**

   The operation failed and the promise has a reason for the failure.

JavaScript Promises…

- Instead of providing a callback, promises has its own method which you call to tell the promise what will happen if it is successful or when it fails.
- The Promise constructor takes a single argument, a function that has two parameters: resolve and reject.
- The resolve function is called when the operation is successful, and the reject function is called when the operation fails.
- The methods a promise provides are "then(…)" for when a successful result is available and "catch(…)" for when something went wrong.

JavaScript Promises...

```
let promise = new Promise(function(resolve, reject) {

  setTimeout(function() {
    resolve('Success!');
  }, 1000);
});

promise.then(function(value) {

  console.log(value); // Output: Success!

}).catch(function(error) {
  console.log(error);
});
```

JavaScript Promises...

Promises can also be chained together using the then() method. This allows you to write code that performs multiple asynchronous operations in a sequence.

```
let promise1 = new Promise(function(resolve, reject) {
  setTimeout(function() {
    resolve(1);
  }, 1000);
});
let promise2 = promise1.then(function(value) {
  console.log(value); // Output: 1
  return value + 1;
});
promise2.then(function(value) {
  console.log(value); // Output: 2
});
```

JavaScript Promises...

```
function step1(data) {
  return new Promise(function(resolve, reject) {
    console.log("Step 1 with " + data + " is complete");
    resolve("more data");
  });
}
function step2(data) {
  return new Promise(function(resolve, reject) {
    console.log("Step 2 with " + data + " is complete");
    resolve("even more data");
  });
}

function step3(data) {
  return new Promise(function(resolve, reject) {
    console.log("Step 3 with " + data + " is complete");
    resolve();
  });
}
```

```
step1("data")
  .then(step2)
  .then(step3)
  .then(function() {
    console.log("All
steps are complete");
  });
```

Async/Await:

- Async/await is a syntactical feature of JavaScript that provides an alternative way of working with promises.
- It allows you to write asynchronous code in a synchronous style, making your code more readable and easier to reason about.
- Async/await works by allowing you to declare a function as "async" and then use the "await" keyword to wait for promises to be fulfilled or rejected.
- When you use the "await" keyword in an async function, it pauses the execution of the function until the promise is resolved or rejected.

Async/Await:

- Async/await is a syntactical feature of JavaScript that provides an alternative way of working with promises.
- It allows you to write asynchronous code in a synchronous style, making your code more readable and easier to reason about.
- Async/await works by allowing you to declare a function as "async" and then use the "await" keyword to wait for promises to be fulfilled or rejected.
- When you use the "await" keyword in an async function, it pauses the execution of the function until the promise is resolved or rejected.

Async/Await:

```
async function example() {
  try {
    let result = await somePromise();
    console.log(result);
  } catch(error) {
    console.log(error);
  }
}
```

Async/Await:

Async/await can also be used to simplify the chaining of promises.

```
async function example() {
  try {
    let result1 = await somePromise1();
     Let result2 = await somePromise2(result1);
    console.log(result2);
  } catch(error) {
    console.log(error);
  }
}
```

The second promise, is dependent on the result of the first promise.

By using the "await" keyword on each promise, we can write the code in a synchronous style and avoid deeply nested callbacks.

HERALD COLLEGE KATHMANDU

Fetch API:

The Fetch API is a modern API for making network requests in JavaScript.

It provides a simple and standardized way to fetch resources from the network, such as images, JSON data, and HTML documents.

```
fetch('https://example.com/data.json')
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console error(error));
```

In this example, the fetch() method is called with a URL as its argument.

The .then() method is called on the Promise object to handle the response.

Fetch API:

In addition, the Fetch API can be used with async/await syntax to simplify the code and make it more readable:

```
async function fetchData() {

  try {

    const response = await fetch('https://example.com/data.json');

    const data = await response.json();

    console log(data);

  } catch(error) {

    console error(error);

  }
```

Before you come for Lab, Research!!

- HTML DOM Events
- HTML DOM Elements
- JavaScript Promise
- JavaScript Axios
- Using Axios
- All about Promise

well that's the end
Goodbye 2o