

Asynchronous JavaScript:

Beyond the basics

In the last lecture, we covered the basics of asynchronous code in JavaScript, including callbacks, promises, `async/await`, and how they can be used to control the flow of your code and handle time-consuming tasks (such as `setTimeout`) without freezing the rest of your application. In this lecture, we will be focusing on the `fetch` method and how it can be used to query information on the web, typically through the use of APIs.

We previously used `localStorage` as a way to store and retrieve data in our JavaScript applications. With `localStorage`, it was easy to use the `getItem()` or `setItem()` methods to instantly read or update data. However, when we want to retrieve data from an external source, such as a web API, we don't know if or when the data we are requesting is available. JavaScript has a built method called **`fetch()`**. The `fetch` method allows us to make network requests and retrieve data from a URL. But, because these requests can be slow, JavaScript wraps the `fetch` response inside of a promise object, so that we can handle it in the background.

Using Fetch

The `fetch` function is a built-in JavaScript function that allows you to retrieve data from a URL, we simply pass in the URL of the resource we want to retrieve as the first argument. The browser then handles the request and returns a promise that we can use to handle the response.

This can be useful for querying information from a website or API. We will be using the [Json placeholder API](#), which is a great resource for testing out different ways of fetching data from an API because it is a completely fake API that can be used for anything. To get started, we can write `fetch(url)` and pass in the URL of the API we want to query. For example, if we want to query the users from the Json placeholder API, we can use the following URL:

<https://jsonplaceholder.typicode.com/users>

Once we run the `fetch` function, it will return a promise with a response object. To handle the promise, we can use the `.then` method and access the response object. For example:

```
fetch("https://jsonplaceholder.typicode.com/users")
  .then(response => console.log(response))
```

This will show the following in the console:

Response

```
{
  type: "cors",
  url: "https://jsonplaceholder.typicode.com/users",
  redirected: false,
  status: 200,
  ok: true,
  statusText: "OK",
  headers: {
    "cache-control": "max-age=43200",
    "content-type": "application/json; charset=utf-8",
    expires: "-1",
```

```

    pragma: "no-cache"
  },
  body: ReadableStream,
  bodyUsed: false
}

```

The response object contains information about the request, such as the headers, status, and the body of the response. The body is in a **readable stream format**, which means we can't directly access the data. Instead, we need to process the data inside of body through another Promise. To do this, we can call the `json()` method on the response object. This returns another promise that resolves with the data in JSON format.

For example, to retrieve the list of users from the Json Placeholder API, we can use the following code:

```

fetch("https://jsonplaceholder.typicode.com/users")
  .then(response => response.json())
  .then(data => console.log(data))

```

In this example, we are calling `fetch` with the URL of the API, then we are converting the response to JSON, and finally, we are logging the data to the console. You can see that the data we get is an array of 10 objects, each representing a user.

It's worth noting that `fetch` returns a promise, and the `response` and `json` methods also return promises, that's why we are chaining the `then` methods to handle the data correctly.

We can rewrite the above code using `async-await` instead of chaining two `.then` methods, for example like this:

```

// first need to declare an async function, because the 'await' keyword
// can only be used inside of async functions.
async function getUsers() {
  const response = await fetch("https://jsonplaceholder.typicode.com/users")
  const data = await response.json()
  console.log(data)
}
getUsers() // call the getUsers function

```

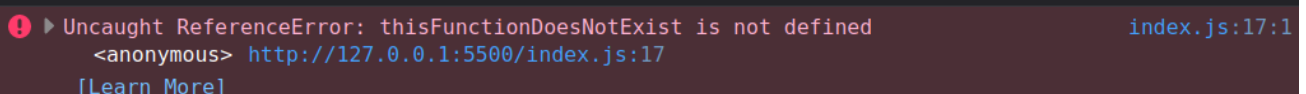
Error handling

Before we continue with the `fetch` API, its important to understand javascript error handling, how and when to use it. First lets look at what happens when we try to call an undeclared function:

```

thisFunctionDoesNotExist()
console.log("Is our script still running?")

```



```

! ▶ Uncaught ReferenceError: thisFunctionDoesNotExist is not defined      index.js:17:1
    <anonymous> http://127.0.0.1:5500/index.js:17
    [Learn More]

```

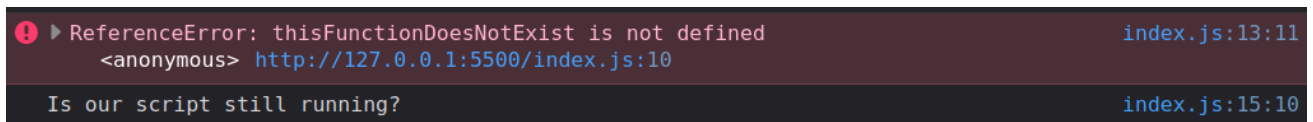
Javascript automatically shows the error for us, and in this case it tells us that this is a reference error, because `thisFunctionDoesNotExist` is not defined. But notice how it says it's an "Uncaught" ReferenceError, and also how this line doesn't execute:

```
console.log("Is our script still running?")
```

This is because we haven't "caught" this type of error, uncaught errors cause javascript to stop execution (it crashes the script). So, how do we catch errors in Javascript ?

We wrap our potentially problematic code in a try-catch block, for example:

```
try {
  thisFunctionDoesNotExist()
} catch (error) {
  console.error(error)
}
console.log("Is our script still running?")
```

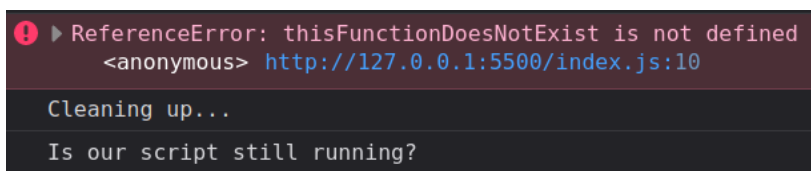


The screenshot shows a browser console with two entries. The first entry is an error: "ReferenceError: thisFunctionDoesNotExist is not defined" at index.js:13:11. The second entry is a log message: "Is our script still running?" at index.js:15:10.

Notice two differences: 1) The error is still logged to the console, but it no longer says "uncaught", and 2) the rest of our script continues to execute.

This is the most basic way to catch errors, and is most often enough. But error handling provides one more block called: `finally`. The finally block executes after either try succeeds or catch block runs, the only point of the finally block is to "clean up" any side effects our code may have caused before executing the rest of the code. (example of side effects: may be mutating a global variable, or changing some state that needs to be reset).

```
try {
  thisFunctionDoesNotExist()
} catch (error) {
  console.error(error)
} finally {
  console.log("Cleaning up...")
}
console.log("Is our script still running?")
```



The screenshot shows a browser console with three entries. The first entry is an error: "ReferenceError: thisFunctionDoesNotExist is not defined" at index.js:10. The second entry is a log message: "Cleaning up...". The third entry is a log message: "Is our script still running?".

If we are using `.then` to resolve promises, we don't need to write the entire try-catch block, the Promise object has built in both a `.catch` and a `.finally` method. You will most likely use `async-await` anyway, but just to show what that looks like, for example:

```
// declare a promise that just contains a string, and immediately resolve it manually.
Promise.resolve("Hi")
  .then((data) => console.log(data, " <- This promise got resolved!"))
  .catch((error) => console.error("Error: This promise refused to say", error))
```

```
.finally(() => console.log("Cleaning up"))
```

The above code will log to console the following:

```
Hi <- This promise got resolved!
Cleaning up
```

Let's try to manually reject a promise, so that we can catch the error:

```
// declare a promise that just contains a string, and immediately reject it manually.
Promise.reject("Hi")
  .then((data) => console.log(data, " <- This promise got resolved!"))
  .catch((error) => console.error("Error: This promise refused to say",
error))
  .finally(() => console.log("Cleaning up"))
```

The above code will log to console the following:

```
! ▶ Error: This promise refused to say Hi
Cleaning up
```

That may have been a lot of information, the key takeaway is this: **Uncaught errors are bad errors**, **caught errors are good errors**, and **finally is only needed if your code has side-effects that needs cleanup**.

Back to the Fetch API, the good news about it is that its very difficult to get the Fetch API to produce an **uncaught error**.

Let's declare a function similar to our `getUsers()` function, but this time with a try-catch-finally block, in this function we will want to get data on a particular user (from the url:

<https://jsonplaceholder.typicode.com/users/1> where 1 is the id of a user), for example:

```
async function getUserById(id) {
  try {
    const response = await fetch(`https://jsonplaceholder.typicode.com/users/${id}`)
    const data = await response.json()
    console.log(data)
  } catch (error) {
    console.error(error)
  } finally {
    console.log("cleaning up")
  }
}
```

```
// let's call this function with a user-id we know exists
getUserById(1)
```

```
▶ Object { id: 1, name: "Leanne Graham", username: "Bret", ema
}
cleaning up
```

And that went well, we get userdata in response, and the the finally block executes.

But what happens if we call it with a userid that doesn't exist ?

```
getUserById(123456789)
```

```
▶ XHR GET https://jsonplaceholder.typicode.com/users/123456789
▶ Object { }
cleaning up
```

Notice how there is **no error**, and the script continues to execute just fine, it logs out an empty object after the 404 Not Found message, and the the finally block executes.

What the 404 Not Found is telling us is simply that we made a request to an url that cannot be found.

Pretty much the only way to get the fetch API to produce an uncaught error is if either the users computer or the server that's hosting our website loses internet-connection. Other than that we generally don't have to worry about uncaught errors.

Checking a the Response Status

So, we generally won't get errors, but it's important to check the status of the response to make sure it is successful before trying to access the data. We can use the `.ok` property on the response object to check if the response is successful. It returns a Boolean value of true or false. For example:

```
async function getUserById(id) {
  const response = await fetch(`https://jsonplaceholder.typicode.com/users/${id}`)
  if (response.ok) {
    const data = await response.json()
    console.log(data)
  }
  else {
    if (response.status === 404) console.log("user not found.")
    else console.log("something else went wrong", response.status)
  }
}
```

So we have `if (response.ok) { ...code... }` and then we have an `else` block as well. In the `response.ok` block we grab the data through `response.json()`, while in the `else` block we can display custom messages based on the `response.status` code.

Then if we call this function in an existing userId, for example 1 we get a response and the data is logged out to console:

```
getUserById(1)
```

But if we call the same function with a non-existent user, for example:

```
getUserById(113987891375)
```

Then it will log out "user not found."

Now that we know how to handle the response data, we can move on to using the fetch() function to make more advanced requests.

POST, PUT, and DELETE requests

- GET requests are used to retrieve data from a server. This is the default fetch() behaviour (what we used so far)
- POST requests are used to submit data to a server. They are typically used to create new resources.
- PUT requests are used to update existing data on a server. They are typically used to update existing resources.
- DELETE requests are used to delete data from a server. They are typically used to delete resources.

Below is an example of using the POST request:

```
async function postComment(userId, title, comment) {
  const response =
    await fetch('https://jsonplaceholder.typicode.com/posts',
    {
      method: "POST",
      body: JSON.stringify({
        userId: userId,
        title: title,
        body: comment
      }),
      headers: {
        'Content-type': 'application/json; charset=UTF-8',
      }
    });
  if (response.ok) {
    const data = await response.json()
    console.log(data)
  }
}
```



```
postComment(1, "Hi", "How are you?")
```

In this example, we are using the `fetch()` function to send a post request to the server. The first argument is the URL that we are sending the request to, and the second argument is an options object.

The options object has a `method` property, which is set to "POST" to indicate that we are sending a post request. The `body` property is used to send data to the server, and in this case, we are sending a JSON object.

The `headers` property is used to set the headers of the request. In this example, we are setting the `Content-Type` to "application/json" to indicate that we are sending a JSON object.

The rest of the code is similar to the previous example, where we are handling the response and handling any errors that may occur.

Another important method is the PUT request, which is used to update existing data on the server. It works similarly to the POST method, but it sends a PUT request instead of a POST request.

```
async function updateUser() {
  const response = await fetch("https://jsonplaceholder.typicode.com/users/1",
  {
    method: 'PUT',
    body: JSON.stringify({
      name: 'Jane Doe',
      email: 'janedoe@example.com'
    }),
    headers: {
      'Content-Type': 'application/json'
    }
  });

  if (response.ok) {
    const data = await response.json();
    console.log(data);
  } else {
    if (response.status === 404) console.log("user not found.")
    else console.log("something else went wrong", response.status)
  }
}
```

In this example, we are using the `fetch()` function to send a put request to the server to update user with id of 1. The first argument is the URL that we are sending the request to, and the second argument is an options object. In this options object, we are passing in a `method` property that is set to "PUT" to indicate that we want to update data on the server.

We are also passing in the `body` property, which is a JSON representation of the data we want to update, in this case, the name and email of the user. The `headers` property is also passed in, which is used to set the content-type of the request to "application/json".

It is also possible to delete data on the server using the DELETE method, which works similarly to PUT method, but it sends a DELETE request instead of a PUT request.

```
async function deleteUser() {  
  const response = await fetch("https://jsonplaceholder.typicode.com/users/1", {  
    method: 'DELETE'  
  });  
  
  if(response.ok) {  
    console.log("User deleted successfully!");  
  } else {  
    if (response.status === 404) console.log("user not found.")  
    else console.log("something else went wrong", response.status)  
  }  
}
```

In this example, we are using the `fetch()` function to send a delete request to the server to delete user with id of 1. The first argument is the URL that we are sending the request to, and the second argument is an options object.

The options object has a `method` property, which is set to "DELETE" to indicate that we want to delete data on the server.