Net Ninja Modern JavaScript Notes

Last Updated: October 15, 2020

Sources

- [YouTube] The Net Ninja Modern JavaScript Tutorial (Link)
- [YouTube] The Net Ninja Asynchronous JavaScript Tutorial (Link)

Modern JavaScript Tutorial 4 – Functions

Callback Functions

- In JavaScript, a method is a function that belongs to an object.
    - We should also note that everything in JavaScript is an object – a function is an object, an array is an object, etc.
- When we pass a function in as an argument to another function, that function passed in is called a callback function.
    - In the example to the right, we are passing a function that logs a value to the console as an argument to myFunc(...).
    - Note that in this case, the function that is provided as an argument to myFunc(...) is an arrow function.
    - When myFunc(...) is called at the bottom of the screenshot, value is provided as an argument to callbackFunc(...) and a value of 50 is logged to the console.

```
const myFunc = (callbackFunc) => {
  // do something
  let value = 50;
  callbackFunc(value);
};

myFunc(value => {
  // do something
  console.log(value);
});
```

- Sometimes, large functions are defined outside of the function that uses it as a callback function, as shown in the Person example.
    - Generally, functions are provided directly inside the function signature as an arrow function argument, so this is what we will most often see.

```
let people = ['mario', 'luigi', 'ryu', 'shaun', 'chun-li'];

const logPerson = (person, index) => {
  console.log(`${index} - heelo ${person}`);
};

people.forEach(logPerson);
```

```
0 - hello mario
1 - hello luigi
2 - hello ryu
3 - hello shaun
4 - hello chun-li
```

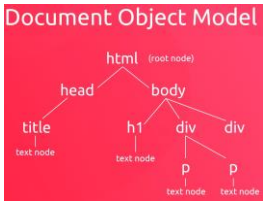
Modern JavaScript Tutorial 6 – JavaScript and the DOM

Selecting HTML Elements

- By using JavaScript with the Browser's Document Object Model (DOM), we can do things like add content to the browser, change CSS styles, react to user events (e.g. clicking), creating popups, etc.
    - If we want to edit a JavaScript file and see the interactions in the browser in real-time, we should use the 'Live Server' extension in VS Code.
    - Then we once we create an HTML page, we just right-click and choose to 'Open with Live Server'.

```
> document
▼#document
    <html lang="en">
    ▼<head>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width,
        initial-scale=1.0">
        <meta http-equiv="X-UA-Compatible" content="ie=edge">
        <title>Document</title>
      </head>
    ▼<body>
        <h1>The DOM</h1>
        <script src="sandbox.js"></script>
        <!-- Code injected by live-server -->
      ▶<script type="text/javascript">…</script>
      </body>
    </html>
```

- The DOM is created by the browser when an HTML document loads inside of it, where the browser creates an object that models the document (called the document object).
    - It contains many different properties about the HTML document, along with many methods that we can use to interact with the HTML document.
    - We can easily view the document by going to the console in developer tools and typing document.



- The DOM describes our webpage as a tree of nodes.
    - When we interact with the DOM, we do so by querying it to get a reference to an element (or set of elements) within the DOM.
- The best way to query the DOM is to use document.querySelector(...), providing a tag, class name, id, etc. as an argument.

    - We can combine selectors as well to increase specificity for the query selection.
    - In developer tools, if a person right-clicks on a tag, they can copy the selector for it directly.
- If we provide document.querySelector('p') as an argument, for example, the line will go through the document object from top-to-bottom and will grab the first 'p' tag, ignoring the rest.

```
const para = document.querySelector('p');

console.log(para);
```

```
<p>hello, world</p>
```

    - By assigning the reference to the object returned from the line to a constant, we now have a reference that we can use later in the JavaScript code.
- Once we have an object reference to a portion of the HTML, we can modify it by changing it, styling it differently, deleting it, etc.
- To select all elements that match a selector criteria, we can use document.querySelectorAll(...) to return a NodeList object.
    - In Shaun's opinion, the two query selectors described so far are the two best ways to select elements from the HTML document.
- A few more ways of getting elements include document.getElementById(...), document.getElementsByClassName(...), and document.getElementsByTagName(...).

Manipulating Content

- To get the inner text between HTML tags, we can use the innerText property (note that this is a property of an object, not a method) to get the text.

```
const para = document.querySelector('p');

console.log(para.innerText);
```

```
hello, world
```

    - To update the inner text, we simply set the property equal to some other text.
- To get the HTML inside of a selector (i.e. all HTML inside of a selector), we can use the innerHTML property.
    - As shown in the example, we can insert text and HTML dynamically using JavaScript and query selectors, along with document object properties.

```
const content = document.querySelector('.content');

//console.log(content.innerHTML);
//content.innerHTML += '<h2>THIS IS A NEW H2</h2>';

const people = ['mario', 'luigi', 'yoshi'];

people.forEach(person => {
  content.innerHTML += `<p>${person}</p>`;
});
```

```
<div class="content">
  <p>this is the content</p>
  <p>mario</p> == $0
  <p>luigi</p>
  <p>yoshi</p>
</div>
```

- Besides HTML properties, we can also change HTML attributes such as href links.
    - The setAttribute(...) method can be used on a document object to get the attribute that you would like to change, as well as set the new attribute that you would like to change to.

```
const link = document.querySelector('a');

console.log(link.getAttribute('href'));
link.setAttribute('href', 'https://www.thenetninja.co.uk');
```

```
<h1>The DOM</h1>
<a href="https://www.thenetninja.co.uk">Link to
</a> == $0
<p class="error">lorem ipsum</p>
```

- Note that the attribute doesn't even have to have been applied to the element for you to apply it – say if you wanted to set a new CSS style on an attribute that doesn't yet have a style attribute applied.
    - A better way to apply a style is to use the style property, since that won't overwrite any CSS styles that you might already have applied to the element.

```
const title = document.querySelector('h1');

//title.setAttribute('style', 'margin: 50px');

console.log(title.style);
console.log(title.style.color);

title.style.margin = '50px';
```

```
<body>
  <h1 style="color: orange; margin: 50px;">The DOM</h1>
  <script src="sandbox.js"></script>
  <!-- Code injected by live-server -->
▶ <script type="text/javascript">…</script>
</body>
```

- If we want to apply new classes to an element, we can use the classList property and the add(...) method.
    - We can also remove classes from an element by using the remove(...) method.
    - Toggle(...) is another classList method that can be useful, since it allows us to toggle whether a class is active on an element or not.

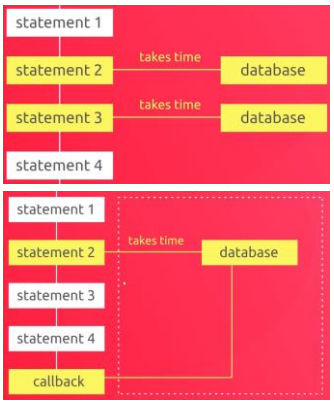
```
const content = document.querySelector('p');

console.log(content.classList);
content.classList.add('error');
```

```
<h1>The DOM</h1>
<p class="error">
```

Asynchronous JavaScript Tutorial #1 – What is Async JavaScript?

- Asynchronous JavaScript allows us, at its most basic form, to start something now and finish it later.
    - It governs how we perform tasks which take some time to complete (e.g. getting data from a database).

- JavaScript by its nature is a synchronous language, which means that it runs one statement at a time, from top to bottom.
    - It therefore runs these statements synchronously in a single thread.
    - If one of the statements takes time to complete, that statement blocks the execution of the code since it must first complete before execution continues.

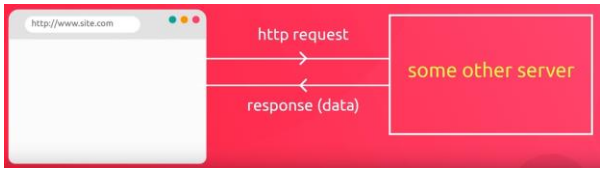


- To prevent this blocking from occurring and stopping the execution of our application, we can use an asynchronous function that allow the execution to continue while the statement finishes up on another thread.
    - What we typically do then is pass a statement a callback function as an argument, which will run later once the data comes back.

```
console.log(1);
console.log(2);

setTimeout(() => {
  console.log('callback function fired');
}, 2000);

console.log(3);
console.log(4);
```

```
1
2
3
4
callback function fired
```

Asynchronous JavaScript Tutorial #2 – HTTP Requests



- We make HTTP requests to get data from another server through API endpoints.
    - Recall that API endpoints is a combination of the host path and the HTTP method (GET, POST, PUT, etc).
- Once we make the request to the API endpoint, we typically get back a response in a JSON format, which looks very similar to a JavaScript object format.
- We'll look at the fetch(...) function separately later, but for now we are going to make HTTP requests using the XMLHttpRequest() object.
    - We supply the request.open(...) method with the HTTP method and the endpoint, and can add an event listener with an asynchronous callback function to output the response once it has been received.

```
const request = new XMLHttpRequest();

request.addEventListener('readystatechange', () => {
  //console.log(request, request.readyState);
  if(request.readyState === 4){
    console.log(request.responseText);
  }
});

request.open('GET', 'https://jsonplaceholder.typicode.com/todos/');
request.send();
```

```
[
  {
    "userId": 1,
    "id": 1,
    "title": "delectus aut autem",
    "completed": false
  },
  {
    "userId": 1,
    "id": 2,
    "title": "quis ut nam facilis et officia qui",
    "completed": false
  },
```

Asynchronous JavaScript Tutorial #3 – Status Codes

- Responses from API endpoints should be accompanied by status codes to indicate whether the response was successful or whether there was an error that was encountered with the request.
    - In our previous example in Tutorial #2, we were not checking for error statuses, which meant that we would receive a blank response without properly handling the response if our endpoint path was incorrect, for instance.

```
const request = new XMLHttpRequest();

request.addEventListener('readystatechange', () => {
  //console.log(request, request.readyState);
  if(request.readyState === 4 && request.status === 200){
    console.log(request, request.responseText);
  } else if(request.readyState === 4){
    console.log('could not fetch the data');
  }
});

request.open('GET', 'https://jsonplaceholder.typicode.com/todoss/');
request.send();
```

```
GET https://jsonplaceholder
oss/ 404
could not fetch the data
```

Asynchronous JavaScript Tutorial #4 – Callback Functions

- From the previous example, it makes some sense for us make this call to the Todo List API reusable.
    - We can do this by assigning the code to a variable as an arrow function.
    - To make the code even more reusable, we can add a callback function to the function arguments that allows us to handle the console logs more effectively.
    - Since the addEventListener(...) code is asynchronous, it will not block our code and execution will continue while the HTTP request completes.
- We should note that not all callback functions are asynchronous, and we would need to check the documentation to know.

```
const getTodos = (callback) => {
  const request = new XMLHttpRequest();

  request.addEventListener('readystatechange', () => {
    if(request.readyState === 4 && request.status === 200){
      callback(undefined, request.responseText);
    } else if(request.readyState === 4){
      callback('coud not fetch data', undefined);
    }
  });

  request.open('GET', 'https://jsonplaceholder.typicode.com/todos/');
  request.send();
};

getTodos((err, data) => {
  console.log('callback fired');
  if(err){
    console.log(err);
  } else {
    console.log(data);
  }
});
```

```
GET https://jsonplaceholder
oss/ 404
could not fetch the data
```

Asynchronous JavaScript Tutorial #5 – Using JSON Data

- JSON data is simply a string formatted in a way that resembles JavaScript objects.
    - It contains data in key-value pairs, and if multiple items are returned, they are formatted as an array.
    - Once we receive back our data, we need to convert the data into a JavaScript object so that we can work on it.

```
[
  { "text": "play mariokart", "author": "shaun" },
  { "text": "by some bread", "author": "Mario" },
  { "text": "do the plumming", "author": "Luigi" }
]
```

- The JSON.parse(...) function will take in a JSON return and will convert it into JavaScript object(s) that can be easily used in the code.
- In its simplest form, JSON is simply a standardized way of transferring data between servers and clients.

```
const getTodos = (resource, callback) => {
  const request = new XMLHttpRequest();

  request.addEventListener('readystatechange', () => {
    if(request.readyState === 4 && request.status === 200){
      const data = JSON.parse(request.responseText);
      callback(undefined, data);
    } else if(request.readyState === 4){
      callback('coud not fetch data', undefined);
    }
  });

  request.open('GET', resource);
  request.send();
};

getTodos('todos/luigi.json', (err, data) => {
  console.log(data);
  getTodos('todos/mario.json', (err, data) => {
    console.log(data);
    getTodos('todos/shaun.json', (err, data) => {
      console.log(data);
    })
  })
```

```
▼ (3) [{…}, {…}, {…}]
  ▶ 0: {text: "do the plumming", author: "Luigi"}
  ▶ 1: {text: "avoid mario", author: "Luigi"}
  ▶ 2: {text: "go kart racing", author: "Luigi"}
    length: 3
  ▶ __proto__: Array(0)
▼ (3) [{…}, {…}, {…}]
  ▶ 0: {text: "make fun of luigi", author: "Mario"}
  ▶ 1: {text: "rescue peach (again)", author: "Mario"}
  ▶ 2: {text: "go kart racing", author: "Mario"}
    length: 3
  ▶ __proto__: Array(0)
▼ (3) [{…}, {…}, {…}]
  ▶ 0: {text: "play mariokart", author: "Shaun"}
  ▶ 1: {text: "buy some bread", author: "Shaun"}
  ▶ 2: {text: "take a nap", author: "Shaun"}
    length: 3
  ▶ __proto__: Array(0)
```

Asynchronous JavaScript Tutorial #6 – Callback Hell

- Sometimes we may find ourselves needing to get JSON data from one API, then reach out to another, and then another after that.
    - When we are chaining together callbacks, we end up in a place called callback hell.
- When we are in callback hell, the logic gets very complicated and the maintainability of the app is lost.

Asynchronous JavaScript Tutorial #7 – Promises

- Thankfully, promises allowing this chaining to occur without creating a mess of the logic.
- When we use promises, the first thing that we do is return a new Promise().
    - A promise is something that is takes some time to do, and ultimately leads to either the promise being resolved and us getting the data we are looking for or the promise will be rejected, and we will get an error.
- A promise takes a function as a parameter, which could be a network request, for instance.
    - Promises automatically have two parameters included – a resolve and a reject parameter.
    - Both the resolve and reject parameters are functions that are built into the Promise API in JavaScript.

```
const getSomething = () => {

  return new Promise((resolve, reject) => {
    // fetch something
    //resolve('some data');
    reject('some error');
  });

};

getSomething().then((data) => {
  console.log(data);
}, (err) => {
  console.log(err);
});
```

- If we had success with our Promise(...) function, we would call the resolve(...) function inside and pass the data into it.
    - If there was an error that was encountered, we would then call the reject(...) function and pass the error data or message.
- When a function contains a promise, we can add a .then(...) method that will fire only once the promise has been successfully resolved.
    - The then(...) method could then take a callback function as an argument that could then run upon the promise completing, and would receive the data that was passed into the resolve(...) function.
    - A second argument inside the then(...) method can be added which will run if the promise is rejected, and we would receive the error that was passed into the reject(...) function.
- As we can see in the first screenshot, adding two callback functions as arguments to the .then(...) method can get a bit messy.
    - Instead of having two arguments, we can just have one .then(...) callback function argument to handle a promise being resolved and can chain on a .catch(...) method to handle the promise being rejected.

```
getSomething().then(data => {
  console.log(data);
}).catch(err => {
  console.log(err);
});
```

```
const getTodos = (resource) => {

  return new Promise((resolve, reject) => {
    const request = new XMLHttpRequest();

    request.addEventListener('readystatechange', () => {
      if(request.readyState === 4 && request.status === 200){
        const data = JSON.parse(request.responseText);
        resolve(data);
      } else if(request.readyState === 4){
        reject('error getting resource');
      }
    });

    request.open('GET', resource);
    request.send();
  });

};

getTodos('todos/luigis.json').then(data => {
  console.log('promise resolved:', data);
}).catch(err => {
  console.log('promise rejected:', err);
});
```

```
promise resolved:
▼ (3) [{…}, {…}, {…}]
  ▶ 0: {text: "do the plumming", author: "Luigi"}
  ▶ 1: {text: "avoid mario", author: "Luigi"}
  ▶ 2: {text: "go kart racing", author: "Luigi"}
    length: 3
  ▶ __proto__: Array(0)
```

> Or is there some way to make one's own function truly asynchronous without leveraging very specific functions of the environment...

Until recently, no. Up through the 5th edition specification, JavaScript *the language* was basically silent on the entire concept of threads and asynchronicity; it was only when you got into environments that it came up. The only way to make something asynchronous was to use a host-provided function, such as `nextTick` (or any of the various operations that completes asynchronously) on NodeJS or `setTimeout` on browsers.

In the ECMAScript 6th edition specification in June 2015, they introduced *promises* into the language. The callbacks hooked up to an ES6 promise via `then` and such are *always* invoked asynchronously (even if the promise is already settled when the callback is attached), and so JavaScript has asynchronicity at a language level now. So if you implement your function so that it returns a promise rather than accepting a callback, you'll know that the `then` callbacks hooked up to it will be triggered asynchronously.

Asynchronous JavaScript Tutorial #8 – Chaining Promises

- One of the most powerful features of promises is that we can chain them together so that we can call one asynchronous task after another if we want to.
    - This gets us out of the callback hell that we previously found ourselves in.
- By returning another promise inside of the first .then(...) function, we can chain another .then(...) function onto it.
    - Only a single .catch(...) function is needed in this promise chain, which will handle errors from any of the .then(...) functions.

```
getTodos('todos/luigi.json').then(data => {
  console.log('promise 1 resolved:', data);
  return getTodos('todos/mario.json');
}).then(data => {
  console.log('promise 2 resolved:', data);
  return getTodos('todos/shaun.json');
}).then(data => {
  console.log('promise 3 resolved:', data);
}).catch(err => {
  console.log('promise rejected:', err);
});
```

```
promise 1 resolved: ▶ (3) [{…}, {…}, {…}]
promise 2 resolved: ▶ (3) [{…}, {…}, {…}]
promise 3 resolved: ▶ (3) [{…}, {…}, {…}]
```

Asynchronous JavaScript Tutorial #9 – The Fetch API

- The native Fetch API provides an easier way to make requests, rather than having to use the XMLHttpRequest() object.
    - There is less code to write, and it is easier to maintain when using this API.
    - This API implements the Promise API under the hood, making it straight forward to handle success and error cases.
- Inside of the fetch(...) function, we pass in an argument that is the resource that we want to fetch.
    - It could be an API endpoint, or could be a local database request or resource, for instance.

- Since the fetch(...) implements the Promise API, we can add .then(...) and .catch(...) methods onto the end of the function to handle asynchronous success or error, firing callback functions inside of the methods upon resolution.
- With the Fetch API, the promise is only ever rejected if we receive a network error, meaning that if we mistype a URL provided into the fetch(...) function, it is still resolved.
    - However, in our resolved response, we will get a status code of 404, for example, indicating that the network resource could not be reached.
    - Typically, we will handle the network error cases such as a status code of 404 inside of the .then(...) method, along with the success status code of 200, letting the .catch(...) method handle the other network errors.
- The Fetch API creates a response object for us when we get data returned to us.
    - To get the actual JSON data from the response, we have to use the response.json() method which will parse the JSON data to be used inside of our code.

```
fetch('todos/luigi.json').then((response) => {
  console.log('resolved', response);
  return response.json();
}).then(data => {
  console.log(data);
}).catch((err) => {
  console.log('rejected', err);
});
```

```
▼(3) [{…}, {…}, {…}]
  ▶0: {text: "do the plumming", author: "Luigi"}
  ▶1: {text: "avoid mario", author: "Luigi"}
  ▶2: {text: "go kart racing", author: "Luigi"}
    length: 3
  ▶__proto__: Array(0)
```

    - This response.json() method will return a promise, which means that we can return that promise inside of the first .then(...) method and chain on another .then(...) method to handle that asynchronous response.

Asynchronous JavaScript Tutorial #10 – Async and Await

- Async and await were two keywords recently introduced into JavaScript (ES8 in 2017).
    - At their most basic, these keywords allow us to chain promises together in a cleaner and more readable way.
    - While promise chains look a lot better than using large callback functions as we did with XMLHttpRequest(), they still get messy when we are chaining a lot of different promises together.
- Using async and await allows us to section off all our asynchronous code into a single async function, and then use the await keyword inside to chain promises together in a much more readable and logical way.
    - When we call an async function, we know that it will return a promise.
- We first start by creating a new function to contain our asynchronous code and create an arrow function with the async keyword in front, turning the function into an asynchronous function.

```
const getTodos = async () => {

  const response = await fetch('todos/luigi.json');
  const data = await response.json();

  return data;

};

console.log(1);
console.log(2);

getTodos()
  .then(data => console.log('resolved:', data));

console.log(3);
console.log(4);
```

```
1
2
3
4
resolved: ▶(3) [{…}, {…}, {…}]
```

    - Inside of the async function, we will now want to handle all our asynchronous code to go out and grab data.
- Instead of having to attach .then(...) methods and chain them to the fetch(...) function, we can use the await keyword to block execution of the code at that point, but only inside of the async function itself and not outside of the function.
    - Once the promise has resolved, we can then use the response object, which itself implements a promise, and await the promise from that.
    - Since the async function itself returns a promise, we have to add only a single .then(...) method to the function call itself to handle the response promise from the function.
- A summary of async and await is as follows:
    - Async and await has bundled up all our asynchronous code inside of a function, which we can call and use whenever we want.
    - The power of the await keyword, which is placed only before a promise, is that we can chain together many different calls that return promises and execute them sequentially in a clean, logical way.
    - The asynchronous function that we create will not block the rest of the code in our application.
- While these new keywords are not supported in all browsers (e.g. Internet Explorer), they are supported in all modern browsers.

Asynchronous JavaScript Tutorial #11 – Throwing Errors

- If we want to reject the promise that an async function returns, we can throw our own custom error.
- Recall from the previous section about the Fetch API that if there is an issue with the JSON, such that it is not in a valid format, for example, the promise will be rejected and can be caught outside of the async function.
    - Also recall that if the resource URL provided into the fetch(...) function is not valid, that promise will still be resolved rather than rejected, which will lead to the promise being rejected in the response.json() line.

```
const getTodos = async () => {

  const response = await fetch('todos/luigis.json');

  if(response.status !== 200){
    throw new Error('cannot fetch the data');
  }

  const data = await response.json();

  return data;

};

getTodos()
  .then(data => console.log('resolved:', data))
  .catch(err => console.log('rejected:', err.message));
```

```
▸GET http://127.0.0.1:5500/chapter_12/todos/luigis.json 404 (Not Found)
rejected: cannot fetch the data
```

- This is does not actually address the problem at its source, so we need to manually check if the network response object has a status code of 200 and throw a new custom error object if not.
    - If the custom error object is thrown, the promise returned by the async function is rejected.
    - This allows us to catch the error outside of the async function, where we can handle it with a callback function.
- Note that a custom error message can be provided inside of the error object.