Janusz Starczewski

NODE.js

Node.js is a powerful, cross-platform, open-source runtime environment that allows developers to build scalable and high-performance web applications using JavaScript. Developed by Ryan Dahl in 2009, Node.js is built on top of the **V8 JavaScript engine**, which is used by Google Chrome. This allows Node.js to execute JavaScript code outside of a web browser and on a server.

One of the key features of Node.js is its **event-driven**, **non-blocking** I/O model, which allows it to handle large volumes of data and simultaneous connections without blocking the event loop. This makes Node.js particularly well-suited for building real-time applications, such as chat applications, multiplayer games, and streaming services.

Getting Started with Node.js

To get started with Node.js, you'll need to install it on your computer. Node.js is available for Windows, macOS, and Linux, and you can download the installer from the official Node.js website. Once you've installed Node.js, you can create a new Node.js project by running the following command in your terminal:

```
npm init
```

This will create a new `package.json` file in your project directory, which will allow you to manage your project's dependencies and scripts.

Working with Modules in Node.js

One of the strengths of Node.js is its modular architecture, which allows you to easily import and use external libraries and packages in your code. Node.js supports two types of modules: built-in modules and external modules.

Built-in modules are part of the Node.js core and provide basic functionality, such as file system operations, network communication, and HTTP requests. You can import a built-in module into your code using the `require()` function:

```
const fs = require('fs');
```

External modules, on the other hand, are third-party packages that you can install and use in your code. You can install an external module using the `npm install` command:

```
npm install <package-name>
```

Once you've installed a package, you can import it into your code using the `require()` function:

```
const express = require('express');
```

Building a Web Server with Node.js

One of the most common use cases for Node.js is building web servers. Node.js provides a built-in `http` module that allows you to create a web server and handle HTTP requests and responses.

Here's a basic example of a Node.js web server:

```
const http = require('http');

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello, world!');
});

server.listen(3000, () => {
  console.log('Server running on port 3000');
});
```

In this example, we're creating a new HTTP server using the `http.createServer()` method. We're then setting the response status code and content type using the `res.statusCode` and `res.setHeader()` methods, and sending the response body using the `res.end()` method.

Finally, we're starting the server by calling the `server.listen()` method and passing in the port number to listen on. When the server starts, we're logging a message to the console to indicate that it's running.

EventEmitters

In Node.js, the EventEmitter class provides a way to handle and propagate events within a Node.js application. An event is a signal that indicates that something has happened in the application, such as a user clicking a button or a file being saved. By using the EventEmitter class, you can create custom events and handle them in your application.

The EventEmitter class is part of the Node.js core, so you can use it in your application without installing any external packages.

Sending Events with EventEmitters

To send an event with an EventEmitter, you first need to create an instance of the EventEmitter class:

```
const EventEmitter = require('events');
const myEmitter = new EventEmitter();
```

Once you have an instance of the EventEmitter class, you can use the **`emit()`** method to send an event:

```
myEmitter.emit('myEvent', arg1, arg2, ...);
```

The first argument to the **`emit()`** method is the name of the event you want to send. You can pass additional arguments to the **`emit()`** method, which will be passed to any event listeners that handle the event.

For example, you could send an event named `buttonClick` with a `clickCount` argument:

```
myEmitter.emit('buttonClick', 3);
```

Receiving Events with EventEmitters

To receive events with an EventEmitter, you need to add a listener function to the EventEmitter instance using the **`on()`** or **`addListener()`** method:

```
myEmitter.on('myEvent', (arg1, arg2, ...) => {
  // handle the event
});
```

The first argument to the **`on()`** method is the name of the event you want to listen for. The second argument is a listener function that will be called whenever the event is emitted. The listener function can take any number of arguments, which will be passed from the **`emit()`** method.

For example, you could listen for the `buttonClick` event and log the `clickCount` argument:

```
myEmitter.on('buttonClick', (clickCount) => {
  console.log(`Button clicked ${clickCount} times`);
});
```

You can also use the `once()` method to listen for an event only once, and then remove the listener automatically:

```
myEmitter.once('myEvent', (arg1, arg2, ...) => {
  // handle the event
});
```

Example of an HTTP server application that uses EventEmitters

```
const http = require('http');
const EventEmitter = require('events');
const myEmitter = new EventEmitter();

const server = http.createServer((req, res) => {
  if (req.url === '/data') {
    myEmitter.emit('getData', (data) => {
      res.write(`Received data: ${data}`);
      res.end();
    });
  } else {
    res.write('Hello World!');
    res.end();
  }
});
```

```
server.listen(3000, () => {
  console.log('Server is running on port 3000');
});

// Simulate getting data from a database
setTimeout(() => {
  const data = 'some data';
  myEmitter.emit('getData', data);
}, 5000);
```

In this example, we create an HTTP server using the createServer() method from the http module. When a request is made to the /data URL, we emit a getData event using our previously created myEmitter instance. We pass a callback function to the emit() method, which will be called when the event is handled by a listener.

We then listen for the getData event and simulate getting some data from a database using a setTimeout() function. When the data is ready, we emit the getData event with the data as an argument.

Finally, in the callback function that we passed to the emit() method, we write the received data to the response object and end the response.

Note that this is just a simple example, and in a real-world application, you would likely use more robust methods for getting data from a database or other sources. Nonetheless, this example demonstrates how EventEmitters can be used to handle events in a Node.js application, even in the context of an HTTP server.

What is a Module in JavaScript?

In simple terms, a module is a piece of reusable JavaScript code. It could be a `.js` file or a directory containing `.js` files. You can export the content of these files and use them in other files.

Modules help developers adhere to the DRY (Don't Repeat Yourself) principle in programming. They also help to break down complex logic into small, simple, and manageable chunks.

Types of Node Modules

There are two main types of Node modules that you will work with as a Node.js developer. They include the following.

- Built-in modules
- Local modules/Third-party modules

Built-in Modules

Node.js comes with some modules out of the box. These modules are available for use when you install Node.js, eg.

- http
- url
- path
- fs
- os

You can use the built-in modules with the syntax below.

```
const someVariable = require('nameOfModule')
```

You load the module with the `require` function. You need to pass the name of the module you're loading as an argument to the `require` function.

Note: The name of the module must be in quotation marks. Also, using `const` to declare the variable ensures that you do not overwrite the value when calling it. You also need to save the returned value from the `require` function in `someVariable`. You can name that variable anything you want. But often, you will see programmers give the same to the variable as the name of the module (see the next example).

```
const http = require('http')

server = http.createServer((req, res) => {
    res.writeHead(200, {'Content-Type': 'text/plain'})
    res.end('Hello World!')
})

server.listen(3000)
```

You use the `require` function to load the built-in `http` module. Then, you save the returned value in a variable named `http`.

The returned value from the `http` module is an object. Since you've loaded it using the `require` function, you can now use it in your code. For example, call the `.createServer` property to create a server.

Local Modules

When you work with Node.js, you create local modules that you load and use in your program. Let's see how to do that.

Create a simple `sayHello` module. It takes a `userName` as a parameter and prints "hello" and the user's name.

```
function sayHello(userName) {
        console.log(`Hello ${userName}!`)
}

module.exports = sayHello
```

First, you need to create the function. Then you export it using the syntax `module.exports`. It doesn't have to be a function, though. Your module can export an object, array, or any data type.

How to load your local modules

You can load your local modules and use them in other files. To do so, you use the `require` function as you did for the built-in modules.

But with your custom functions, you need to provide the path of the file as an argument. In this case, the path is `'./sayHello'` (which is referencing the `sayHello.js` file).

```
const sayHello = require('./sayHello')
sayHello("Maria") // Hello Maria!
```

Once you've loaded your module, you can make a reference to it in your code.

Third-Party Modules

A cool thing about using modules in Node.js is that you can share them with others. The Node Package Manager (NPM) makes that possible. When you install Node.js, NPM comes along with it.

With NPM, you can share your modules as packages via the NPM registry. And you can also use packages others have shared.

How to use third-party packages

To use a third-party package in your application, you first need to install it. You can run the command below to install a package.

```
npm install <name-of-package>
```

For example, there's a package called `capitalize`. It performs functions like capitalizing the first letter of a word.

Running the command below will install the capitalize package:

```
npm install capitalize
```

To use the installed package, you need to load it with the `require` function.

```
const capitalize = require('capitalize)
```

And then you can use it in your code, like this for example:

```
const capitalize = require('capitalize')
console.log(capitalize("hello")) // Hello
```

This is a simple example. But there are packages that perform more complex tasks and can save you loads of time.

For example, you can use the Express.js package which is a Node.js framework. It makes building apps faster and simple.

Frameworks for Node.js

Open-Source Framework	Description
Express.js	Express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications. This is the most popular framework as of now for Node.js.
Geddy	Geddy is a simple, structured web application framework for Node.js based on MVC architecture.
Locomotive	Locomotive is MVC web application framework for Node.js. It supports MVC patterns, RESTful routes, and convention over configuration, while integrating seamlessly with any database and template engine. Locomotive builds on Express, preserving the power and simplicity you've come to expect from Node.
Koa	Koa is a new web framework designed by the team behind Express, which aims to be a smaller, more expressive, and more robust foundation for web applications and APIs.
Total.js	Totaljs is free web application framework for building web sites and web applications using JavaScript, HTML and CSS on Node.js
Hapi.js	Hapi is a rich Node.js framework for building applications and services.
Keystone	Keystone is the open source framework for developing database-driven websites, applications and

Open-Source Framework	Description
	APIs in Node.js. Built on Express and MongoDB.
Derbyjs	Derby support single-page apps that have a full MVC structure, including a model provided by Racer, a template and styles based view, and controller code with application logic and routes.
Sails.js	Sails makes it easy to build custom, enterprise-grade Node.js apps. It is designed to emulate the familiar MVC pattern of frameworks like Ruby on Rails, but with support for the requirements of modern apps: data-driven APIs with a scalable, service-oriented architecture. It's especially good for building chat, realtime dashboards, or multiplayer games; but you can use it for any web application project - top to bottom.
Meteor	Meteor is a complete open source platform for building web and mobile apps in pure JavaScript.
Mojito	This HTML5 framework for the browser and server from Yahoo offers direct MVC access to the server database through the local routines. One clever feature allows the code to migrate. If the client can't run JavaScript for some reason, Mojito will run it on the server -- a convenient way to handle very thin clients.
Restify	Restify is a node.js module built specifically to enable you to build correct REST web services.
Loopback	Loopback is an open-source Node.js API framework.
ActionHero	actionhero.js is a multi-transport Node.JS API Server with integrated cluster capabilities and delayed

Open-Source Framework	Description
	tasks.
Frisby	Frisby is a REST API testing framework built on node.js and Jasmine that makes testing API endpoints easy, fast, and fun.
Chocolate.js	Chocolate is a simple webapp framework built on Node.js using Coffeescript.

Express.js

Express.js is a popular web application framework for Node.js that provides a set of features and tools for building robust and scalable web applications. It is designed to make it easy to create RESTful APIs, web applications, and other types of HTTP-based services.

One of the key benefits of using Express.js is that it provides a simple and intuitive API that makes it easy to build web applications quickly and efficiently. Express.js also has a large and active community of developers who contribute to its development, maintain numerous packages, and provide support through forums and other channels.

Basic Concepts of Express.js

Routing: Express.js provides a routing API that makes it easy to define routes for handling HTTP requests. Routes are defined using HTTP methods such as GET, POST, PUT, and DELETE, and can match patterns in the URL path and/or query parameters.

Middleware: Middleware functions in Express.js are functions that are executed in a sequence for each incoming request. They can modify the request and response objects, and can also be used to handle errors, authenticate users, and perform other types of common web application tasks.

Templates: Express.js provides support for rendering HTML templates using popular templating engines such as EJS, Handlebars, and Jade. This makes it easy to create dynamic web pages that can be customized based on user input, database queries, and other types of data.

Static Files: Express.js makes it easy to serve static files such as images, CSS files, and client-side JavaScript files. This can be done using the built-in `express.static()` middleware, which maps static file requests to a specified directory on the file system.

Getting Started with Express.js

To get started with Express.js, you first need to install it using npm:

```
npm install express
```

Once you have installed Express.js, you can create a new web application by creating a new JavaScript file and importing the Express.js module:

```
const express = require('express');
const app = express();
```

You can then define routes and middleware functions using the `app` object:

```
// Define a route for handling GET requests to the root path
app.get('/', (req, res) => {
  res.send('Hello World!');
});

// Define a middleware function to log requests to the console
app.use((req, res, next) => {
  console.log(`${req.method} ${req.url}`);
  next();
});
```

In this example, we define a route for handling GET requests to the root path (`/`) and a middleware function that logs requests to the console. The `req` and `res` parameters in the route function correspond to the request and response objects, respectively.

Single-Page Application

To create a Single-Page Application (SPA) with Express.js, you'll need to set up your server to handle the routing and serve the initial HTML file, while the client-side framework takes over and handles subsequent navigation and rendering. Here's a general approach to creating an SPA with Express.js:

1. Set up your project:
   - Create a new directory for your project.
   - Initialize a new Node.js project using `npm init`.
   - Install Express.js and any other dependencies you may need.
2. Create the Express.js server:
   - Create a new JavaScript file (e.g., `server.js`) to define your server.
   - Import Express.js and create an instance of the Express application:

```
const express = require('express');
const app = express();
```

- Set up static file serving to serve your client-side assets (HTML, CSS, JavaScript):

```
app.use(express.static('public'));
```

Replace `'public'` with the appropriate directory where your client-side files reside.

3.Define the catch-all route:

- Create a catch-all route that serves your SPA's main HTML file:

```
app.get('*', (req, res) => {
  res.sendFile(path.join(__dirname, 'public', 'index.html'));
});
```

Adjust the `path.join()` parameters to reflect the correct file path to your HTML file.

4. Handle API routes (optional):

- If your SPA communicates with a backend API, define additional routes to handle API requests. These routes can handle CRUD operations, authentication, and other API-related tasks.

5. Set up your client-side framework:

- Choose a client-side framework/library, such as React, Angular, or Vue.js.
- Follow the documentation and best practices of your chosen framework to build your SPA's components, routing, and state management.

6. Build and bundle your client-side assets:

- Use the build tools provided by your chosen client-side framework to compile and bundle your JavaScript, CSS, and other assets into a production-ready format.

7. Start the Express.js server:

- Add a script to your `package.json` file to start the server:

```
"scripts": {
  "start": "node server.js"
}
```

- Run `npm start` in the terminal to start the Express.js server.

8. Test your SPA:

- Open your browser and navigate to `http://localhost:3000` (or the appropriate port, as configured in your Express.js server).
- Verify that your SPA loads successfully and that client-side routing and rendering are working as expected.

By following these steps, you'll have an Express.js server set up to serve your SPA's initial HTML file, while the client-side framework takes over to handle subsequent navigation and rendering. Remember to adjust the file paths and configurations based on the specifics of your project.

REST Representational State Transfer

What is this REST for?

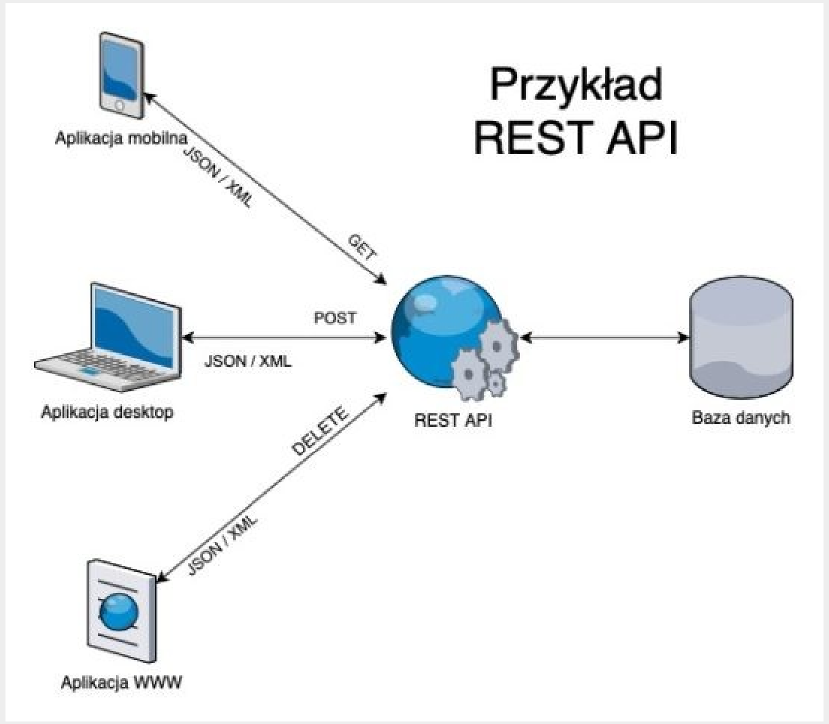
There are a huge number of services on the Internet that exchange data with each other. Let's assume that each of them has its own rules for exchanging and accessing information. Without established rules, the integration of such systems would be very difficult to implement and maintain. That is why solutions have been created that create certain standards.
REST has gained immense popularity, among others thanks to its simplicity, speed and universality, and its implementation is possible in many programming languages. Thanks to this, systems exchanging information with each other do not have to be implemented in the same technology, because all the magic is hidden in the data format! REST has many libraries both on the frontend and backend, which significantly speed up the work on applications.
It is worth remembering that REST is not a synonym for HTTP! HTTP is a communication protocol, and REST is an architectural solution that uses this protocol.

What are the general rules in RESTful?

In order for REST applications to be consistent and the implementation of communication between them easier, rules have been adopted, the fulfillment of which makes the application RESTful. It is not a strict standard, but only a set of rules.

1. Uniform Communication Interface

The server should provide an API that will be understood by all applications communicating with it. The data returned should have the same format and data range, and a well-designed interface should meet the needs of multiple devices and applications.



An example of communication of many devices with REST API

2. Division into client-server applications (client-server)

The separation of applications allows for their independent development and operation. Such a division definitely increases the possibilities of scaling, portability and extensibility, and error handling is much easier. This approach definitely facilitates the implementation of point number 1. Separating the application into client-server layers will facilitate integration with other client services in the future.

It will be much more efficient to install a web application based on HTML with JS and which connects to an independent API than configuring the entire application, which is a monolith built of frontend and backend.

3. Stateless

On the server side, there should be no mechanisms holding client data that would be needed for the proper operation of the system. The API should not keep the state of the application, but only interpret the information sent by the client and function on this basis. The client should always send a complete set of information needed for the correct execution of the request.

This rule is strongly related to the user's session, which is often implemented on the server side. It is thanks to it that it is verified whether the user has access to the system and what permissions he has. In the case of RESTful API, each request that goes to the server should contain a special token in its headers, e.g. JWT. It is on its basis that the API can decide whether the request should be processed or rejected.

4. Data cache (Cacheability)

Due to the fact that the API can handle very high traffic, "tricks" are used to reduce the network and hardware load. One of the ways is to use the mechanism of browsers that locally "remember" server responses, which are based on the GET and POST methods. For this purpose, special HTTP headers are used, which define whether the response is to be remembered, and if so, when it should be refreshed or deleted.

Example of HTTP headers in the server response

- Expires – data expiration date. When the deadline expires, the resource must be fetched from the server and can be remembered again.
- Cache-Control – defines whether the resource should be saved and for how long.
- ETag – a unique identifier / token, directly related to the saved resource. It changes when the value is updated.
- Last-Modified – specifies when the resource was updated / modified.

Another solution that can be used is Redis or Memcached, which allow you to cache entire server responses for specific requests.

The request comes to the cache of the server, which checks if it has data for this request. If so, it fetches them and returns them to the client. In such a scenario, downloading resources from the API is omitted, which automatically offloads it in many aspects.

5. Separated layers (Layered system)

Communication and data exchange between client applications and the API should not be burdened with information about external websites and services used by the server. This allows you to maintain transparency and separation of logic from the web layer.

6. Sending the code to the client's application (Code on demand)

The API can send ready-made code fragments to client applications for processing and launching them (e.g. JS scripts). This rule is the only one of the six that is optional.

RESTful Application Programming Interface

To create a RESTful API with Express.js, include these steps:

1. Set up your project

2. Create the Express.js server:

- Create a new JavaScript file (e.g., `server.js`) to define your server.
- Import Express.js and create an instance of the Express application:

```
const express = require('express');
const app = express();
```

- Set up middleware to parse incoming request bodies as JSON:

```
app.use(express.json());
```

3.Define your API routes:

- Define routes to handle different HTTP methods (GET, POST, PUT, DELETE) and corresponding endpoints.
- For example, to handle a GET request at `/api/users`:

```
app.get('/api/users', (req, res) => {
  // Handle the request and send the response
  res.json({ message: 'Get all users' });
});
```

- Add additional routes to handle other API operations (e.g., POST, PUT, DELETE).

4.Test your API:

- Start the Express.js server by adding a script to your `package.json` file:

```
"scripts": {
  "start": "node server.js"
}
```

- Run `npm start` in the terminal to start the Express.js server.
- Use tools like Postman or curl to send HTTP requests to your API endpoints and verify the responses.

5.Connect to a database (optional):

- If your RESTful API requires data storage, you can connect to a database (e.g., MongoDB, MySQL) using appropriate database drivers or ORMs.
- Configure database connection parameters and set up the necessary queries or models to handle database operations.

6.Implement CRUD operations:

- Use the appropriate HTTP methods and corresponding routes to handle Create, Read, Update, and Delete (CRUD) operations on your API resources.
- For example, to handle creating a new user:

```
app.post('/api/users', (req, res) => {
  // Access the request body and create a new user in the database
  const newUser = req.body;
  // Perform database operation and handle success/error cases
  res.json({ message: 'User created successfully' });
});
```

- Implement similar routes for other CRUD operations.

7.Handle error cases:

- Add error-handling middleware to catch and handle any errors that occur during API requests and database operations.
- Return appropriate error responses (status codes and error messages) to the client.

8.Enhance API security (optional):

- Implement authentication and authorization mechanisms to secure your API endpoints.

- Use middleware functions to validate user credentials, protect sensitive routes, and enforce access controls.

9. Deploy your API (optional):

- Choose a hosting platform to deploy your Express.js API to a production environment.
- Configure any necessary environment variables and deployment settings.
- Deploy your API and verify that it works in the production environment.

By following these steps, you can create a RESTful API using Express.js. Customize the routes, database connections, and error handling to match the specific requirements of your API.

RESTful CRUD

To implement all CRUD (Create, Read, Update, Delete) operations in a RESTful API with Express.js, follow these steps:

1. Set up your project and create the Express.js server as described in the previous instructions.

2. Define your API routes for handling CRUD operations. Below is an example of how to implement all CRUD operations for a resource like "users":

```
// Get all users
app.get('/api/users', (req, res) => {
  // Retrieve all users from the database
  // Send the retrieved users as a response
  res.json({ message: 'Get all users' });
});
```

```
// Get a specific user by ID
app.get('/api/users/:id', (req, res) => {
  const userId = req.params.id;
  // Retrieve the user with the given ID from the database
  // Send the retrieved user as a response
  res.json({ message: `Get user with ID: ${userId}` });
});

// Create a new user
app.post('/api/users', (req, res) => {
  const newUser = req.body;
  // Create a new user in the database using the received data
  // Send a success message or the created user as a response
  res.json({ message: 'User created successfully' });
```

```
});

// Update a user by ID
app.put('/api/users/:id', (req, res) => {
  const userId = req.params.id;
  const updatedUser = req.body;
  // Update the user with the given ID in the database using the received
data
  // Send a success message or the updated user as a response
  res.json({ message: `User with ID ${userId} updated successfully` });
});

// Delete a user by ID
app.delete('/api/users/:id', (req, res) => {
  const userId = req.params.id;
```

```
    // Delete the user with the given ID from the database
    // Send a success message or the deleted user as a response
    res.json({ message: `User with ID ${userId} deleted successfully` });
  });
```

Note: The code snippets above represent the structure of the routes and their corresponding handlers. You'll need to implement the actual logic to interact with the database or any other data source you're using, eg.

```
  app.get('/api/users', async (req, res) => {try {
    const users = await User.find();
    res.json(users);
  } catch (error) {
    console.error('Error fetching users:', error);
    res.status(500).json({ message: 'Error fetching users' });
  }
});
```

3. Connect to a database:

- Set up a database connection using a database driver or an ORM (e.g., MongoDB with Mongoose, MySQL with Sequelize).
- Configure the database connection parameters and establish the connection.

4.Implement the database operations for CRUD functionality:

- Inside the route handlers, use the appropriate database queries or ORM methods to perform the CRUD operations on the respective resource (e.g., "users").
- Handle success and error cases accordingly, returning appropriate responses to the client.

5.Add error handling middleware:

- Implement error-handling middleware to catch and handle any errors that occur during API requests or database operations.
- Return appropriate error responses (status codes and error messages) to the client.

6.Test your API:

- Start the Express.js server and use tools like Postman or curl to send HTTP requests to the defined API endpoints.
- Verify that the API routes for all CRUD operations work as expected and return the appropriate responses.

By following these steps, you can implement all CRUD operations in your RESTful API using Express.js. Customize the routes, database operations, and error handling to fit the specific requirements of your API and the resources you are working with.

Conclusions

Node.js is a powerful and flexible runtime environment for building web applications using JavaScript. With its event-driven, non-blocking I/O model and modular architecture, Node.js is particularly well-suited for building real-time applications and web servers. By learning Node.js, you can expand your skills as a web developer and build scalable and high-performance applications for the modern web.

EventEmitters provide a powerful way to handle and propagate events within a Node.js application. By creating custom events and adding event listeners, you can build applications that respond to user actions, network events, and other types of signals. By mastering the EventEmitter class, you can take your Node.js applications to the next level and build robust and responsive software.

You learned about what Node modules are and the three types of node modules. You also learned about how to use the different types in your application.

Express.js is a flexible web application framework for Node.js that provides a rich set of features and tools for building robust and scalable web applications. By mastering the basic concepts of routing, middleware, templates, and static files, you can create web applications quickly and efficiently using Express.js. Additionally, the active community of developers and the vast ecosystem of packages and tools make it easy to extend and customize Express.js to meet your specific needs.

Asysync Code

So what are all these things about? Handling asynchronous code! But what is asynchronous (async) code to begin with? Consider this example.

```
const getUser = () => {
  setTimeout(() => {
    return { name: 'FirstStudent' }
  }, 2000)
}

const user = getUser() // This doesn't actually fetch the user
console.log(user.name) // This won't work
```

This code won't work!

We have a (ES6 arrow) function which executes `setTimeout()` and then tries to return an object after 2 seconds. That doesn't work because the `getUser()` function gets executed and its result gets used immediately. The immediate return value of the function is `void` though. Returning something after 2 seconds won't work. The same would be the case for functions where you reach out to a web server, access the file system or do something else which isn't finished immediately.

The reason why it doesn't work is that JavaScript code runs in a non-blocking way. That means, that it won't wait for async code to execute and return a value - it will only execute line for line until it's done. We can of course still work with asynchronous operations, it just doesn't work as attempted in the above code snippet.

Callbacks to the Rescue!

Since the above code doesn't work like that, we need to change something. Callbacks are the oldest way of working with data **once it's there** and continuing execution of the other code whilst the async code is executing.

Here's the revised code.

```
const getUser = callbck => {
  setTimeout(() => {
    callbck({ name: 'FirstStudent' })
  }, 2000)
}
```

```
getUser(user => {
  console.log(user.name) // Prints 'FirstStudent' after 2 seconds
})
```

What changed? We're now not returning anything in `getUser()` - it wouldn't have worked anyways!

Instead, we now pass a **callback function** to `getUser()`. Inside `getUser()`, this callback function is received as a normal function argument and executed once the timer completes. The cool thing is, that we can now also pass an argument to the callback function.

```
const getUser = callbck => {
  setTimeout(() => {
    callbck({ name: 'FirstStudent' }) // <- Passing an argument to the
callback function
  }, 2000)
}
```

Why is this helpful? It allows us to pass data to code which is defined somewhere else but which will only run once our timer completes! In the above example, it's an anonymous function but of course you could also use a named function.

```
// Use an anonymous function
getUser(user => {
  console.log(user.name) // Prints 'FirstStudent' after 2 seconds
})
// Alternatively, use a named function:
getUser(handleUser) // <- This also works, just make sure to NOT execute the
function here (handleUser() would be wrong!)
const handleUser = user => {
  console.log(user.name) // Prints 'FirstStudent' after 2 seconds
}
```

Notice that code execution doesn't stop whilst waiting for the timer.

```
const getUser = callbck => {
  setTimeout(() => {
    callbck({ name: 'FirstStudent' })
  }, 2000)
}
getUser(user => {
  console.log(user.name) // Prints 'FirstStudent' after 2 seconds
})
console.log('This prints before "FirstStudent" gets printed!') // <- This does
indeed print before 'FirstStudent'
```

So ... callbacks are pretty awesome, right? We can postpone code execution and work with data once we actually got it. And we don't even block the execution of our main program.

#

Welcome to Callback Hell

Unfortunately, it's not that easy. The idea behind callbacks indeed isn't bad and there is a reason why it's still getting used all over the web. But when working with a lot of dependent asynchronous operations, you quickly end up in callback hell. Consider this example.

```
const checkStud = callbck => {
  // In reality, you of course don't have a timer but will probably reach out
to a server
  setTimeout(() => {
    callbck({ isStud: true })
  }, 2000)
}
const getUser = (studInfo, callbck) => {
  if (!studInfo.isStud) {
    callbck(null)
    return
  }
  setTimeout(() => {
    callbck({ name: 'FirstStudent' })
  }, 2000)
}
checkStud(StudInfo => {
  getUser(StudInfo, user => {
    console.log(user.name)
  })
})
```

This part here is pretty bad:

```
checkStud(StudInfo => {
  getUser(StudInfo, user => {
    console.log(user.name)
  })
})
```

It could quickly grow to use multiple callbacks and nesting them into each other won't make your code easier to read, understand or maintain.

This problem is referred to as **callback hell**. It really can be a hell because you'll have a hard time changing one of your many layers of callbacks. Debugging it can be a pain and it certainly isn't readable anymore once you reach three or four levels of nesting.

Also don't forget error handling. It's not trivial to catch errors and handle them correctly when resting in callback hell. Indeed, you'll go even deeper into callback hell once you start handling different outcomes of your various asynchronous functions. Chances are, you want to run other asynchronous code if you encounter an error. Can you imagine where you'll end up with this?

#

Promising a Better Future

Callbacks are okay for single asynchronous operations but they certainly aren't perfect, we know that now. Fortunately, we're not the only ones discovering this issue. ES6 introduces a solution: Promises! Indeed, with the help of various third-party libraries, you could and can use the concept of promises in an ES5 code already. Here's how a promise works in JavaScript.

```
const getUser = () => {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve({ name: 'FirstStudent' })
    }, 2000)
  })
}
getUser().then(user => {
  console.log(user.name)
})
```

The `getUser()` function might look confusing, focus on the bottom part of the code though. We simply call `getUser()` there and then use `then()` to handle the asynchronous value. Inside `getUser()`, we create a new promise. The constructor function of that object receives a function as an argument. That function is executed automatically and can also receive two arguments: `resolve` and `reject`. Both are function you may execute inside the function passed to the promise constructor.

When calling `resolve(data)`, you resolve (= complete) the promise and return `data` to the function executed in the `then` block.

You could also call `reject(err)` to throw an error. I'll come back to error handling later.

It might not immediately look that much better than callbacks but consider that we only use one asynchronous operation here! The real power of promises can be seen once we start using dependent async operations.

```
checkStud()
```

```
  .then(studStatus => {
    return getUser(studStatus) // returns a new promise which may use the
studStatus we fetched
  })
  .then(user => {
    console.log(user.name) // prints the user name
  })
```

This code is much more readable than its callback alternative! And of course this argument gets even stronger once we start chaining more async operations. Inside `then()`, you can simply return the result of a function call. And the result of that function call can be used in the next `then()` block - you'll receive it as an argument there. If the returned result is a promise, JavaScript will wait for its completion and resolve it for you. Awesome!

#

What about errors?

You can handle errors with ease, too! Simply add a `catch()` block to your chain and it will catch any errors thrown by any of your promises.

```
checkStud()
  .then(studStatus => {
    return getUser(studStatus) // returns a new promise which may use the
studStatus we fetched
  })
  .then(user => {
    console.log(user.name) // prints the user name
  })
  .catch(error => {
    // handle error here
  })
```

Promises offer a real improvement over callback functions and they give you a chance of escaping hell which doesn't sound too bad. ES6 also offers some other nice features you can use with promises - you may have a look at Promise.all() or Promise.race() for example.

#

Are we happy?

Promises are awesome - are we happy then? Maybe. Promises are pretty great and see a lot of usage these days. You can really build predictable and manageable pipelines for asynchronous operations with them. They do have one limitation though: You may only handle one asynchronous operation with each promise. That's fine for sending HTTP requests and reacting to responses for example. It's not really a great solution if you want to handle asynchronous operations which don't end after one "value".

What would be an example for an asynchronous operation which might be run multiple times? Think about user events. They're certainly asynchronous as you can't block your code to wait for them to occur. You can't really handle them with promises though. You could handle one event (e.g. click on a button) but thereafter your promise is resolved and can't react to further clicks.

```
const button = document.querySelector('button')
const handleClick = () => {
  return new Promise(resolve => {
    button.addEventListener('click', () => {
      resolve(event)
    })
  })
}
handleClick().then(event => {
  console.log(event.target)
})
```

In this example, we're adding an event listener to a button in our DOM. This happens inside the promise but since a promise can only resolve once, we're only reacting to the first click. Subsequent clicks will go into the void. It's this reason as well as one other important advantage which made Observables very popular. Now what's that again?

#

RxJS Observables

IMPORTANT

#

RxJS 6 was released!

Please note that RxJS 6 was released. Updating is easy and described in detail in this article. For the CDN drop-in import (which we're using in this article), have a look at this example snippet.

Promises aren't bad, not at all! You may very well stick to them and I'll actually come back to promises later in this article. But whilst being relatively new to the JavaScript world, RxJS Observables already gained quite some ground. There are good reasons for that. Here are the two biggest arguments for using observables over promises.

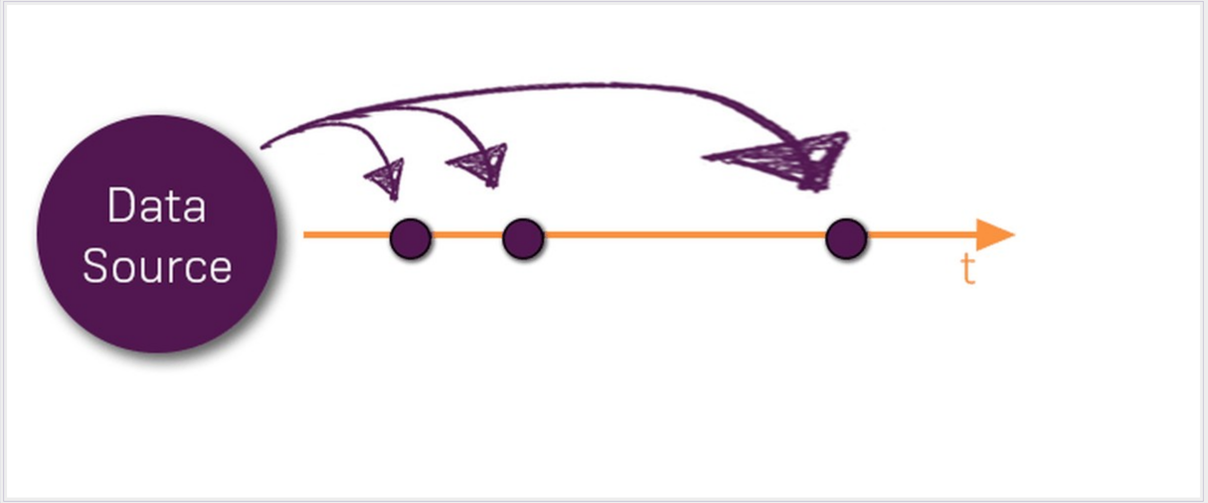
- You're working with **streams of data** instead of single values

- You got an amazing toolset of operators at your disposal to manipulate, transform and work with your async data

#

Streams of Data

Let's start with that "streams of data" thing. What exactly do I mean with that expression?



The data source (for example a button getting clicked) may emit multiple values. It could also only emit one - we don't know! Or maybe we do even know that we'll only receive one data object (e.g. HTTP request + response) - this can still be treated as a stream of event(s) and hence we may use RxJS Observables. More about arguments speaking for observables in all kind of situations will follow later.

For now, let's focus on that stream of data. As mentioned earlier, you can't handle that stream just like that with promises. You can do that with observables though. Here's how it could look like.

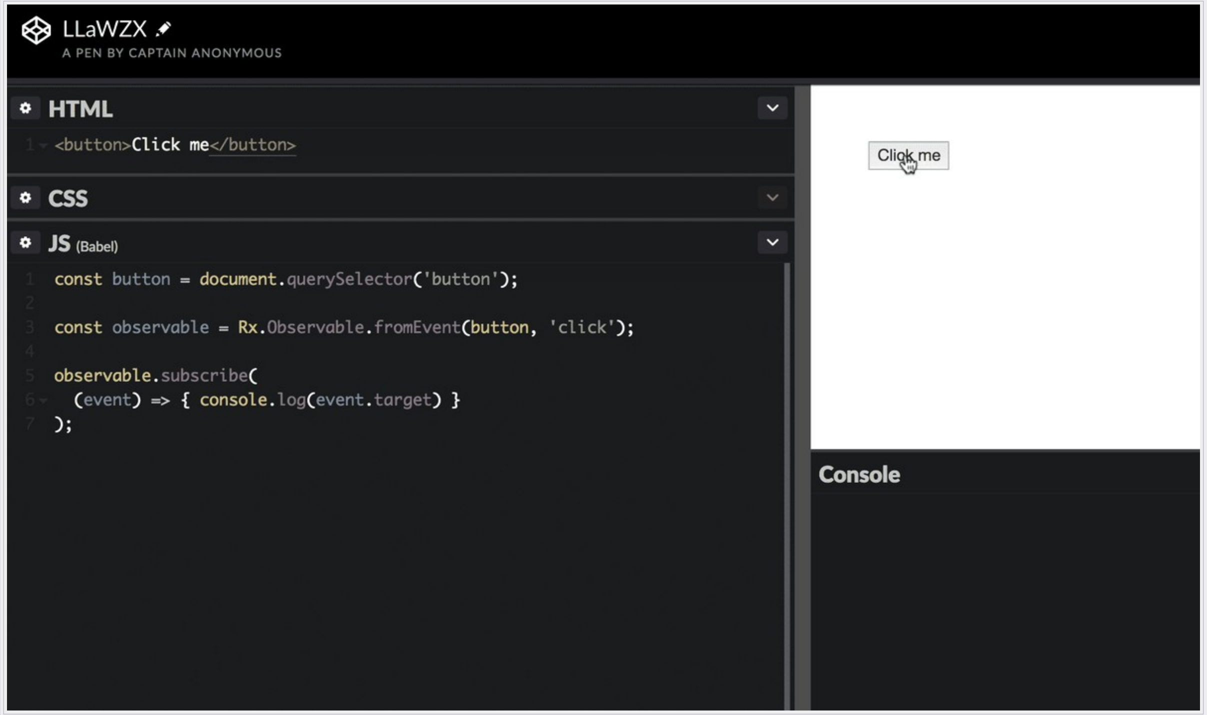
```
const button = document.querySelector('button')
const observable = Rx.Observable.fromEvent(button, 'click')
```

This actually doesn't do anything yet (i.e. we don't react to clicks) but we did set up an observable here. As the name implies, an observable **observes** something. In this case, it observes click events on the button we passed to `fromEvent()`. You could say that clicks on that button are our **data source** now.

That's of course nice but not that useful one its own. We want to react to these clicks, right? No worries, we can! We have an observable which is now watching clicks on that button. With that, we can now set up a subscription on that observable.

```
const button = document.querySelector('button')
const observable = Rx.Observable.fromEvent(button, 'click')
observable.subscribe(event => {
  console.log(event.target)
})
```

With `subscribe()`, we actually subscribe to all the data pieces the observable recognizes. Remember? We do have a stream of data pieces. Whenever a new piece appears, our subscription gets informed. We then react by passing a function as the first argument to `subscription()`. We can pass two other functions but I'll come back to these. In the first function we pass, we receive the data **for each data emission** the observable recognizes. Put in other words: The function we pass to `subscribe()` gets executed whenever a new value is emitted. In our case, whenever the button gets clicked.



```html
<button>Click me</button>
```

```js
const button = document.querySelector('button');

const observable = Rx.Observable.fromEvent(button, 'click');

observable.subscribe(
  (event) => { console.log(event.target) }
);
```

Check this pen to run it yourself.

Being able to react to an infinite amount of asynchronous (or also synchronous!) stream of data is pretty awesome. As you can see, you also have a very clean and simply syntax. You can also listen/ react to two other things: Errors and completion.

#

Errors

If your observable data source emits an error, you probably want to react to that. An example would of course be a HTTP request which errors out. In promises you had `catch()`, with observables you may simply pass another function to `subscribe()`.

```
observable.subscribe(
  event => {
    console.log(event.target)
  },
  error => {
    console.log(error)
  } // <- Handle the error here
)
```

Of course this doesn't make much sense for an observable watching button clicks. This can't throw an error. But many data sources (e.g. HTTP requests, validation, authentication) can throw errors. In such cases, the second function passed to `subscribe()` gets fired - you can implement any logic you want to handle errors inside that function.

#

Completion of an Observable

Some, but not all, observables also complete eventually. Our click-watching observable doesn't. How would it? You don't know if the user is going to click again or not. Other observables do complete though. The good old HTTP request does for example. In such a case, the first function passed to `subscribe()` still gets fired as described above. You can handle the emitted value there. But additionally, a third function gets executed - if you provided it.

```
observable.subscribe(
  event => {
    console.log(event.target)
  },
  error => {
    console.log(error)
  },
  () => {
    console.log('Completed!')
  } // <- Gets executed once the observable completes - it doesn't receive any
argument, no data
)
```

This third argument receives no data, it simply is a place where you can execute any code you want to execute once you know that the observable finished.

These are the basics about observables and you can probably already see why they might be an useful alternative to promises and callbacks. But what do you do if the observable returns a new observable? When using promises, you could simply chain `then()` calls to handle promises resolving new promises. How does that work for observables? And what about that second big advantage I outlined earlier - the many operators?

#

Observables & Operators

As mentioned before, another big advantage of observables are the many operators you can use on them. Thus far, we haven't used any though. `subscribe()` isn't really a special operator but a vital tool to get any use out of observables.

Let's go back to the issue of observables returning new observables. Have a look at the following code.

```
const button = document.querySelector('button');
const observable = Rx.Observable.fromEvent(button, 'click');
observable.subscribe(
  (event) => {
    const secondObservable = Rx.Observable.timer(1000);
    secondObservable.subscribe(
      (data) => console.log(data); // <- Back in callback hell?
    );
  }
);
```

Here, we create a new observable on every click. This observable will then emit one (and only one) value (0) after 1 second. It works fine but to me, it looks a lot like callback hell. Are observables just a more elegant way of getting there? Not at all! We just have to use one of the amazing operators the RxJS library ships with: switchMap().

```
const button = document.querySelector('button')
const observable = Rx.Observable.fromEvent(button, 'click')
observable
  .switchMap(event => Rx.Observable.timer(1000)) // <- use the data of the
first observable in the second one (if you want) and return the new observable
  .subscribe(data => console.log(data))
```

Here's a link to the codepen.

Isn't that beautiful code? We easily map the value of our first (outer) observable into a new (inner) one. We could use the data (in our case the event object) there if we wanted. In the example, we ignore it and instead simply return a new observable which fires after 1 second and returns 0.

There are way more operators than just `switchMap()` and it are these operators which give observables a clear edge over promises - even in cases where you don't really work with a stream of data (like the famous HTTP request). You can simply treat everything, even synchronous data, as a stream and use the awesome operators. Consider this example.

```
const observable = Rx.Observable.of({ name: 'FirstStudent' })
observable.pluck('name').subscribe(data => console.log(data))
```

Here's the code.

This example doesn't have anything asynchronous about it - we simply use RxJS observables here to easily retrieve a value out of an object. Of course there are easier ways to achieve the same result - but keep in mind it's a trivial example. The key takeaway is: Observables are awesome due to their data-stream nature and observables. For both async and sync data.

#

Async & Await

We found the clear winner, didn't we? Observables own everything. Well, observables are amazing and I can only recommend using them. But if you're dealing with an issue where the operators don't help you (even though that's hard to imagine), you might not really face a disadvantage when using promises.

There's one thing which can be annoying when using both promises or observables: You're writing code "in block", not like you normally do. Wouldn't it be nice if you could write async code just like you write synchronous one? Like that?

```
stud = checkStud() // <- async operation
user = getUser(stud) // <- async operation
console.log(user.name)
```

That would be nice but it's of course not possible due to the way JavaScript works. It doesn't wait for your asynchronous operations to finish, hence you can't mix them with your synchronous code like that. But with a **new feature added by ES8**, you suddenly can!

```
async function fetchUser() {
  const stud = await checkStud() // <- async operation
  const user = await getUser(stud) // <- async operation
  return user
}
fetchUser().then(user => console.log(user.name))
```

Here's the code.

In the example, you see that we still use `then()` to react to the data passed by our promise. But before we do that, we actually use two promises (both `checkStud()` as well as `getUser()` return a promise!) to fetch a user. Even though we work with promises here, we write code just like we write "normal", synchronous code. How does that work?

The magic happens via two keywords: `async` and `await`. You put `async` in front of a function to turn it into an async function. Such a function will in the end always resolve as a promise - even though it doesn't look like one. But you could rewrite the example like this.

```
function fetchUser() {
  return checkStud()
    .then(stud => {
      return getUser(stud)
    })
    .then(user => {
      return user
    })
}
fetchUser().then(user => console.log(user.name))
```

This also gives you a clue about what `await` does: It simply **waits** (awesome, right?) for a promise to resolve. It basically does the same `then()` does. It waits for the promise to resolve and then takes the value the promise resolved to and stores it into a variable. The code after `await checkStud()` will only execute once it's done - it's like chaining `then()` calls therefore.

What about handling errors? That's pretty easy, too. Since we're now writing "synchronous" code again (we aren't), we can simply use `try-catch` again.

```
async function fetchUser() {
  try {
    const stud = await checkStud() // <- async operation
    const user = await getUser(stud) // <- async operation
    return user
  } catch (error) {
    return { name: 'Default' }
  }
}
fetchUser().then(user => console.log(user.name))
```

Async/ await gives you a powerful tool for working with promises. It doesn't work with observables though. The reason being that behind the scenes, async/ await really only uses promises.

#

Which Approach Should You Use?

We had a look at four different approaches:-**Callbacks** with the danger of entering callback hell -**Promises** to escape callback hell-**Observables** to handle streams of data and apply operator magic -**async/ await** to write "synchronous" code with promises

#

Which approach should you use?

Use **Callbacks** if you got no other choice or only handle one async operation. The code will then still be perfectly manageable and understandable. Callback functions aren't bad per se - there just exist better alternatives in many cases.

One such case are multiple chained (or dependent) asynchronous operations. You quickly enter callback hell when trying to use callbacks in such a situation. **Promises** are a great tool to handle your operations in a structured and predictable way.

In all cases where you use promises, you might also use **observables**. It's not strictly better but if there's an operator which makes your life easier or you simply love observables, there's no downside in using them. There's a strong argument to be made for observables once you handle data streams, i.e. operations where you're not just getting back one single value. Promises can't handle such situations (easily).

Finally, **async/ await** is an awesome tool for cases where you don't really want or need to use observables but still want to use promises. You can write "synchronous" code with async/ await and handle your promise chains even easier.

Ultimately, it of course also comes down to your taste and the environment you're working in. For everything but callbacks, you'll probably need a transpiler and/ or polyfill. If you only target environments where ES6 or even ES8 is supported natively, that's not true anymore. For observables, you'll always need the RxJS library - that of course means that you'll increase the codebase you ship in the end. Might not be worth it if you're only handling trivial cases.

Async Function in JavaScript

Overview

Whenever we work on Javascript, we have to deal with tasks dependent on other tasks. Let us say we want to download a picture, edit it, and save it. The very first thing we need to do is to download the picture. Only after that we can edit and then save it. Hence we can see that every task is dependent on the previous one.

To achieve this functionality, we will end up with many nested callbacks known as callback hell. Luckily JS involved from callbacks to promises (ES2015) and then further to a more straightforward format of async/await (ES2017).

Scope

What are async functions in javascript and their syntax?

Parameters that the async function accepts and the value returned by async functions.

The execution order of async and await functions.

The importance and functionality of await expression inside an async function in javascript.

Migrating a Promise chain to an async format function using await and try-catch block.

The browsers that support async await functionality.

Syntax of Javascript Async Function

An async function in javascript is a combination of a **generator** and **promises**.

Async functions are built on top of traditional javascript **promises**. Thus they are just a higher level of generalization or an abstraction layer over-promise.

The purpose of **async / await** is to simplify the code syntax necessary to consume the promise returned by an API.

The async function in javascript is always prefixed with the async keyword and followed by the syntax of a simple javascript function. It operates asynchronously via the event loop and always makes a function returns a promise.

Syntax:

```
// Async function declaration
async function fun1(param0) {
  // promise and other statements
}

// Async function expression
const func2 = async function () {
  // promise and other statements
}

// Async arrow function
const func3 = async () => {
  // promise and other statements
}
```

```
async function fun4(param0, param1, /* … ,*/ paramN) {
  // promise and other statements
}
```

Example:

```
// a simple async function in javascript that accepts no parameters
async function custom() {
  return 'Hello'
}
```

Parameters of Javascript Async Function

Lets us consider the following async function in javascript:

```
async function fun1(param0, param1, . . ., paramN) {

    // promise and other statements

}
```

Here the parameters are:

name Optional:

The name of the async function. (fun1)

param Optional:

They are the name of the arguments that are passed to the async function. (param0)

statements Optional:

This is the body of the async function in javascript. It can consist of a promise.

Return Value of Javascript Async Function

The Async function in javascript always returns a promise. Even if the async function in javascript is not explicitly returning a promise, the async function will internally return a promise.

This promise will be either resolved with the value returned by the function or rejected with a thrown exception from within the async function, which will depend on the conditions.

```
// simple function which returns a promise

const doSomethingAsync = () => {
  return new Promise((resolve) => {

    setTimeout(() => resolve('Promise returned'), 2000)

  })

}



// async function in the javascript example which resolves the promise

const doSomething = async () => {

  doSomethingAsync().then((res) => console.log(res))

}



console.log('Example that promise is returned by async function')
```

doSomething()

Output:

For example, that promise is returned by an async function

Promise returned

Note:

The main purpose of async/ await functions is to simplify the syntax of promises when they (promise) are used synchronously. It is similar to the combination of promise and generator.

Java Course Online for BeginnersTARUN LUTHRA

Java Course Online for Beginners

44k+ enrolled

Free Python Certification Course: Master the essentialsRAHUL JANGHU

Free Python Certification Course: Master the essentials

28k+ enrolled

Free C++ Course: Learn the EssentialsPRATEEK NARANG

Free C++ Course: Learn the Essentials

24k+ enrolled

35,262+ learners have attended these Courses.

Example of Async Function in Javascript

Understanding an async function in javascript with the help of a real-world example is helpful:

Let us suppose we wish to have some tomato juice made at home, but there are a few steps to be followed:

We need to buy some tomatoes from the market.

We need to peel the tomatoes.

We need to add it to a juicer to make the juice.

Code using async await:

```
// function to buy tomatoes which were step 1

function buyTomatoes() {
  return new Promise((resolve, reject) => {

    setTimeout(() => resolve(console.log('Buy tomatoes from the store.')), 1000)

  })

}



// function to peel tomatoes which was step 2

function peelTomatoes() {

  return new Promise((resolve, reject) => {

    setTimeout(

      () => resolve(console.log('Peel off the skin of the tomatoes.')),

      1000,
    )
  })
}


// function to add tomatoes to the juicer which was step 3
function addTomatoesInJuicer() {
  return new Promise((resolve, reject) => {
    setTimeout(
      () => resolve(console.log('Add peeled pieces of tomatoes to juicer.')),
      1000,
    )
  })

}



// async function in javascript to resolve the promises by all of the above three functions one by one

async function executeTheProcess() {

  await buyTomatoes()

  await peelTomatoes()

  await addTomatoesInJuicer()

}



// execution of the process begins from here
```

```
executeTheProcess()
```

Output:

```
Buy tomatoes from the store.

Peel off the skin of the tomatoes.

Add peeled pieces of tomatoes to the juicer.
```

Benefits of using an async function in javascript:

In the case of API calls, fetching data from the database or any information from the server.

In the case of using the timeouts functionality.

Increasing the responsiveness and performance of the application in case of extended running programs.

Organizing the code to look clean and readable.

What is Javascript Async Function?

Async/Await is just syntactic sugar for promises.

They help us in the writing of asynchronous and promise-based behavior, in a cleaner and readable manner. They are built on top of the existing promises in javascript.

Promise code:

```
function greet() {

  if (true) Promise.resolve('Hello!')
  else Promise.reject('NO')
}
```

Async code:

```
async function greet() {
  return 'hello!'
}
```

async functions in javascript are marked with the keyword async and inside these async functions, we write promise-based code as it was synchronous. We just need to use the await operator whenever a promise is returned. This operator halts the execution of the current function until the promise is settled.

internal-working-of-async-functions

Explanation of working of an async function in javascript:

```
// function which returns a promising value to the myfunc()

const doSomething = () => Promise.resolve('Something')



// async function in javascript with await to store the resolved promise value.

async function myfunc() {

  console.log('In the function!')

  const res = await doSomething()

  console.log(res)
}



console.log('Before function!')

// calling the main async myfunc()

myfunc()

console.log('After function!')
```

Output:

```


Before function!

In the function!

After function!
```

Something

Explanation and working of the above code in depth:

Foremost, a console.log() is encountered by the engine at line 11. It gets added to the main call stack after which Before function! gets logged into the console.

Then myfunc() is called and hence it is pushed onto the main call stack, after this the function starts its execution. Here the engine encounters the second console.log() on line 6, which gets popped onto the main call stack, after which Before function! gets logged into the console and the function gets popped from the call stack.

After that await expression is encountered inside the myfunc(); here, the function doSomething() gets into the call stack and ultimately returns a resolved promise. When this promise is resolved, and a value is returned from the function doSomething(), await is encountered. The engine immediately suspends the async function execution. The rest of the async function in javascript gets halted and is set to run inside the microtask queue.

Now the engine jumps out of the async function and continues the execution of the code in the caller function. Here the engine encounters another console.log() on line 14 which gets popped into the main call stack, after which After function! gets logged into the console. Finally, the call stack is empty, and hence event loop checks to see if there are any pending tasks in the microtasks queue for execution.

myfunc() again gets popped into the call stack and continues its execution where it earlier left off. The variable res gets its value from the resolved promise from doSomething(), another console.log() on line 8 is popped onto the stack, Something is printed in the console, and the async function is popped from the call stack.

Note:

There are two types of queues to manage async behavior, a micro task queue which can only contain process.nextTick(), async, and promise callbacks; second is a microtask queue which can only contain setTimeout(), setInterval(), and setImmediate().

More Examples

Async Functions and Execution Order

The sequential execution order of the async function in javascript can be explained using the following example:

```
function resolveAfter2Seconds() {

  console.log('Starting slow promise')

  return new Promise((resolve) => {

    setTimeout(() => resolve('Slow promise is done'), 2000)

  })

}



function resolveAfter1Second() {

  console.log('Starting fast promise')
  return new Promise((resolve) => {

    setTimeout(() => resolve('Fast promise is done'), 1000)

  })

}



async function sequentialStart() {

  // 1. Execution gets here almost instantly

  const slow = await resolveAfter2Seconds()

  // 2. this runs 2 seconds after 1.

  console.log(slow)
  const fast = await resolveAfter1Second()

  // 3. this runs 3 seconds after 1.

  console.log(fast)

}



// total execution time taken is 3 seconds

sequentialStart()
```

Output:

```
Starting slow promise

The slow promise is done
```

```
Starting fast promise

The fast promise is done
```

In the above example, the sequential execution starts instantly and reaches line 17; after that, execution gets suspended for 2 seconds until the first function is resolved. After that, there is another delay of 1 second for the second function to resolve. So in total, the code takes 3 seconds to execute.

Await Function

The await operator causes an async function in javascript execution to pause until a promise is resolved or rejected that is fulfilled. It also resumes the execution after the execution of the promise is settled. If the promise is resolved, then the value of that awaiting expression has the value of the promise. If the promise is rejected, that value is thrown by the await expression.

After the wait delays the continuance of the async function in javascript, the execution of succeeding statements follows. If await is the last expression in the execution order, execution continues by returning to the function's caller a pending Promise for completion of the await's function and resuming execution of that caller.

Syntax:

```
let value = await promise
```

Example:

```
function resolveAfter2Seconds() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve('Promise is resolved')

    }, 2000)

  })

}



// async function in javascript

async function fun1() {

  // res holds the value after a promise is resolved

  const res = await resolveAfter2Seconds()

  console.log(res)

  console.log('After statements')
```

```
}



// calling the function

fun1()
```

Output:

```


Promise is resolved

After statements
```

Rewriting a Promise Chain with an Async Function

Writing an async function in javascript using the promise syntax and then chaining them can reduce the code readability, and maintaining the code can get more complex. Therefore as async await has a simpler syntax, along with its efficient error handling abilities, it seems a better solution than promises chaining.

Real-world example of Promise chaining of food order:

```
function restaurantCustomer() {
    return get customer()
    .then(customer => {
        return getOrder(customer);
    }).then(order => {
        return prepare food(order);
    }).then(meal => {
        return serve food(meal);
    }).then(food => {

        return eat food(food, customer);

    }).catch(show error);
```

Now to convert it into async await we can follow the following steps:

Conversion of .then() calls to await:

In the classic promise syntax, the promise was resolved using the .then() function and was rejected using the .catch() keyword. But in the new async await format, the function will be called using the await operator, which halts the execution of the current function until the called function is settled.

Using async keyword:

All the function that uses await must be followed by the async keyword.

Error handling using the try-catch block:

We will wrap the entire code in a try block and then catch errors if any.

Promise chain migrated to async await format code of the above example of food order:

```
async function restaurantCustomer() {

  try {

    let customer = await get customer()

    let order = await getOrder(customer)

    let meal = await prepare food(order)

    let food = await serve food(meal)

    return await eat food(food)

  } catch (e) {
    show error(e)

  }

}
```

This format of async awaits benefits over Promise Chains because of simple and flexible code, easy chaining, Efficient error handling, and no nesting of callbacks; hence layout is simplified.

Supported Browsers

Async function js statements are supported in the following browser version:

Browser	Version
Chrome	55

Edge	15
Firefox	52
Opera	42
Safari	10.1
Deno	1.0
Node.js	7.6.0
Samsung Internet	6.0

Conclusion

Async function in javascript is a way to execute the asynchronous code synchronously and clearly.

Async functions are generally a combination of generators and promises, basically syntactic sugar for promises.

The async/await is a very useful tool for writing clean code hence helping in avoiding callback hells.

When a function is prefixed with async, then the function always returns a promise which can be handled by the await keyword.

Even if the function is not returning a promise explicitly, the engine will internally make the function return a promise.

Await returns the resultant value when the promise is resolved, and if the promise is rejected, the await operator throws the rejected value.

Await operator can only be used inside an async function.

The async function in javascript follows sequential order of execution.

Async await provides us with a way to migrate from promise chaining and maintain a readable and more manageable code using await expression and a try-catch block.

Node Js Callback Hell

quiz

Challenge Inside! : Find out where you stand! Try quiz, solve problems & win rewards!

Go to Challenge

Overview

Javascript is a synchronous and single-threaded programming language, but some tasks in javascript can run in parallel due to its asynchronous nature. In javascript, through the use of Callback functions, we can perform asynchronous tasks. Callbacks seem easy to implement but can lead to the problem of callback hell, which makes the code unable to read and maintain. Although the callback hell in Node.js can be easily avoided by many ways; a few of them are using async/ await and Promises.

Scope

Asynchronous nature of programming in Node.js.

What are callbacks in Node.js?

What is Node js callback hell?

Asynchronous and synchronous nature of programming using callbacks.

Examples of callbacks and callback hell in node js.

Methods to avoid callback hell using Promises, async / await, splitting of the function, and by writing comments.

Pre-requisites

There are few general prerequisites :

Node.js must be installed on your development system whether it be windows, macOS, or Linux.

You must be comfortable constructing and executing functions in JavaScript before learning how to use them asynchronously with help of callbacks.

Asynchronous Programming in Node.js

The earlier websites were mainly static, and the data displayed by those websites was in HTML form which was not very interactive. Presently, the websites have become very user-friendly (using javascript) and include various intensive operations like fetching data from any external API. To handle these types of operations, a developer needs to know Asynchronous programming in Nodejs.

Javascript is a single-threaded, synchronous programming language which means the execution of statements takes place one step at a time. However, if we want to fetch data from any API, it can take any amount of undetermined time, which can depend on Network speed, Server availability, and connection traffic.

If API calls were performed in a synchronous manner, then the browser would not be able to maintain interactive behavior (like taking any kind of input from the user, like scrolling, mouse click, or button press) until that API request is complete. This is known as blocking nature. In contrast, asynchronous programming is a technique that ensures non-blocking code execution.

To prevent blocking nature, the browser has numerous Web APIs that JavaScript can access that are asynchronous in nature. It means the statements can run in parallel with other operations. The Event Loop in Node.js was designed to aid with the interactions between the main application thread and asynchronous components. Hence asynchronous programming in Node.js is very important and essential.

Example: Suppose we want to download an image from the internet, then edit the downloaded image (cropping or something else) and then save the edited image on the hard disk of our system.

The above process can be defined using three simple steps:

Downloading the image from the internet.

Editing the image.

Saving the edited image on the hard disk.

All the steps are dependent on the previous steps; hence we need asynchronous programming in Nodejs. This can be achieved by using callbacks in Node.js.

```
// we are assuming all the below functions are defined somewhere in your program

downloadImageFunction(args, function() {

  editImageFunction(args, function() {

    saveImageFunction(args, function() {

    });
  });

});
```

What is a Callback?

A callback function is a simple javascript function that is passed as an argument to another function and is executed when the other function has completed its execution. In layman's terms, a callback is generally used as a parameter to another function. Callbacks in Node.js are so common that you probably used callbacks yourself without understanding that they are called callbacks.

A simple example of a function that everyone must have used must be,

```
// Adding event to a button when it is clicked

const btn = document.getElementById('button');
```

```
btn.addEventListener('click', function(e) {

    // Displaying the alert when the button was clicked

    alert("The button was clicked");

})
```

Simplifying the above example:-

```
// Adding event to a button when it is clicked

const btn = document.getElementById('button');



function clicked (e) {
     // Displaying the alert when the button was clicked

    alert("The button was clicked");

}



// Adds clicked function as a callback to the event listener

btn.addEventListener('click', clicked)

})
```

Now we can clearly see that the clicked function is a parameter to the addEventListener function and is a callback. Also, the function that takes another function as an argument is called a higher-order function.

Syntactic Code Example of a Higher-Order Function and a Callback

```
// A function

function fnc() {

  console.log('Just a simple javascript function');

}



// A function which is taking another function as an parameter

function higherOrderFunction(callback) {

  // Calling the function which is passed as the parameter, it is referred to as a callback

  callback();

}
```

```
// Passing a function

higherOrderFunction(fnc)
```

The callback functions are used a lot in asynchronous Node.js, but it does not mean that callbacks are not used in synchronous programming.

Free Python Certification Course: Master the essentialsRAHUL JANGHU

Free Python Certification Course: Master the essentials

28k+ enrolled

Free C++ Course: Learn the EssentialsPRATEEK NARANG

Free C++ Course: Learn the Essentials

24k+ enrolled

35,262+ learners have attended these Courses.

Callbacks in Synchronous Programming

There are many functions that use callbacks synchronously. One of them is when you use some of the array methods available in JavaScript.

```
let arr = [1, 2, 3, 4, 5, 6];



arr.map(number =>

  number * 10

)
```

Output :

```
[10, 20, 30, 40, 50, 60]
```

Explanation:

In the above example, we are using the array.map() method, which takes in a callback as an argument.

As the function traverses through the given array, each element of the array will be passed in as a parameter into our callback function. The callback function is passed in the map function, intended to be used later when our code executes.

When our code executes, the array.map() function will use our callback function for every single index in our array. We defined what we want the callback function to do, which in this example is to multiply 10 to every argument(in the above case, the number in our array at the current iteration) we pass into the callback function, which is all happening synchronously.

Callbacks in Asynchronous Programming

An example of an asynchronous use of callbacks is when we use the setTimeout() function. setTimeout takes in two arguments, a callback function and how long to delay in milliseconds before running that callback function.

Example :

```
// this function takes a callback which is executed after 5 second
const foodOrder = (ordNumber, callback) => {
   setTimeout(() => {
     const foodDescription = {
        order: 3,
        meal: 'Burger',
        drink: 'Sprite'
     }
     callback(foodDescription);
   }, 5000)
}


// calling the foodOrder function and passing a callback function as a parameter
foodOrder(5, (foodDescription) => {
  console.log(foodDescription)
});
```

Output :

```
// After 3 Seconds the function prints

{ order: 3, meal: 'Burger', drink: 'Sprite' }
```

Explanation:

In this above example, we have passed a callback function as the second argument to foodOrder, allowing us to use it in our asynchronous setTimeout function. This callback is used for logging our foodData in the console after 5 seconds.

What is Callback Hell in NodeJs?

Callback functions are a useful way to confirm the delayed execution of a function until another function completes its execution and returns with data. However, we may need to nest callbacks inside callbacks, and this nested nature of callbacks can scale horizontally and become unreadable as well as messy if you have a lot of consecutive asynchronous requests that rely on each other. This nesting of callbacks is known as node js callback hell or the Pyramid of Doom.

Here is an example of nested callbacks:

```
// nesting of callbacks

// node js callback hell

function pyramidOfDoom() {



// first setTimeout function

setTimeout(() => {

    console.log(1)

    // second nested setTimeout function
    setTimeout(() => {

        console.log(2)

        // third nested setTimeout function

        setTimeout(() => {

            console.log(3)

        }, 500)

    }, 2000)

}, 1000)

};
```

Output :

```
1

2

3
```

In the above code, each new setTimeout() is nested inside a higher order function, creating a pyramid of doom or node js callback hell.

Example of Callback Hell in NodeJs

An example showing how node js callback hell is constructed, let us imagine we’re trying to make a burger. To make a burger, we need to follow these steps:

Get ingredients (we’re gonna assume it’s veg burger)

Cook the Aloo patty

Get burger buns

Put the cooked patty and vegies between the buns

Serve the burger

Let’s say we can’t make the veg burger ourselves. We have to instruct a helper on the steps to make the burger. After every instruction, we have to wait for the helper to finish his task, then give him the next instruction. Hence if we want to wait for this program in JavaScript, we need to use callbacks.

To make the burger, we have to get the ingredients first.

```
// node js callback hell example

const makeBurger = () => {

  getIngredients(function(potato) {

    // We can only cook potato after we get it with other ingredients.
  });

};
```

Now we need to instruct the helper to cook the patty. And after that get the buns.

```
// node js callback hell example

const makeBurger = () => {

  getBeef(function(beef) {



  });

};

const makeBurger = () => {

    getIngredients(function(potato) {
        // We can only cook potato after we get it with other ingredients.

        cookPatty(potato, function(cookedPatty) {

            // get the buns

            getBuns(function(buns) {

            // Put patty in bun

            });

      });

    });

};
```

After we get the buns, we need to put the patty between the buns.

```
// node js callback hell example
```

```
const makeBurger = () => {

  getIngredients(function(potato) {

    // We can only cook potato after we get it with other ingredients.

    cookPatty(potato, function(cookedPatty) {

      // get the buns

      getBuns(function(buns) {

        // Put patty in bun

        putAlooBetweenBuns(buns, potato, function(burger) {

        // Serve the burger

        });

      });
    });

  });

};
```

Finally, we can serve the burger! But we can’t return burger from makeBurger because it’s asynchronous. We need to accept a callback to serve the burger.

```
// node js callback hell example

const makeBurger = () => {

  getIngredients(function(potato) {

    // We can only cook potato after we get it with other ingredients.

    cookPatty(potato, function(cookedPatty) {

      // get the buns
```

```
            getBuns(function(buns) {
                // Put patty in bun
                putAlooBetweenBuns(buns, potato, function(burger) {
                    // Serve the burger
                    nextStep(burger);
                });
            });
       });
    });
};
```

```
// function to Make and serve the burger

makeBurger(function (burger) => {

  serve(burger);

})
```

A working and running example of callback hell :

```
// Example asynchronous function

function asynchronousFunc(params, callback) {

    // This function throws an error if no parameters are passed

    if (!params) {

    return callback(new Error('Something went wrong with this request'));
    } else {

      // returning a random number so it may seem like the contrived asynchronous function

    return setTimeout( () =>

        callback(null, {body: params + ' function is executed. ' + Math.floor(Math.random() * 10)}),

        500)

    }

}



// node js callback hell

// Nesting of the callbacks

function callbackHell() {
```

```
asynchronousFunc('First', function first(error, response) {

  if (error) {

    console.log(error);

    return;

  }

  console.log(response.body);

  asynchronousFunc('Second', function second(error, response) {

    if (error) {

      console.log(error);

      return;

    }
```

```
      console.log(response.body);

      // passing null parameter so that other nested functions inside this request will not execute

      asynchronousFunc(null, function third(error, response) {

        if (error) {

          console.log(error);

          return;

        }

        console.log(response.body);

      })

    })

  })
```

```
};

// Execute the pyramid of doom
callbackHell();
```

Output :

```
First function is executed. 0
Second function is executed. 4
Error: Something went wrong with this request
```

Explanation:

The function and second request to asynchronousFunc are executed because a parameter are passed. The third request to asynchronousFunc is passed with a null parameter. Hence this request throws an error, which affects all the other functions nested inside this one. Hence the executions stops at this request.

Ways to Avoid NodeJs Callback Hell In

Let us take our makeBurger example again and try to avoid callback hell using the following solutions:

Writing Comments

The makeBurger() callback hell in Nodejs is simple to understand to us. We can read it. It just doesn’t look readable and straightforward.

If someone is reading makeBurger() for the first time, they may think, “What is the need for so many nested callbacks to make a simple function?”.

```
// Makes a burger

// makeBurger contains four steps:

// 1. Get ingredients (we’re gonna assume it’s veg burger)

// 2. Cook the Aloo patty

// 3. Get burger buns

// 4. Put the cooked patty and vegies between the buns

// 5. Serve the burger

// We have used callbacks here because each step is asynchronous.

//   We have to wait for the helper to finish the previous step before we can start the next step
```

```
const makeBurger = () => {
  getIngredients(function(potato) {
    cookPatty(potato, function(cookedPatty) {
      getBuns(function(buns) {
        putAlooBetweenBuns(buns, potato, function(burger) {
          nextStep(burger);
        });
      });
   });
  });
};



// function to Make and serve the burger

makeBurger(function (burger) => {

  serve(burger);

})
```

Now the readability of the program has been increased due to the comments.

Split Functions into Smaller Functions

Let us write the step-by-step imperative code of all the functions under makeBurger().

For getIngridients(), the first callback, we have to go to the fridge and get the ingredients like tomatoes, potatoes, and onions. Suppose there are fridges on the top and bottom floor of the kitchen. We need to go to the bottom fridge.

```
// avoiding node js callback hell by splitting the big method

const getIngredients = callback => {

    const storage = bottomStorage;

    const aloo = getPotatoFromFridge(fridge);

    // other ingredients also similarly

    callback(potato);

};
```

To cook potatoes, we need to wash and peel the potatoes and then fry those potatoes, and wait for 10-20 minutes.

```
// avoiding node js callback hell

const cookPatty = (potatoes, callback) => {

    // cleaning and peeling of potatoes then

    const workInProgress = deepFry(potatoes);

    setTimeout(function() {

        callback(workInProgress);

    }, 20 * 60 * 1000);

};
```

Similarly, we can write other functions also under the makeBurger() function, but this will lead to a large codebase.

Using Promises

We can use promises instead to improve the syntax of the makeBurger(). Promises can make callback hell in the Node.js program much easier to manage and read. Instead of the nested callbacks code you saw in the original example, you will have the following code:

```
// avoiding node js callback hell using promises

const makeBurger = () => {

  return getIngredients()

    .then(potatoes => cookPotatoes(potatoes))

    .then(cookPotatoes => getBuns(potatoes))

    .then(bunsAndAloo => putAlooBetweenBuns(bunsAndAloo));

};
```

```
// Make and serve burger

makeBurger().then(burger => serve(burger));
```

Further simplifying the above code,

```
// avoiding node js callback hell using chaining the promises

const makeBurger = () => {

  return getPotatoes()

    .then(cookPotatoes)

    .then(getBuns)

    .then(putAlooBetweenBuns);
};



// Make and serve burger using then and  catch

makeBurger().then(serve);
```

We can observe that the above code is much more easier to read and manage than the original example. Now to convert the callback-based code into promise-based code, we need to create a new promise for each callback. Then we need to resolve the promise if the callback is successfully executed or reject the promise if the callback fails.

```
// avoiding node js callback hell

const getIngredients = _ => {

    const storage = bottomStorage;

    const aloo = getPotatoFromFridge(fridge);
```

```
  // other ingredients also similarly



  //returning a new promise for the callback

  return new Promise((resolve, reject) => {

    if (potatoes) {

      resolve(potatoes);

    } else {

      reject(new Error(“No more potatoes in the storage fridge!”));

    }

  });

};
```

```
const cookPatty = (potatoes) => {

    // cleaning and peeling of potatoes then

    const workInProgress = deepFry(potatoes);



    //returning a new promise for the callback

    return new Promise((resolve, reject) => {                   setTimeout(function() {

            resolve(workInProgress);

        }, 20 * 60 * 1000);

    });
};
```

In Node.js, each of the other functions that contain a callback can be written using the same syntax.

Syntax:

```
// The function that’s defined for you

const functName = (param1, param2, callback) => {

  // Do stuff here

  callback(err, stuff);

};
```

```
// How you use the function

functionName(param1, param2, (err, stuff) => {

    if (err) {

         console.error(err);

     }

     // Do stuff

});
```

4. Using Async/ await

With Async/ await functions, we can write makeBurger() as if it is synchronous in nature.

```
// avoiding node js callback hell using async await
// making the makeBurger function async and awaiting for the response from the promises

const makeBurger = async () => {

    const potatoes = await getPotatoes();

    const cookedPotatoes = await cookPotatoes(potatoes);

    const buns = await getBuns();

    const burger = await putAlooBetweenBuns(cookedPotatoes, buns);

    return burger;

};



// Make and serve burger

makeBurger().then(serve);
```

One improvement can be made to the makeBurger() method here. You can likely get two helpers to getBuns and getPotatoes at the exact same time. This means you can await them both with Promise.all().

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

  // using Promise.all so that we can wait for both the functions to get completely executed

  const [potatoes, buns] = await Promise.all(getPotatoes, getBuns);

  const cookedPotatoes = await cookPotatoes(potatoes);

  const burger = await putAlooBetweenBuns(cookedPotatoes, buns);

  return burger;

};
```

```
// Make and serve burger

makeBurger().then(serve);
```

Conclusion

Javascript is a synchronous and single-threaded programming language, but some tasks in javascript can run in parallel due to its asynchronous nature.

Asynchronous nature can be achieved with the help of callbacks in Node.js.

Callbacks are used in some synchronous methods of Node.js like array.map().

Callbacks are basically methods that are passed as the parameters to the other function.

Functions that take other functions as a parameter are known as Higher Order functions.

Nesting of callbacks can lead to an unreadable and not easy to manageable codebase commonly known as callback hell in Node.js or pyramid of doom.

Node js callback hell can be avoided using Promises and async / await.

Splitting of the functions and by writing comments can also be used to avoid callback hell in Node js.