8. Async

Asynchronous code execution in JavaScript **1**

Basic Architecture **2**

Non-Blocking example: **3**

Blocking example: **5**

Callbacks **6**

Callback hell **6**

Promises **7**

Chaining **9**

Async/Await **10**

Error handling with async/await **11**

An important consideration regarding async/await 11

Implementation **11**

Asynchronous code execution in JavaScript

According to Wikipedia: *Asynchrony in computer programming refers to the occurrence of events independently of the main program flow and ways to deal with such events.*

In programming languages like Java or C# the “main program flow” happens on the main thread and “the occurrence of events independently of the main program flow” means that some code is running on a different thread that is executed in parallel to the “main program flow”.

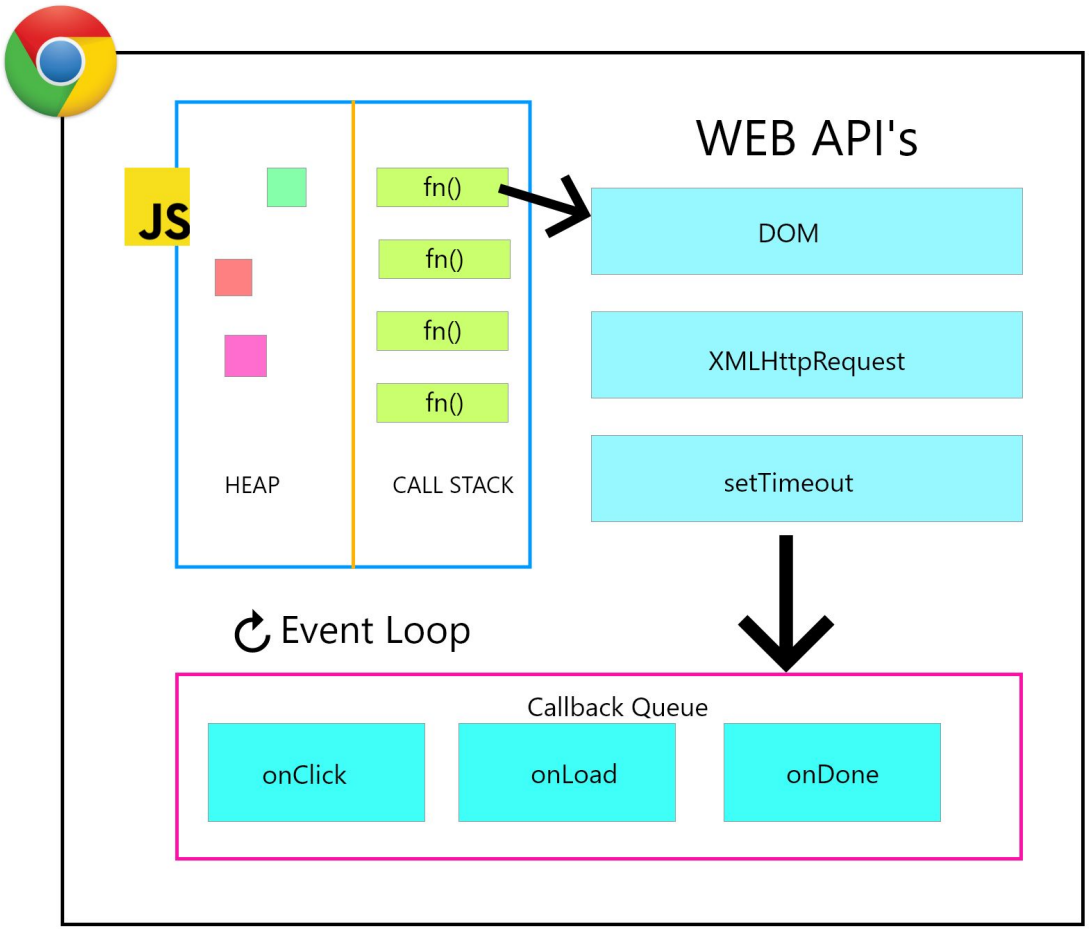
This is not the case with JavaScript. That is because a JavaScript program is single threaded and all code is executed in a sequence, not in parallel. In JavaScript this is handled by using what is called an “asynchronous non-blocking I/O model”. What that means is that while the execution of JavaScript is blocking, I/O operations are not.

I/O operations can be:

1. fetching data over the internet with Ajax or over WebSocket connections,
2. querying data from a database such as MongoDB,
3. accessing the filesystem with the NodeJS “fs” module.

All these kinds of operations are done in parallel to the execution of your code and **it is not JavaScript that does these operations**; to put it simply, the underlying engine does it.

Basic Architecture



1. **Heap** - Objects are allocated in a heap which is just a name to denote a large mostly unstructured region of memory

2. **Stack** - This represents the single thread provided for JavaScript code execution. Function calls form a stack of frames (more on this below)

3. **Browser or Web APIs** are built into your web browser, and are able to expose data from the browser and surrounding computer environment and do useful complex things with it. They are not part of the JavaScript language itself, rather they are built on top of the core JavaScript language, providing you with extra tools to use in your JavaScript code. For example, the Geolocation API provides some simple JavaScript constructs for retrieving location data so you can say, plot your location on a Google Map. In the background, the browser is actually using some complex lower-level code (e.g. C++) to communicate with the device's GPS hardware (or whatever is available to determine position data), retrieve position data, and return it to the browser environment to use in your code. But again, this complexity is abstracted away from you by the API

Non-Blocking example:

```
function main() {
  console.log('A');
  setTimeout(() => {
    console.log('B');
  }, 0);
  console.log('C');
}
main();
//  Output:
//  A
//  C
//  B
```

Here we have the main function which has 2 console.log commands logging ‘A’ and ‘C’ to the console. Between them is a setTimeout call which logs ‘B’ to the console with 0ms wait time.

Function Flow:

1. The call to the main function is first pushed into the stack (as a frame).
2. Then the browser pushes the first statement in the main function into the stack which is console.log(‘A’). This statement is executed and upon completion that frame is popped out. Letter A is logged to the console.
3. The next statement (setTimeout() with callback exec() and 0ms wait time) is pushed into the call stack and execution starts. setTimeout function uses a Browser API to delay a callback to the provided function. The frame (with setTimeout) is then popped out once the handover to the browser is complete (for the timer).
4. console.log(‘C’) is pushed to the stack while the timer runs in the browser for the callback to the exec() function. In this particular case, as the delay provided was 0ms, the callback will be added to the message queue as soon as the browser receives it (ideally).
5. After the execution of the last statement in the main function, the main() frame is popped out of the call stack, thereby making it empty. For the browser to push any message from the queue to the call stack, the call stack has to be empty first. That is why even though the delay provided in the setTimeout() was 0 seconds, the callback to exec() has to wait till the execution of all the frames in the call stack is complete.
6. Now the callback exec() is pushed into the call stack and executed. The letter B is logged to the console. This is the event loop of javascript.

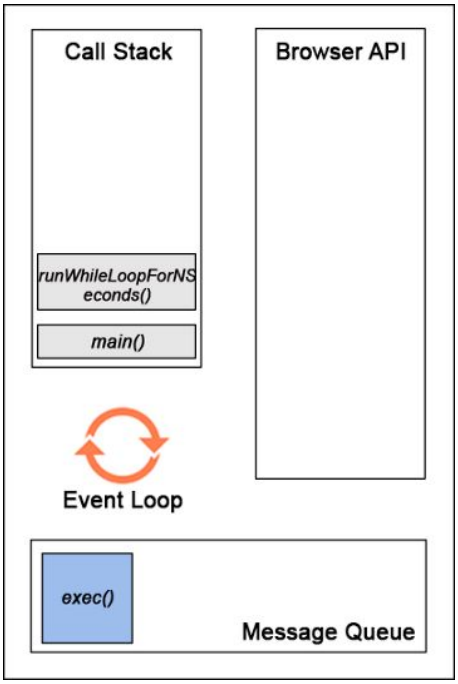
The delay parameter in setTimeout(function, delayTime) does not stand for the precise time delay after which the function is executed. It stands for the minimum wait time after which at some point in time the function will be executed.


Call Stack
Browser API
main()
Event Loop
Message Queue
1
Call Stack
Browser API
console.log('A')
main()
Event Loop
Message Queue
2
Call Stack
Browser API
setTimeout(function exec() { console.log('B')} ,0);
main()
Event Loop
Message Queue
3
Call Stack
Browser API
exec()
timer running
console.log('C');
main()
Event Loop
Message Queue
4
Call Stack
Browser API
Event Loop
exec()
Message Queue
5
Call Stack
Browser API
exec()
Event Loop
Message Queue
6

Blocking example:

```
function runWhileLoopForNSeconds(sec: number) {
  let start = Date.now(),
    now = start;
  while (now - start < sec * 1000) {
    now = Date.now();
  }
}
function main() {
  console.log('A');
  setTimeout(() => {
    console.log('B');
  }, 0);
  runWhileLoopForNSeconds(3);
  console.log('C');
}
main();
// Output
// A
// ...Main thread is blocked in a while loop for 3 seconds...
// C
// B
```

1. The function runWhileLoopForNSeconds() does exactly what its name describes. It constantly checks if the elapsed time from the time it was invoked is equal to the number of seconds provided as the argument to the function. The main point to remember is that while loop (like many others) is a blocking statement meaning its execution happens on the call stack and does not use the browser APIs. So it blocks all succeeding statements until it finishes execution.
2. So in the above code, even though setTimeout has a delay of 0s and the while loop runs for 3s, the exec() call back is stuck in the message queue. The while loop keeps on running on the call stack (single thread) until 3s has elapsed. And after the call stack becomes empty the callback exec() is moved to the call stack and executed.



3. So the delay argument in setTimeout() does not guarantee the start of execution after the exact time elapsed. It serves as a minimum time for the delay part.

Callbacks

For JavaScript to know when an asynchronous operation has a result (a result being either returned data or an error that occurred during the operation), it points to a function that will be executed once that result is ready.

This function is what we call a “callback function”. Meanwhile, JavaScript continues its normal execution of code. This is why frameworks that do external calls of different kinds have APIs where you provide callback functions to be executed later on.

In our example with the setTimeout call, the callback is the arrow function calling console log.

Here is another example, fetching data from a URL using a module named “request”:

```
import request from 'request';

const sendRequest = request(
  'https://www.somepage.com',
  (error, response, body) => {
    if (error) {
      // Handle error.
    } else {
      // Successful, do something with the response and the body.
    }
  },
);
```

As you can see, `request` takes a function as its last argument. This function is not executed together with the code above. It is saved to be executed later once the underlying I/O operation of fetching data over HTTP(s) is done.

Callback hell

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

```
import request from 'request';

const sendRequest = request(
  'https://www.somepage.com',
  (error, response, body) => {
    if (error) {
      // Handle error.
    } else {
      // Successful, do something with the response and the body.
      request(
        `https://www.somepage.com/${response}`,
        (nextError, nextResponse, nextBody) => {
          if (nextError) {
            // Handle error.
          } else {
            // Successful, do something with the response and the body.
          }
        },
      );
    }
  },
);
```

As you can see in the example if we want to do multiple asynchronous calls in sequence we need to nest more and more callbacks into each other. This is an anti-pattern called a callback hell.

Promises

To deal with the callback hell, the Promises were introduced in ES6.

A promise is an object that wraps an asynchronous operation and notifies when it's done. This sounds exactly like callbacks, but the important difference is in the usage of Promises. Instead of providing a callback, a promise has its own methods which you call to tell the promise what will happen when it is successful or when it fails.
The methods a promise provides are `then(…)` for when a successful result is available (it was **resolved**) and `catch(…)` for when something went wrong (it was **rejected**).

Using a promise looks like this:

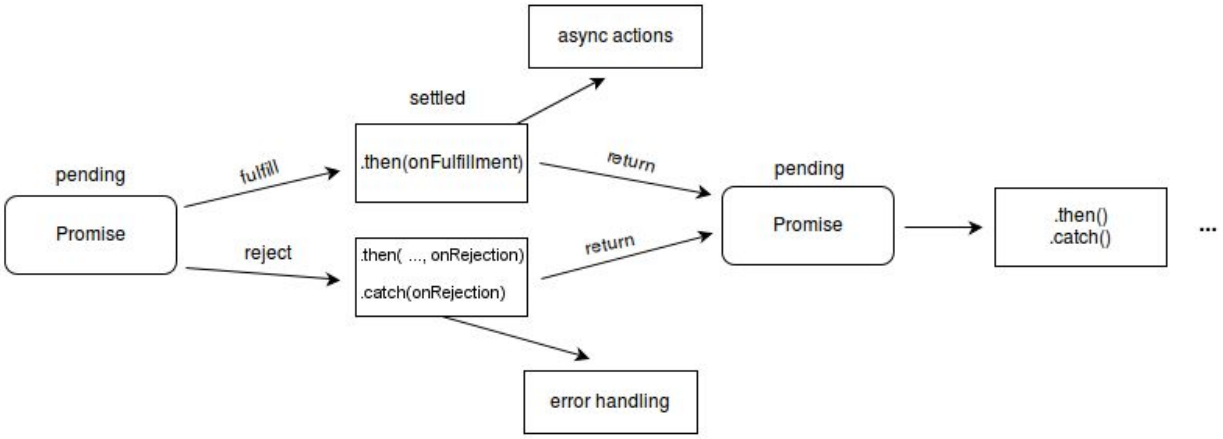
```
someAsyncOperation(someParams)
  .then(result => {
    // Do something with the result
  })
  .catch(error => {
    // Handle error
  });
```

One important side note here is that “someAsyncOperation(someParams)” is not a Promise itself but a function that returns a Promise.

The true power of promises is shown when you have several asynchronous operations that depend on each other, just like in the example above for the callback hell.
So let’s revisit the case where we have a request that depends on the result of another request. This time we are going to use a method called `fetch` that is similar to `request` but it uses promises instead of callbacks. This is also to point out that callbacks and promises are not interchangeable.

Chaining

As the Promise.prototype.then() and Promise.prototype.catch() methods return promises, they can be chained.



Using fetch, the code would instead look like this:

```
fetch('http://www.somepage.com')
  // Get the text from Response object
  // this is also async operation returning promise
  .then(response => response.text())
  // Now we can use the text to create new fetch promise
  .then(text => fetch(`http://www.somepage.com/${text}`))
  .then(response => {
    // Reponse being the result of the second request
    // Handle response
  })
  // If any of the above promises are rejected
  // this catch will handle the error
  .catch(error => {
    // Handle error
  });
```

Instead of nesting callbacks inside callbacks inside callbacks, you chain `.then()` calls together making it more readable and easier to follow. Every `.then()` should either return a new Promise or just a value or object which will be passed to the next `.then()` in the chain.

Another important thing to notice is that even though we are doing two different asynchronous requests we only have one `.catch()` where we handle our errors. That's because any error that occurs in the **Promise chain** will stop further execution and an error will end up in the nearest `.catch()` in the chain.

Async/Await

Async/Await is a language feature that is a part of the ES8 standard. It is the next step in the evolution of handling asynchronous operations in JavaScript. It gives you two new keywords to use in your code: `async` and `await`.

Async is for declaring that a function will handle asynchronous operations and **await** is used to declare that we want to "await" the result of an asynchronous operation inside a function that has the async keyword.

```
const fetchJson = async (path: string) => {
  const result = await fetch(`http://www.somepage.com/${path}`);
  const json = await result.json();
  return json;
};
```

You can only use the `await` keyword on "awaitable" functions. A function is "awaitable" if it returns a Promise (all functions marked `async` must return a Promise).

That basically means that this will work:

```
const myFetch = (url: string) =>
  new Promise<string>((resolve, reject) => {
    if (!url) {
      reject(`Url can't be empty.`);
      return;
    }
    resolve('Hi');
  });

const fetchJson = async (path: string) => {
  const result = await myFetch(`http://www.somepage.com/${path}`);
  return result;
};
```

Error handling with async/await

Inside the scope of an async function you can use try/catch for error handling and even though you await an asynchronous operation, any errors will end up in that catch block:

```
const fetchJson = async (path: string) => {
  try {
    const result = await myFetch(`http://www.somepage.com/${path}`);
    return result;
  } catch (error) {
    // Handle error
  }
};
```

As with promises and .catch() you also need only one try/catch to catch all errors from any number of awaited promises inside the block.

An important consideration regarding async/await

Async/await may make your asynchronous calls look more synchronous but it is still executed the same way as if it were using a callback or promise based API. The asynchronous I/O operations will still be processed in parallel and the code handling the responses in the async functions will not be executed until that asynchronous operation has a result. Also, even though you are using async/await you have to sooner or later resolve it as a Promise in the top level of your program. This is because async and await are just syntactic sugar for automatically creating, returning and resolving Promises.

Firebase and Firestore

Firebase is a collection of cloud services and tools managed by Google. While these services are primarily aimed at enterprise customers, many of them have also free plans that you can use in your hobby projects. In the next assignment you will be using Firestore, which is a cloud database from Firebase.

We will be loosely following the official Firebase setup guide and follow up guides for Firestore and Auth.

Creating a project

First you need to log in to firebase using a google account on their homepage. After you are logged in you can create a new project. Since we want to only use the Firestore, we won't be needing all the google analytics for our project.

Registering application

Console is the web interface of your firebase apps. Here you can find and manage all the tools and services. Now to use the services in our code we need to add a new app to the project. You can think of them as the frontend applications communicating with the project services. In our case it will be one ReactJS web application.
You can either add a new application in the overview or through the **Project settings** in the left panel. After we create the app, under **Firebase SDK snippet > Config** you can see a code snippet for `firebaseConfig` object that we will later use in our code.

Cloud Firestore

We can now select the **Cloud Firestore** in the Develop section in the left panel and create a new database for our app. By selecting **test mode** we can skip setting up request permissions for 30 days, which will speed up the setup. Next you need to select the location where the data will be physically stored, **eur3 (europe-west)** should be the best option.

Now that we have a database set up we need to create some collections. A collection represents a table of the database (although not so strict).

Authentication

Authentication is another service provided by Firebase. There are many supported options like Google, Facebook or even Github sign in, but for demonstration purposes we will be using plain old **Email/Password** sign in since it doesn't require any other setup (normally I would recommend against using this outdated method). In the console go to **Authentication** tab and enable the Email/Password sign in.

Implementation: Init

After setting everything up in the Firebase console, we can get to implementation. Start by adding the firebase package.

```
yarn add firebase
```

Now we need to initialize the firebase app. Add new file **utils/firebase.ts** where all Firebase related code will be placed. Copy `firebaseConfig` from the project settings page mentioned before.

```
import firebase from "firebase/app";
import "firebase/firestore";
import "firebase/auth";

// Your web app's Firebase configuration
const firebaseConfig = {
```

```
  apiKey: "API_KEY",
  authDomain: "PROJECT_ID.firebaseapp.com",
  databaseURL: "https://PROJECT_ID.firebaseio.com",
  projectId: "PROJECT_ID",
  storageBucket: "PROJECT_ID.appspot.com",
  messagingSenderId: "SENDER_ID",
  appId: "APP_ID"
};

// Initialize Firebase
firebase.initializeApp(firebaseConfig);
```

Implementation: Auth

Documentation on how to implement Authentication can be found here. Since provided examples are for general JS, we need to come up with a React solution. For the onAuthStateChanged which provides user login status, we can write a custom hook that saves this info into a state.

```
// Hook providing logged in user information
export const useLoggedInUser = () => {
  // Hold user info in state
  const [user, setUser] = useState<firebase.User>();

  // Setup onAuthStateChanged once when component is mounted
  useEffect(() => {
    firebase.auth().onAuthStateChanged(u => setUser(u ?? undefined));
  }, []);

  return user;
};

// Sign up handler
export const signUp = (email: string, password: string) =>
  firebase.auth().createUserWithEmailAndPassword(email, password);

// Sign in handler
export const signIn = (email: string, password: string) =>
```

```
  firebase.auth().signInWithEmailAndPassword(email, password);

// Sign out handler
export const signOut = () => firebase.auth().signOut();
```

Since firebase takes care of the user session now we no longer need the `useLoggedIn` custom hook from the previous lesson. Replace that with the new hook and also update everything else related (logout button etc.).

```
const App: FC = () => {
  // Styles
  const classes = useStyles();

  // Login state
  const user = useLoggedInUser();
```

We also need to update the Login component. Both `signUp` and `signIn` are async functions returning a Promise. We can demonstrate both approaches here. What's also nice is that on error Firebase returns an informative error message which we can directly show to the user.

```
<CardActions>
  <Button
    variant='text'
    size='large'
    color='primary'
    // Handling promise with async/await
    onClick={async () => {
      try {
        await signUp(user, password);
      } catch (err) {
        setError(err.message);
      }
    }}
  >
    Create account
  </Button>
  <Button
```

```
    variant='text'
    size='large'
    color='primary'
    // Handling promise with chained handlers
    onClick={() =>
      signIn(user, password).catch(err => setError(err.message))
    }
  >
    Login
  </Button>
</CardActions>
```

Implementation: Firestore

Last thing to implement remains the Firestore database. Similar to `firebase.auth()` all firestore related functions can be accessed through `firebase.firestore()`. Since we will be accessing data from firestore all over the application, we can simply export this object from **utils/firebase.ts** to make it easier to use.

```
// Firestore database
export const db = firebase.firestore();
```

To demonstrate how to work with data in firestore, we will implement a simple reviews system. On the **/about** page will be all reviews pulled from the **reviews** collection and then we will be able to submit a new review on **/review** page which will be accessible from **/about**. We can start by declaring types for the data we will be using. I've decided to put these types to **utils/firebase.ts** since we are using the `User` type from there.
Another step we can do is to export the specific collection we will be using to make it safer and also this way we can properly type it so it's easier to work with.

```
// Simplified user type for referencing users
export type User = Pick<firebase.User, 'uid' | 'email'>;

// Holds information about a review
export type Review = {
  by: User;
  stars: number;
  description?: string;
}
```

```
// We can simply cast this type to narrow our collection to Review type
// Safer way would be to use .withConverter() method
export const reviewsCollection = db.collection('reviews')
  as firebase.firestore.CollectionReference<Review>;
```

Now we can update the `About` page and add a `ReviewPreview` component that renders a single review (you can find code for these in the next assignment). In the `About` page we need to add code that fetches all reviews from firestore db. Since we want to subscribe to changes only once we wrap the function in useEffect that runs when component mounts and saves the response inside component's state.

```
import { reviewsCollection, Review } from '../utils/firebase';

const About: FC = () => {
  const [error, setError] = useState<string>();
  const [reviews, setReviews] = useState<Review[]>([]);
  useEffect(() => {
    // Call .onSnapshot() to listen to changes
    reviewsCollection.onSnapshot(
      snapshot => {
        // Access .docs property of snapshot
        setReviews(snapshot.docs.map(doc => doc.data()));
      },
      err => setError(err.message),
    );
  }, []);
```

The data from state can be used just like any other value.

```
{reviews.map((r, i) => (
  <Grid key={i} xs={12} sm={6} item>
    <ReviewPreview {...r} />
  </Grid>
))}
```

Lastly, onto the `Review` page. Again complete code can be found in the assignment, example below shows how to **add a new document** to a collection.

```
const handleSubmit = async () => {
  try {
    // Call .add() and pass new Record as an argument
    await reviewsCollection.add({
      stars,
      description,
      by: {
        uid: user?.uid ?? '',
        email: user?.email ?? '',
      },
    });
    // After awaiting previous call we can redirect back to /about page
    push('/about');
  } catch (err) {
    setError(err.what);
  }
};
```

There are many more methods for managing the data which you can find nicely explained in the firebase documentation.