




Chapter 2

# Asynchronous Programming



1

Asynchronous Programming

## Outline

- Overview of asynchronous programming patterns
- Callbacks
- ES6 Promises
- Async/await
- Exception Handling
- Event-driven programming using the EventEmitter

M.Romdhani, 2021

2

2

## Overview of JavaScript Asynchronous Programming Patterns

3

### What's asynchronous in a Node.js application ?

Asynchronous  
Programming

#### ■ In short, most things

#### ■ In long :

- Any network calls
- Timers (setTimeout, setInterval)
- Filesystem access
- Anything else that can be offloaded

M.Romdhani, 2021

4

4

## The Three Asynchronous Programming Patterns

Asynchronous  
Programming

### ■ Callbacks (The old way)

- In Javascript, callback function is a function that is passed as an argument to another function. The function that takes in a function/functions (callback function) as an argument/arguments is called higher-order function.

### ■ Promises:

- A Promise is an object that presents a value that may not be available yet, but will be resolved at some point in the future.
- It allows us to associate handlers with an asynchronous action that will eventual return a success value or a failure reason.

### ■ Async-Await (The newest way)

- Async-await is a way to write functions that contains promises/asynchronous code that looks synchronous.

M.Romdhani, 2021

5

5

## The Three Asynchronous Programming Patterns

Asynchronous  
Programming

### ■ Callbacks

```

GetUser(function(err,user) {
  GetProfile(user, function(err,profile){
    GetAccount(profile, function(err,acc){
      GetReport(acc, function(err, report){
        SendStatistics(report, function(e){
          //...
        })
      })
    })
  })
})

```

### ■ Promises

```

GetUser()
  .then (GetProfile)
  .then (GetAccount)
  .then (GetReport)
  .then (SendStatistics)
  .then(function(success){
    console.log(success);
  }).catch(function(e){
    console.log(e) ;
  })

```

M.Romdhani, 2021

### ■ Async/await

```

async function SendAsync(){
  let user = await GetUser(1);
  let profile = await GetProfile(user);
  let account = await GetAccount(profile);
  let report = await GetReport(account);
  let send = SendStatistics(report);

  console.log(send) ;
}

```

6

6

## Callbacks

7

### A callback is

Asynchronous  
Programming

- « just » a function
- In examples, usually anonymous functions (pass function(){} directly)
- According to some style guides, should be an arrow function ( ()=> {})
- Called when the async operation finishes

```
fs.readFile('file.md', 'utf-8', function (err, content) {  
  if (err) {  
    console.log('error happened during reading the file')  
    return console.log(err)  
  }  
})
```

M. Romdhani, 2021

8

8

## Node-style Callback

- Called with any error as the first argument/parameter, if there is no error, null is passed
- Called with any number of « output » data as the other arguments

```
(err, data) => { /* more logic */ }
```

## Node-style callback problems

### 1. Callback Hell

```
router.get('/path/to/something', (req, res, next) => {
  doSomething(arg1, arg2, (err, data1) => {
    if (err) return next(err);
    doAnotherThing(arg3, arg2, data1, (err2, data2) => {
      if (err2) return next(err2);
      somethingCompletelyDifferent(arg1, arg42, (err3, data3) => {
        if (err3) return next(err3);
        doSomethingElse((err4, data4) => {
          if (err4) return next(err4);
          res.render('page', { data });
        });
      });
    });
  });
});
```

### 2. Shadowing variables

- err in myAsyncFn is different than err in myOtherAsyncFn callback even having the same named

```
myAsyncFn((err, data) => {
  if (err) handle(err)
  myOtherAsyncFn(data, (err, secondData) => {
    fun(data, secondData, (err) => {
      if (err) handle(err)
    })
    fn(data, secondData, (err) => {
      if (err) handle(err)
    })
  })
})
```

## Node-style callback problems

Asynchronous  
Programming

### 3. Duplicated Error Handling

```
myAsyncFn((err, data) => {  
  if (err) handle(err)  
  myOtherAsyncFn(data, (err, secondData) => {  
    fun(data, secondData, (err) => {  
      if (err) handle(err)  
    })  
    fn(data, secondData, (err) => {  
      if (err) handle(err)  
    })  
  })  
})
```

M.Romdhani, 2021

11

11

**Promises**

12

## What is a Promise?

Asynchronous  
Programming

- A promise is an object that will return a value in future. Because of this “in future” thing, Promises are well suited for asynchronous JavaScript operations.

- Promises vs. Callbacks

- Promises reduces the amount of nested code
- Promises allow you to visualize the execution flow easily
- Promises let you handle all errors at once at the end of the chain.

- Using the Promise Object

```
const returnPromise = new Promise((resolve, reject) => {
  if(true){
    resolve("return true.");
  }else{
    reject('return false')
  }
});

returnPromise
  .then(value => value)
  .then(newVal => console.log(newVal))
  .catch(rejectVal => console.log(rejectVal))
```

M.Romdhani, 2021

13

13

## Converting Node-styled callbacks to promises

Asynchronous  
Programming

- Creating a raw promise

```
fs.readFile('some-file', (err, data) => {
  if (err) {
    // Handle error
  } else {
    // Do something with data
  }
})
```



```
const fs = require('fs')
function readFile(file, encoding) {
  return new Promise(function(resolve, reject) {
    fs.readFile(file, encoding, function(err, data) {
      if (err) return reject(err) // Rejects the promise with `err` as the reason
      resolve(data) // Fulfills the promise with `data` as the value
    })
  })
}

let promise = readFile('some-file')
promise.then(data => { /* Do something with data */ })
  .catch(err => { /* Handle error */ })
```

14

## Converting Node-styled callbacks to promises

Asynchronous  
Programming

### ■ Using the Node's `util.promisify` module (Starting from Node version 8)

```
fs.readFile('some-file', (err, data) => {
  if (err) {
    // Handle error
  } else {
    // Do something with data
  }
})
```



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

const readFilePromise = util.promisify(fs.readFile)

readFilePromise(filePath, options)
  .then(data => { /* Do something with data */ })
  .catch(err => { /* Handle error */ })
```

M.Romdhani, 2021

15

15

## Converting Node-styled callbacks to promises

Asynchronous  
Programming

### ■ Using the Node's `fs/promises` module (Starting from Node version 10)

```
fs.readFile('some-file', (err, data) => {
  if (err) {
    // Handle error
  } else {
    // Do something with data
  }
})
```



```
const fs = require("fs").promises;
readFile(filePath, options)
  .then(data => { /* Do something with data */ })
  .catch(err => { /* Handle error */ })
```

M.Romdhani, 2021

16

16

Asynchronous  
Programming

## Node Promise Style problems

- Lots of tightly scoped functions
- Very Verbose way of returning/passing multiple things

M.Romdhani, 2021

17

17

## Async/await

18

## Async/await

Asynchronous  
Programming

- Async/await provides a way to work with Promises easily

- Async functions

```
async function f1() {
  return 1;
}
async function f2() { //Equivalent to f1()
  return Promise.resolve(1);
}

f1().then(alert); // 1
```

- Await

- The keyword await makes JavaScript wait until that promise settles and returns its result. **Only used inside an async function.**

```
async function f() {
  let promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("done!"), 1000)
  });

  let result = await promise; // wait until the promise resolves (*)
  alert(result); // "done!"
}
```

M.Romdhani

f();

19

19

## Error handling in async/await

Asynchronous  
Programming

- If a promise resolves normally, then await promise returns the result. But in the case of a rejection, it throws the error, just as if there were a throw statement at that line
- We can catch that error using try..catch, the same way as a regular throw:

```
async function f() {
  try {
    let response = await fetch('http://no-such-url');
  } catch(err) {
    alert(err); // TypeError: failed to fetch
  }
}
```

- If we don't have try..catch, then the promise generated by the call of the async function f() becomes rejected. We can append .catch to handle it.

M.Romdhani, 2021

20

20

## Example : Rewriting a Promise using Async/await

Asynchronous  
Programming

- Given this Promise Code. Let's rewrite it using async/await

```
function loadJson(url) {
  return
    fetch(url)
      .then(response => {
        if (response.status == 200) {
          return response.json();
        } else {
          throw new Error(response.status);
        }
      });
}

loadJson('no-such-user.json')
  .catch(alert); // Error: 404
```

M.Romdhani, 2021

21

21

## Example : Rewriting a Promise using Async/await

Asynchronous  
Programming

- The solution

```
async function loadJson(url) { // (1)
  let response = await fetch(url); // (2)

  if (response.status == 200) {
    let json = await response.json(); // (3)
    return json;
  }

  throw new Error(response.status);
}

loadJson('no-such-user.json')
  .catch(alert); // Error: 404 (4)
```

- (1) The function loadJson becomes async.
- (2) All .then inside are replaced with await.
- (3) We can return response.json() instead of awaiting

M.Romdhani, 2021

22

22

## Exception Handling

23

## Exceptions in JavaScript

Asynchronous  
Programming

- A JavaScript code throws an exception when a particular statement generates an error. Instead of executing the next code statement, the JavaScript engine checks for the existence of exception handling code.
- If no exception handlers have been defined, the engine returns from the function that threw the exception. This process is repeated for each function on the call stack until it finds an exception handler.

M.Romdhani, 2021

24

24

## Types of Error Objects

Asynchronous  
Programming

- There are nine types of built-in error objects in JavaScript, which are the foundation for exception handling:
  - **Error** — represents generic exceptions. It is most often used for implementing user-defined exceptions.
  - **EvalError** — occurs when the eval() function is used improperly.
  - **RangeError** — used for errors that occur when a numeric variable or parameter is outside of its valid range.
  - **ReferenceError** — occurs when a non-existent variable is accessed.
  - **SyntaxError** — occurs when the JavaScript language rules are broken. For static-typed languages, this happens during compilation time. In JavaScript, it happens during runtime.
  - **TypeError** — occurs when a value does not match the expected type. Calling a non-existent object method is a common cause of this type of exception.
  - And many others

M.Romdhani, 2021

25

25

## Throwing Exceptions

Asynchronous  
Programming

- JavaScript allows developers to trigger exceptions via the throw statement.
 

```
if (denominator === 0) {
  throw new RangeError("Attempted division by zero");
}
```

  - Each of the built-in error objects takes an optional “message” parameter that gives a human-readable description of the error.

M.Romdhani, 2021

26

26

## Handling Exceptions

Asynchronous  
Programming

### ■ The “try” Clause

- JavaScript, quite similar to other programming languages, has the try, catch, finally statements, which gives us control over the flow of exceptions on our code.

- Here is a sample:

```
try {
  // a function that potentially throws an error
  someFunction();
} catch (err) {
  // this code handles exceptions
  console.log(e.message);
} finally {
  // this code will always be executed
  console.log('finally');
}
```

M.Romdhani, 2021

27

27

## Handling Exceptions in Asynchronous Code

Asynchronous  
Programming

### ■ async/await

- Let's define a standard function that just throws an error:

```
async function foo() {
  throw new Error();
}
```

- When an error is thrown in an async function, a rejected promise will be returned with the thrown error, equivalent to:

```
return Promise.Reject(new Error())
```

M.Romdhani, 2021

28

28

## Handling Exceptions in Asynchronous Code

Asynchronous  
Programming

### ■ Let's see what happens when `foo()` is invoked:

```
try {
  foo();
} catch(err) {
  // This block won't be reached.
} finally {
  // This block will be reached before the Promise is rejected.
}
```

Since `foo()` is `async`, it dispatches a Promise. The code does not wait for the `async` function, so there is no actual error to be caught at the moment. The `finally` block is executed and then the Promise rejects.

### ■ This can be handled by just adding the `await` keyword when invoking `foo()` and wrapping the code in an `async` function:

```
async function run() {
  try {
    await foo();
  } catch(err) {
    // This block will be reached now.
  } finally {
    // This block will be reached at the end.
  }
}
```

M.Romdhani, 2021

29

29

## Handling Exceptions in Asynchronous Code

Asynchronous  
Programming

### ■ Promises

#### ■ Let's define a function that throws an error outside of the Promise:

```
function foo(x) {
  if (typeof x !== 'number') {
    throw new TypeError('x is not a number');
  }
  return new Promise((resolve, reject) => {
    resolve(x);
  });
}
```

#### ■ Now let's invoke `foo` with a string instead of a number:

```
foo('test')
  .then(x => console.log(x))
  .catch(err => console.log(err));
```

- This will result in an `Uncaught TypeError: x is not a number` since the `catch` of the promise is not being able to handle an error that was thrown outside of the Promise.

M.Romdhani, 2021

30

30

## Handling Exceptions in Asynchronous Code

Asynchronous  
Programming

- To catch such errors, you need to use the standard try and catch clauses

```
try {
  foo('test')
  .then(x => console.log(x))
  .catch(err => console.log(err));
} catch(err) {
  // Now the error is handed
}
```

M.Romdhani, 2021

31

31

## Handling Exceptions in Asynchronous Code

Asynchronous  
Programming

- There are two main rules for working with the error-first callback approach:

1. The first argument of the callback is for the error object. If an error occurred, it will be returned by the first err argument. If no error occurred, err will be set to null.
2. The second argument of the callback is the response data.

```
function asyncFoo(x, callback) {
  // Some async code...
}

asyncFoo('testParam', (err, result) => {
  if (err) {
    // Handle error.
  }
  // Do some other work.
});
```

If there is an err object, it's better not to touch or rely on the result parameter.

M.Romdhani, 2021

32

32

Asynchronous  
Programming

M.Romdhani, 202133

33

Event-driven programming using the  
EventEmitter

34

## Events in Node.js

Asynchronous  
Programming

- **Node.js is an asynchronous event-driven JavaScript runtime. It has an event-driven architecture which can perform asynchronous tasks.**
- **Node.js has 'events' module which emits named events that can cause corresponding functions or callbacks to be called.**
  - Functions(Callbacks) listen or subscribe to a particular event to occur and when that event triggers, all the callbacks subscribed to that event are fired one by one in order to which they were registered.
  - All objects that emit events are instances of the **EventEmitter** class. The event can be emitted or listen to an event with the help of EventEmitter.

M.Romdhani, 2021

35

35

## EventEmitter

Asynchronous  
Programming

- **The EventEmitter class is at the core of Node asynchronous event-driven architecture.**
- **The EventEmitter is a module that facilitates communication/interaction between objects in Node.**
- **The concept is quite simple: emitter objects emit named events that cause previously registered listeners to be called. So, an emitter object basically has two main features:**
  - Emitting name events.
  - Registering and unregistering listener functions.
- **It's kind of like a **pub/sub** or **observer** design pattern.**

M.Romdhani, 2021

36

36

## EventEmitter :an illustration

Asynchronous  
Programming

### ■ A simple example of the use of EventEmitter

```
// Importing events
const EventEmitter = require('events');

// Initializing event emitter instances
var eventEmitter = new EventEmitter();

// Registering to myEvent
eventEmitter.on('myEvent', (msg) => {
  console.log(msg);
});

// Triggering myEvent
eventEmitter.emit('myEvent', "First event");
```

### ■ Output:

First event

M.Romdhani, 2021

37

37

## EventEmitter main properties and methods

Asynchronous  
Programming

| EventEmitter methods                            | Description   |
|---|---|
| <code>addListener(event, listener)</code>       | Adds a listener to the end of the listeners array for the specified event. No checks are made to see if the listener has already been added.  |
| <code>on(event, listener)</code>                | It can also be called as an alias of <code>emitter.addListener()</code>   |
| <code>once(event, listener)</code>              | Adds a one-time listener for the event. This listener is invoked only the next time the event is fired, after which it is removed.            |
| <code>emit(event, [arg1], [arg2], [...])</code> | Raise the specified events with the supplied arguments.   |
| <code>removeListener(event, listener)</code>    | Removes a listener from the listener array for the specified event. Caution: changes array indices in the listener array behind the listener. |
| <code>removeAllListeners([event])</code>        | Removes all listeners, or those of the specified event.   |

M.Romdhani, 2021

38

38