

# ES2017 Async Functions

## Goals

- Explain what the **async** keyword does
- Explain what the **await** keyword does
- Manage asynchronous code using **async** / **await**
- Refactor code using other patterns (e.g. callbacks, promises) to **async** / **await**

## The async keyword

### async Overview

- The **async** keyword is part of ES2017
- You can declare any function in JavaScript as **async**
- **async** functions always return promises!
- Inside of an **async** function, you can write code that looks synchronous, even if it isn't (*more on this later*)

### Our First async Example

demo/async-examples.js

```
// not async, obvs
function friendlyFn() {
  return "hello!!! omg so nice to meet you!"
}

friendlyFn();
// "hello!!! omg so nice to meet you!"
```

demo/async-examples.js

```
// omg async
async function asyncFriendlyFn() {
  return "hello!!! omg so nice to meet you!"
}

asyncFriendlyFn();
// Promise {<resolved>: "hello!!! omg so nice to meet you!"}

asyncFriendlyFn().then(msg => console.log(msg));
// "hello!!! omg so nice to meet you!"
```

### Similar Behavior, Using Promises

demo/async-examples.js

```
// omg async
async function asyncFriendlyFn() {
  return "hello!!! omg so nice to meet you!"
}

asyncFriendlyFn();
// Promise {<resolved>: "hello!!! omg so nice to meet you!"}

asyncFriendlyFn().then(msg => console.log(msg));
// "hello!!! omg so nice to meet you!"
```

demo/async-examples.js

```
// similar behavior to async
function friendlyFnPromise() {
  return Promise.resolve("hello!!! omg so nice to meet you!")
}

friendlyFnPromise();
// Promise {<resolved>: "hello!!! omg so nice to meet you!"}

friendlyFnPromise().then(msg => console.log(msg));
// "hello!!! omg so nice to meet you!"
```

## What about Rejection?

- Inside of **async** functions, the return value is wrapped in a resolved promise.
- If you want to reject instead of resolve, simply throw an error inside of the **async** function!

## Rejection Example

demo/async-examples.js

```
async function oops() {
  throw "you shouldn't have invoked me!!"
}

oops();
// Promise {<rejected>: "you shouldn't have invoked me!!"}

oops().catch(err => console.log(err));
// "you shouldn't have invoked me!!"
```

# The **await** keyword

## **await** Overview

- Inside of an **async** function, we can use the **await** keyword
- **await** pauses the execution of the **async** function
- Can **await** any async operation returning a promise (eg other **async** functions!)
- The **await** keyword waits for promise to resolve & extracts its resolved value
- It then resumes the **async** function's execution
- Think of the **await** keyword like a pause button

## Using **await**

demo/await-examples.js

```
async function getStarWarsData() {  
  console.log("starting!");  
  let movieData = await $.getJSON(  
    "https://swapi.dev/api/films/");  
  // these lines do NOT run until the promise is resolved!  
  console.log("all done!");  
  console.log(movieData);  
}  
  
getStarWarsData();
```

No **.then** or callback necessary!

## Using **async** / **await**

### Object **async**

- We can also place **async** functions as methods inside objects!
- Make sure to prefix the name of the function with the **async** keyword

demo/await-examples.js

```
let starWars = {  
  genre: "sci-fi",  
  async logMovieData() {  
    let url = "https://swapi.dev/api/films/";  
    let movieData = await $.getJSON(url);  
    console.log(movieData.results);  
  }  
};  
  
starWars.logMovieData();
```

**Note:** Async functions and promises

Remember that **async** functions **always** return promises. In the example above, `starWars.logMovieData()` returns a resolved promise with a value of **undefined**, since the function itself has no return value.

If you wanted to do something with the movie data besides **console.log** it, you'd need to **return** the data from the **async** function, and then chain a **.then** on the end of `starWars.logMovieData()`.

The moral here is that using **async** / **await** doesn't absolve you from your responsibility to learn about promises. If anything, it's the opposite: if you don't understand promises well, it will be harder for you to debug code using **async** or **await**.

## Class async

- We can also make **async** instance methods with ES2015 **class** syntax

demo/pokemon.js

```
class Pokemon {
  constructor(id) {
    this.id = id;
  }

  async logName() {
    let url = `https://pokeapi.co/api/v2/pokemon/${this.id}/`;
    let response = await $.getJSON(url);
    console.log(response.name);
  }
}

let pokemon = new Pokemon(10);

pokemon.logName();
// "caterpie"
```

## Handling errors

- If a promise is rejected using **await**, an error will be thrown.
- We can use a **try/catch** statement to handle errors!

demo/await-examples.js

```
async function getUser(user) {
  try {
    let url = `https://api.github.com/users/${user}`;
    let response = await $.getJSON(url);
    console.log(`${response.name}: ${response.bio}`);
  } catch (e) {
    console.log("User does not exist!");
  }
}
```

demo/await-examples.js

```
getUser("mmmaaattttt");  
// Matt Lane: Co-founder at @rithmschool.  
// Teacher of how the internet works.  
// Check us out at rithmschool.com  
  
getUser("nopenouserhereomggoaway");  
// User does not exist!
```

## Refactoring Async Code

### Callbacks Revisited

demo/refactoring.js

```
let baseURL = "https://pokeapi.co/api/v2/pokemon";  
  
$.getJSON(`${baseURL}/1/`, p1 => {  
  console.log(`The first pokemon is ${p1.name}`);  
  $.getJSON(`${baseURL}/2/`, p2 => {  
    console.log(`The second pokemon is ${p2.name}`);  
    $.getJSON(`${baseURL}/3/`, p3 => {  
      console.log(`The third pokemon is ${p3.name}`);  
    });  
  });  
});
```

### Promises Revisited

demo/refactoring.js

```
let baseURL = "https://pokeapi.co/api/v2/pokemon";  
  
$.getJSON(`${baseURL}/1/`)  
  .then(p1 => {  
    console.log(`The first pokemon is ${p1.name}`);  
    return $.getJSON(`${baseURL}/2/`);  
  })  
  .then(p2 => {  
    console.log(`The second pokemon is ${p2.name}`);  
    return $.getJSON(`${baseURL}/3/`);  
  })  
  .then(p3 => {  
    console.log(`The third pokemon is ${p3.name}`);  
    return $.getJSON(`${baseURL}/3/`);  
  });
```

### async / await Revisited

demo/refactoring.js

```
async function catchSomeOfEm() {
  let baseURL = "https://pokeapi.co/api/v2/pokemon";
  let p1 = await $.getJSON(`${baseURL}/1/`);
  let p2 = await $.getJSON(`${baseURL}/2/`);
  let p3 = await $.getJSON(`${baseURL}/3/`);

  console.log(`The first pokemon is ${p1.name}`);
  console.log(`The second pokemon is ${p2.name}`);
  console.log(`The third pokemon is ${p3.name}`);
}

catchSomeOfEm();
```

- Above we are making three requests sequentially.
- Each request must wait for the previous request before starting.
- But the requests are totally independent!
- This can really slow down our applications... so how do we fix it?

## Parallel Requests using *async / await*

demo/refactoring.js

```
async function catchSomeOfEmParallel() {
  let baseURL = "https://pokeapi.co/api/v2/pokemon";
  let p1Promise = $.getJSON(`${baseURL}/1/`);
  let p2Promise = $.getJSON(`${baseURL}/2/`);
  let p3Promise = $.getJSON(`${baseURL}/3/`);

  let p1 = await p1Promise;
  let p2 = await p2Promise;
  let p3 = await p3Promise;

  console.log(`The first pokemon is ${p1.name}`);
  console.log(`The second pokemon is ${p2.name}`);
  console.log(`The third pokemon is ${p3.name}`);
}

catchSomeOfEmParallel();
```

Start the requests in parallel rather than in sequence!

## Another option with *Promise.all*

demo/refactoring.js

```
async function catchSomeOfEmParallel2() {
  let baseURL = "https://pokeapi.co/api/v2/pokemon";
  let pokemon = await Promise.all([
    $.getJSON(`${baseURL}/1/`),
  ])
```

```
$.getJSON(`${baseUrl}/2/`),
$.getJSON(`${baseUrl}/3/`)
]);

console.log(`The first pokemon is ${pokemon[0].name}`);
console.log(`The second pokemon is ${pokemon[1].name}`);
console.log(`The third pokemon is ${pokemon[2].name}`);
}

catchSomeOfEmParallel2();
```

- We can use ***Promise.all*** to await multiple resolved promises
- Here we are simply waiting for an array of promises to resolve!

## Looking Ahead

### Coming Up

- Practice with ***async / await***
- An introduction to Node.js!