async/await

Two types of asynchrony

We have been working with two broad types of asynchronous events:

1. **Inherently asynchronous events**
   - Example: `addEventListener('click')`. There is no such thing as a synchronous click event.

2. **Annoyingly asynchronous events**
   - Example: **`fetch()`**. This function would be easier to use if it were synchronous, but for performance reasons it's asynchronous

Asynchronous `fetch()`

The usual asynchronous `fetch()` looks like this:

```
function onJsonReady(json) {
  console.log(json);
}

function onResponse(response) {
  return response.json();
}

fetch('albums.json')
    .then(onResponse)
    .then(onJsonReady);
```

Synchronous `fetch()`?

A hypothetical synchronous `fetch()` might look like this:

```
// THIS CODE DOESN'T WORK
const response = fetch('albums.json');
const json = response.json();
console.log(json);
```

This is a lot cleaner code-wise!!

However, a synchronous `fetch()` would freeze the browser as the resource was downloading, which would be terrible for performance.

async / await

What if we could get the best of both worlds?

- Synchronous-*looking* code
- That actually ran asynchronously

```
// THIS CODE DOESN'T WORK
const response = fetch('albums.json');
const json = response.json();
console.log(json);
```

async / await

What if we could get the best of both worlds?

- Synchronous-*looking* code
- That actually ran asynchronously

```
// But this code does work:
async function loadJson() {
  const response = await fetch('albums.json');
  const json = await response.json();
  console.log(json);
}
loadJson();
```

async / await

What if we could get the best of both worlds?

- Synchronous-*looking* code
- That actually ran asynchronously

```
// But this code does work:
async function loadJson() {
  const response = await fetch('albums.json');
  const json = await response.json();
  console.log(json);
}
loadJson();
```

???

async functions

A function marked `async` has the following qualities:

- It will behave more or less like a normal function if you don't put `await` expression in it.

- An `await` expression is of form:
  - `await` ***promise***

async functions

A function marked `async` has the following qualities:

- If there is an `await` expression, **the execution of the function will pause** until the `Promise` in the `await` expression is resolved.
  - Note: The browser is not blocked; it will continue executing JavaScript as the async function is paused.
- Then when the `Promise` is resolved, the execution of the function continues.
- The `await` expression evaluates to the resolved value of the `Promise`.

```
function onJsonReady(json) {
  console.log(json);
}
function onResponse(response) {
  return response.json();
}
fetch('albums.json')
    .then(onResponse)
    .then(onJsonReady);
```

The methods in purple return Promises.

```
async function loadJson() {
  const response = await fetch('albums.json');
  const json = await response.json();
  console.log(json);
}
loadJson();
```

```
function onJsonReady(json) {
  console.log(json);
}
function onResponse(response) {
  return response.json();
}
fetch('albums.json')
    .then(onResponse)
    .then(onJsonReady);
```

The variables in blue are the values that the Promises "resolve to".

```
async function loadJson() {
  const response = await fetch('albums.json');
  const json = await response.json();
  console.log(json);
}
loadJson();
```

async functions

```
async function loadJson() {
  const response = await fetch('albums.json');
  const json = await response.json();
  console.log(json);
}
loadJson();
```

async functions

```
async function loadJson() {
  const response = await fetch('albums.json');
  const json = await response.json();
  console.log(json);
}
loadJson();
```

async functions

```
 async function loadJson() {
➡  const response = await fetch('albums.json');
   const json = await response.json();
   console.log(json);
 }
➡loadJson();
```

Since we've reached an `await` statement, two things happen:

1. `fetch('albums.json');` runs
2. The execution of the `loadJson` function is paused here until `fetch('albums.json');` has completed.

async functions

```
async function loadJson() {
  const response = await fetch('albums.json');
  const json = await response.json();
  console.log(json);
}
loadJson();
console.log('after loadJson');
```

At the point, the JavaScript engine will return from `loadJson()` and it will continue executing where it left off.

async functions

```
async function loadJson() {
  const response = await fetch('albums.json');
  const json = await response.json();
  console.log(json);
}
loadJson();
console.log('after loadJson');
```

async functions

```
async function loadJson() {
  const response = await fetch('albums.json');
  const json = await response.json();
  console.log(json);
}
loadJson();
console.log('after loadJson');
```

async functions

```
async function loadJson() {
  const response = await fetch('albums.json');
  const json = await response.json();
  console.log(json);
}
loadJson();
console.log('after loadJson');
```

async functions

```
async function loadJson() {
  const response = await fetch('albums.json');
  const json = await response.json();
  console.log(json);
}
loadJson();
console.log('after loadJson');
```

If there are other events, like if a button was clicked and we had a event handler for it, JavaScript will continue executing those events.

async functions

```
async function loadJson() {
  const response = await fetch('albums.json');
  const json = await response.json();
  console.log(json);
}
loadJson();
console.log('after loadJson');
```

When the `fetch()` completes, the JavaScript engine will resume execution of `loadJson()`.

Recall: `fetch()` resolution

```
function onResponse(response) {
  return response.json();
}
fetch('albums.json')
    .then(onResponse)
```

Normally when `fetch()` finishes, it executes the `onResponse` callback, whose parameter will be response.

In Promise-speak:

- The return value of `fetch()` is a `Promise` that **resolves to** the `response` object.

async functions

```
async function loadJson() {
  const response = await fetch('albums.json');
  const json = await response.json();
  console.log(json);
}
loadJson();
console.log('after loadJson');
```

The value of the `await` expression is the value that the `Promise` resolves to, in this case `response`.

async functions

```
async function loadJson() {
  const response = await fetch('albums.json');
➡ const json = await response.json();
  console.log(json);
}
loadJson();
console.log('after loadJson');
```

async functions

```
async function loadJson() {
  const response = await fetch('albums.json');
  const json = await response.json();
  console.log(json);
}
loadJson();
```

Since we've reached an `await` statement, two things happen:

1. `response.json();` runs
2. The execution of the `loadJson` function is paused here until `response.json();` has completed.

async functions

```
async function loadJson() {
  const response = await fetch('albums.json');
  const json = await response.json();
  console.log(json);
}
loadJson();
```

If there are other events, like if a button was clicked and we had a event handler for it, JavaScript will continue executing those events.

async functions

```
async function loadJson() {
  const response = await fetch('albums.json');
  const json = await response.json();
  console.log(json);
}
loadJson();
```

async functions

```
async function loadJson() {
  const response = await fetch('albums.json');
  const json = await response.json();
  console.log(json);
}
loadJson();
```

When the `response.json()` completes, the JavaScript engine will resume execution of `loadJson()`.

Recall: `json()` resolution

```
function onJsonReady(jsObj) {
  console.log(jsObj);
}
function onResponse(response) {
  return response.json();
}
fetch('albums.json')
    .then(onResponse)
    .then(onJsonReady);
```

Normally when `json()` finishes, it executes the `onJsonReady` callback, whose parameter will be **`jsObj`**.

In Promise-speak:

- The return value of `json()` is a `Promise` that **resolves to** the **`jsObj`** object.

async functions

```
async function loadJson() {
  const response = await fetch('albums.json');
➡ const json = await response.json();
  console.log(json);
}
loadJson();
```

The value of the `await` expression is the value that the `Promise` resolves to, in this case `json`.

async functions

```
async function loadJson() {
  const response = await fetch('albums.json');
  const json = await response.json();
  console.log(json);
}
loadJson();
```

async functions

```
async function loadJson() {
  const response = await fetch('albums.json');
  const json = await response.json();
  console.log(json);
}
loadJson();
```

async functions

```
async function loadJson() {
  const response = await fetch('albums.json');
  const json = await response.json();
  console.log(json);
}
loadJson();
```

Note that the JS execution does ***not*** return back to the call site, since the JS execution already did that when we saw the first `await` expression.

Returning from async

Q: What happens if we return a value from an async function?

```
async function loadJson() {
  const response = await fetch('albums.json');
  const json = await response.json();
  console.log(json);
  return true;
}
loadJson();
```

Returning from async

A: async functions must always return a `Promise`.

```
async function loadJson() {
  const response = await fetch('albums.json');
  const json = await response.json();
  console.log(json);
  return true;
}
loadJson();
```

If you return a value that is **not** a `Promise` (such as `true`), then the JavaScript engine will automatically wrap the value in a `Promise` that resolves to the value you returned.

Returning from async

```
function loadJsonDone(value) {
  console.log('loadJson complete!');
  // Prints "value: true"
  console.log('value: ' + value);
}

async function loadJson() {
  const response = await fetch('albums.json');
  const json = await response.json();
  console.log(json);
  return true;
}
loadJson().then(loadJsonDone)
console.log('after loadJson');
```

More `async`

- Constructors cannot be marked `async`

- But you can pass `async` functions as parameters to:
  - `addEventListener` (Browser)
  - on (NodeJS)
  - `get/put/delete/`etc (ExpressJS)
  - Wherever you can pass a function as a parameter

`async` / `await` availability

Browsers:

- All major browsers support `async` /`await`, but it's pretty recent: Edge + Safari support came ~1 month ago

NodeJS:

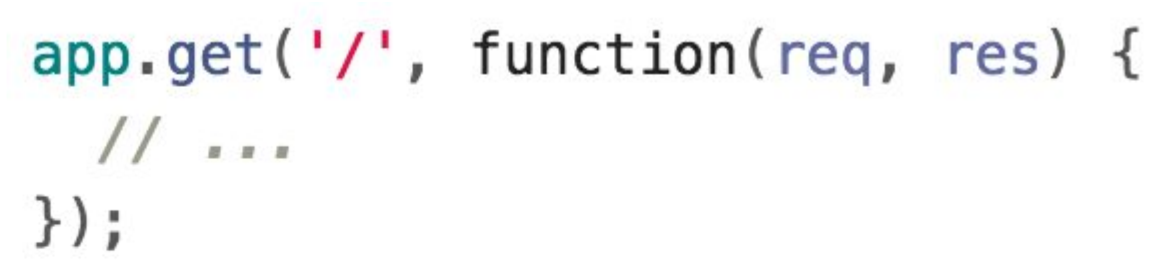
- `async` /`await` available in v7.5+... which is why we need you to install v7 instead of v6

(FYI, underneath the covers `async`/`await` is implemented by generator functions, another functional programming construct)

Async and Express routes

Recall: ExpressJS routes

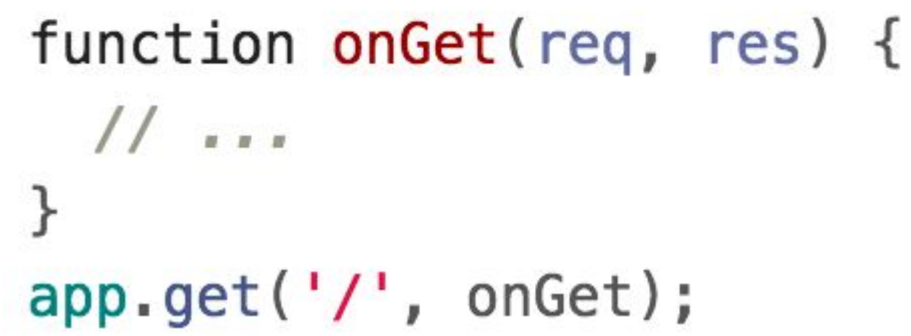
We've been seeing ExpressJS routes that look like this, with an anonymous function parameter:



```
app.get('/', function(req, res) {
  // ...
});
```

ExpressJS routes

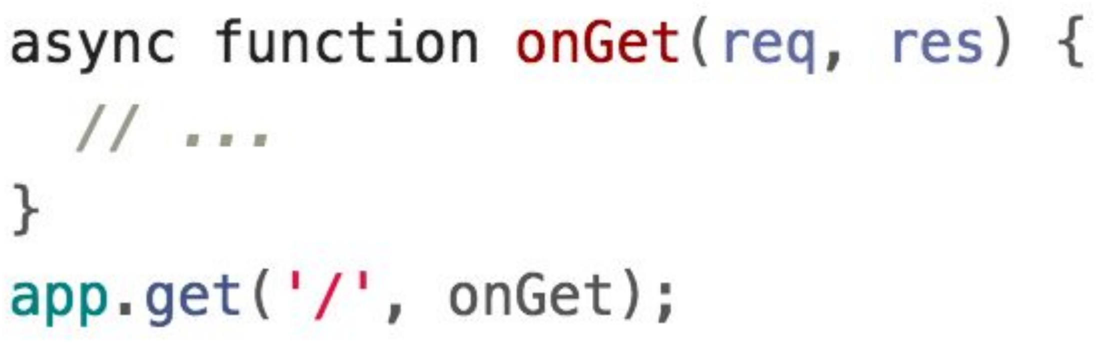
Of course, they can also be written like this, with a named function parameter:



```
function onGet(req, res) {
  // ...
}
app.get('/', onGet);
```

ExpressJS routes

This callBack named function behaviour can be transformed using async.



```
async function onGet(req, res) {
  // ...
}
app.get('/', onGet);
```

Which works about the same as a non-async function, except when you write an `await` inside of it.

`async` / `await` availability

Browsers:

- All major browsers support `async` /`await`, but it's pretty recent: Edge + Safari support came ~1 month ago

NodeJS:

- `async` /`await` available in v7.5+... which is why we need you to install v7 instead of v6

(FYI, underneath the covers `async`/`await` is implemented by generator functions, another functional programming construct)