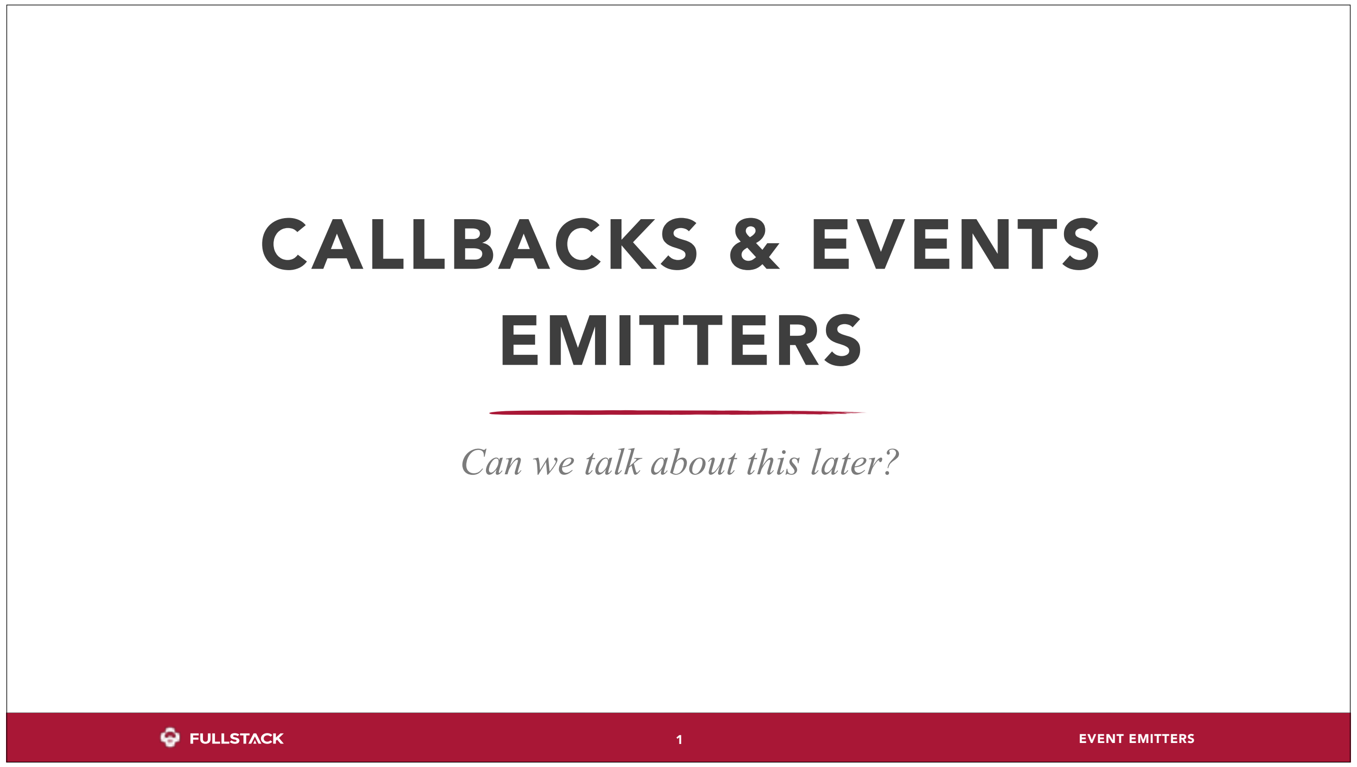
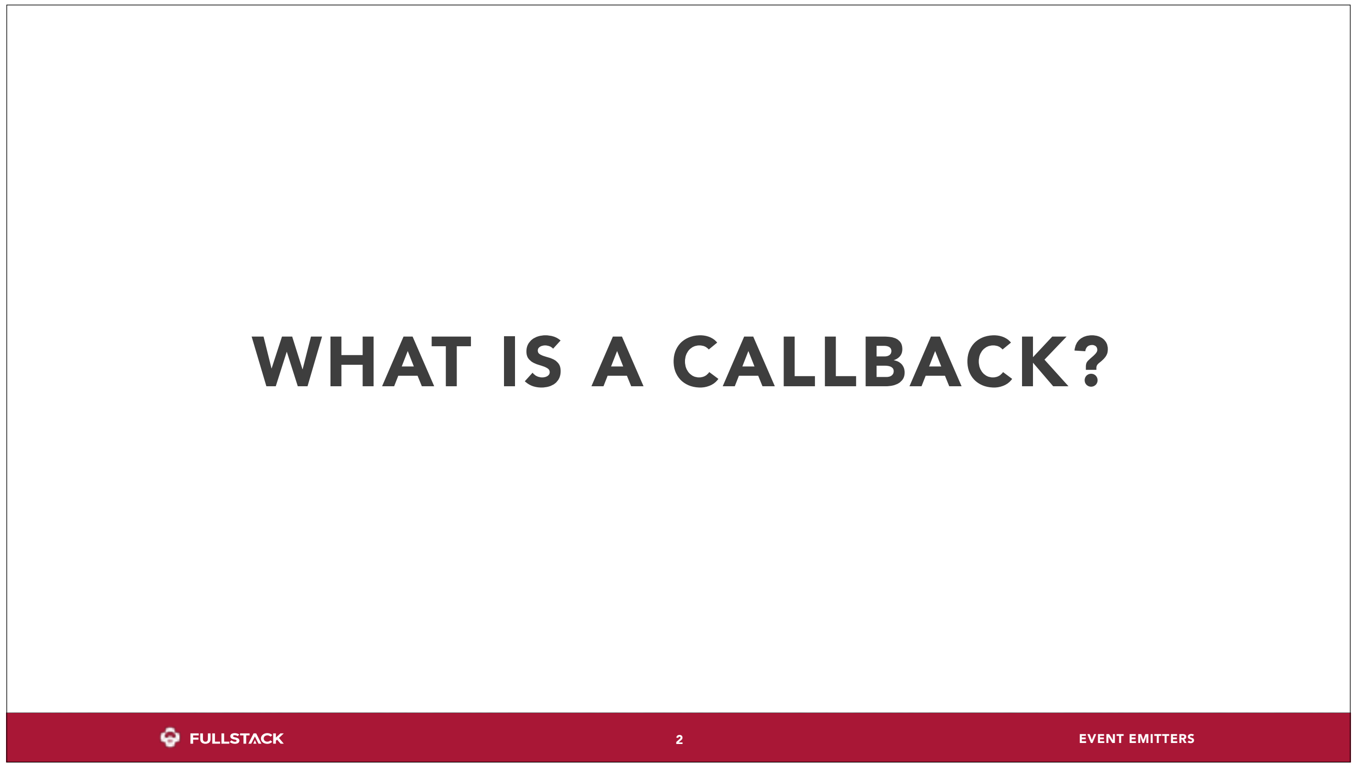


CALLBACKS & EVENTS EMITTERS

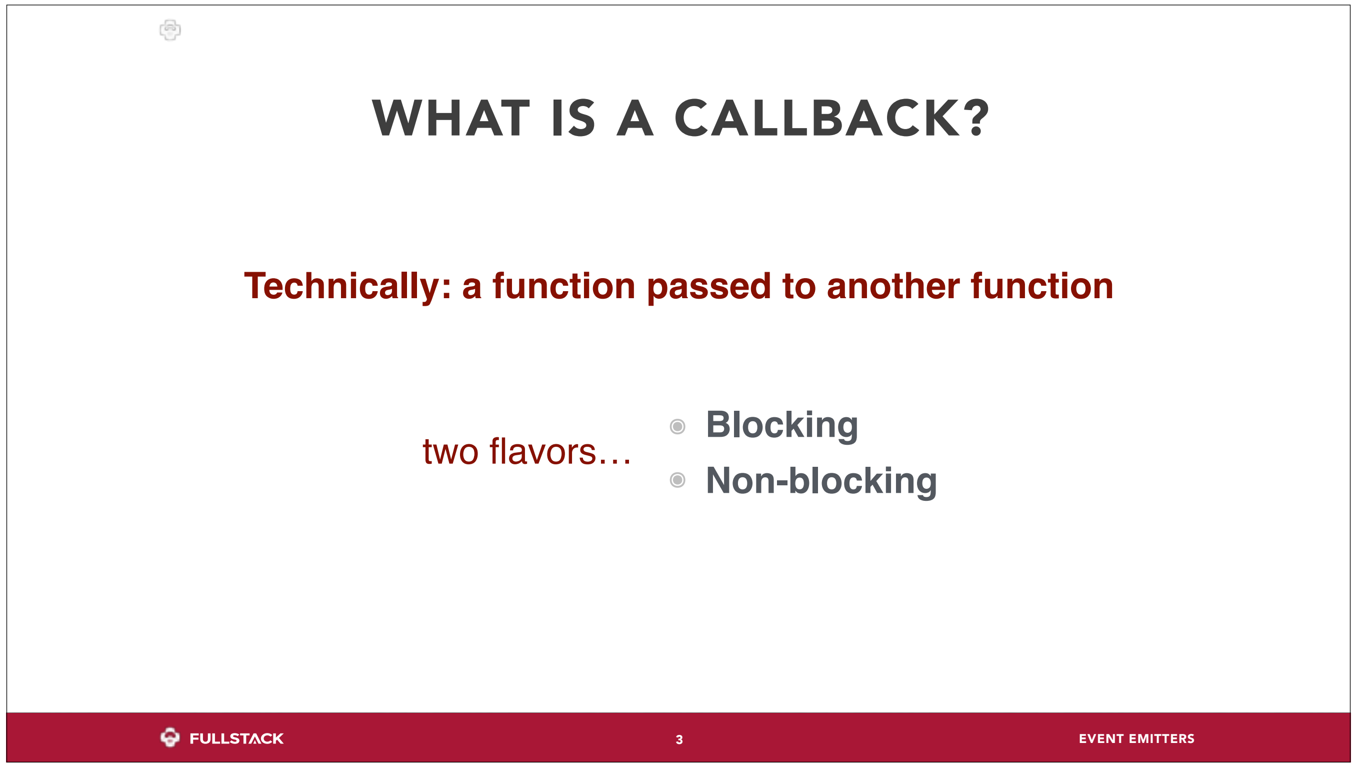
Can we talk about this later?

On Thursday we learned about Promises, and how they are an object that represents an eventual result of an async operation, such as querying our database. We learned how they help us have async code that is more readable, powerful, and portable than using what we were calling ‘error first’ callback such as in fs.readFile. For the next 30 minutes, we’re going to discuss all the ways that we handle actions that are deferred, or will happen in the future, and not in the order that we write the code.



WHAT IS A CALLBACK?

We're about to learn more about the coding pattern that saves us from callback hell - Promises. But first, let's talk a bit more about callbacks.



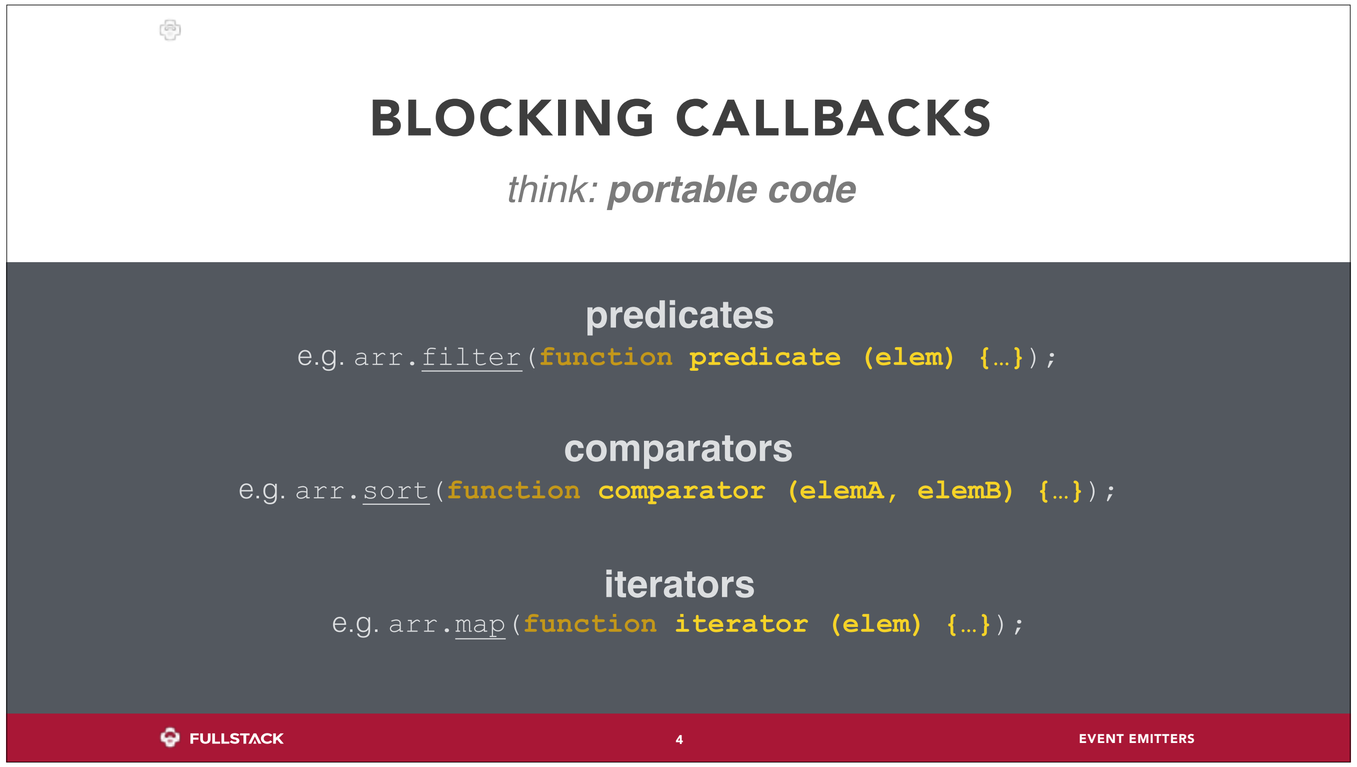
WHAT IS A CALLBACK?

Technically: a function passed to another function

two flavors…

- **Blocking**
- **Non-blocking**

The definition is simple, and we use callbacks for so many things. Sometimes callbacks are part of an operation that is blocking the call stack, and sometimes they are instead non-blocking, they are the function we give to an asynchronous function saying ‘when you’re done, run this function’



BLOCKING CALLBACKS

*think: **portable code***

predicates

e.g. `arr.filter(function predicate (elem) {…});`

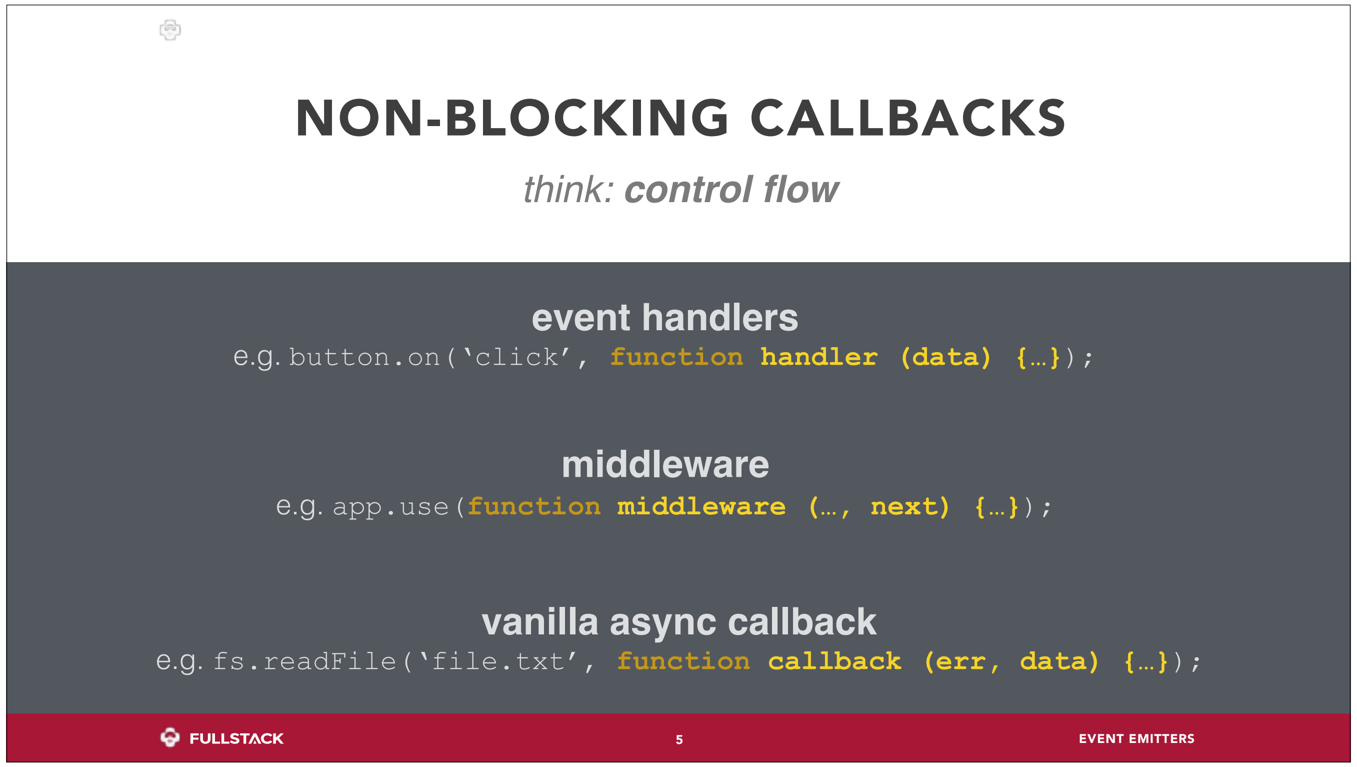
comparators

e.g. `arr.sort(function comparator (elemA, elemB) {…});`

iterators

e.g. `arr.map(function iterator (elem) {…});`

These callbacks can be thought of as building blocks of code. The function we are passing our callbacks to is going to decide how to use our callback. usually, we are controlling the behavior of the function/method in some way, or possibly decorating / augmenting our callback.



NON-BLOCKING CALLBACKS

think: ***control flow***

event handlers

e.g. `button.on(‘click’, function handler (data) {…});`

middleware

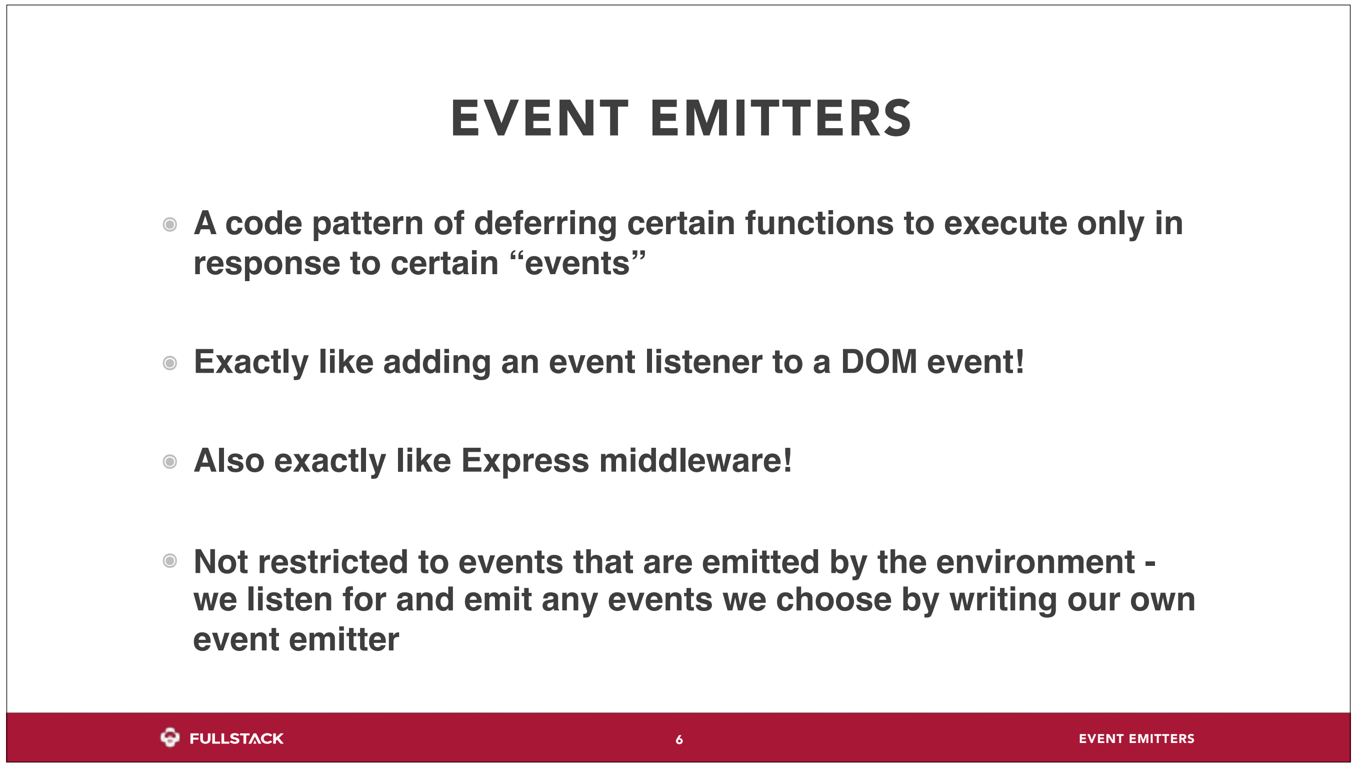
e.g. `app.use(function middleware (…, next) {…});`

vanilla async callback

e.g. `fs.readFile(‘file.txt’, function callback (err, data) {…});`

“At some point in the future, execute this function”

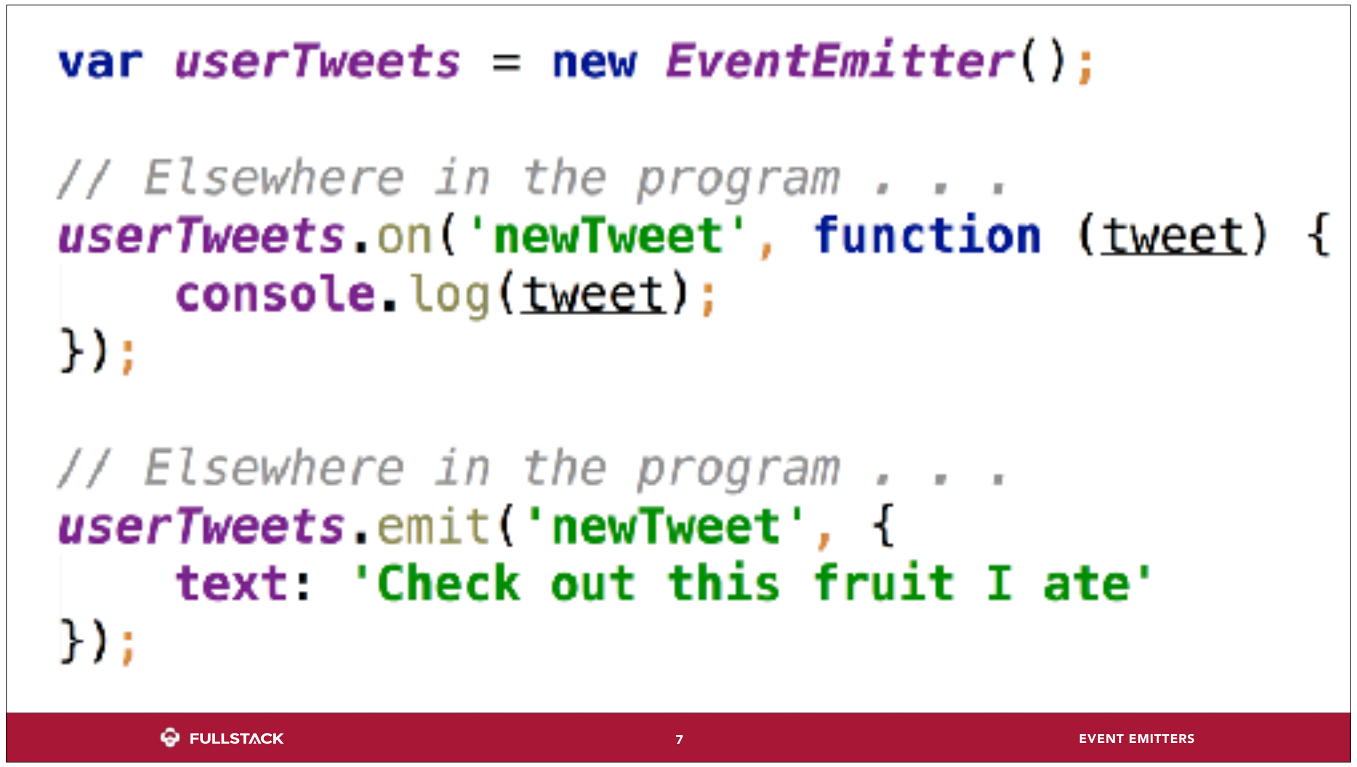
In all these examples, we are saying ‘hold on to this function, and when ____ happens, then execute it’



EVENT EMITTERS

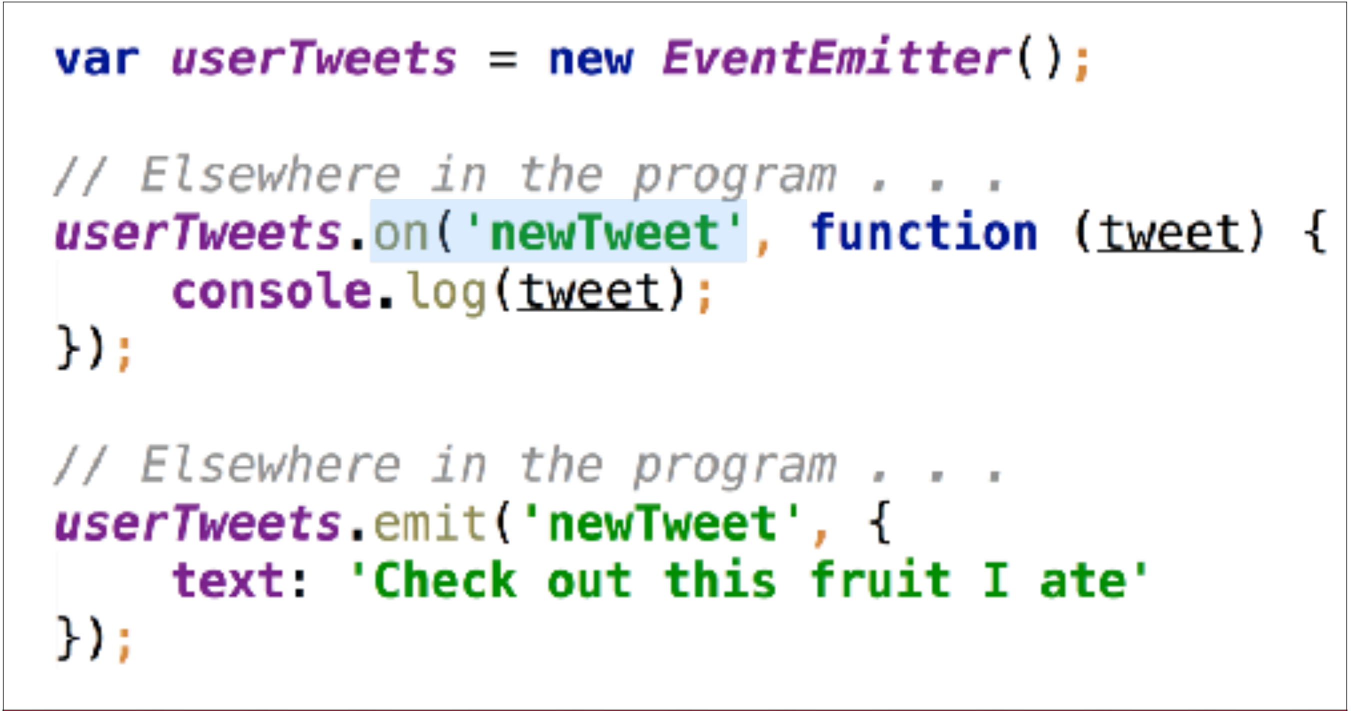
- A code pattern of deferring certain functions to execute only in response to certain “events”
- Exactly like adding an event listener to a DOM event!
- Also exactly like Express middleware!
- Not restricted to events that are emitted by the environment - we listen for and emit any events we choose by writing our own event emitter

Though we associate ‘event emitters’ with the first example from the previous slide, in a way, ALL async functions are event emitters. When an async operation being run by another process is done, it will emit an event, which triggers the execution of our callback.

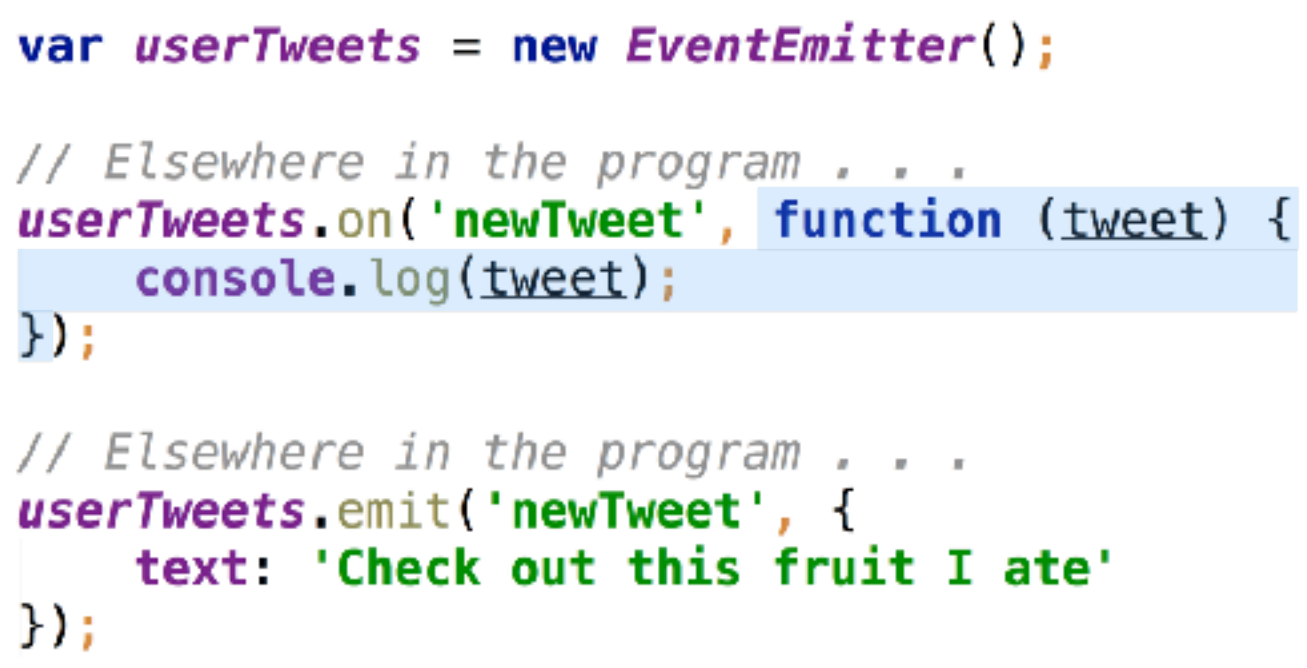


```
var userTweets = new EventEmitter();

// Elsewhere in the program . . .
userTweets.on('newTweet', function (tweet) {
    console.log(tweet);
});

// Elsewhere in the program . . .
userTweets.emit('newTweet', {
    text: 'Check out this fruit I ate'
});
```

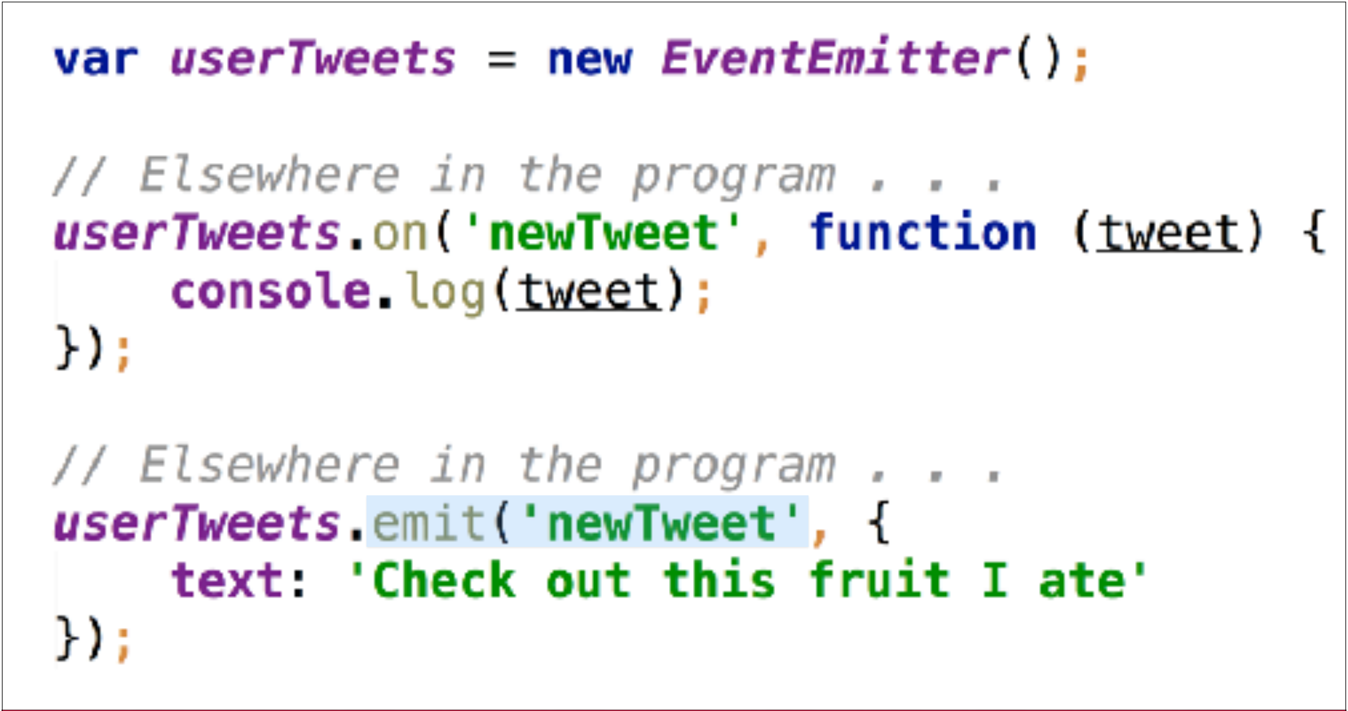
Let’s imagine that we have a twitter clone app. When there is a new tweet added to the database, we want the rest of the program to know about it! So, we register an event ‘newTweet’. When ‘newTweet’ is emitted with some payload, our callback will be invoked with that payload as the argument.



```
var userTweets = new EventEmitter();

// Elsewhere in the program . . .
userTweets.on('newTweet', function (tweet) {
    console.log(tweet);
});

// Elsewhere in the program . . .
userTweets.emit('newTweet', {
    text: 'Check out this fruit I ate'
});
```



```
var userTweets = new EventEmitter();

// Elsewhere in the program . . .
userTweets.on('newTweet', function (tweet) {
    console.log(tweet);
});

// Elsewhere in the program . . .
userTweets.emit('newTweet', {
    text: 'Check out this fruit I ate'
});
```



```
var userTweets = new EventEmitter();

// Elsewhere in the program . . .
userTweets.on('newTweet', function (tweet) {
    console.log(tweet);
});

// Elsewhere in the program . . .
userTweets.emit('newTweet', {
    text: 'Check out this fruit I ate'
});
```

```javascript
var userTweets = new EventEmitter();

// Elsewhere in the program . . .
userTweets.on('newTweet', function (tweet) {
    console.log(tweet);
});

// Elsewhere in the program . . .
userTweets.emit('newTweet', {
    text: 'Check out this fruit I ate'
});
```

EVENT EMITTERS

- Objects that can “emit” specific events with a payload to any amount of registered listeners
- An instance of the “observer/observable” a.k.a “pub/sub” pattern
- Feels at-home in an *event*-driven environment

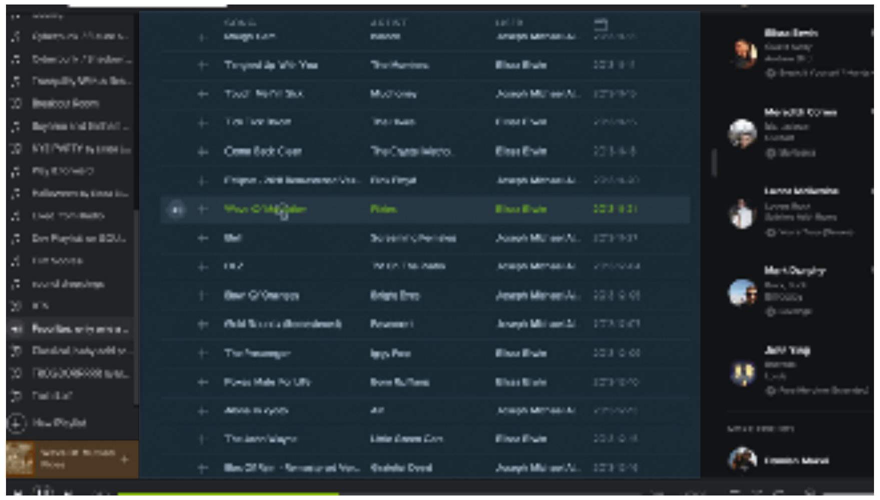
PRACTICAL USES

- Connect two decoupled parts of an application

```
var currentTrack = new EventEmitter();
```

```
currentTrack.emit('changeTrack', newTrack);
```

```
currentTrack.on('changeTrack', function (newTrack) {
    // Display new track!
});
```

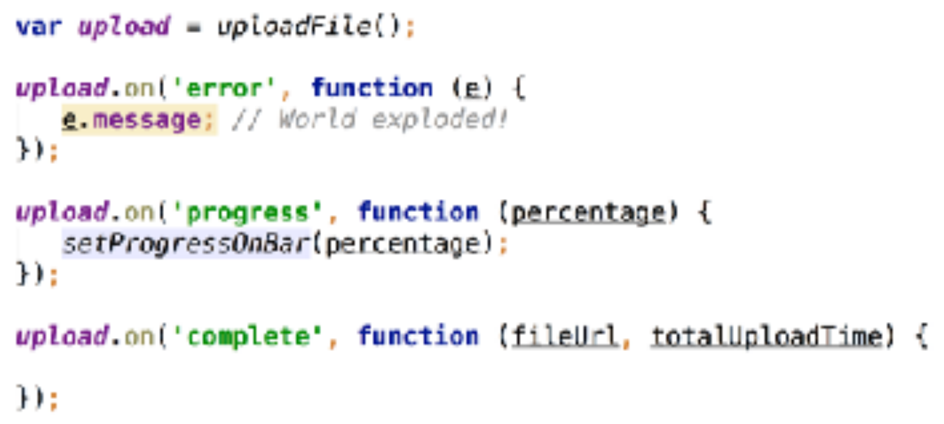


When I want to change the song, two parts of my app need to change—I need to update how that song's title and info looks, and also update the currently playing song in the bottom left corner.

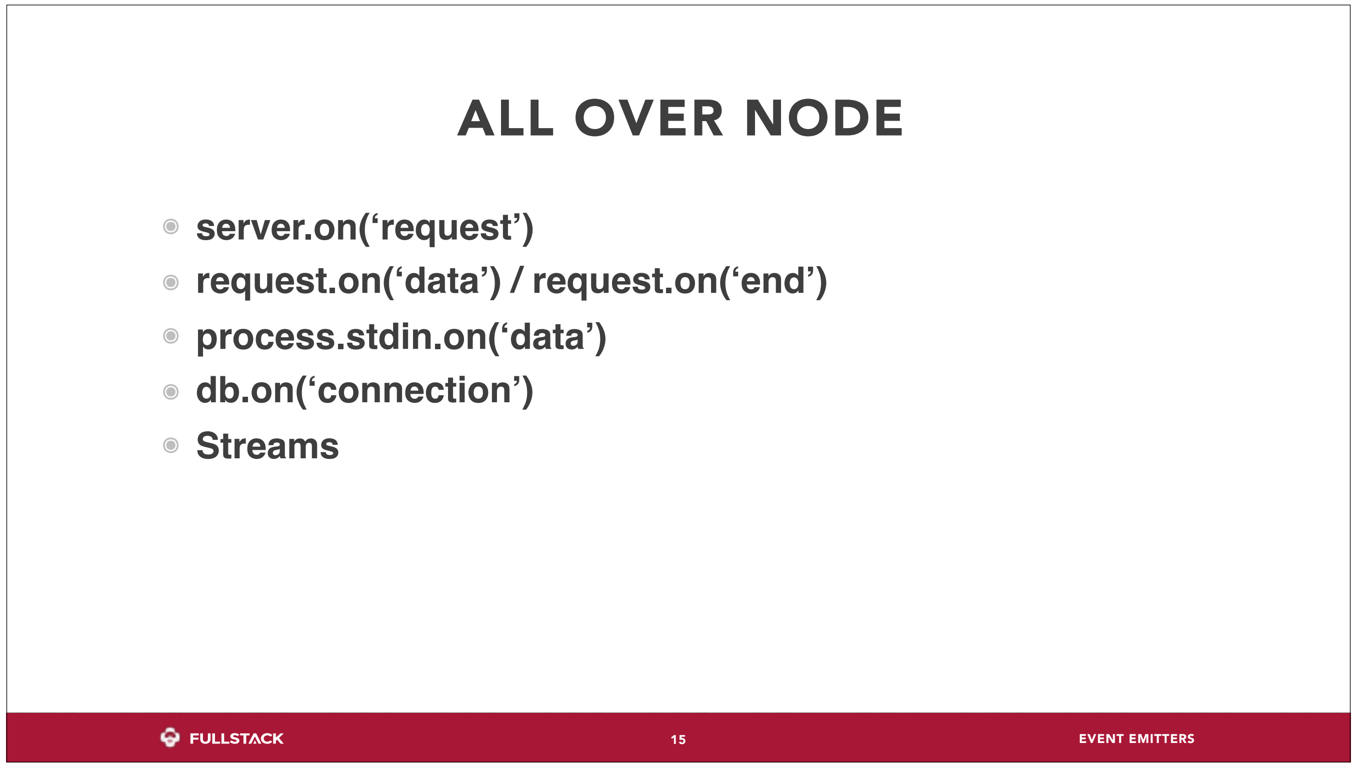
The `currentTrack` Event Emitter object can be used as a link between these two parts of my application, so that when I change the song, all the parts of my app know to update.

PRACTICAL USES

- **Represent multiple asynchronous events on a single entity.**



```
var upload = uploadFile();

upload.on('error', function (e) {
    e.message; // World exploded!
});

upload.on('progress', function (percentage) {
    setProgressOnBar(percentage);
});

upload.on('complete', function (fileUrl, totalUploadTime) {

});
```



ALL OVER NODE

- server.on('request')
- request.on('data') / request.on('end')
- process.stdin.on('data')
- db.on('connection')
- Streams

Take a look at what body-parser is doing
Do a quick example of using it to create our own events.
Lab