JavaScript Promises and Async Await

Frank Walsh

Recap -Javascript Characteristics

- JavaScript is single threaded
    - shares a thread with a load of other stuff
- JavaScript is event driven
    - Events happen – we write code to deal with them
- JavaScript can be Asynchronous
    - Order of operation results may differ from order they were called

Recap - Async Code

What will be the console output be:

```
console.log('setTimeout');
    for(var i = 0; i < 5; i++) {

     setTimeout(()=> {console.log('i: '+i)}, 1);

   }
```

Hint: the for loop will finish before the 1st callback.

What have Events ever done for us...

- Great for things that can happen multiple times
- Great if you don't really care about what happened before you attached the listener
- Great if it's a straight-forward, stand-alone event with a quick resolution time

E.g. key press event on control.

document.getElementById("demo").addEventListener("keypress", myFunction);

- But...

The Callback

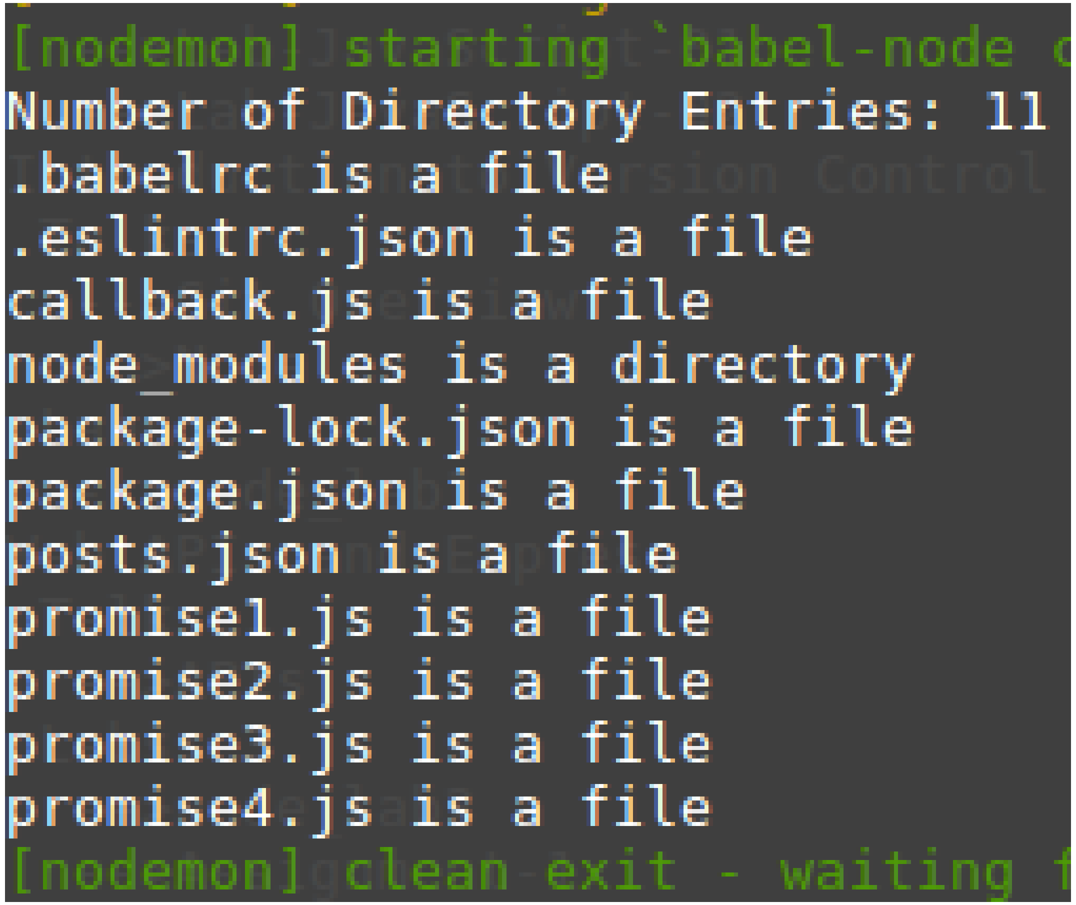
- The traditional way of handling asynchronous events
  - Function that is registered as the event handler for something we're fairly sure will happen in the future

```
componentDidMount : ()=> {
      request.get('http://0.0.0.0:3000/friends')
        .end( (error, response) => { . . . Callback code ...}
   }
   …….
   filterFriends : (event) => { . . . Callback . . . .}
   render: ()=>{
       . . . . . . . .
       <input type="text”  …. onChange={this.filterFriends} />
```

Callbacks in Node

Display info on a directory's contents

```
import fs from 'fs';
fs.readdir('.', (err, filenames) => {
 if (err) throw err;
 console.log(`Number of Directory Entries:
   ${filenames.length}`);
 filenames.forEach((name) => {
  fs.stat(name, (err, stats) => {
    if (err) throw err;
    let result = stats.isFile() ? 'file' : 'directory';
    console.log(name, 'is a', result);
  });
 });
});
```



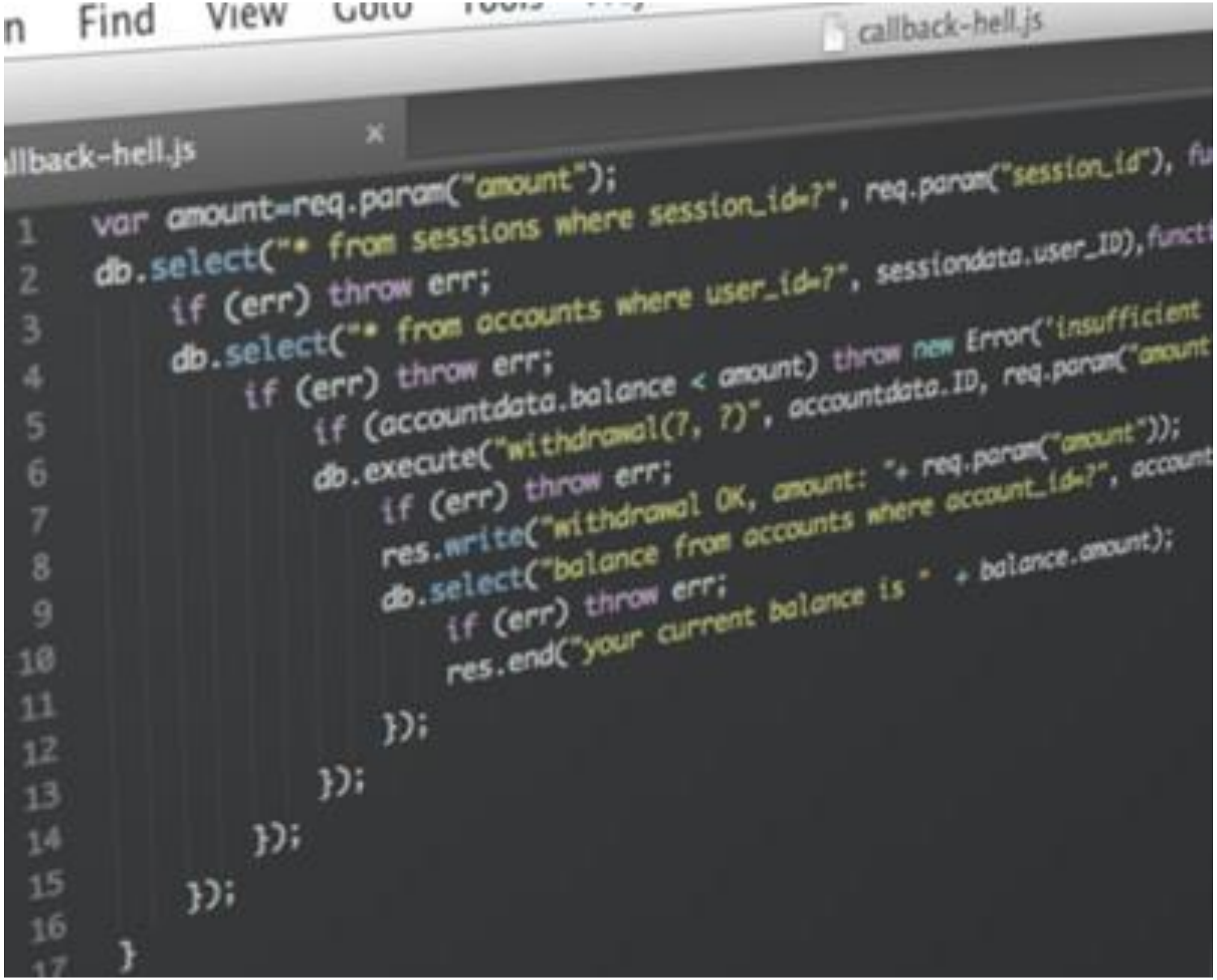
```
[nodemon] starting `babel-node c
Number of Directory Entries: 11
.babelrc is a file
.eslintrc.json is a file
callback.js is a file
node_modules is a directory
package-lock.json is a file
package.json is a file
posts.json is a file
promise1.js is a file
promise2.js is a file
promise3.js is a file
promise4.js is a file
[nodemon] clean exit - waiting f
```

Callbacks vs. Events

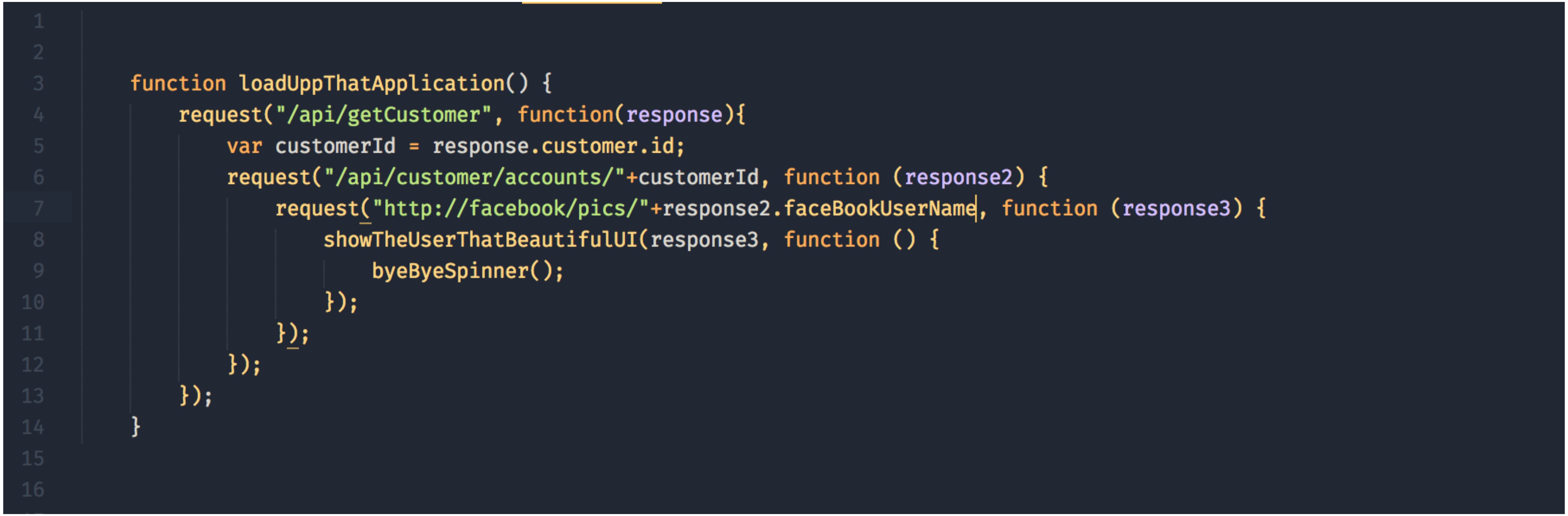
- Generally speaking a 'callback' is under the control of the detecting process.
  - you tell the Node event Manager to "call myEvent when HTTP Get request happens"
  - The Event manager calls myEvent when a request is received
- An event is like a tweet; anyone can read and respond to it. While a callback is like a text message; only the person(s) you send it to can read and respond to it.
- **Events** - Think of a Server (Employee) and Client (Boss).One Employee can have many Bosses. The Employee Raises the event, when he finishes the task, and the Bosses may decide to listen to the Employee event or not. The employee is the publisher and the bosses are subscriber.
- **Callback** - The Boss specifically asked the employee to do a task and at the end of task done, the Boss wants to be notified. The employee will make sure that when the task is done, he notifies only the Boss that requested, not necessary all the Bosses. The employee will not notify the Boss, if the partial job is done. It will be only after all the task is done. Only one boss requested the info, and employee only posted the reply to one boss.

Callback Hell

- One aspect of callback hell is code readability / understandability
- Each new callback causes another level of indentation
- And this is typically the "success path"
    - It gets more complicated if you want to handle errors.



```
var amount=req.param("amount");
db.select("* from sessions where session_id=?", req.param("session_id"), fu
    if (err) throw err;
    db.select("* from accounts where user_id=?", sessiondata.user_ID),funct
        if (err) throw err;
            if (accountdata.balance < amount) throw new Error('insufficient
            db.execute("withdrawal(?, ?)", accountdata.ID, req.param("amount
                if (err) throw err;
                res.write("withdrawal OK, amount: "+ req.param("amount"));
                db.select("balance from accounts where account_id=?", account
                    if (err) throw err;
                    res.end("your current balance is " + balance.amount);
                });
            });
        });
    });
}
```



```
function loadUppThatApplication() {
    request("/api/getCustomer", function(response){
        var customerId = response.customer.id;
        request("/api/customer/accounts/"+customerId, function (response2) {
            request("http://facebook/pics/"+response2.faceBookUserName, function (response3) {
                showTheUserThatBeautifulUI(response3, function () {
                    byeByeSpinner();
                });
            });
        });
    });
}
```

Callback Hell – Multiple requests

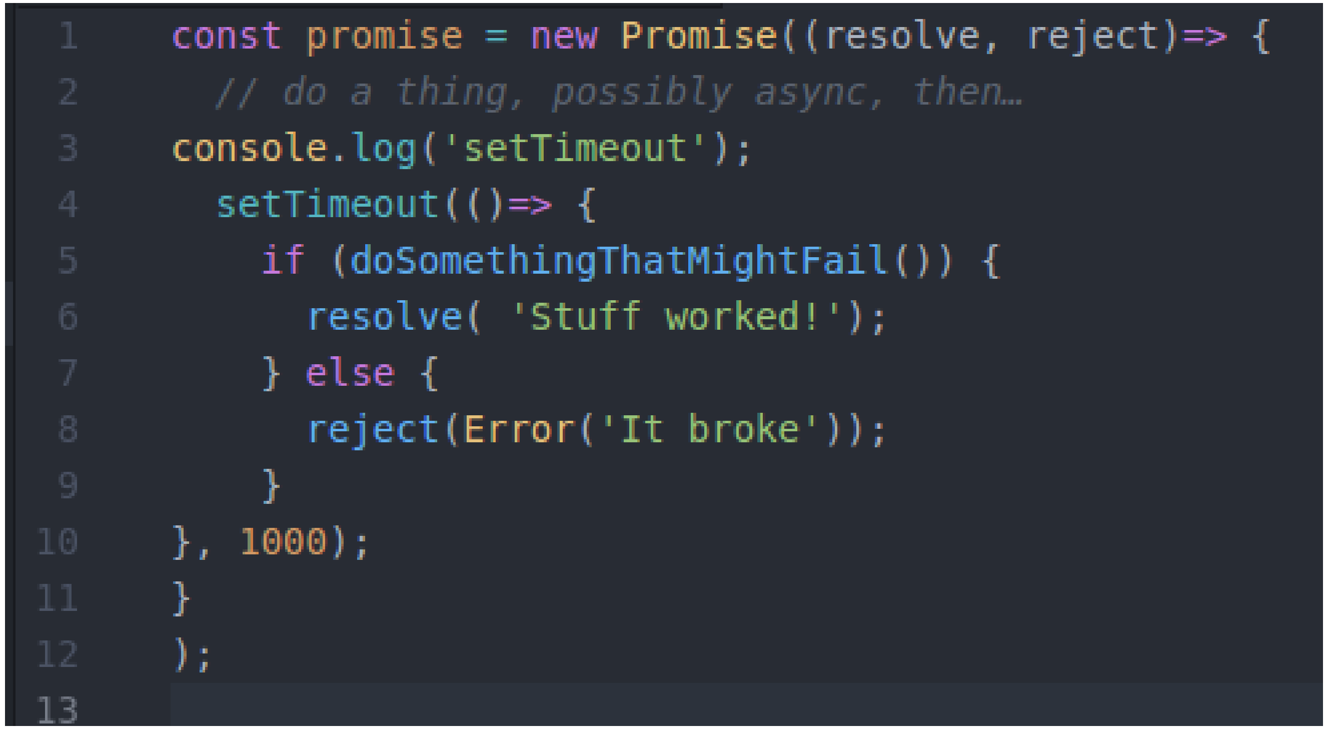
Promises

- A promise is the eventual result of an asynchronous operation or computation.
- Promises are:
  - an abstraction useful in async programming
  - an associated API that allows us to use this abstraction in our programs.
- A promise can be:
  - **fulfilled** - The action relating to the promise succeeded
  - **rejected** - The action relating to the promise failed
  - **pending** - Hasn't fulfilled or rejected yet
  - **settled** - Has fulfilled or rejected

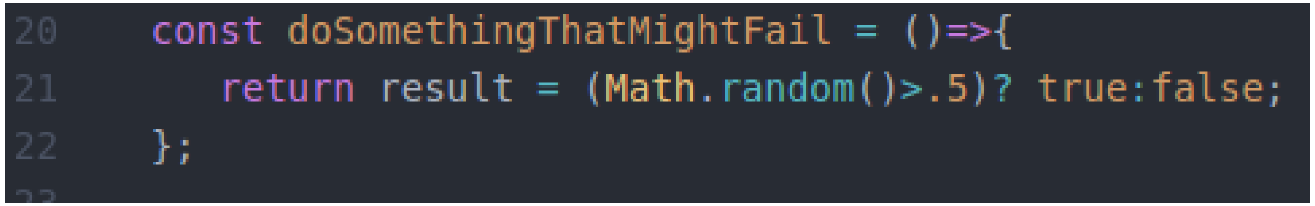
Promise Genealogy

- Nothing new...
  - First proposed in 1976 by Daniel P. Friedman and David Wise, and Peter Hibbard called it eventual. A similar concept, termed future, was introduced in 1977 in a paper by Baker and Carl Hewitt
- Native support in Javascript now but 3rd party libraries have been around for a while:
  - Q,when,WinJS,RSVP.js
- Although APIs can differ, Promise implementations follow a standardized behaviour (Promises/A+)
  - As does Javascript.

JavaScript promise dummy implementation

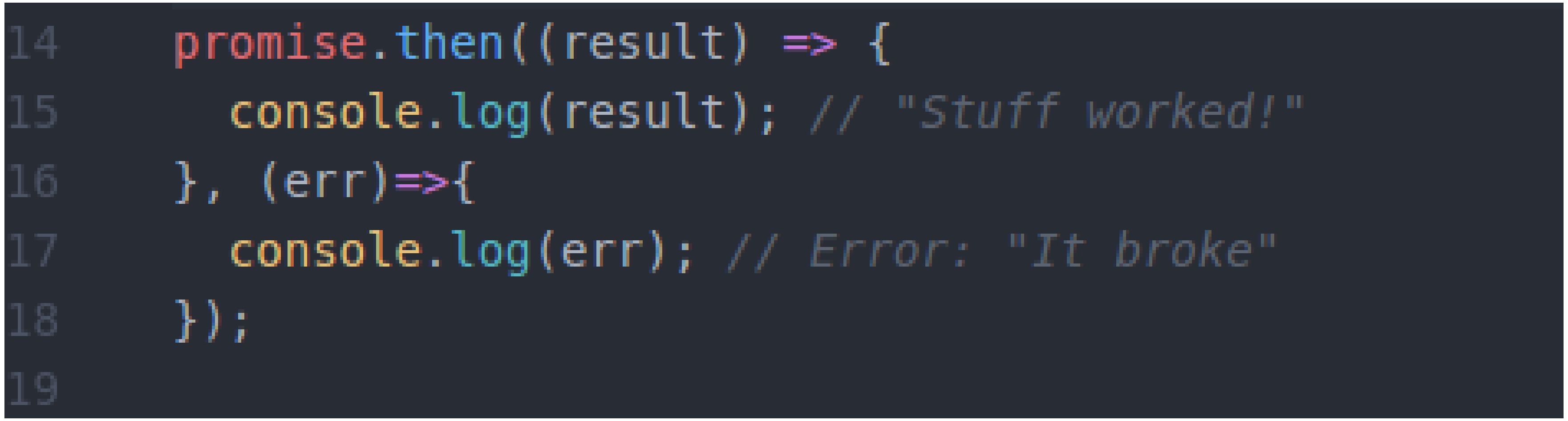


```
const promise = new Promise((resolve, reject)=> {
  // do a thing, possibly async, then…
console.log('setTimeout');
  setTimeout(()=> {
    if (doSomethingThatMightFail()) {
      resolve( 'Stuff worked!');
    } else {
      reject(Error('It broke'));
    }
}, 1000);
}
);
```



```
const doSomethingThatMightFail = ()=>{
   return result = (Math.random()>.5)? true:false;
};
```

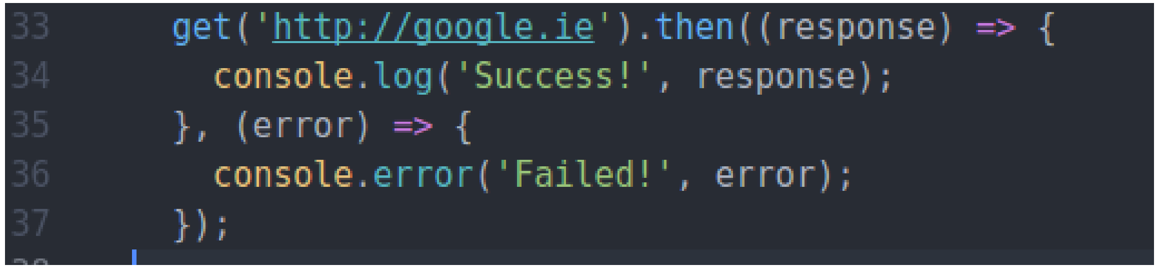
JavaScript Dummy implementation



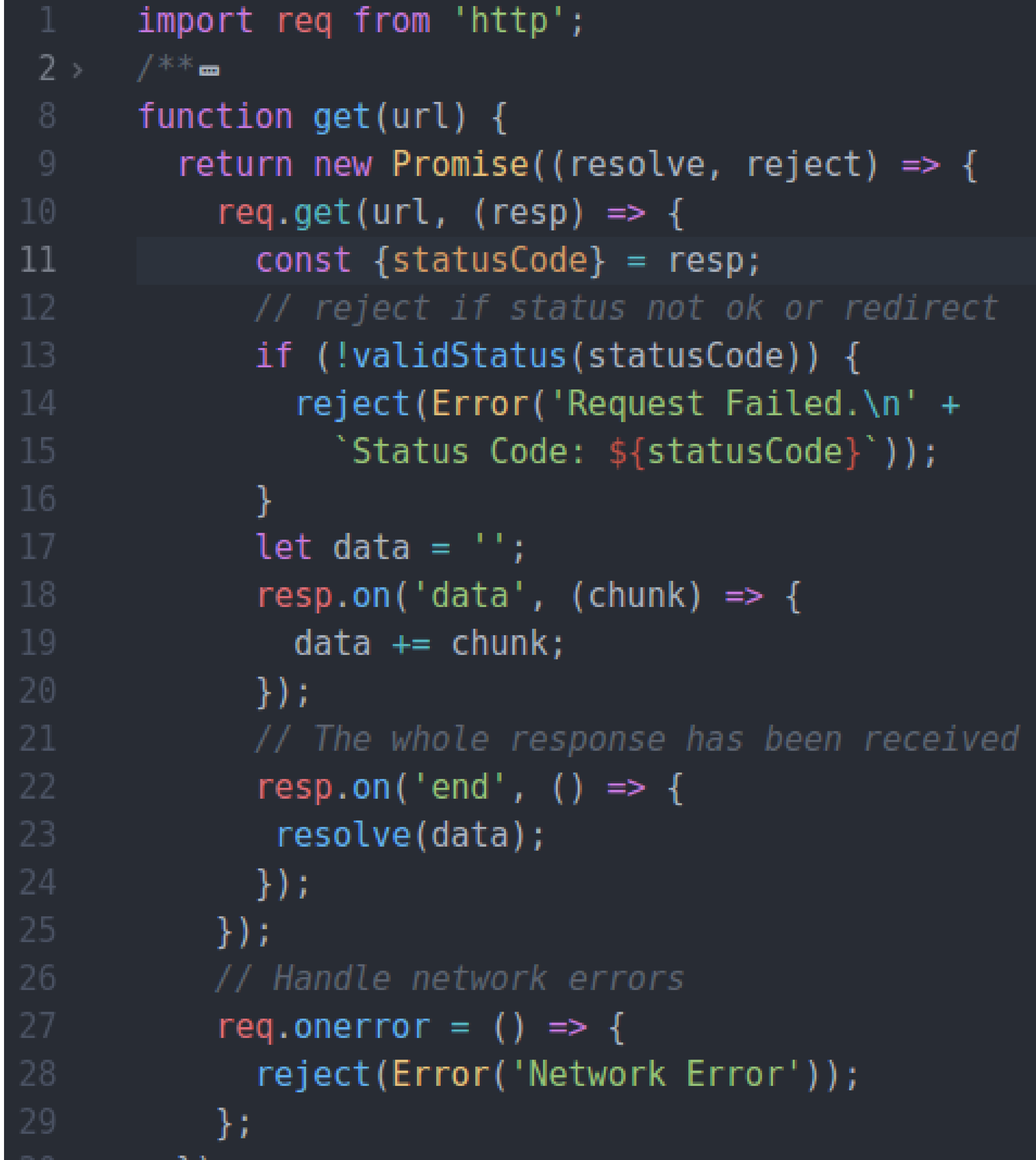
```javascript
promise.then((result) => {
  console.log(result); // "Stuff worked!"
}, (err)=>{
  console.log(err); // Error: "It broke"
});
```

HTTP Request Promise

- Node HTTP API updated to use a promise.
- The *get(…) function* makes HTTP requests by abstracting the http module.
- Http module isn’t very user friendly compared to other solutions. Can now make a request like this:



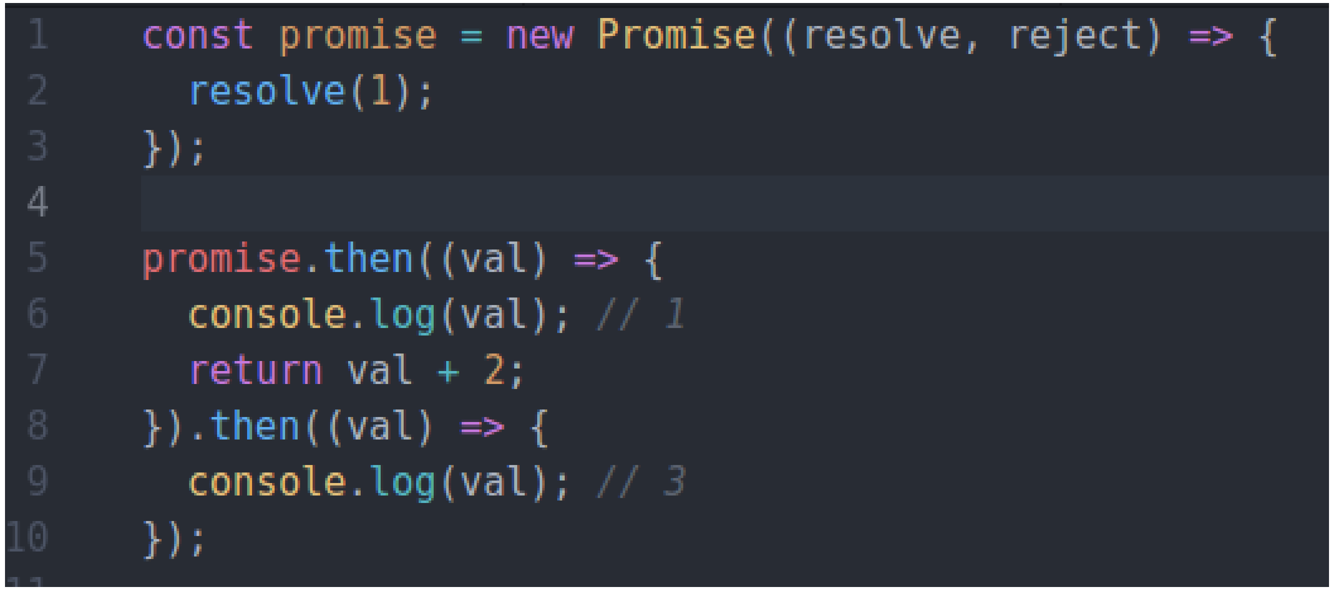
```
get('http://google.ie').then((response) => {
  console.log('Success!', response);
}, (error) => {
  console.error('Failed!', error);
});
```



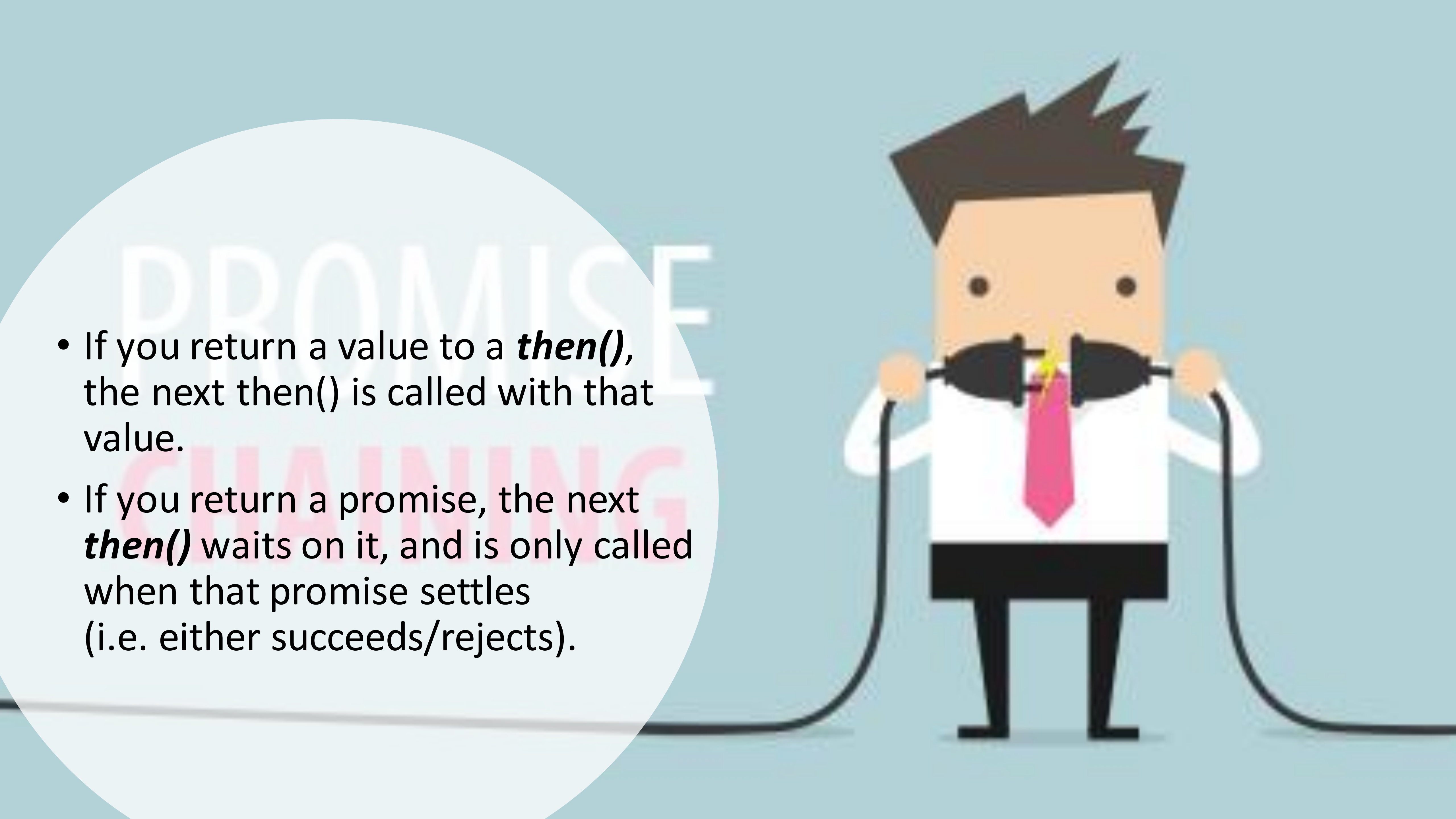
```
import req from 'http';
/**…
function get(url) {
  return new Promise((resolve, reject) => {
    req.get(url, (resp) => {
      const {statusCode} = resp;
      // reject if status not ok or redirect
      if (!validStatus(statusCode)) {
        reject(Error('Request Failed.\n' +
          `Status Code: ${statusCode}`));
      }
      let data = '';
      resp.on('data', (chunk) => {
        data += chunk;
      });
      // The whole response has been received
      resp.on('end', () => {
       resolve(data);
      });
    });
    // Handle network errors
    req.onerror = () => {
      reject(Error('Network Error'));
    };
```

Chaining

- you can chain thens together to transform values or run additional async actions one after another.
- The alternative to "Callback Hell"



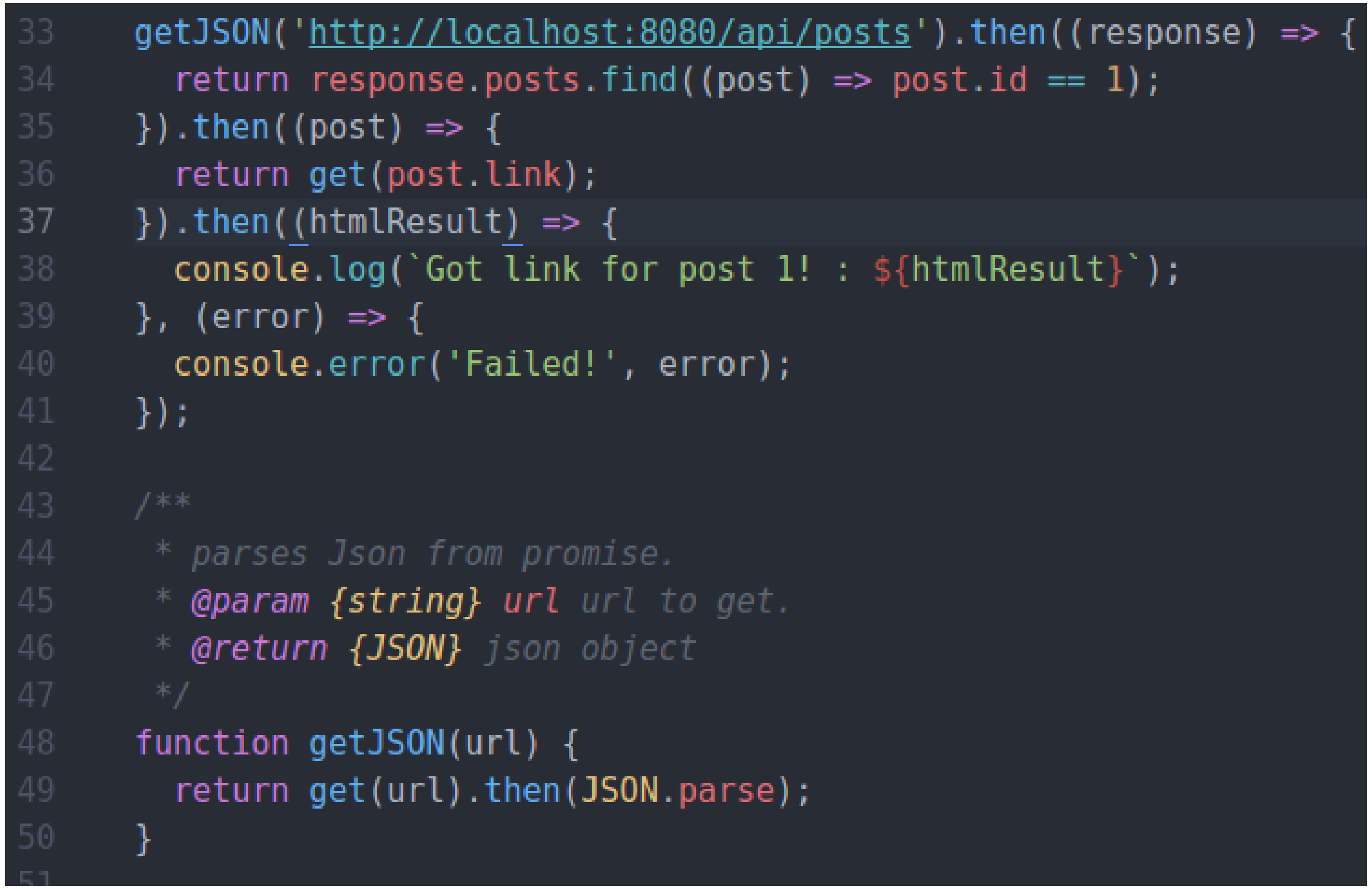
```
const promise = new Promise((resolve, reject) => {
  resolve(1);
});

promise.then((val) => {
  console.log(val); // 1
  return val + 2;
}).then((val) => {
  console.log(val); // 3
});
```



- If you return a value to a ***then()***, the next then() is called with that value.
- If you return a promise, the next ***then()*** waits on it, and is only called when that promise settles
(i.e. either succeeds/rejects).

Chaining

- Problem: Request Hacker News Posts, then request link for 1st post (id ==1), then display link
- **NOTE: error callback applies to whole chain**
  - No need to specify error handler for each promise.



```
getJSON('http://localhost:8080/api/posts').then((response) => {
  return response.posts.find((post) => post.id == 1);
}).then((post) => {
  return get(post.link);
}).then((htmlResult) => {
  console.log(`Got link for post 1! : ${htmlResult}`);
}, (error) => {
  console.error('Failed!', error);
});

/**
 * parses Json from promise.
 * @param {string} url url to get.
 * @return {JSON} json object
 */
function getJSON(url) {
  return get(url).then(JSON.parse);
}
```

Error Handling

- then() takes two arguments, one for fulfillment(success), one for rejection(failure)

```
get('story.json').then((response) => {
  console.log("Success!", response);
}, (error) => {
  console.log("Failed!", error);
})
```

Error Handling - catch(...)

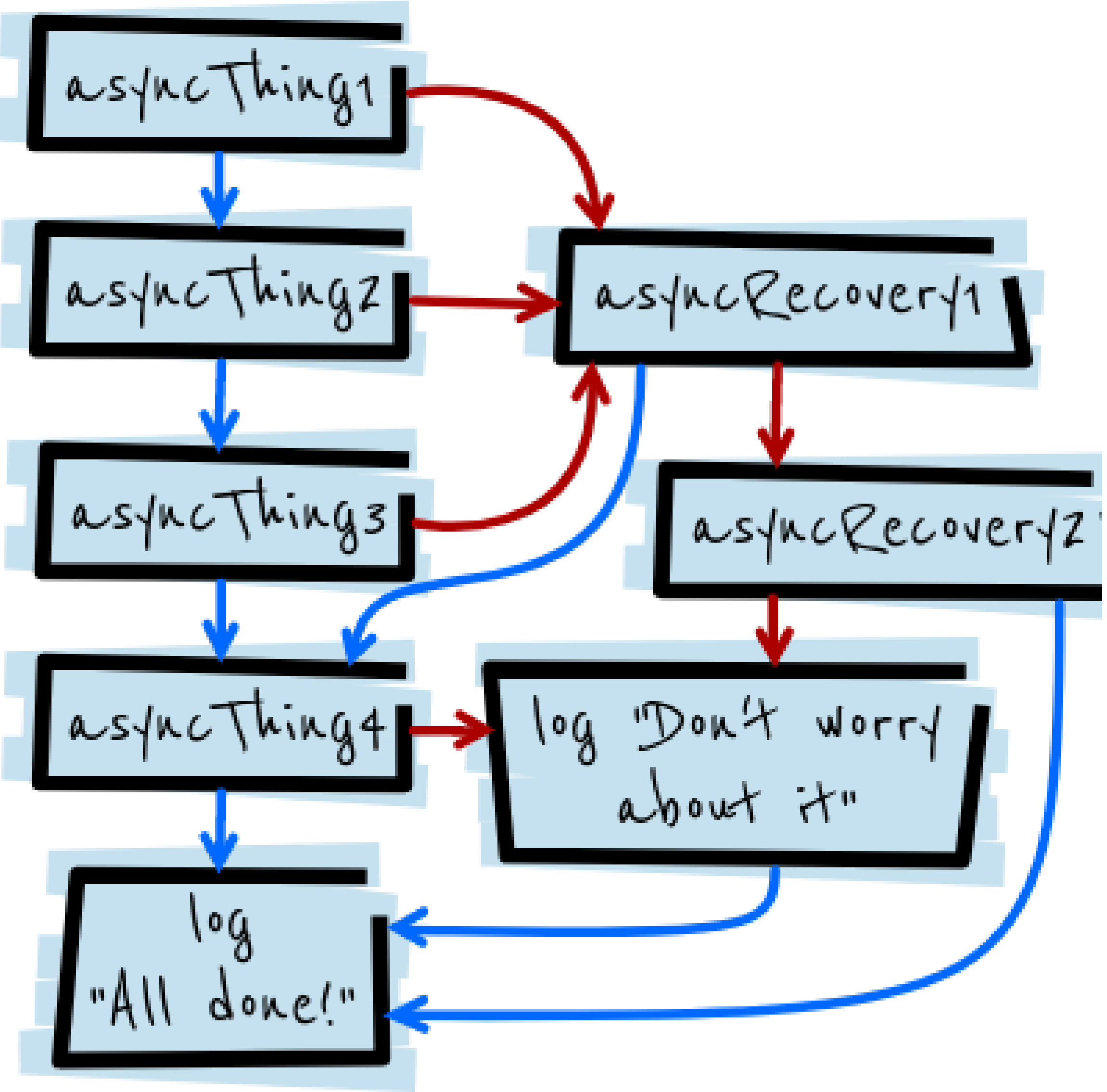
- You can also use catch() to handle promise rejects:
- Reacts slightly different to previous...

```
promise.then((result) => {
  console.log(result); // "Stuff worked!"
}).catch((err)=>{
  console.log(err); // Error: "It broke"
});
```

Rejection forwarding

- A Promise rejection will skip forward to the next then() with a rejection callback (or catch()):

```
asyncThing1().then(function()  {
    return asyncThing2();
   }).then(function() {
    return asyncThing3();
   }).catch(function(err) {
    return asyncRecovery1();
   }).then(function() {
    return asyncThing4();
   }, function(err) {
    return asyncRecovery2();
   }).catch(function(err) {
    console.log("Don't worry about it");
   }).then(function() {
    console.log("All done!");
   })
```



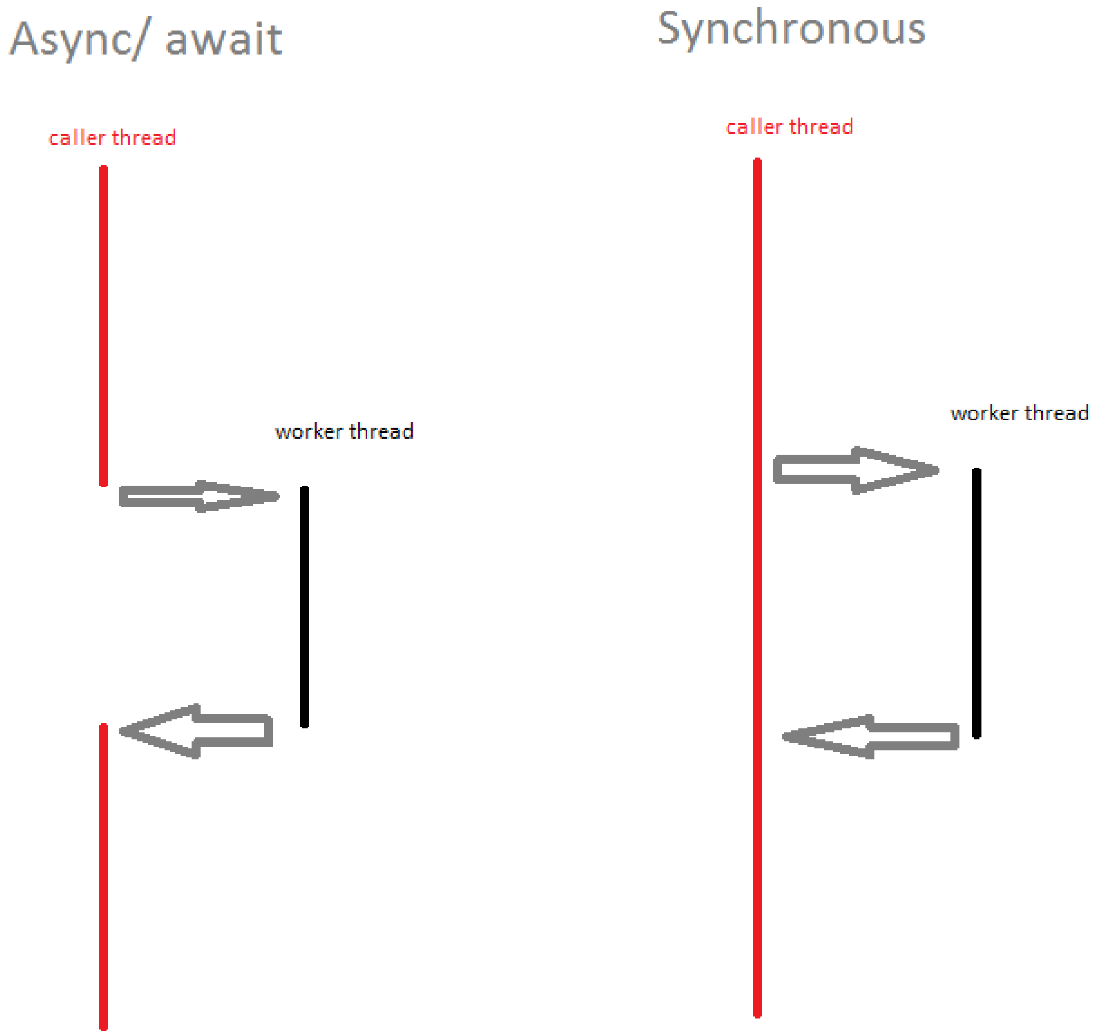
https://developers.google.com/web/fundamentals/primers/promises#error_handlingck to add text

Async Functions

using async/await

Hot off the press(ish): Async Awaits

- **async/await** and **promises** are essentially the same under the hood
- **async** is a keyword
    - Used in function declaration
- **await** is used during the promise handling
    - must be used within an **async function**
- **async** functions return a promise, regardless of what the return value is within the function
- **Available now** in most good browsers as well as Node.js



Promise vs. Async Await

Promise

```
promise.then((result) => {
  console.log(result); // "Stuff worked!"
}, (err)=>{
  console.log(err); // Error: "It broke"
});
```

Async

```
async function doSomethingAsync() {
  try{
    let result = await promise();
    console.log(result);
  }catch (err){
    console.log(err);
  }
}
```

Wrapper Function

- As an Async function always returns a Promise.
    - can *wrap* the async function to catch errors...
    - Can drop try/catch.
- Makes code more readable.

```
const asyncWrapper = fn => {
    return Promise.resolve(fn)
      .catch(err => {return err.message});
};


async function doSomethingAsync() {
    const result = await asyncWrapper(promise());
    console.log(result);
}
```

Example: HackerNews with async await

```
async function getJSON(url) {
  return JSON.parse(await get(url));
}
```

```
async function getPostLinkHTML(url, postId) {
  const posts = await getJSON(url);
  const post = posts.find((post) => post.id == postId);
  const htmlResult = await get(post.link);
  console.log(`Got link for post 1! : ${htmlResult}`);
}


getPostLinkHTML('http://localhost:8080/api/posts', 1);
```

Parallelism

- Previous example needed to be sequential
    - Had to get data back from API **BEFORE** getting link URL
- Should only be sequential if you need to be...

Takes 1000ms

```
async function series() {
  await wait(500); // Wait 500ms…
  await wait(500); // …then wait another 500ms.
  return "done!";
}
```

Takes 500ms

```
async function parallel() {
  const wait1 = wait(500); // Start a 500ms timer asynchronously…
  const wait2 = wait(500); // this timer happens in parallel.
  await wait1; // Wait 500ms for the first timer…
  await wait2; // …by which time this timer has already finished.
  return "done!";
}
```

Sources

- https://developers.google.com/web/fundamentals/primers/promises
- https://stackoverflow.com/questions/2069763/difference-between-event-handlers-and-callbacks
- https://medium.com/@Abazhenov/using-async-await-in-express-with-node-8-b8af872c0016