Async Programming

Non-blocking programming

Agenda

- Single thread
- Multi thread
- JavaScript threading paradigm
- Problem with single thread
- Javascript support for asynchronous programming

Learning Objectives

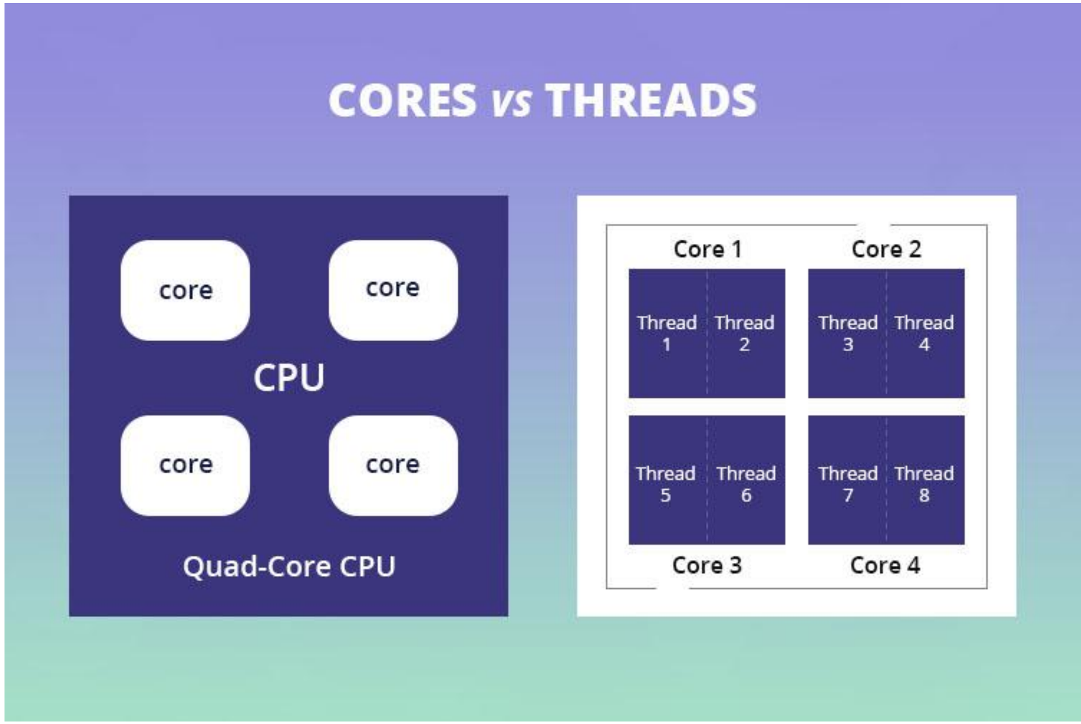
- You know what asynchronous programming is
- You know what the challenges in async programming are
- You know what the concept callback, promise, async-await are
- You know how to use promises and async-await constructs

Thread

"A thread in computer science is short for a thread of execution. Threads are a way for a program to divide (termed "split") itself into two or more simultaneously (or pseudo-simultaneously) running tasks. Threads and processes differ from one operating system to another but, in general, a thread is contained inside a process and different threads in the same process share same resources while different processes in the same multitasking operating system do not."

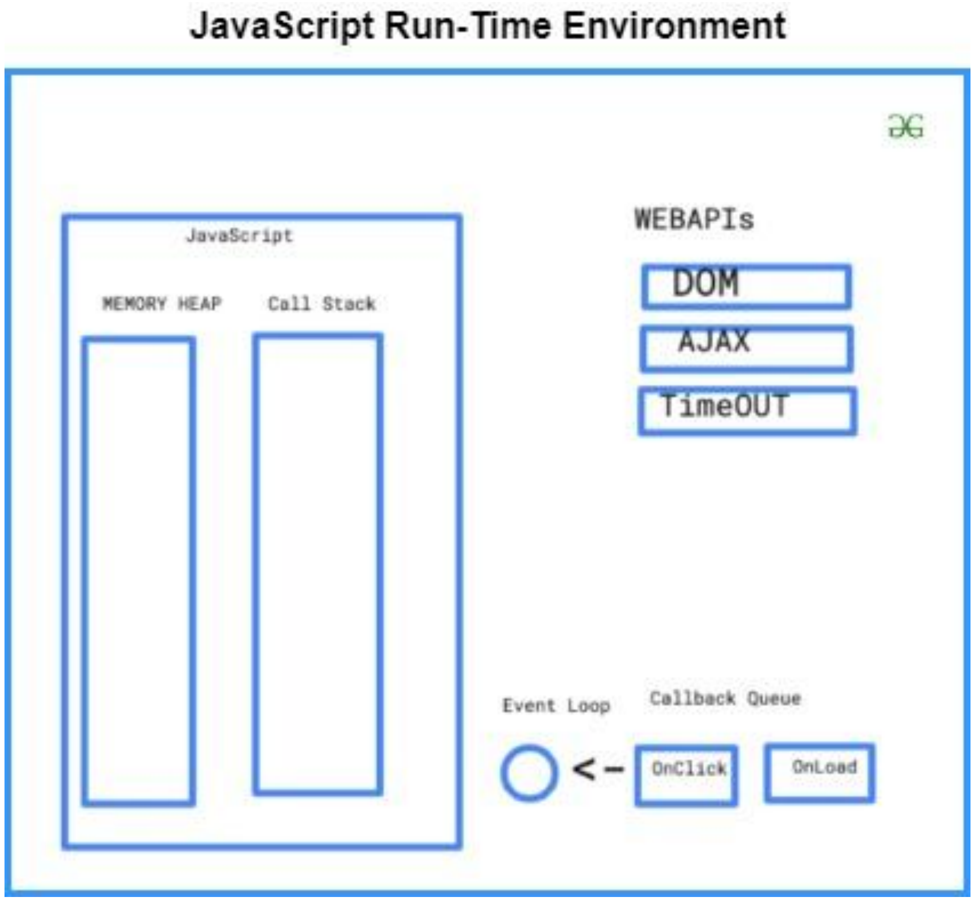
Single Thread vs Multiple Threads

Single threaded processes contain the execution of instructions in a single sequence. In other words, one command is processes at a time. The opposite of single threaded processes are multithreaded processes. These processes allow the execution of multiple parts of a program at the same time.



JavaScript Thread Model

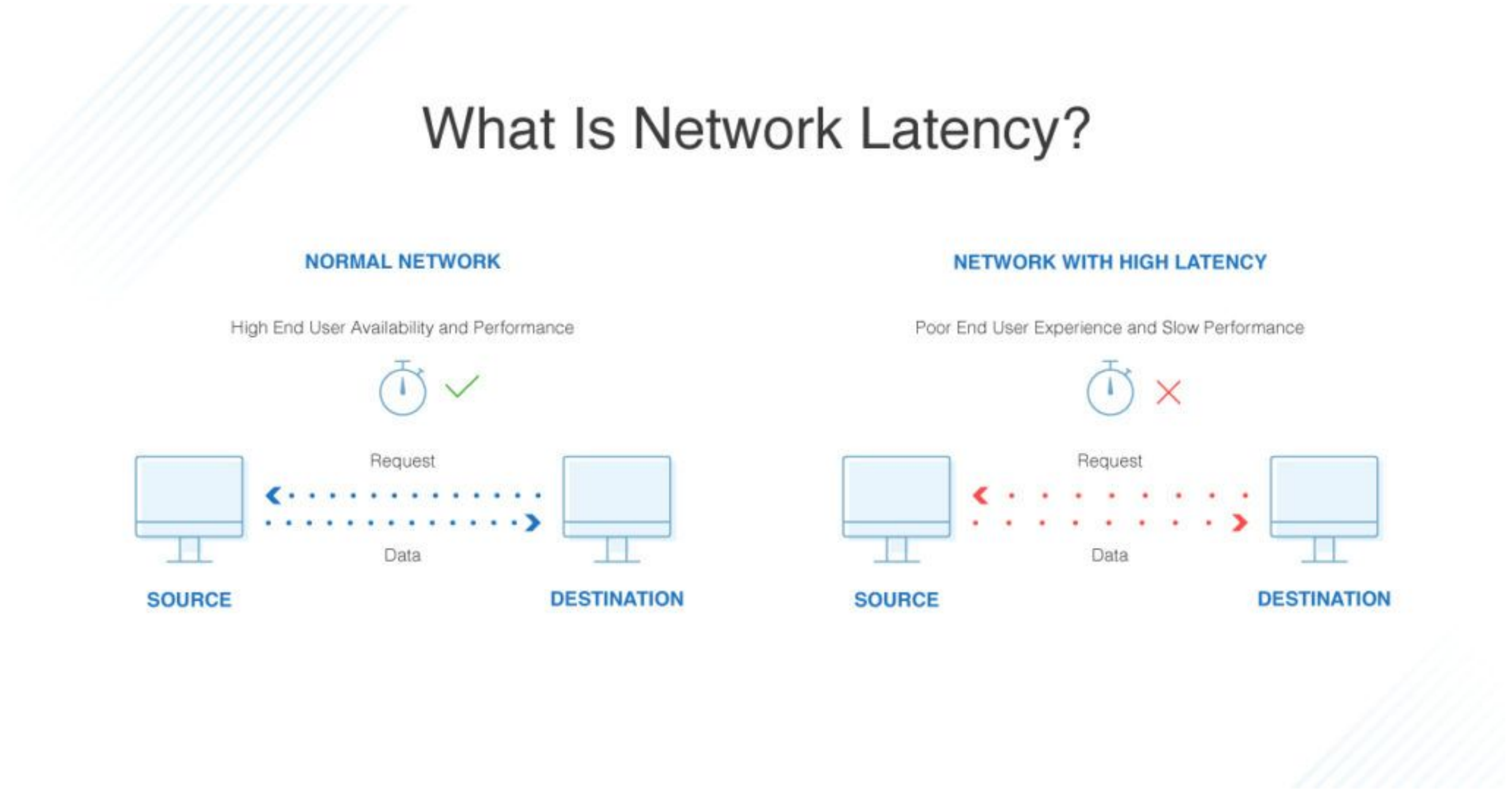
- JavaScript is a single-threaded language.
- Has only one call stack.
- Easy to program
- No concurrency issues, no deadlock



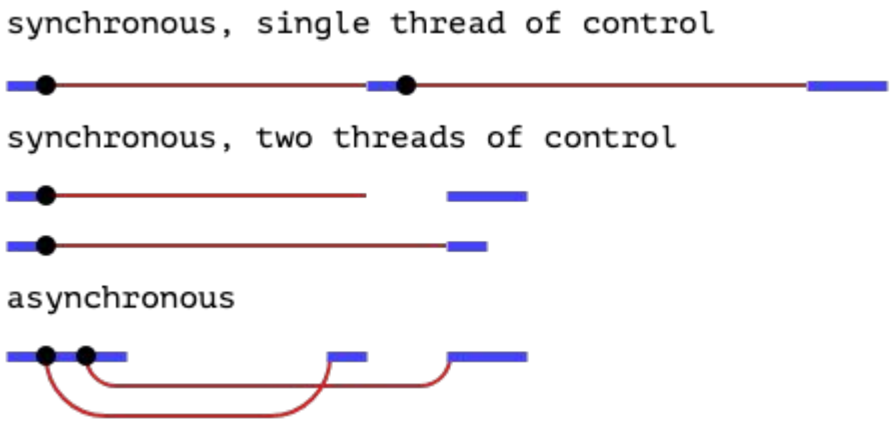
The Problem

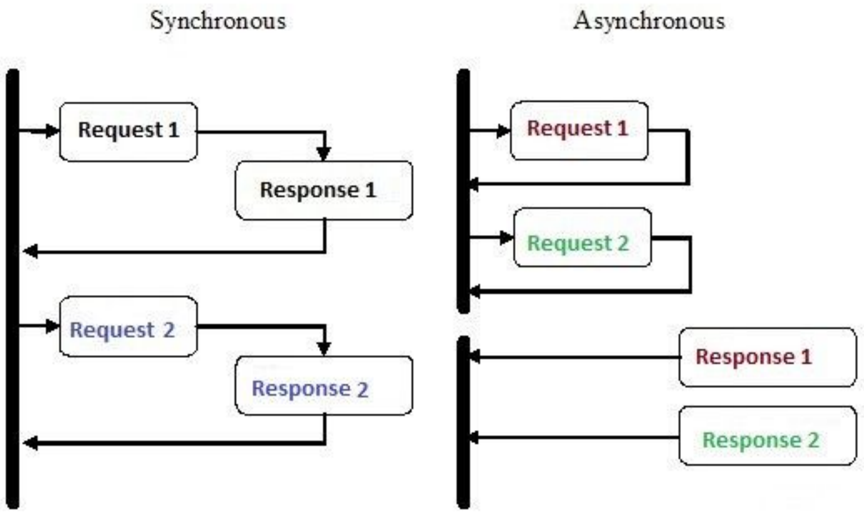
- Applications were running sequentially in an isolated process before
- Nowadays, there are plenty of APIs around the world, one can consume.
- The network latency is the main problem if you have a dependency to an external process, especially over the network.
- Network is not always stable and can not guarantee the same throughput.
- Some packages can even be lost in the web traffic.

- So, how can I be sure that my application still run under these conditions?
- Handling such scenarios is now the mainstream because of cloud computing


What Is Network Latency?
NORMAL NETWORK
High End User Availability and Performance
Request
Data
SOURCE
DESTINATION
NETWORK WITH HIGH LATENCY
Poor End User Experience and Slow Performance
Request
Data
SOURCE
DESTINATION

The problem with waiting


synchronous, single thread of control
synchronous, two threads of control
asynchronous



Sync vs Async

Solution

We need a solution which enables us waiting and handling the asynchronous response celeverly.

It should give us a handle just after trigger the request. Somewhere after comes the response, which should be distinguishable from other requests and response.

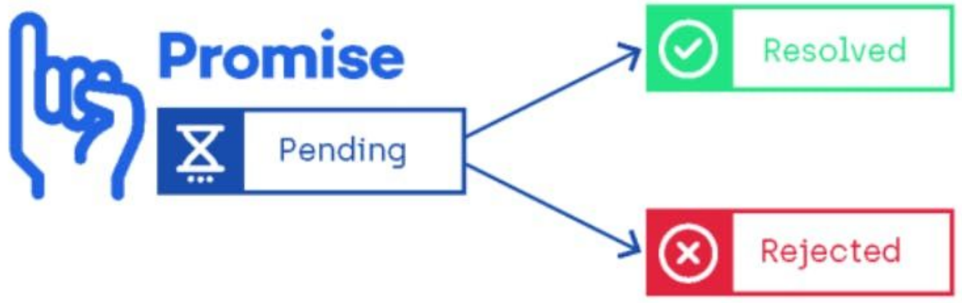
It should notify us if there are any errors in the communication. It may occur some network problems (lost packages, networks gaps, connection timeouts, server errors, etc...).

I should be able to use this mechanism im my code as if the result is there. So enables us using asynchronous object in a synchron way.

Javascript Support

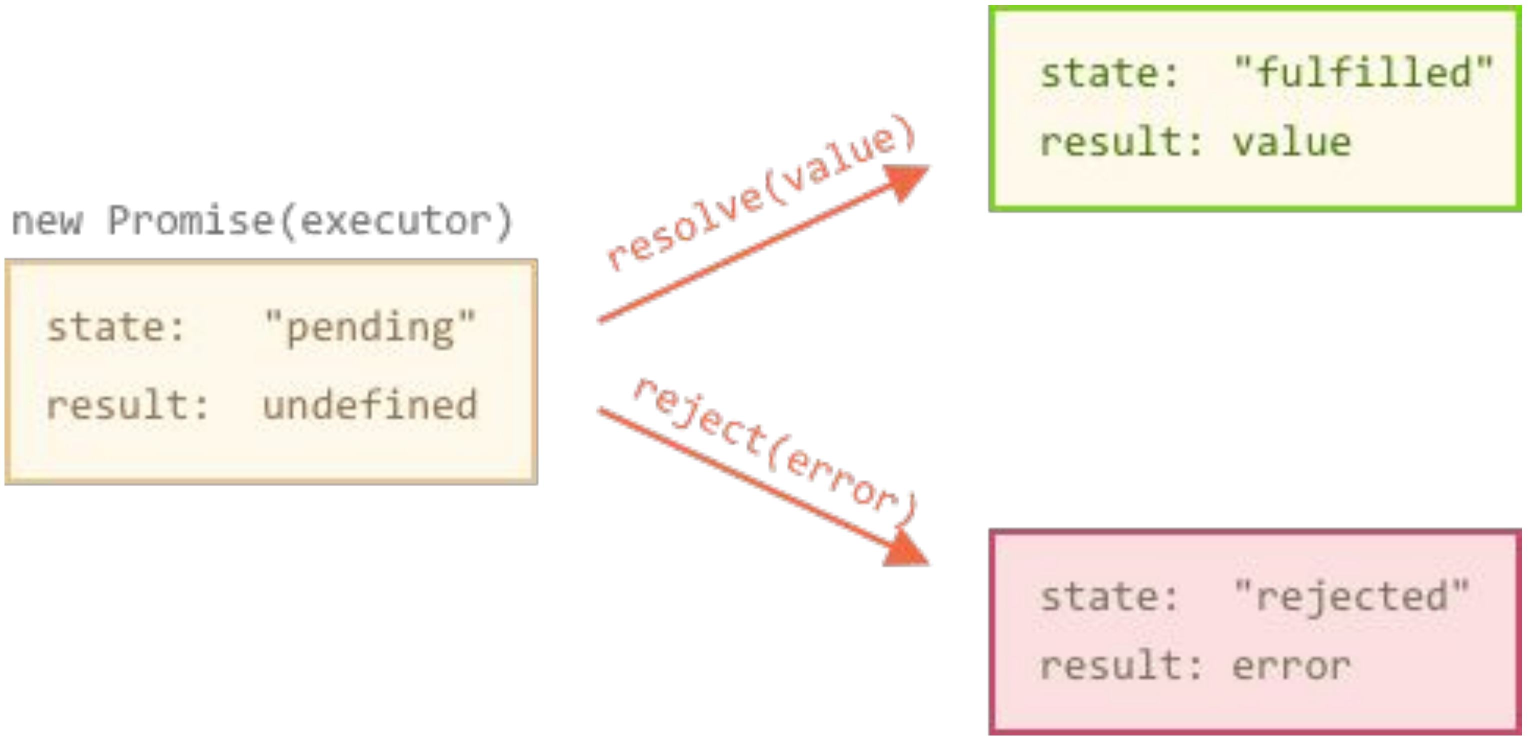
- Primarily, js has such asynchron support through its callback ability. So you can pass through a success callback and an error callback. But it is an oldy
- The new approach is called PROMISE.
    - CLIENT: “Promise me, please, you come back! Don’t leave me alone!”
    - PROMISE: “Ooh babe, no worry! I actually always be with you.”
    - Promises provide us the asynchronous communication in an elegant way
    - The promise object can be used after its creation in the code event it has not been resolved.
- Async Await is the newest approach!
    - It basically eliminates the need of promises as well (actually they are used behind the scenes).
    - More elegant solution
- Asynchronous programming ≠ Multi-threading

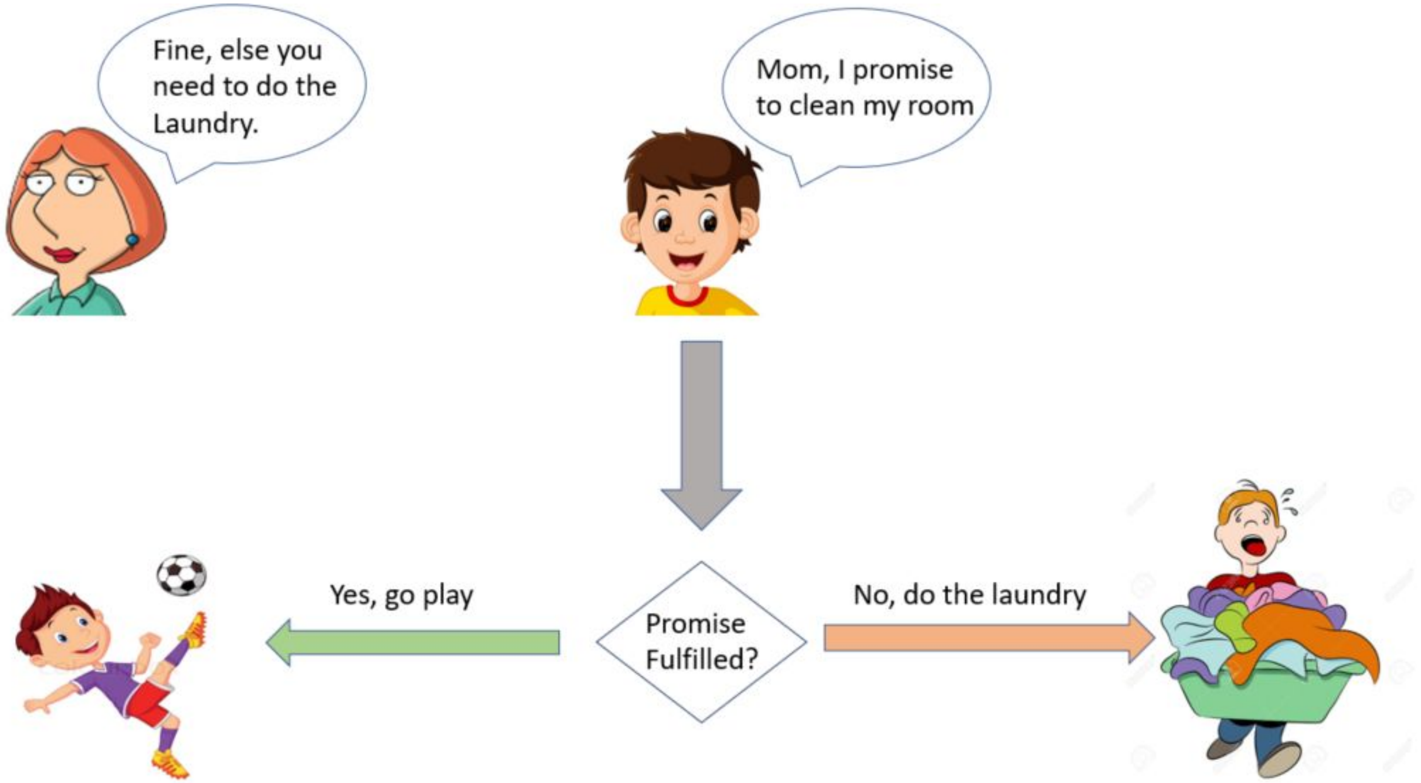
Promises

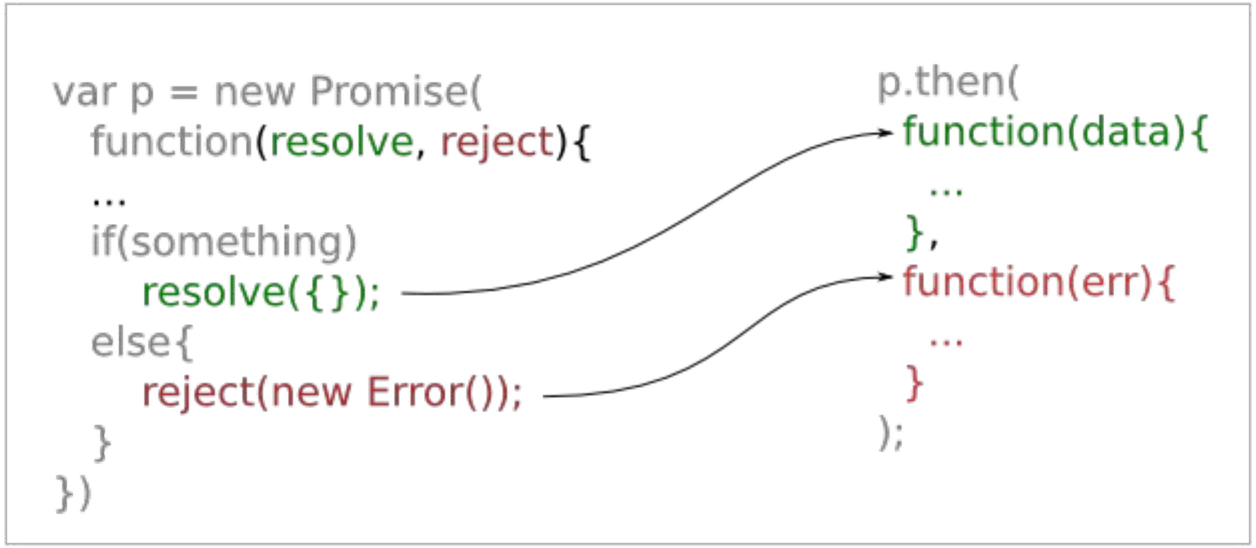

Promise
Pending
Resolved
Rejected

What is a Promise?

- Asynchronicity is always part of web programming, for javascript as well
- Javascript solved this problem first by utilizing the callbacks
- Promises are syntax improvements and enhancements for the asynchronous programming
- Promises returns special objects encapsulating the future (value)
- A promise is waiting for the response, therefore promises has states
    - Pending, Resolved, Rejected
- It has a waterfall approach
    - make the call
    - wait for the result
    - result has come
    - get the result by listening response with ".then()"
    - Listen the errors by using ".catch()"


new Promise(executor)
state: "pending"
result: undefined
resolve(value)
reject(error)
state: "fulfilled"
result: value
state: "rejected"
result: error

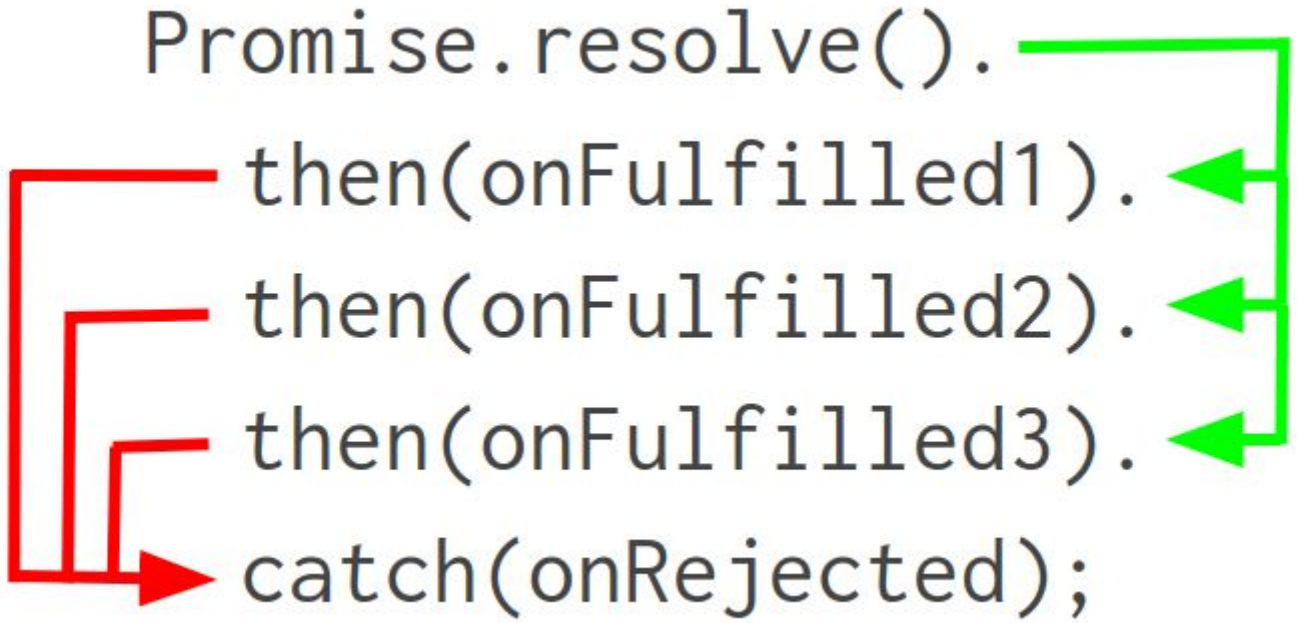

Fine, else you need to do the Laundry.
Mom, I promise to clean my room
Promise Fulfilled?
Yes, go play
No, do the laundry



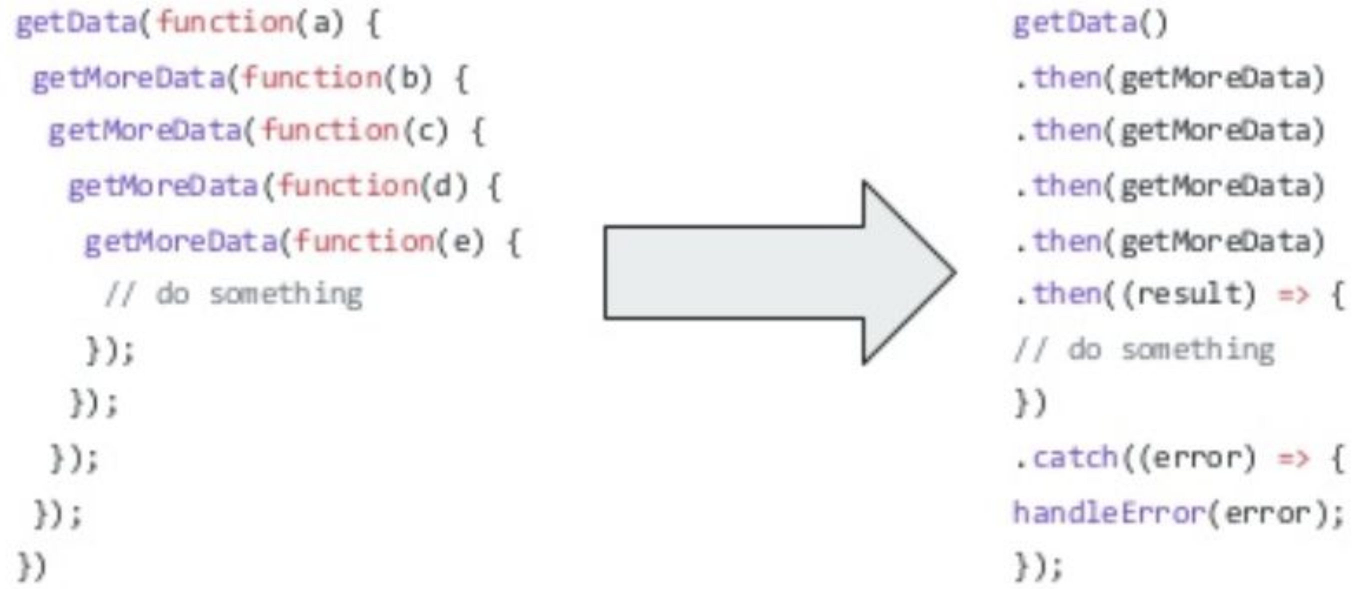
```
var p = new Promise(
  function(resolve, reject){
  ...
  if(something)
     resolve({});
  else{
     reject(new Error());
  }
})
```

```
p.then(
 function(data){
   ...
 },
 function(err){
   ...
 }
);
```

Chaining Promises



```
Promise.resolve().
  then(onFulfilled1).
  then(onFulfilled2).
  then(onFulfilled3).
  catch(onRejected);
```



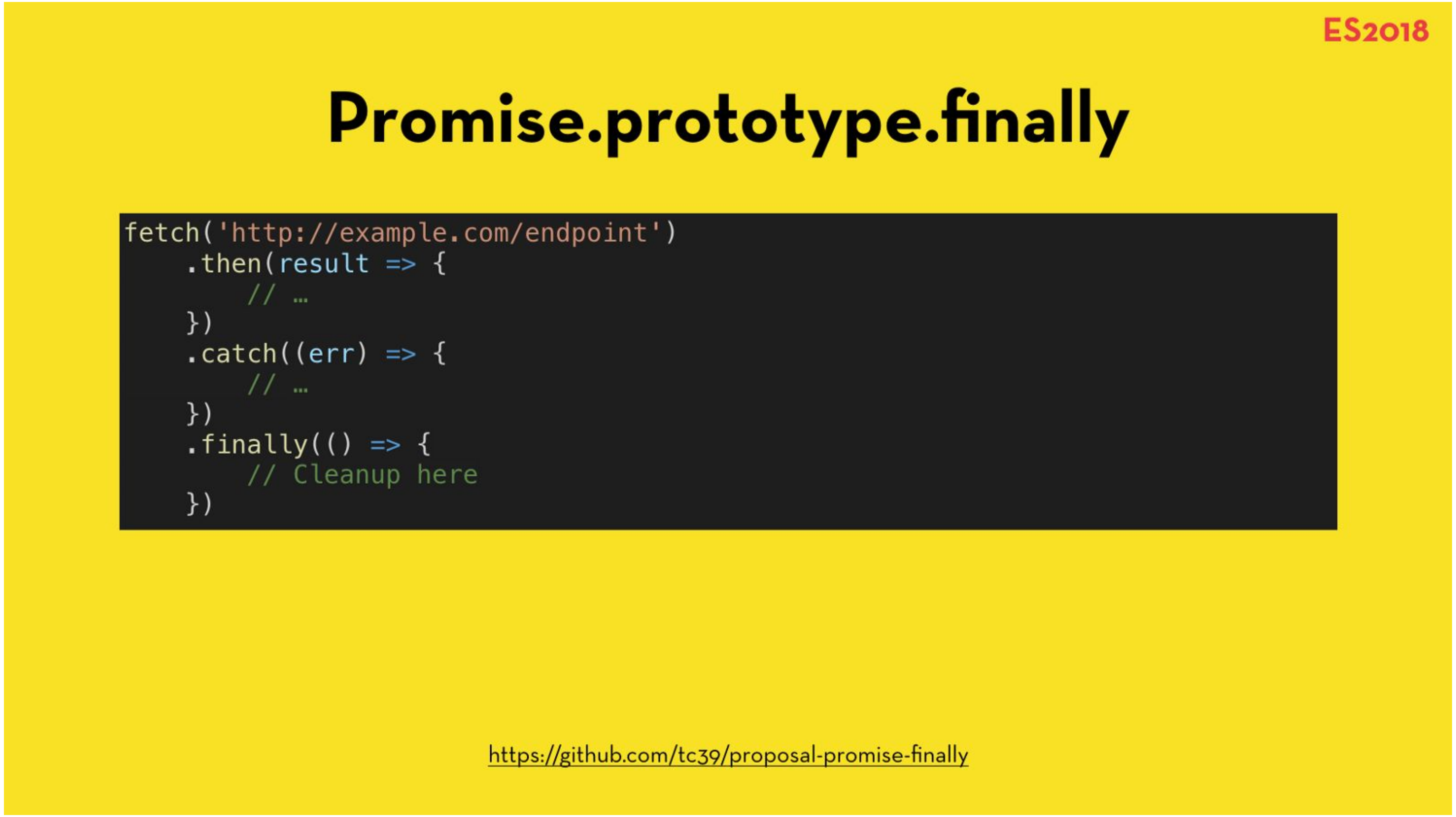
```
getData(function(a) {
 getMoreData(function(b) {
  getMoreData(function(c) {
   getMoreData(function(d) {
    getMoreData(function(e) {
     // do something
    });
   });
  });
 });
})
```

```
getData()
.then(getMoreData)
.then(getMoreData)
.then(getMoreData)
.then(getMoreData)
.then((result) => {
// do something
})
.catch((error) => {
handleError(error);
});
```

The solution to the callback-hell problem

Different Methods of the Promise

https://dev.to/hem/gif-cheatsheet-for-javascript-promise-api-methods-promise-all-promise-allsettled-promise-race-promise-any-1l2o



ES2018

Promise.prototype.finally

```
fetch('http://example.com/endpoint')
    .then(result => {
        // …
    })
    .catch((err) => {
        // …
    })
    .finally(() => {
        // Cleanup here
    })
```

https://github.com/tc39/proposal-promise-finally

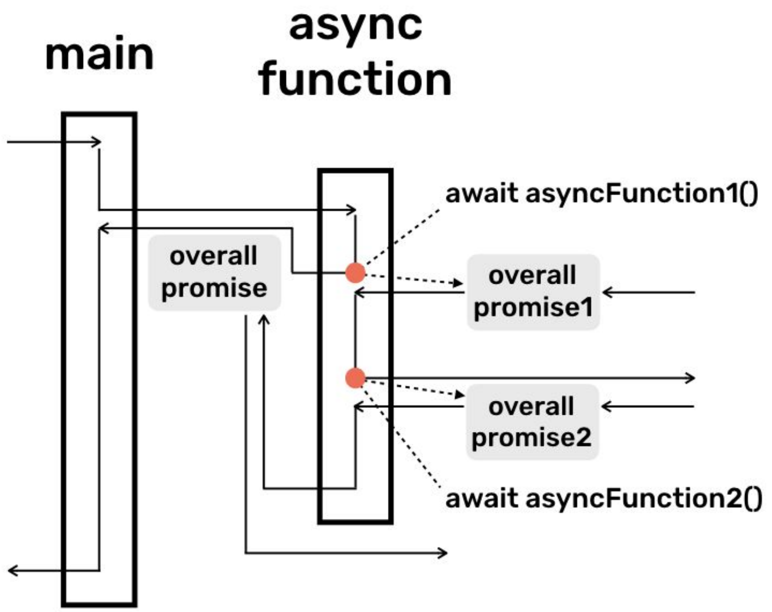
Fetching from an web api

async – await

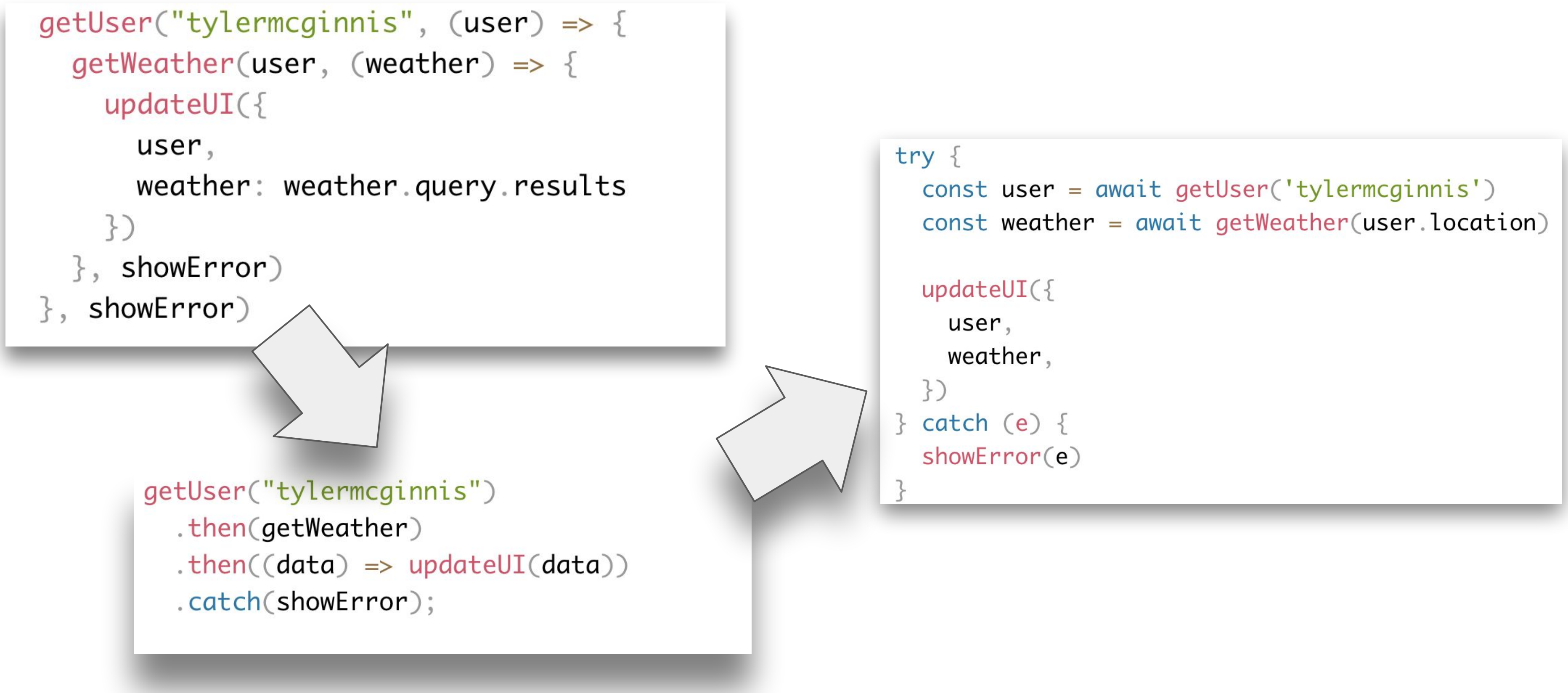
Async () => { Await }

What is async-await?

- It is just better than promise
- Promises create a new processing channel apart from the main programming flow
- It somehow makes the programming harder. You need to understand a future object and use it now
- async-await has a different approach and gives us the chance to settle async calls like sync calls. All the promise-based mechanisms and objects are handled by javascript itself, not the programmer!


main
async function
await asyncFunction1()
overall promise
overall promise1
overall promise2
await asyncFunction2()

Async calls



```
getUser("tylermcginnis", (user) => {
  getWeather(user, (weather) => {
    updateUI({
      user,
      weather: weather.query.results
    })
  }, showError)
}, showError)
```

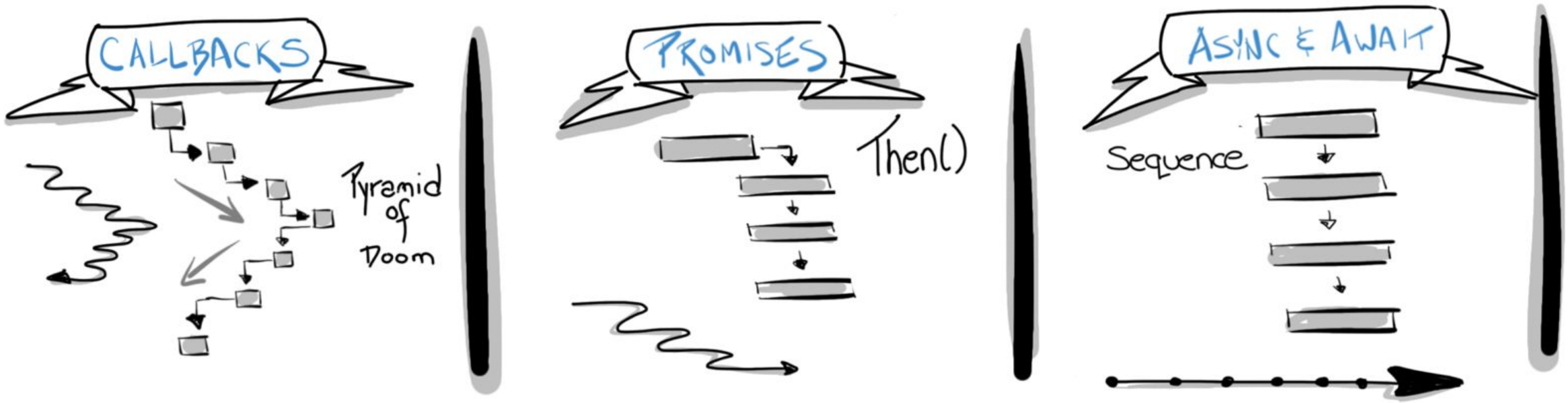
```
getUser("tylermcginnis")
  .then(getWeather)
  .then((data) => updateUI(data))
  .catch(showError);
```

```
try {
  const user = await getUser('tylermcginnis')
  const weather = await getWeather(user.location)

  updateUI({
    user,
    weather,
  })
} catch (e) {
  showError(e)
}
```

```javascript
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!"
}

f();
```

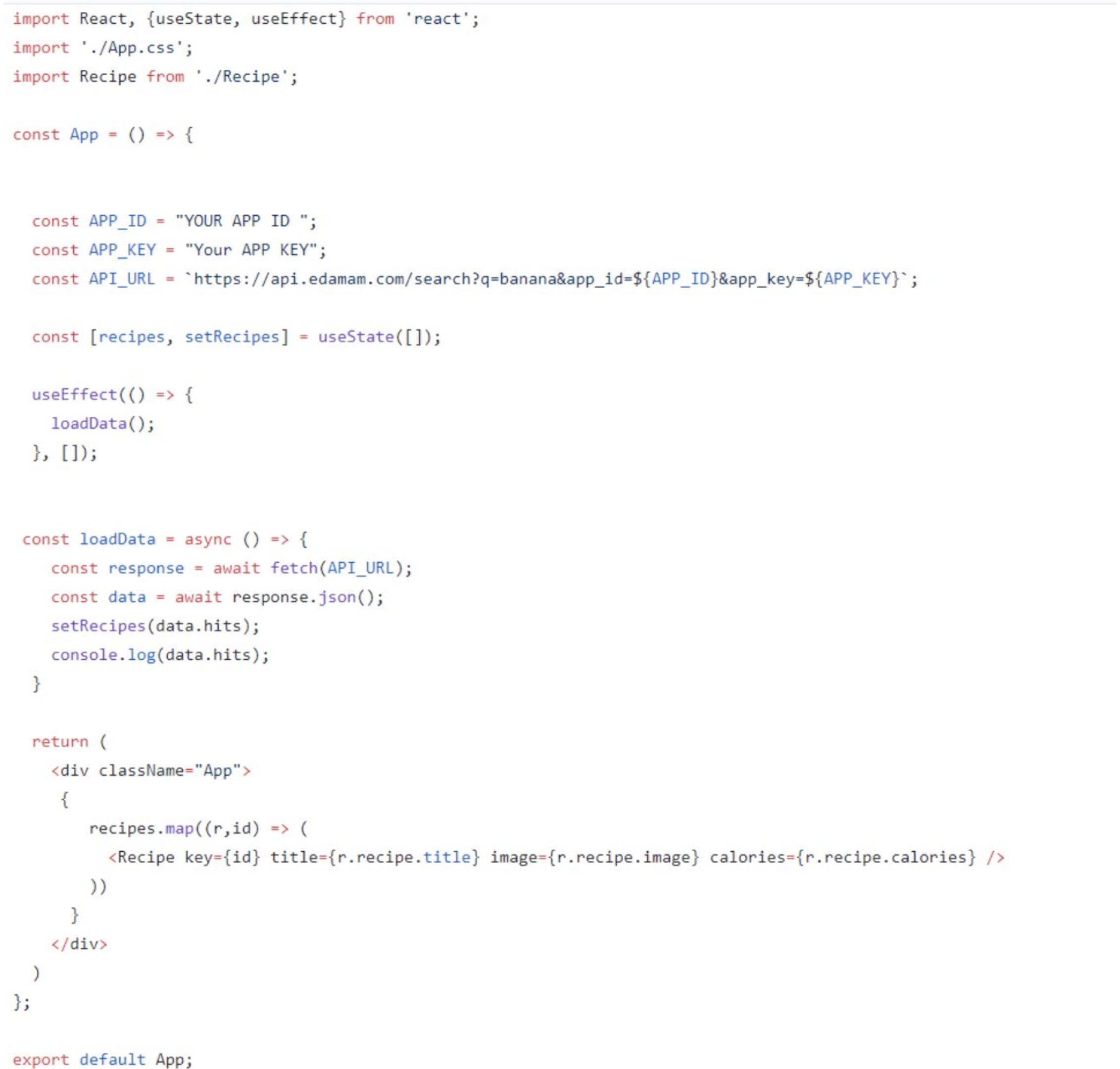
```javascript
async function showAvatar() {

  // read our JSON
  let response = await fetch('/article/promise-chaining/user.json');
  let user = await response.json();

  // read github user
  let githubResponse = await fetch(`https://api.github.com/users/${user.name}
  let githubUser = await githubResponse.json();

  // show the avatar
  let img = document.createElement('img');
  img.src = githubUser.avatar_url;
  img.className = "promise-avatar-example";
  document.body.append(img);

  // wait 3 seconds
  await new Promise((resolve, reject) => setTimeout(resolve, 3000));

  img.remove();

  return githubUser;
}

showAvatar();
```

Asynchronous TypeScript



Differences between approaches

Using async in react



```
import React, {useState, useEffect} from 'react';
import './App.css';
import Recipe from './Recipe';

const App = () => {


  const APP_ID = "YOUR APP ID ";
  const APP_KEY = "Your APP KEY";
  const API_URL = `https://api.edamam.com/search?q=banana&app_id=${APP_ID}&app_key=${APP_KEY}`;

  const [recipes, setRecipes] = useState([]);

  useEffect(() => {
    loadData();
  }, []);


 const loadData = async () => {
    const response = await fetch(API_URL);
    const data = await response.json();
    setRecipes(data.hits);
    console.log(data.hits);
  }

  return (
    <div className="App">
     {
        recipes.map((r,id) => (
          <Recipe key={id} title={r.recipe.title} image={r.recipe.image} calories={r.recipe.calories} />
        ))
      }
    </div>
  )
};

export default App;
```

Demo