ASYNCHRONOUS PROGRAMMING

by Anastasiia Derkach

softserve

Concurrency model and event loop

Callbacks

Promises

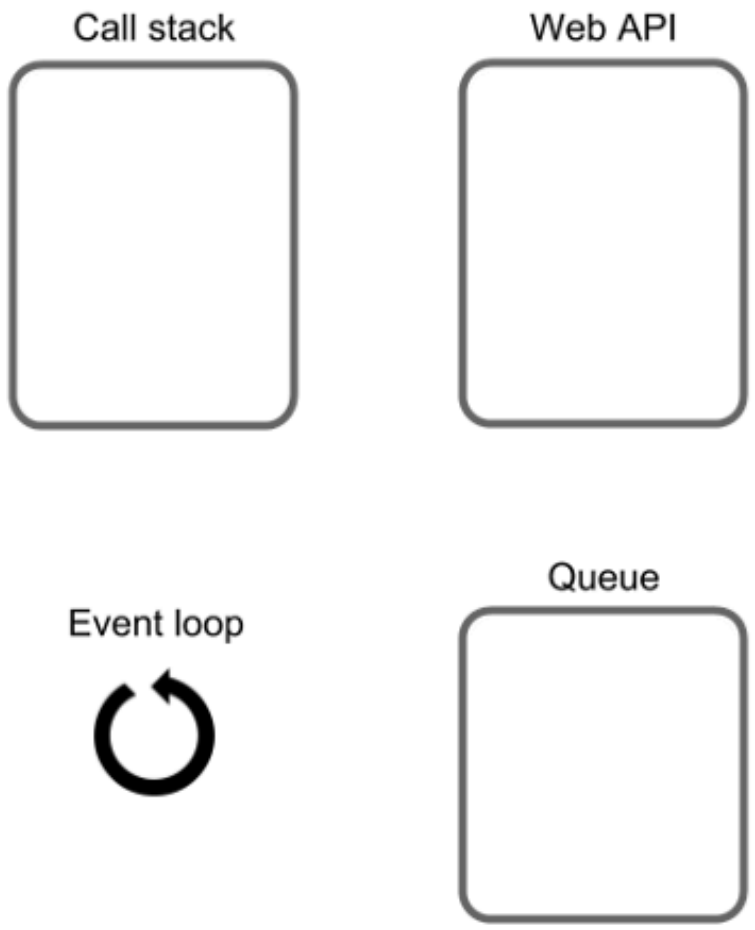
Generators

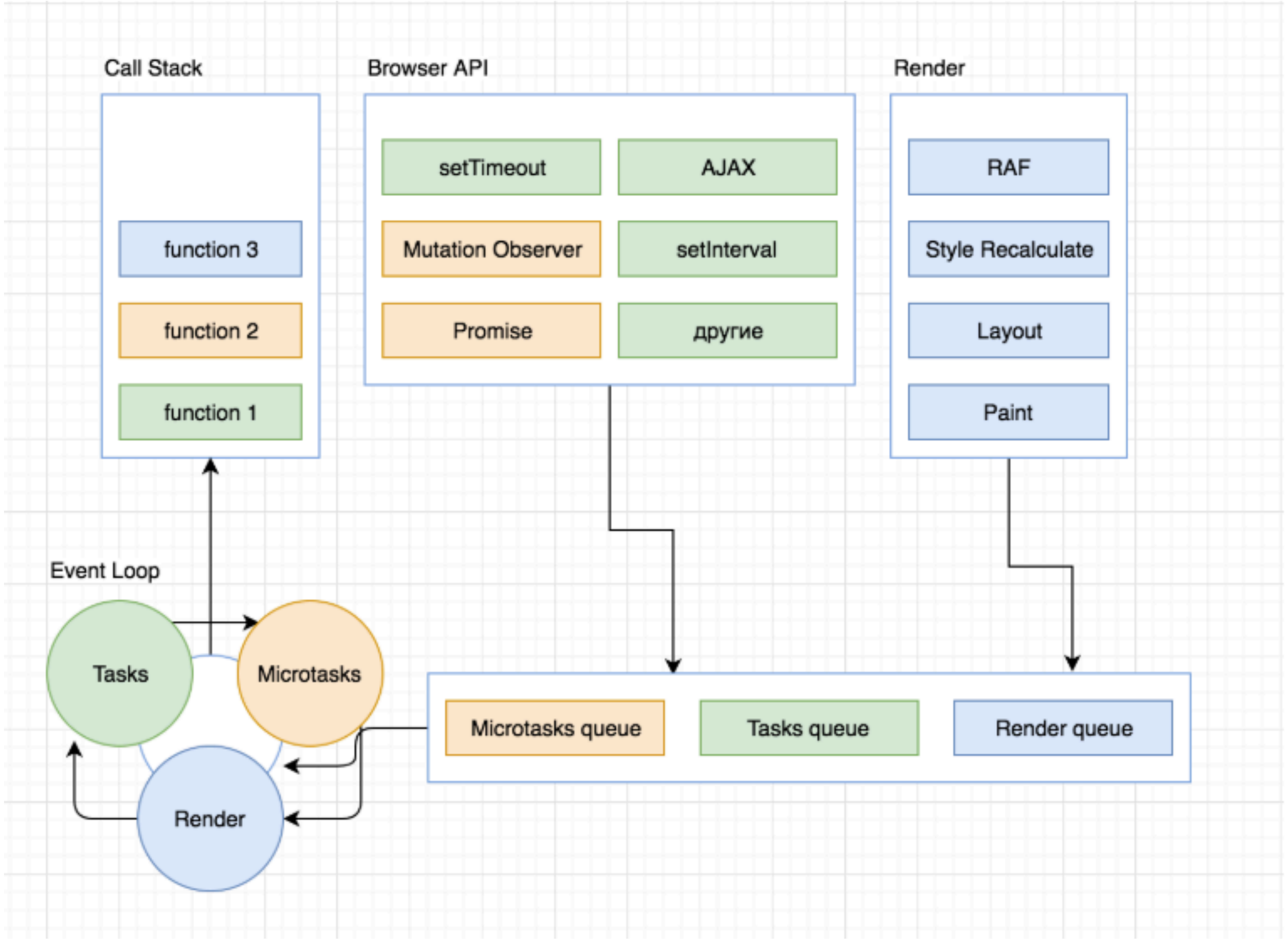
Async/Await

softserve

Javascript is a single threaded programming language, which means it has a single call stack and can do one thing at a time.

softserve

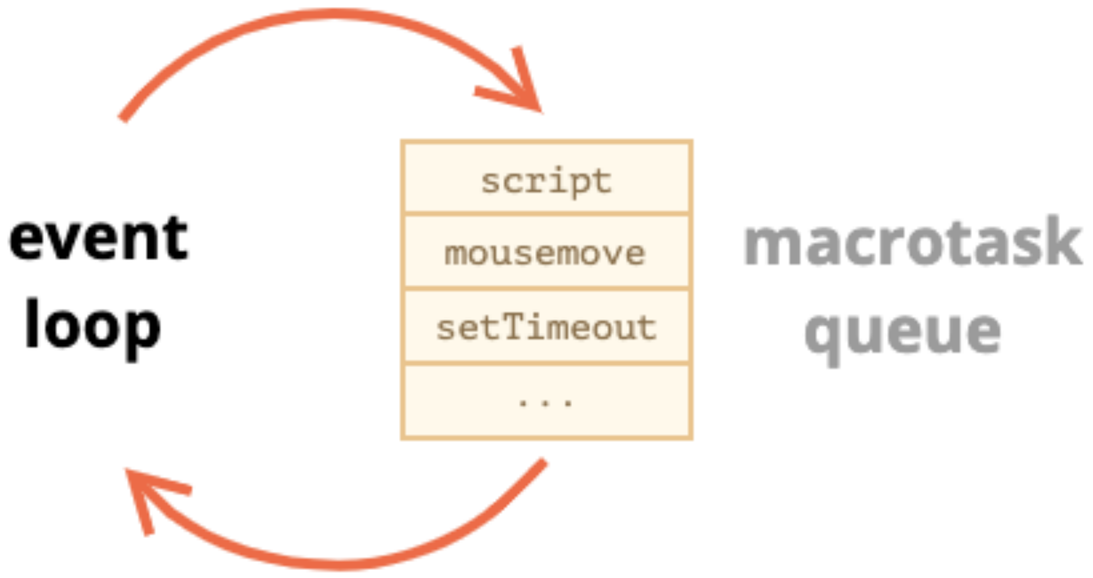

Call stack
Web API
Event loop
Queue


Call Stack
function 3
function 2
function 1
Browser API
setTimeout
AJAX
Mutation Observer
setInterval
Promise
другие
Render
RAF
Style Recalculate
Layout
Paint
Event Loop
Tasks
Microtasks
Render
Microtasks queue
Tasks queue
Render queue

softserve

TASK

- a timeout or interval created with setTimeout() or setInterval();
- when an external script <script src="..."> loads, the task is to execute it.
- an event fires, adding the event's callback function to the task queue.


event
loop
script
mousemove
setTimeout
...
macrotask
queue

first come - first served

Two more details

- rendering never happens while the engine executes a task. Doesn’t matter if the task takes a long time. Changes to DOM are painted only after the task is complete.
- If a task takes too long, the browser can’t do other tasks, process user events, so after a time it raises an alert like “Page Unresponsive” suggesting to kill the task with the whole page.

softserve

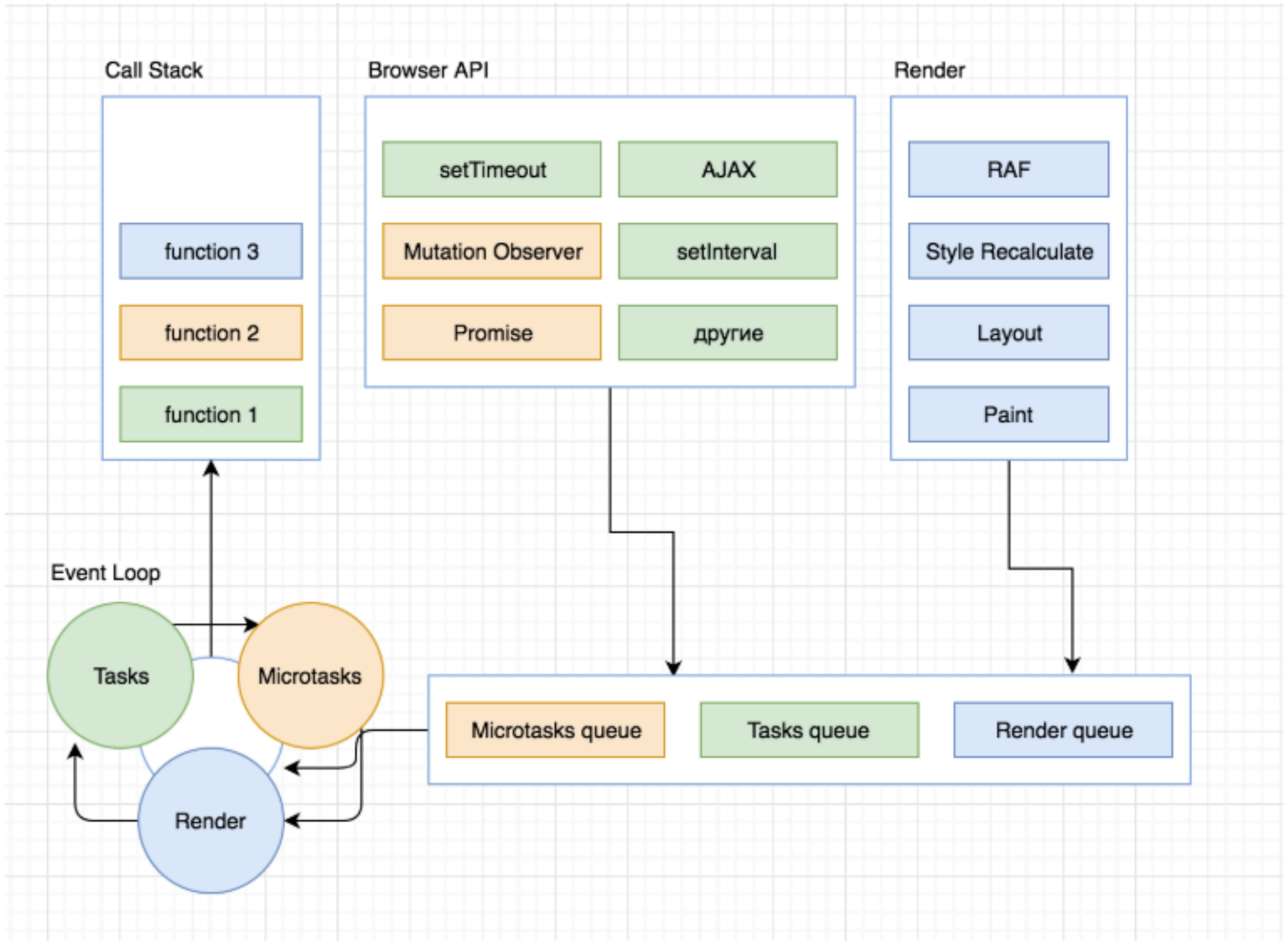
WHY TO USE?

- splitting CPU-hungry tasks
- doing something after the event

softserve

MICROTASKS

- usually created by promises
- come solely from our code
- special function queueMicrotask(func)


Call Stack
function 3
function 2
function 1
Browser API
setTimeout
AJAX
Mutation Observer
setInterval
Promise
другие
Render
RAF
Style Recalculate
Layout
Paint
Event Loop
Tasks
Microtasks
Render
Microtasks queue
Tasks queue
Render queue

softserve

```
setTimeout(() => alert("timeout"));

Promise.resolve()
  .then(() => alert("promise"));

alert("code");
```

```
let callback = () => log("Regular timeout callback has run");

let urgentCallback = () => log("*** Oh noes! An urgent callback has run!");

log("Main program started");
setTimeout(callback, 0);
queueMicrotask(urgentCallback);
log("Main program exiting");
```

```
Main program started
Main program exiting
*** Oh noes! An urgent callback has run!
Regular timeout callback has run
```

softserve

```javascript
let callback = () => log("Regular timeout callback has run");

let urgentCallback = () => log("*** Oh noes! An urgent callback has run!");

let doWork = () => {
  let result = 1;

  queueMicrotask(urgentCallback);

  for (let i=2; i<=10; i++) {
    result *= i;
  }
  return result;
};

log("Main program started");
setTimeout(callback, 0);
log(`10! equals ${doWork()}`);
log("Main program exiting");
```

```
Main program started
10! equals 3628800
Main program exiting
*** Oh noes! An urgent callback has run!
Regular timeout callback has run
```

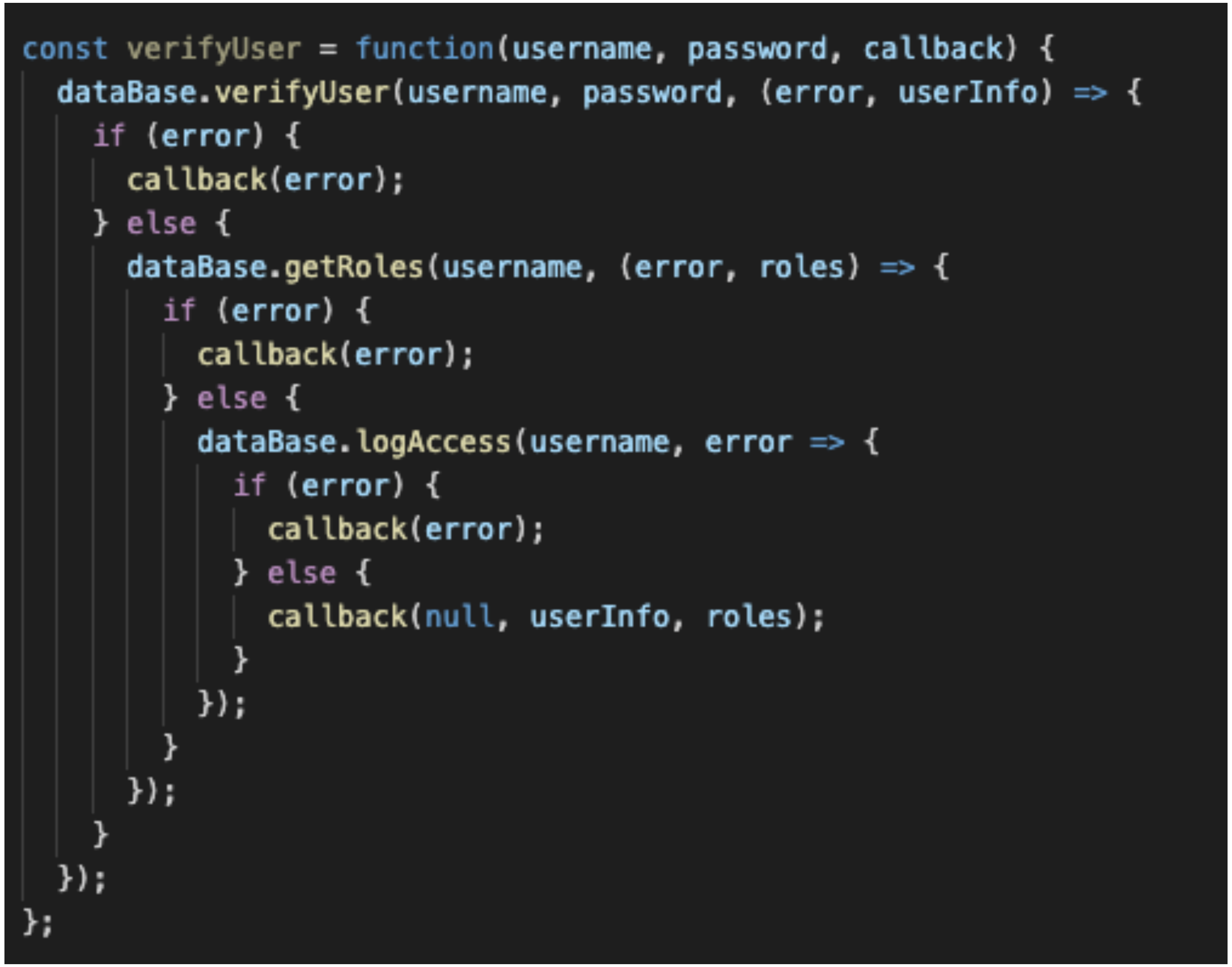
softserve

CALLBACKS

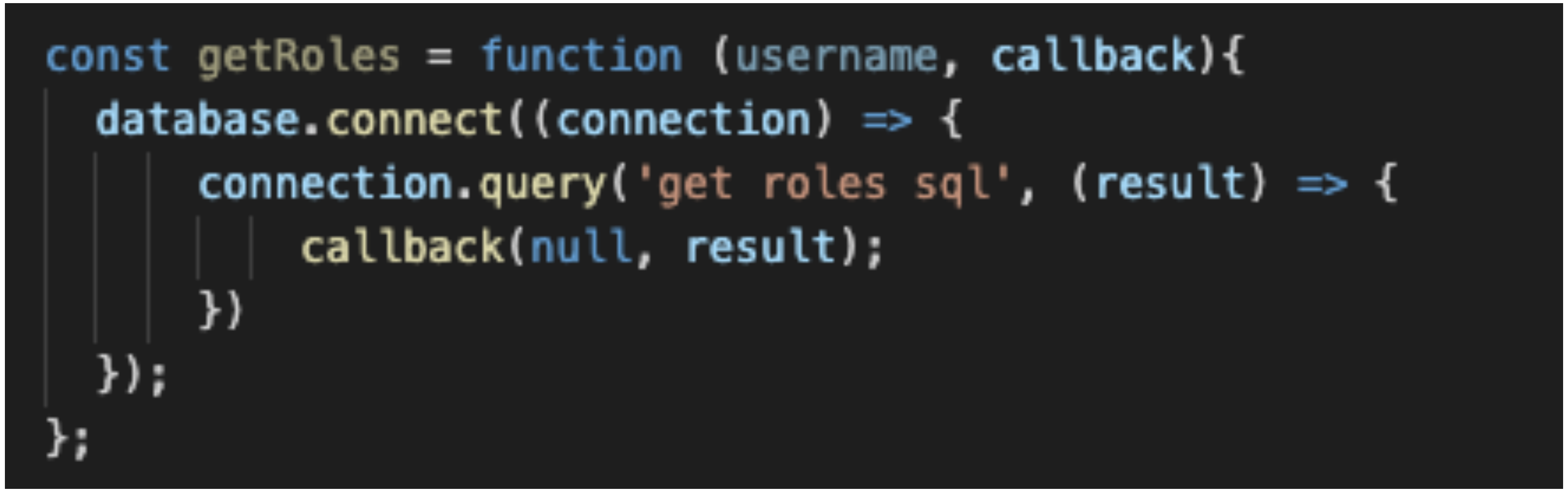
Callbacks are just the name of a convention for using JavaScript functions.

CALLBACK HELL

softserve



```
const verifyUser = function(username, password, callback) {
  dataBase.verifyUser(username, password, (error, userInfo) => {
    if (error) {
      callback(error);
    } else {
      dataBase.getRoles(username, (error, roles) => {
        if (error) {
          callback(error);
        } else {
          dataBase.logAccess(username, error => {
            if (error) {
              callback(error);
            } else {
              callback(null, userInfo, roles);
            }
          });
        }
      });
    }
  });
};
```



```
const getRoles = function (username, callback){
  database.connect((connection) => {
      connection.query('get roles sql', (result) => {
          callback(null, result);
      })
  });
};
```

WHY IS THAT BAD?

softserve

- difficult to maintain
- the DRY principle has absolutely no value in this case
- error handling

softserve

FIXED THIS?

- don't nest functions. Give them names and place them at the top level of your program
- handle **every single error** in every one of your callbacks
- splitting your code into small pieces

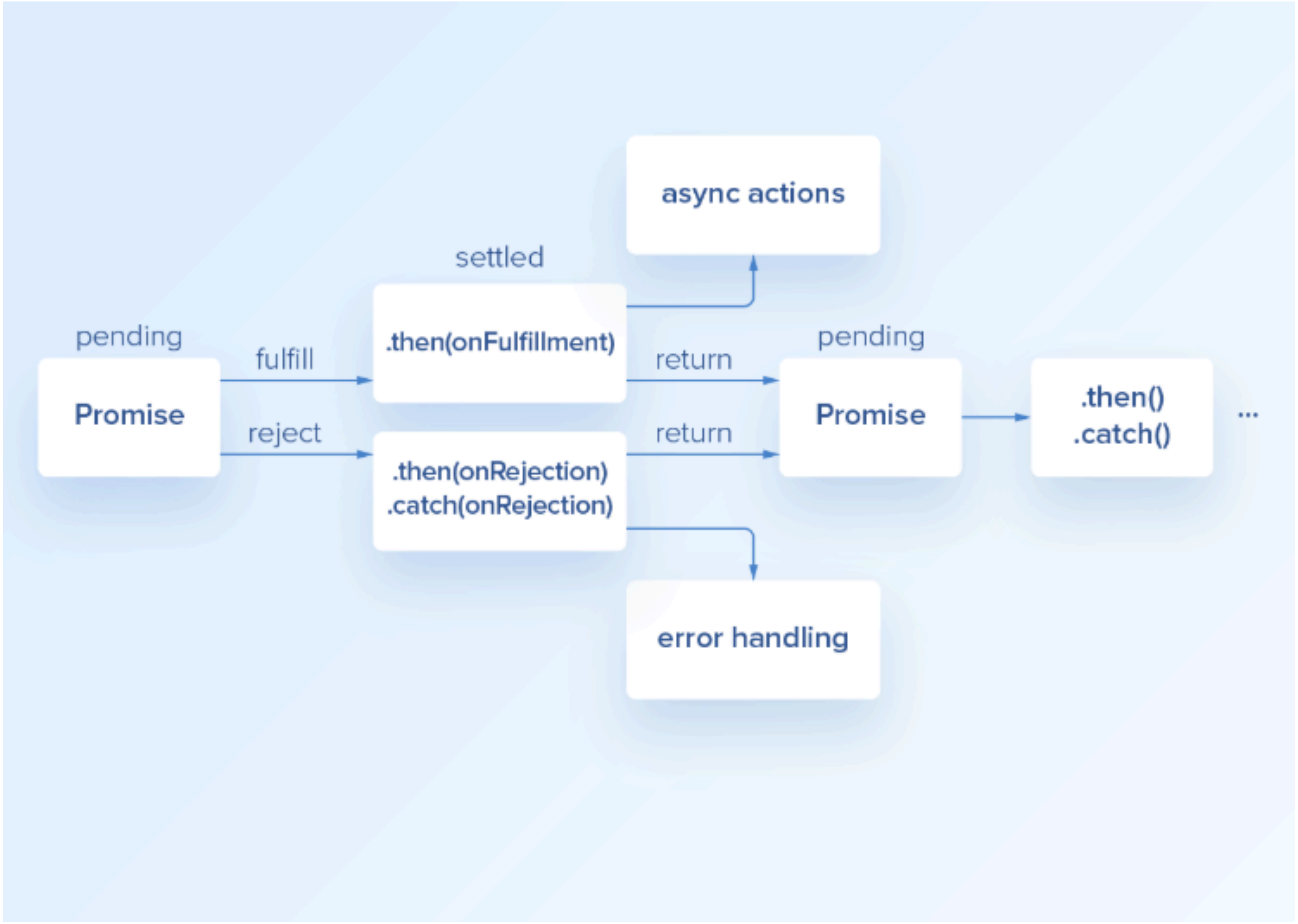
soft**serve**

PROMISES

softserve

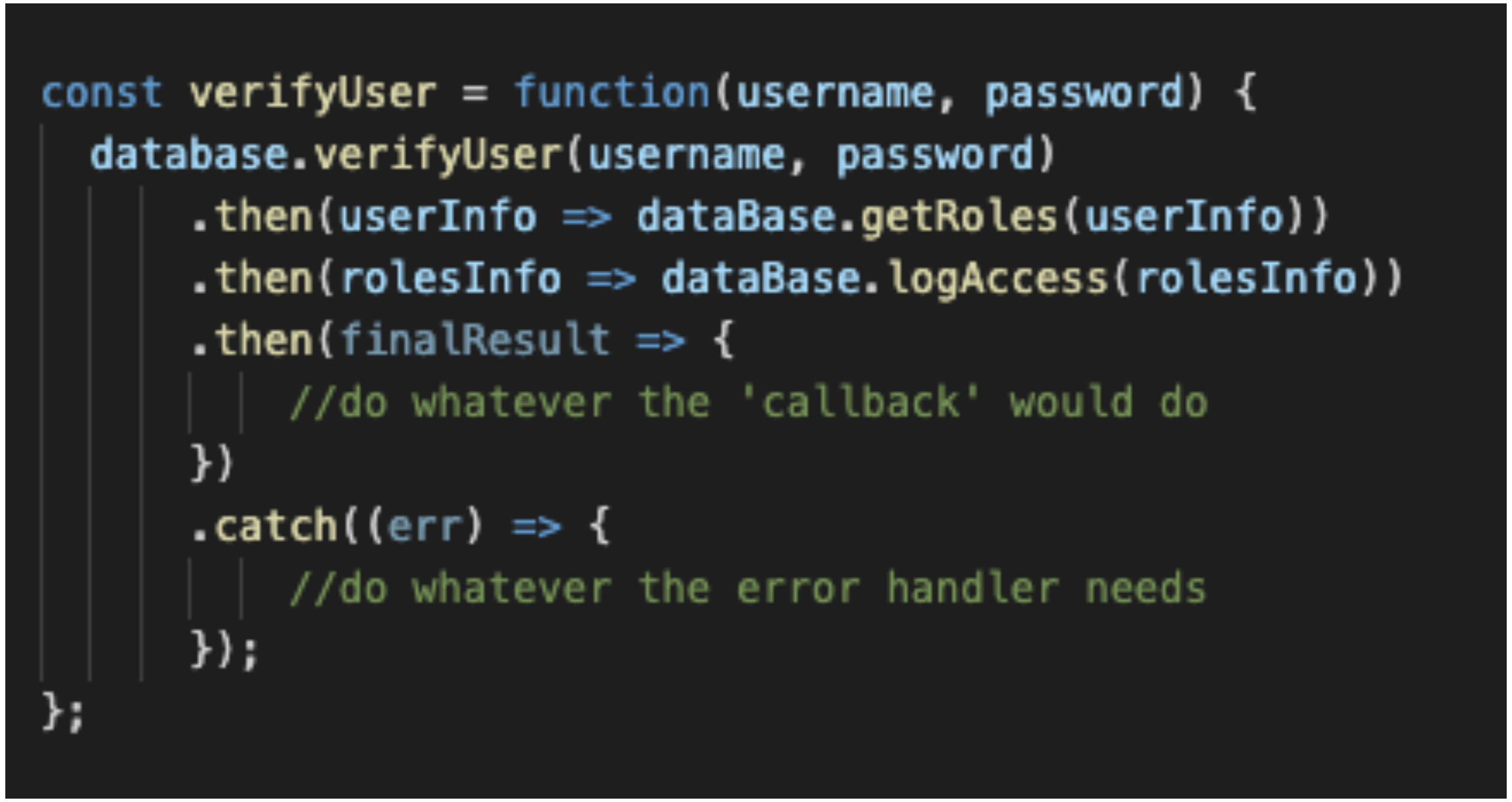
A promise represents the eventual result of an asynchronous operation.

Promises utilize callbacks as well.


async actions
settled
pending
fulfill
.then(onFulfillment)
return
pending
Promise
reject
return
Promise
.then()
.catch()
...
.then(onRejection)
.catch(onRejection)
error handling

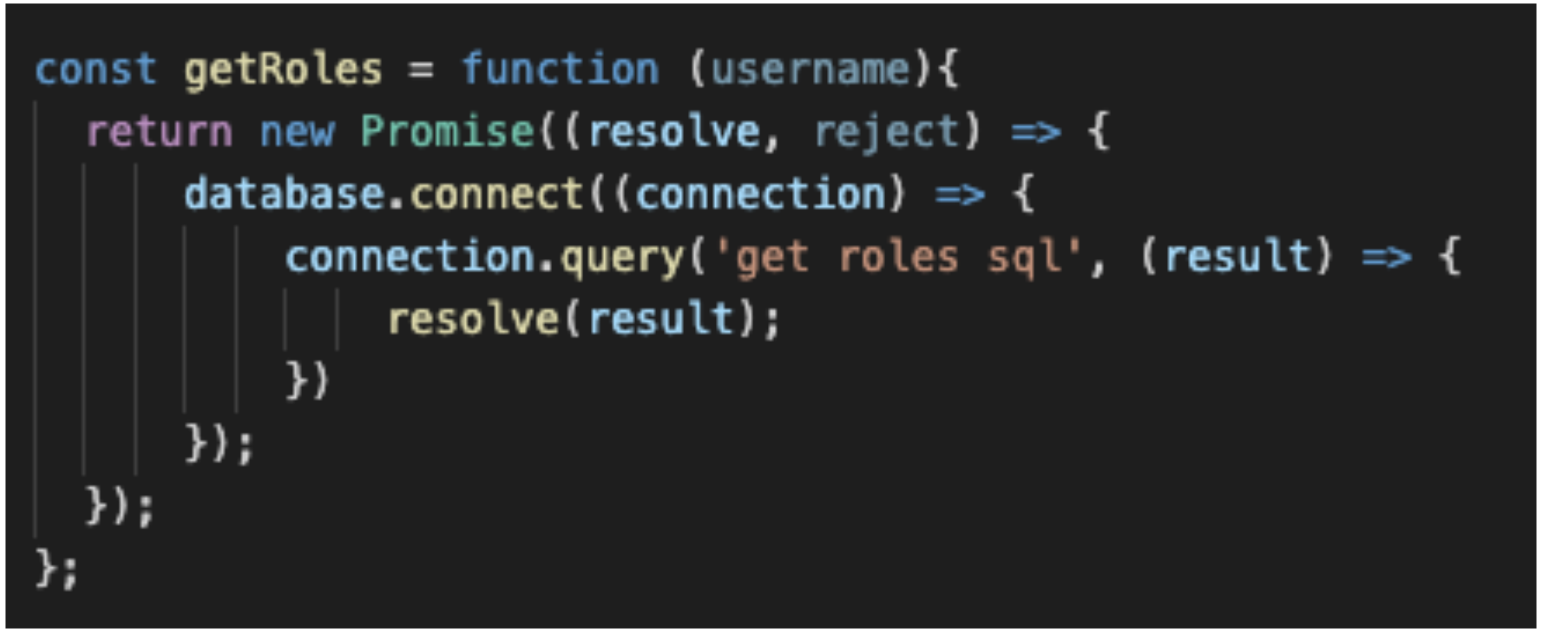
softserve

```
let myFirstPromise = new Promise((resolve, reject) => {
  // We call resolve(...) when what we were doing asynchronously was
successful, and reject(...) when it failed.
  // In this example, we use setTimeout(...) to simulate async code.
  // In reality, you will probably be using something like XHR or an HTML5 API.
  setTimeout( function() {
    resolve("Success!")  // Yay! Everything went well!
  }, 250)
})
```



```
const verifyUser = function(username, password) {
  database.verifyUser(username, password)
      .then(userInfo => dataBase.getRoles(userInfo))
      .then(rolesInfo => dataBase.logAccess(rolesInfo))
      .then(finalResult => {
        //do whatever the 'callback' would do
      })
      .catch((err) => {
        //do whatever the error handler needs
      });
};
```

softserve



```javascript
const getRoles = function (username){
  return new Promise((resolve, reject) => {
    database.connect((connection) => {
      connection.query('get roles sql', (result) => {
        resolve(result);
      })
    });
  });
};
```

METHODS

Promise.all(iterable) - wait for all promises to be resolved, or for any to be rejected.

Promise.allSettled(iterable) - wait until all promises have settled (each may resolve or reject).

Promise.race(iterable) - wait until any of the promises is resolved or rejected.

Promise.reject(reason) - Returns a new Promise object that is rejected with the given reason.

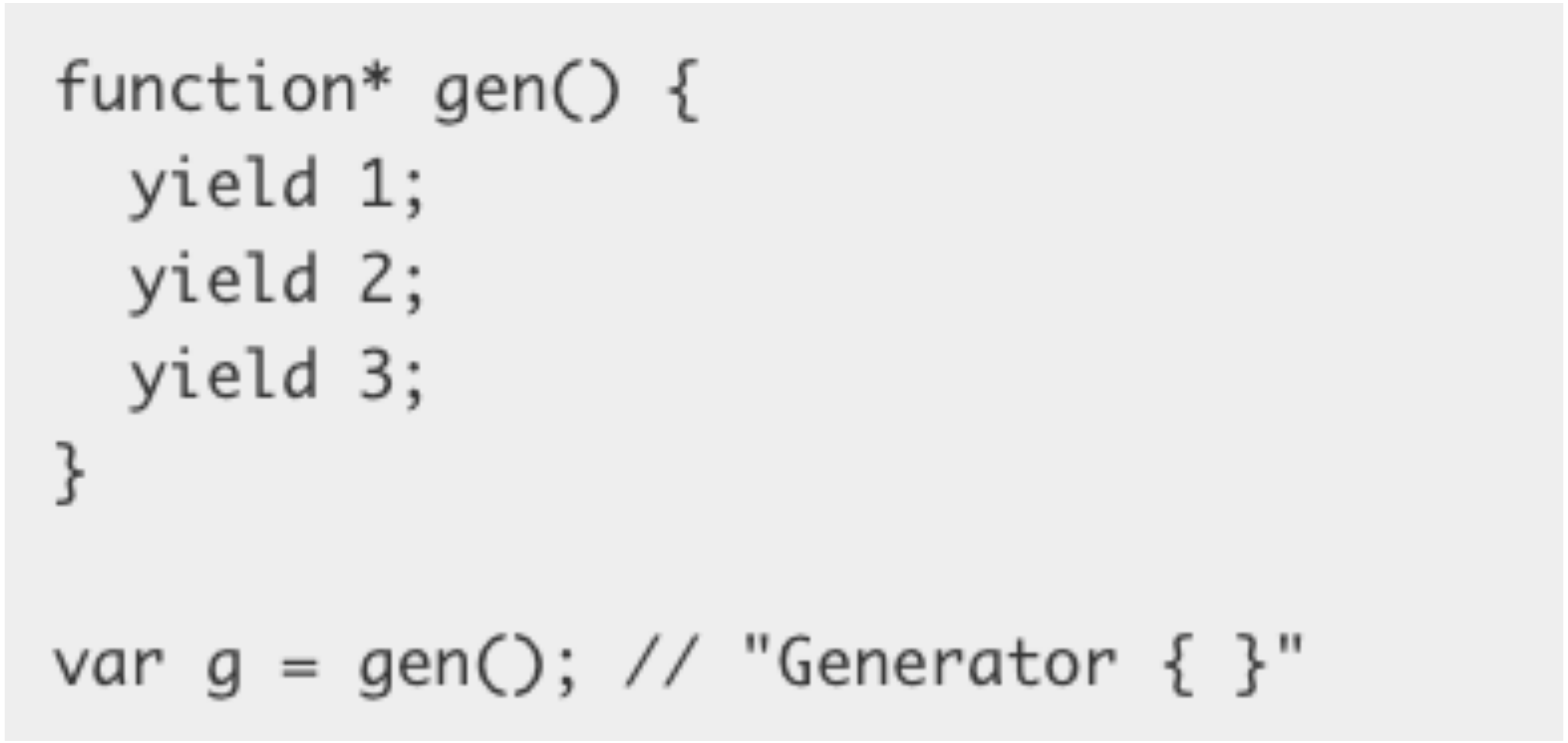
Promise.resolve(value) - returns a new Promise object that is resolved with the given value.

CREATING PROMISES

GENERATORS

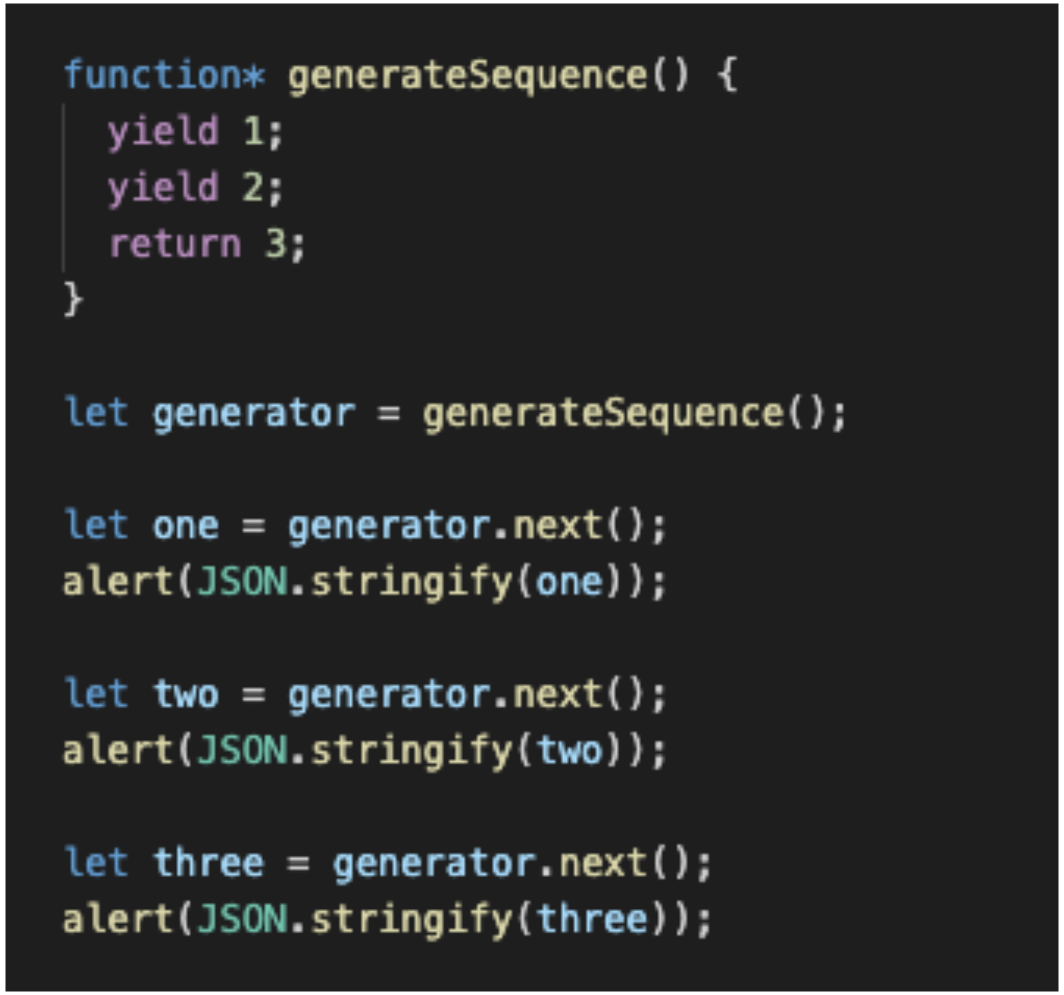
They were introduced in ES6 *(also known as ES2015)*.

Wouldn't it be nice, that when you execute your function, you could pause it at any point, calculate something else, do other things, and then return to it, even with some value and continue?

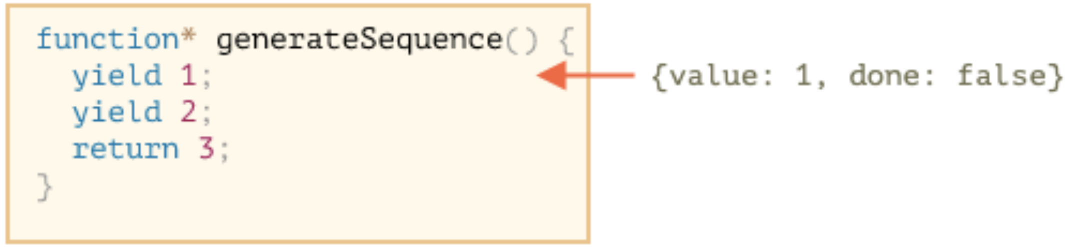


```
function* gen() {
  yield 1;
  yield 2;
  yield 3;
}

var g = gen(); // "Generator { }"
```

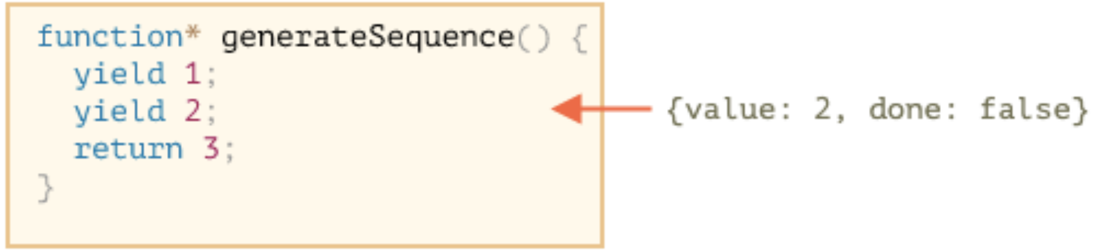
softserve



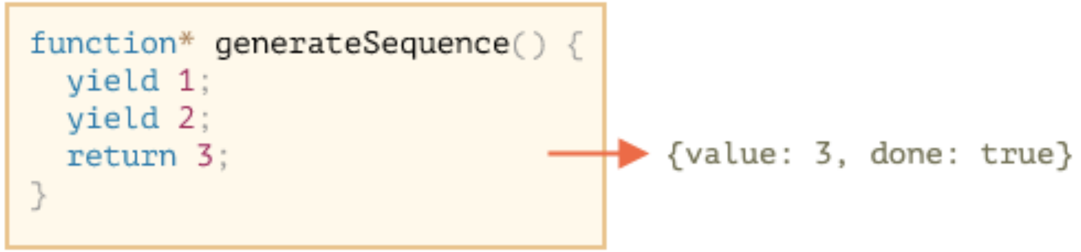
```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}

let generator = generateSequence();

let one = generator.next();
alert(JSON.stringify(one));

let two = generator.next();
alert(JSON.stringify(two));

let three = generator.next();
alert(JSON.stringify(three));
```



```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}
```



```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}
```



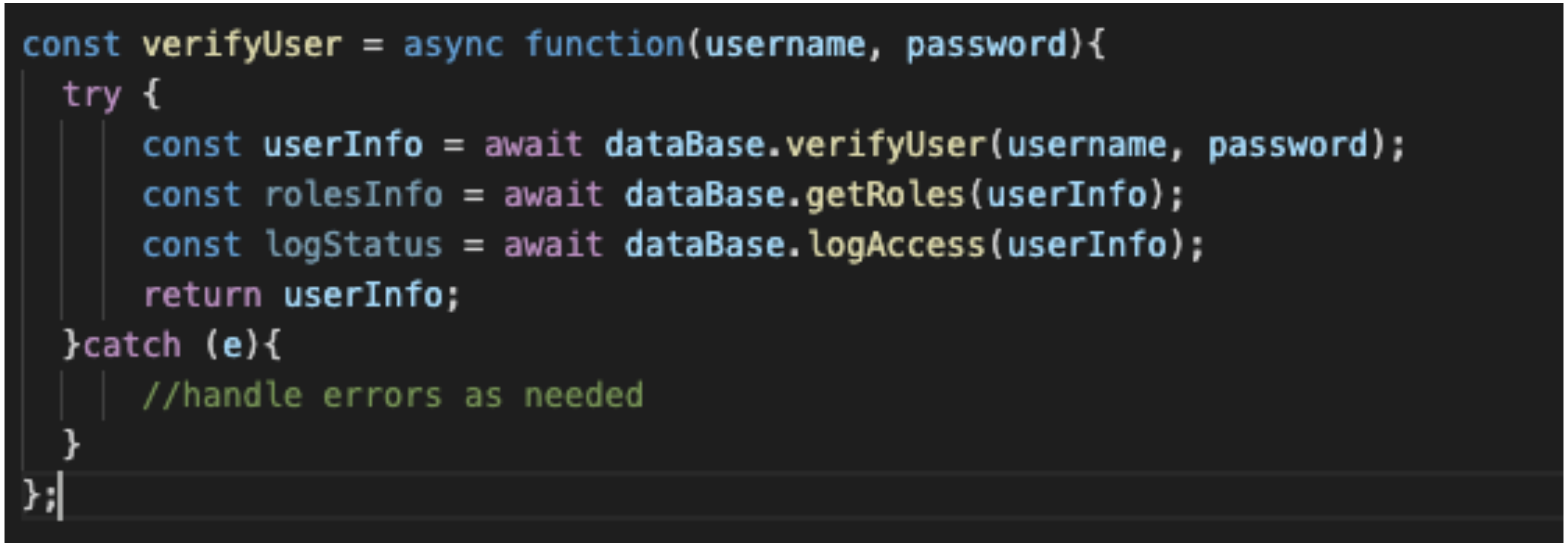
```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}
```

softserve

WHY TO USE

ASYNC/AWAIT

softserve



```
const verifyUser = async function(username, password){
  try {
      const userInfo = await dataBase.verifyUser(username, password);
      const rolesInfo = await dataBase.getRoles(userInfo);
      const logStatus = await dataBase.logAccess(userInfo);
      return userInfo;
  }catch (e){
      //handle errors as needed
  }
};
```

Async is for declaring that a function will handle asynchronous operations and await is used to declare that we want to **“await”** the result of an asynchronous operation inside a function that has the async keyword.

softserve

ERROR HANDLING WITH ASYNC/AWAIT

softserve

```
async function getSomeData(value){
    try {
        const result = await fetchTheData(value);
        return result;
    }
    catch(error){
        // Handle error
    }
}
```

softserve

```
async function fetchTheFirstData(value){
    return await get("someUrl", value);
}


async function fetchTheSecondData(value){
    return await getFromDatabase(value);
}


async function getSomeData(value){
    try {
    const firstResult = await fetchTheFirstData(value);
        const result = await
fetchTheSecondData(firstResult.someValue);
        return result;
    }
    catch(error){
        // Every error thrown in the whole “awaitable” chain will
end up here now.
    }
}
```

The asynchronous I/O operations will still be processed in parallel and the code handling the responses in the async functions will not be executed until that asynchronous operation has a result.

Regardless of the method you choose, always handle every error and keep your code simple.

softserve

RESOURCES USED

- https://medium.com/codebuddies/getting-to-know-asynchronous-javascript-callbacks-promises-and-async-await-17e0673281ee
- https://blog.risingstack.com/asynchronous-javascript/
- http://callbackhell.com/
- https://blog.risingstack.com/node-js-at-scale-understanding-node-js-event-loop/
- https://javascript.info/event-loop
- https://developer.mozilla.org/en-US/docs/Web/API/HTML_DOM_API/Microtask_guide

softserve

THANK YOU

softserve