Fighting Spaghetti Code with Promises and Generators

Michael Jaser
michael.jaser@peerigon.com

@mmeaku

peerigon

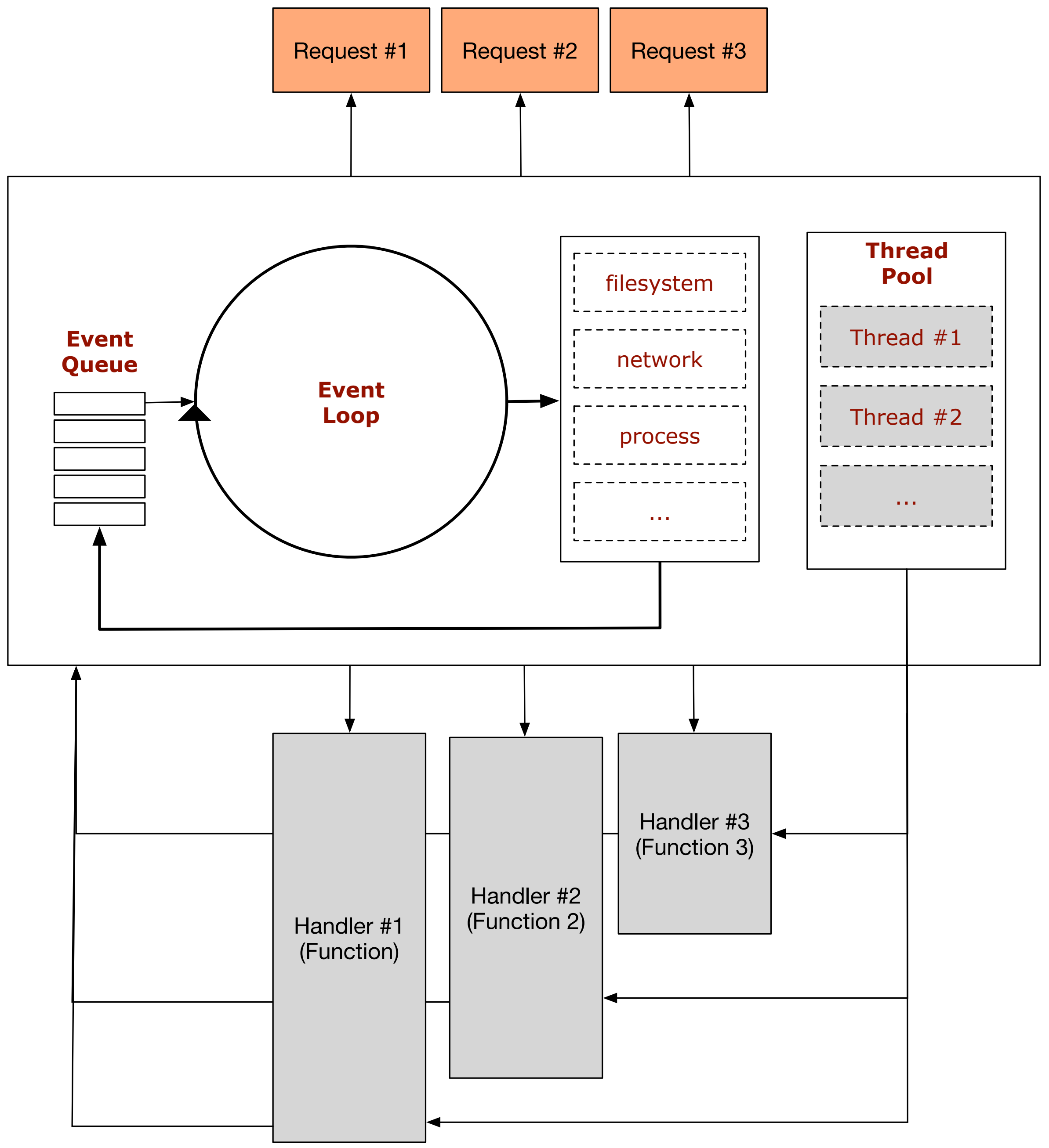
Asynchronous / Non Blocking

- non-blocking IO
- no need to handle threads manually
- high IO concurrency
- single threaded computation

- perfect fit for for high concurrency applications

BUT!

- asynchronous code is harder to read
- flow control is complicated



Asynchronous Function Calls

```
fs.readFile("./data.json", function(err, data) {

      if(err) {
        throw err;
      }

      console.log(data.toString());

});
```

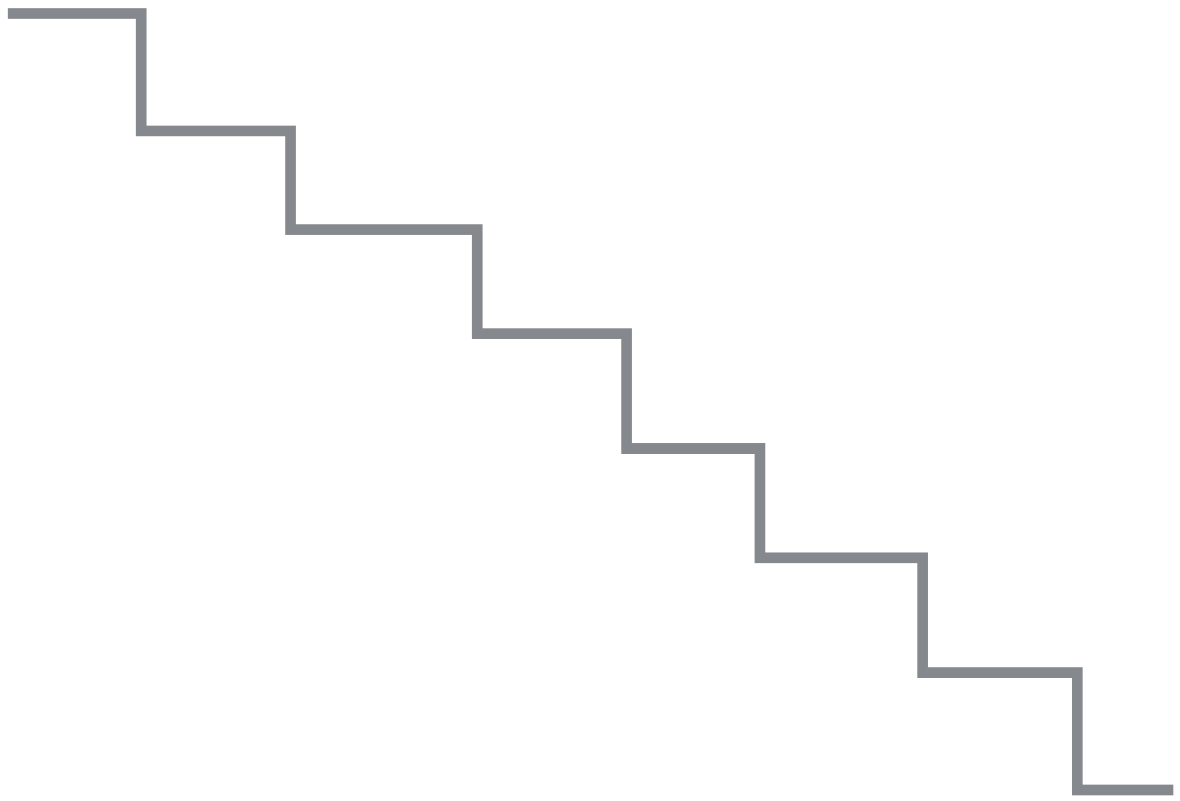
Anonymous callback function

Handle potential error

Result of the operation

Nested Asynchronous Function Calls

```
function storeFileInDb(fileName, callback) {

    //1. load the file
    fs.readFile(fileName, function(err, data) {

        if(err) {
            callback(err);
            return;
        }

        //2. save the file content to the database
       db.addEntry(data, function(err, result) {

           if(err) {
               callback(err);
               return;
           }

           callback(null, result);
       })
    });
}
```



Parallel function calls

```
function loadArticles(callback) {

    var results = [];
    var expectedCalls = 2;
    var canceled = false;

    function onLoaded(err, result) {
        if (err) {
            canceled = false;
            callback(err);
            return;
        }

        results.push(result);

        if (results.length === expectedCalls && !canceled) {
            callback(null, results);
        }
    }

    http.get("http://de.wikipedia.org/wiki/Node.js", function (res) {
        onLoaded(null, res)
    })
    .on("error", onLoaded);

    http.get("http://de.wikipedia.org/wiki/JavaScript", function (res) {
        onLoaded(null, res)
    })
    .on("error", onLoaded);
}
```

← Request 1

← Request 2

Source: https://upload.wikimedia.org/wikipedia/commons/9/93/Spaghetti.jpg

Spaghetti all the way down....

Promises to the rescue

"In computer science, future, promise, and delay refer to constructs used for synchronization in some concurrent programming languages. They describe an object that acts as a proxy for a result that is initially unknown, usually because the computation of its value is yet incomplete."

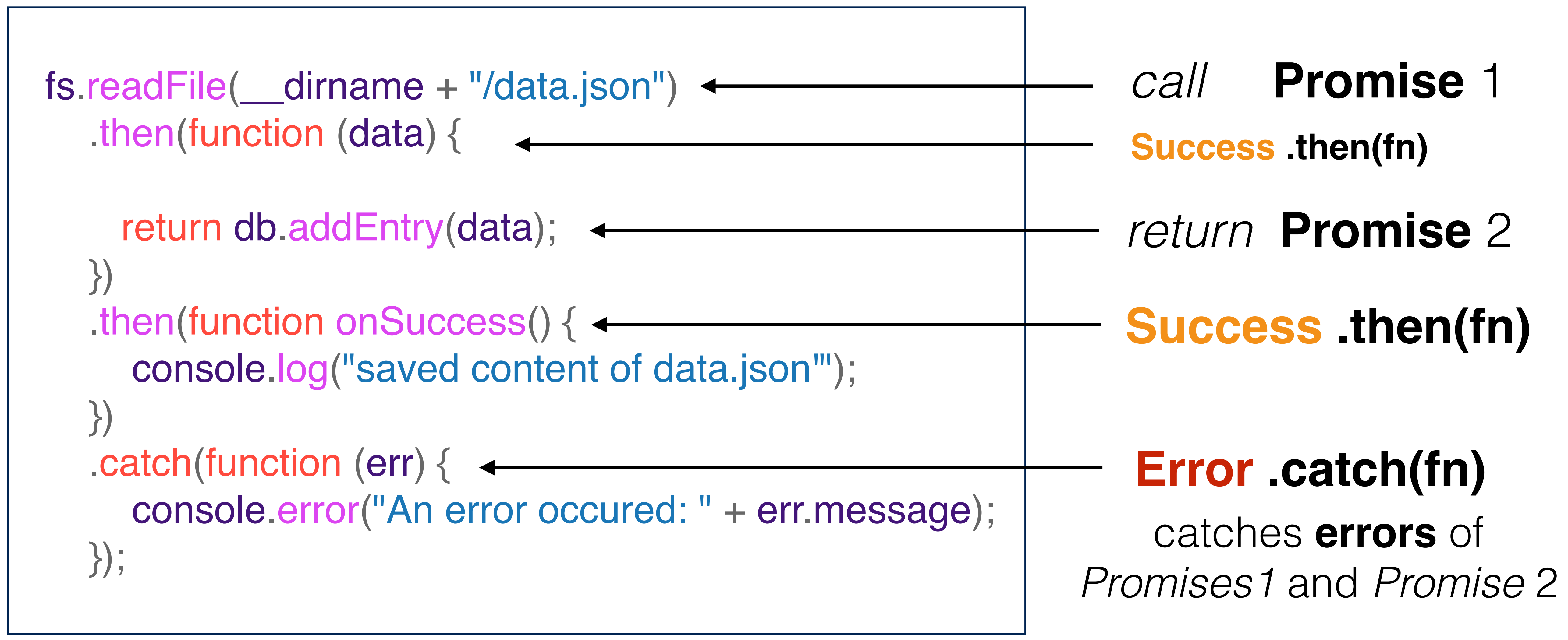
Source: https://en.wikipedia.org/wiki/Futures_and_promises (2015-06-30)

Function Call using **Promises**

```
fs.readFile("./data.json")
    .then(function onSuccess(data) {
        console.log(data.toString());
    })
    .catch(function onError(err) {
        console.error(err.message);
    });
```

Success	**.then(fn)**
Error	**.catch(fn)**

Chained Function Calls using **Promises**



```
fs.readFile(__dirname + "/data.json")
    .then(function (data) {

      return db.addEntry(data);
    })
    .then(function onSuccess() {
        console.log("saved content of data.json"");
    })
    .catch(function (err) {
        console.error("An error occured: " + err.message);
    });
```

Parallel Function Calls using **Promises**

```
Promise.all([
    http.get("http://de.wikipedia.org/wiki/Node.js"),
    http.get("http://de.wikipedia.org/wiki/JavaScript")
])
    .then(function (results) {
        //results[0] => http://de.wikipedia.org/wiki/Node.js
        //results[1] => http://de.wikipedia.org/wiki/JavaScript
    })
    .catch(function (err) {
        //gets called if an error happens in a handler
    });
```

Implementing a Promise

```
function readFile(fileName) {

    return new Promise(function(resolve, reject) {

        fs.readFile(function(err, content) {

            if(err) {
                reject(err);
                return;
            }

            resolve(content);
        });
    });
}
```

return new **Promise()**

reject on **Error**

resolve with **Result**

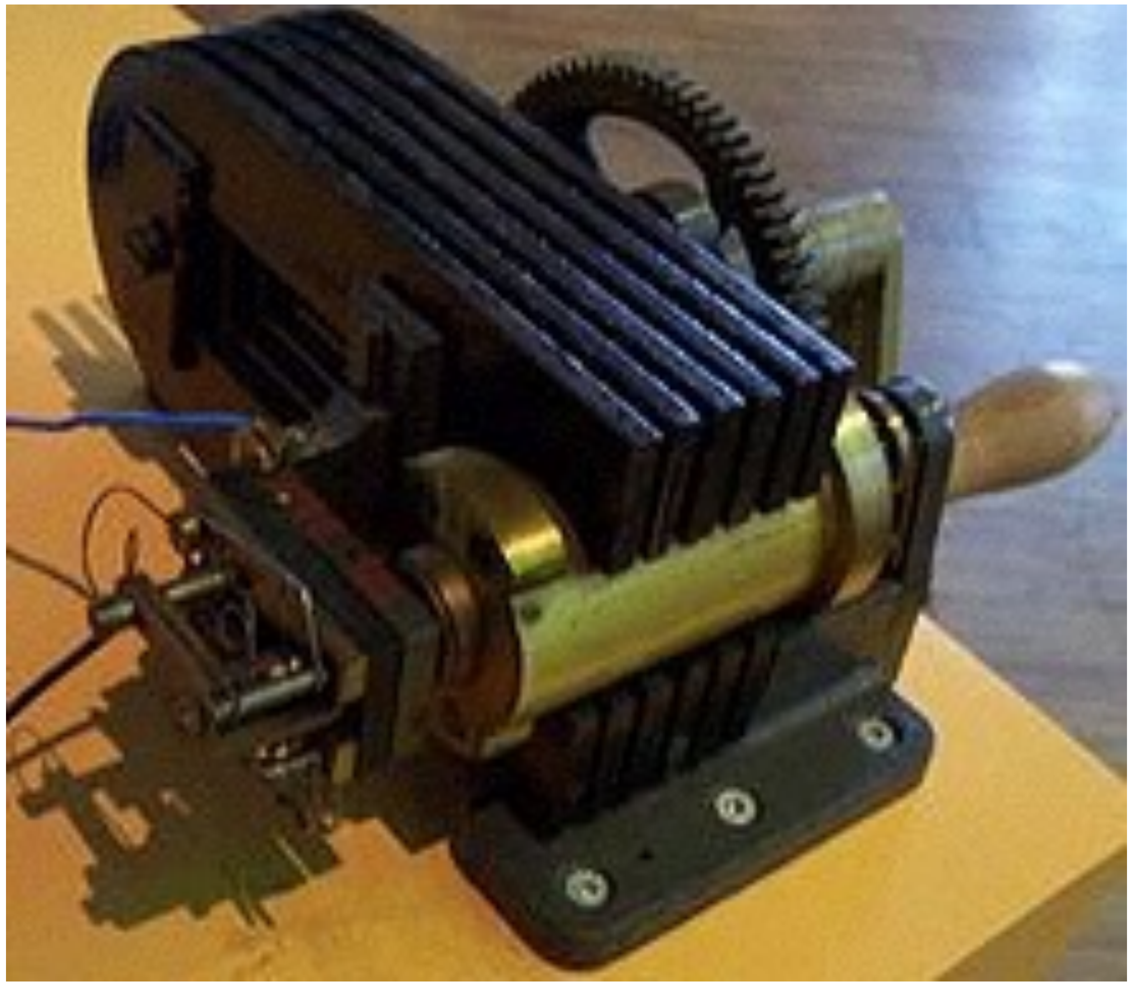
Promises (ES 2015)

- easier to read than callback
- less nesting needed
- simple interface
- standardized in ES2015
    - supported by Browsers
    - supported by Node.js

Generators (ES 2015)

In computer science, a generator is a special routine that can be used to control the iteration behaviour of a loop. In fact, all generators are iterators. A generator is very similar to a function that returns an array, in that a generator has parameters, can be called, and generates a sequence of values. However, instead of building an array containing all the values and returning them all at once, a generator yields the values one at a time, which requires less memory and allows the caller to get started processing the first few values immediately. In short, a generator looks like a function but behaves like an iterator.

Source: https://en.wikipedia.org/wiki/Generator_(computer_programming) (2015-06-30)



Generator as dynamic Iterators

Produce

```
function* fibGen (n) {
    var current = 0, next = 1, swap;
    for (var i = 0; i < n; i++) {
        swap = current;
        current = next;
        next = swap + next;
        yield current
    }
}
```

Consume

```
var gen = fibGen(20);

console.log(gen.next());
console.log(gen.next());
console.log(gen.next());

/*
 { value: 1, done: false }
 { value: 1, done: false }
 { value: 2, done: false }
 */
```

Generators for asynchronous functions

```
//create a generator function which yields promises
function* routeHandler(userId) {

   let user = yield getUser(userId);
   let hash = yield createHash(user.password);
}

//call the generator function using a wrapper
//the wrapper calls next on every resolve
async(routeHandler("abc"));
```

Call **getUser**
Wait till it *resolves*

Call **createHash**
Wait till it *resolves*

Wrap function call with **async** helper

HOLY GENERATORS BATMAN
HOW DO THEY WORK
memegenerator.net

```
function spawn(generator) {

    function handle(result) {
        // result => { done: [Boolean], value: [Object] }

        //was last yield
        if (result.done) {
            return Promise.resolve(result.value);
        }

        return Promise.resolve(result.value)
            .then(
            //call the next promise recursively
            function onSuccess(res) {
                return handle(generator.next(res));
            },
            //pass err to generator
            function onError(err) {
                return handle(generator.throw(err));
            });
    }

    try {
        //call next, which returns a promise
        return handle(generator.next());
    } catch (err) {
        return Promise.reject(err);
    }
}
```

Async Wrapper

Generator
+
Promise

1. Call yielded **Promise 1**
2. When **Promise 1** *resolves*, *return* the result calling *next(res)*
3. *C*all the next yielded **Promise n** *recursively*
4. If no more **Promises** are left, resolve

ES 2016: **Async / Await**

```
function getUserPromise(userId)

async function routeHandler(userId) {

    try {
        let user = await getUser(userId);
        let hash = await createHash(user.password);

        console.log("hash is " + hash);
    }
    catch (err) {
        console.error(err.message);
    }
}

routeHandler("user-1");
```

Same as before but without additional libraries. No more need for **async** wrapper function as the native **async** keyword can be used

What about Browsers?

.fetch()

"The fetch() method of the GlobalFetch interface starts the process of fetching a resource. This returns a promise that resolves to the Response object representing the response to your request."

https://developer.mozilla.org/de/docs/Web/API/GlobalFetch/fetch

@fetchstandard

fetch.spec.whatwg.org

Fetch GitHub Keys

```
/**
 * fetch SSH keys form GitHub
 * @param userName
 * @returns {Promise}
 */
function fetchKey(userName) {
    return fetch("https://github.com/" + userName + ".keys")
        .then(function (response) {
            return response.text();
        });
}
```

Fetch GitHub Members

```
/**
 * fetch members or organization from GitHub
 * @param orgName
 * @returns {Promise}
 */
function fetchMembers(orgName) {

    return fetch(`https://api.github.com/orgs/${orgName}/members`)
        .then(function (res) {
            return res.json();
        })
        .then(function (members) {
            //we want only login names
            return members.map(function (member) {
                return member.login;
            });
        });
}
```

Fetch Members

```
fetchMembers("peerigon")
    .then(function (members) {

        console.log(members.join(", "))
   })
    .catch(function (err) {

        console.error(err);
    });
```

```
spawn(function* () {

    var members = yield fetchMembers("peerigon");

    console.log(members.join(", "));
});
```

Fetch Members + Keys

Promises

```
fetchMembers("peerigon")
    .then(function (members) {
        return Promise.all(members.map(fetchKey));
    })
    .then(function(keys) {
        console.log(keys);
    })
    .catch(function (err) {
        console.error(err);
    });
```

Promises + Generator

```
spawn(function* () {
    var members, keys;

    try {
        members = yield fetchMembers("peerigon");
        keys = yield Promise.all(members.map(fetchKey));

        console.log(keys);
    }
    catch (err) {
        console.error(err);
    }
});
```

Conclusion

- **Promises** allow us to write *maintainable* and *more readable* code with fairly little *overhead*
- **Generators** make it easy to call asynchronous functions in a synchronous style and are great in conjunction with **Promises**
- **Promises** are production ready and should be used by everyone right now!
- **Generators** are fairly new, but will make things very convenient with the standardized **async/await** with the drawback of adding some more overhead

Thank you

Questions?