Mechanics of Promises *(2)*

Understanding JavaScript Promise Generation & Behavior

Topics

- Async functions
- Async IIFE
- Sequential vs Parallel

Async functions

Async Functions

- An async function can contain an await expression, that pauses the execution of the async function and waits for the passed Promise's resolution.
- If the promise fulfills, you get the value back. If the promise rejects, the rejected value is thrown

```
async function myAsyncFunction() {
  try {
    const fulfilledValue = await promise;
  }
  catch (rejectedValue) {
    // …
  }
}
```


Ok, y’all already knew all these - just recapping.

Async Functions return promises

- Async functions always return a promise, whether you use await or not.
- That promise resolves with whatever the async function returns, or rejects with whatever the async function throws.

.

Q: What does this function return?

```
async function getLuckyNumber() {
  let v;
  try {
    v = await readFileAsync('num.txt');
  } catch(e) {
   // Return Fallback Data
    v = 42;
  }
  return v;
}
```

A: A promise

Q: What does this function return?

```
async function getLuckyNumber() {
  let v;
  try {
    v = await readFileAsync('num.txt');
  } catch(e) {
   // Return Fallback Data
    v = 42;
  }
  return v;
}
```

```
// Somewhere else in your code...
console.log(getLuckyNumber());
```

A: A promise

Now, this is important!!!! If somewhere else in your code you try to console.log a call to getProcessedData. What is it going to log out???

Q: What does this function return?

```
async function getLuckyNumber() {
  let v;
  try {
    v = await readFileAsync('num.txt');
  } catch(e) {
   // Return Fallback Data
    v = 42;
  }
  return v;
}
```

```
// Somewhere else in your code...
console.log(await getLuckyNumber());
```


So… If in any other part of my application I need to console log the value that getProcessedData resolve to, you HAVE to use await again!

Now, let’s talk more about where you can use the “await” keyword

Awaiting at Top-Level

Awaiting at top level

- As you know, the `await` keyword can only be used inside an `async` function.
- This presents a challenge: How to structure code that needs to `await` at top level?

Let me give you an example….

Awaiting at top level

```
const express = require("express")
const models = require("./models")

const app = express();


await models.User.sync()
await models.Page.sync()
app.listen(PORT, () => {
  console.log(`Server is listening on port ${PORT}!`)
})
```


This is an example application using express and sequelize.
To start the application, we want, at top-level, call `app.listen` after we synchronize the database models.

Problem is: Database synchronization returns a promise, so we need to await for it. But we can only `await` inside an `async` function, so the code above will NOT work.
What do we need to do in order to make it work?

Awaiting at top level

```
const express = require("express")
const models = require("./models")

const app = express();

const init = async () => {
  await models.User.sync()
  await models.Page.sync()
  app.listen(PORT, () => {
    console.log(`Server is listening on port ${PORT}!`)
  })
}

init();
```


Well, we already solved this problem previously - we just create an async function with whatever name we want (here we used `init`) and we call it immediately.

This already works great, but some may say that it can be further shortened…. If we wrap the function definition in parentheses, it becomes an expression and we can immediately invoke it.

Awaiting at top level

```
const express = require("express")
const models = require("./models")

const app = express();

(async () => {
  await models.User.sync()
  await models.Page.sync()
  app.listen(PORT, () => {
    console.log(`Server is listening on port ${PORT}!`)
  })
})();
```


This is called an “Immediately Invoked Function Expression (IIFE)”.

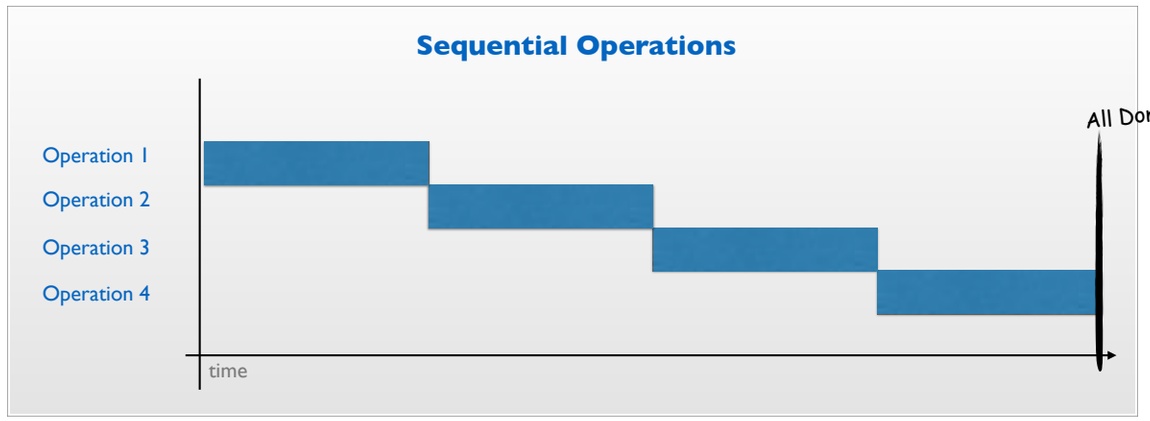
IIFEs were extensively used in the past to produce a lexical scope and protect against polluting the global environment - factors that were solved for quite a while by using modules.

With async/await though, IIFEs are useful simply to provide a starting point in your application where you can use the `await` keyword.

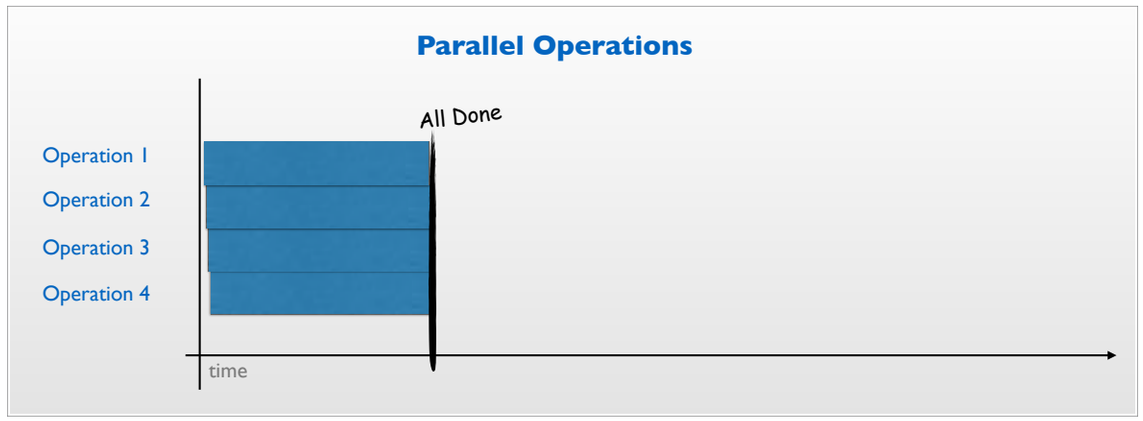
You will see us using this pattern more frequently in the future.

Sequential vs Parallel

Sequential vs Parallel



Sequential vs Parallel



Sequential vs Parallel

```
try {
  const number = await readFileAsync('/luckyNumber.txt');
  const charm = await readFileAsync('/luckyCharm.txt');
  const color = await readFileAsync('/luckyColor.txt');
} catch (error) {
  console.error(error);
}
```

What do you think? Are these operations performed in sequence or in parallel?
Follow up question: How do we make these operations in parallel?

Promises are eager

- A promise will start doing whatever task you give it as soon as the promise constructor is invoked.
- In other words, the task will run whether you `await` for the promise or not.

Sequential vs Parallel

```
try {
  const number = await readFileAsync('/luckyNumber.txt');
  const charm = await readFileAsync('/luckyCharm.txt');
  const color = await readFileAsync('/luckyColor.txt');
} catch (error) {
  console.error(error);
}
```

Sequential vs Parallel

```
try {
  const numberP = readFileAsync('/luckyNumber.txt');
  const charmP = readFileAsync('/luckyCharm.txt');
  const colorP = readFileAsync('/luckyColor.txt');
} catch (error) {
  console.error(error);
}
```

Sequential vs Parallel

```
try {
  const numberP = readFileAsync('/luckyNumber.txt');
  const charmP = readFileAsync('/luckyCharm.txt');
  const colorP = readFileAsync('/luckyColor.txt');
  const number = await numberP;
  const charm = await charmP;
  const color = await colorP;
} catch (error) {
  console.error(error);
}
```

But cumbersome...

Promise.all([promises])

- **Returns a single promise that resolves when all of the promises in the argument have resolved.**
- **Rejects if any of the passed promises are rejected.**
  - If any of the passed-in promises reject, Promise.all rejects with the value of the promise that rejected

Sequential vs Parallel

```
try {
  const numberP = readFileAsync('/luckyNumber.txt');
  const charmP = readFileAsync('/luckyCharm.txt');
  const colorP = readFileAsync('/luckyColor.txt');
  const number = await numberP;
  const charm = await charmP;
  const color = await colorP;
} catch (error) {
  console.error(error);
}
```


So, this is what we had before. We fire all tasks without the await keyword to make them start in parallel, but instead of individually waiting for each result…

Sequential vs Parallel

```
try {
  const numberP = readFileAsync('/luckyNumber.txt');
  const charmP = readFileAsync('/luckyCharm.txt');
  const colorP = readFileAsync('/luckyColor.txt');

  const values = await Promise.all([numberP, charmP, colorP])
  console.log(values); // Array [42, "Four-leaf clover", "Red"]

} catch (error) {
  console.error(error);
}
```

We use Promise.all.

Sequential vs Parallel

```
const numberP = readFileAsync('/luckyNumber.txt');
const charmP = readFileAsync('/luckyCharm.txt');
const colorP = readFileAsync('/luckyColor.txt');

try {
  const values = await Promise.all([numberP, charmP, colorP])
  console.log(values); // Array [42, "Four-leaf clover", "Red"]

} catch (error) {
  console.error(error);
}
```

By the way, because promise.all is the only one you’re waiting, it’s the only piece of code that might throw - so you don’t need to wrap all tasks in your try…catch block.


PROCEED
WITH
CAUTION

Sequential vs Parallel

- A given asynchronous operation may depend on the result of a previous one.

```
const tryGetRich = async () => {

 let num = await readFileAsync('/luckyNumber.txt')
 let success = await bookmaker.bet(num)

 if(success) {
   console.log("I'm rich!")
 }

}
```


Sometimes one asynchronous operation depends on the result of a previous one - in these cases, they need to be sequential.