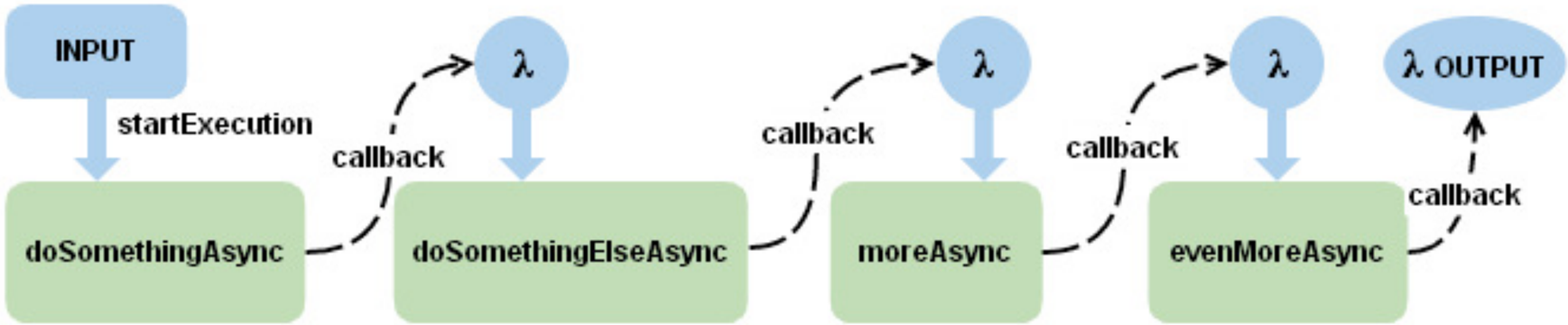
Promises, Callbacks, async/await

Full Stack Web Development

Agenda



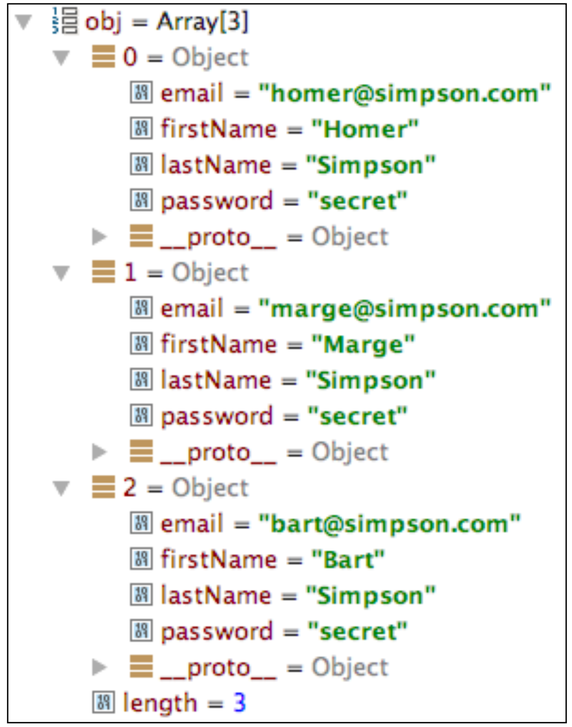
- Task: read a JSON file
- Callback Style
- Promise Style
- Async / Await

Task: Read a JSON File

- Read a json file into a string variable
- Parse that file into a JavaScript Object
- Print out the Javascript Object
- Deal with errors in an orderly manner:
  - File not Found
  - File not Correct JSON format

users.json

```
[
  {
    "firstName": "Homer",
    "lastName": "Simpson",
    "email": "homer@simpson.com",
    "password": "secret"
  },
  {
    "firstName": "Marge",
    "lastName": "Simpson",
    "email": "marge@simpson.com",
    "password": "secret"
  },
  {
    "firstName": "Bart",
    "lastName": "Simpson",
    "email": "bart@simpson.com",
    "password": "secret"
  }
]
```



memory

File System

Stability: 2 - Stable

File I/O is provided by simple wrappers around standard POSIX functions. To use this module do `require('fs')`. All the methods have asynchronous and synchronous forms.

The asynchronous form always takes a completion callback as its last argument. The arguments passed to the completion callback depend on the method, but the first argument is always reserved for an exception. If the operation was completed successfully, then the first argument will be `null` or `undefined`.

When using the synchronous form any exceptions are immediately thrown. You can use try/catch to handle exceptions or allow them to bubble up.

Here is an example of the asynchronous version:

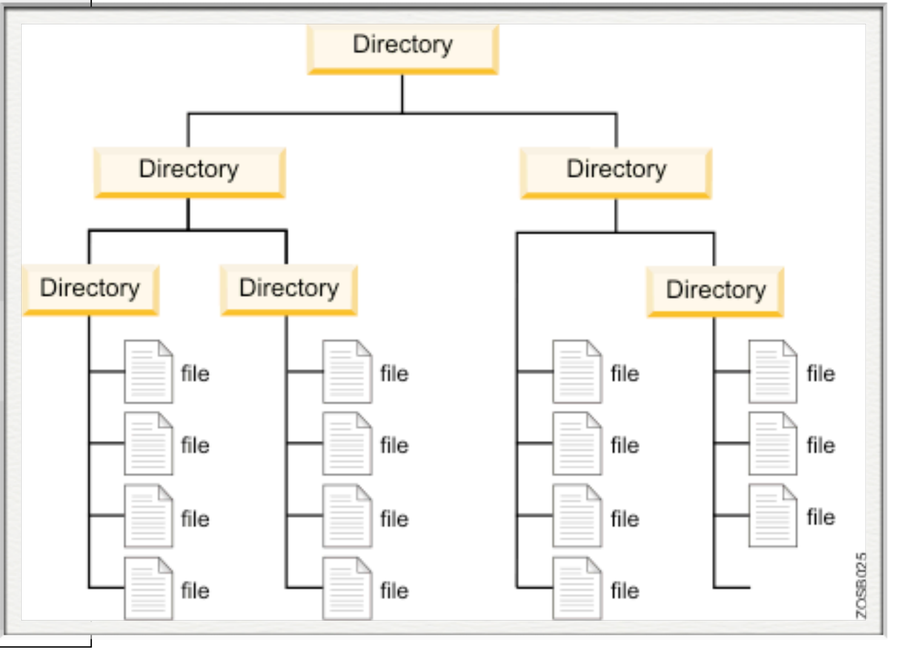
```
const fs = require('fs');

fs.unlink('/tmp/hello', (err) => {
  if (err) throw err;
  console.log('successfully deleted /tmp/hello');
});
```

Here is the synchronous version:

```
const fs = require('fs');

fs.unlinkSync('/tmp/hello');
console.log('successfully deleted /tmp/hello');
```



fs.readFile(file[, options], callback)

Added in: v0.1.29

- `file` <String> | <Buffer> | <Integer> filename or file descriptor
- `options` <Object> | <String>
  - `encoding` <String> | <Null> default = `null`
  - `flag` <String> default = `'r'`
- `callback` <Function>

Asynchronously reads the entire contents of a file. Example:

```
fs.readFile('/etc/passwd', (err, data) => {
  if (err) throw err;
  console.log(data);
});
```

The callback is passed two arguments `(err, data)`, where `data` is the contents of the file.

If no encoding is specified, then the raw buffer is returned.

If `options` is a string, then it specifies the encoding. Example:

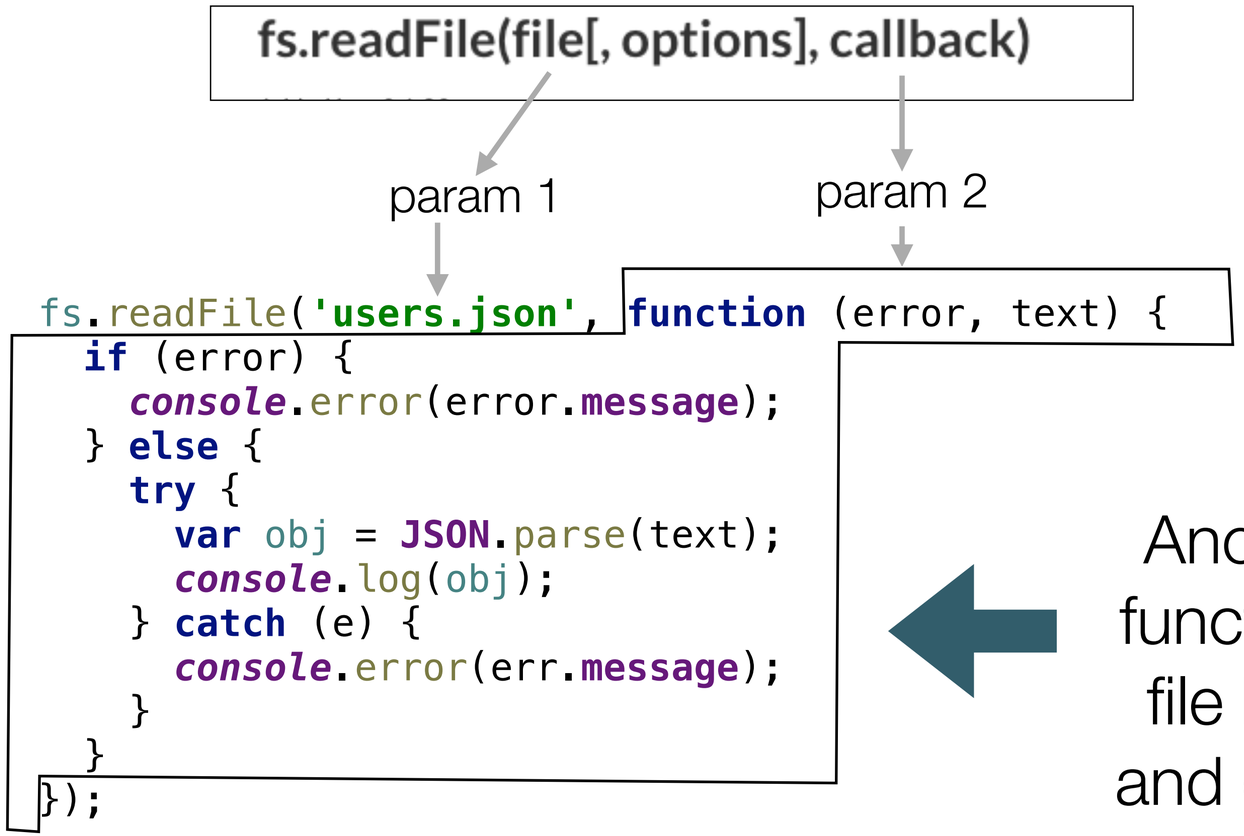
```
fs.readFile('/etc/passwd', 'utf8', callback);
```

Any specified file descriptor has to support reading.

Note: If a file descriptor is specified as the `file`*, it will not be closed automatically.*

Anonymous Function

```
fs.readFile('users.json', function (error, text) {
  if (error) {
    console.error(error.message);
  } else {
    try {
      var obj = JSON.parse(text);
      console.log(obj);
    } catch (e) {
      console.error(err.message);
    }
  }
});
```



Anonymous callback function invoked when file has been located and opened by the OS

Named Function

```
function readFileSimple(error, text) {
  if (error) {
    console.error(error.message);
  } else {
    try {
      var obj = JSON.parse(text);
      console.log(obj);
    } catch (e) {
      console.error(err.message);
    }
  }
};
```

```
fs.readFile('users.json', readFileSimple);
```

Named Callback function

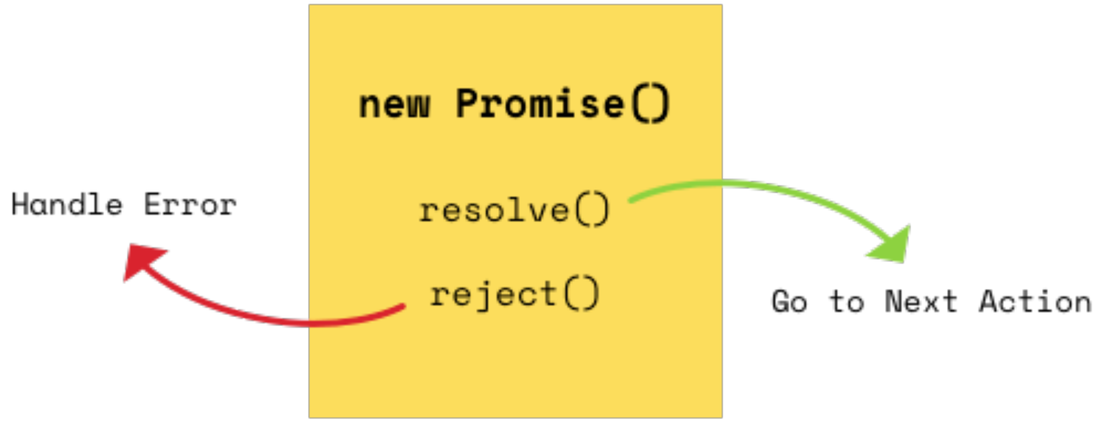
Pass named function to readFile function

param 1

param 2

```
fs.readFile(file[, options], callback)
```

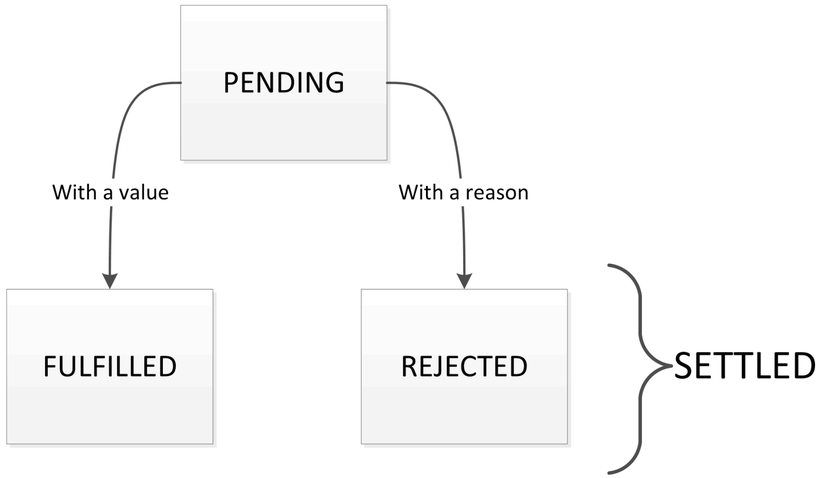
Promises



- Promises provide a simpler alternative for executing, composing, and managing asynchronous operations when compared to traditional callback-based approaches.
- Replacing Callbacks in many libraries and applications
- Implements a simpler and more robust pattern for asynchronous programming

- A promise can be in one of 3 states:
  - **Pending** - the promise's outcome hasn't yet been determined, because the asynchronous operation that will produce its result hasn't completed yet.
  - **Fulfilled** - the asynchronous operation has completed, and the promise has a value.
  - **Rejected** - the asynchronous operation failed, and the promise will never be fulfilled. In the rejected state, a promise has a reason that indicates why the operation failed.
- When a promise is pending, it can transition to the fulfilled or rejected state. Once a promise is fulfilled or rejected, however, it will never transition to any other state - and is regarded as **Settled**

Promise States



Callback

```
var fs = require('fs');

fs.readFile('users.json', function (error, text) {
  if (error) {
    console.error(error.message);
  } else {
    try {
      var obj = JSON.parse(text);
      console.log(obj);
    } catch (e) {
      console.error(err.message);
    }
  }
});
```

Callback

```
var fs = require('fs');

fs.readFile('users.json', function (error, text) {
  if (error) {
    console.error(error.message);
  } else {
    try {
      var obj = JSON.parse(text);
      console.log(obj);
    } catch (e) {
      console.error(err.message);
    }
  }
});
```

Promise

```
fs.readFile("users.json")
  .then(function(text){
    try {
      var obj = JSON.parse(text);
      console.log(obj);
    } catch (err) {
      console.error(err.message);
    }
  })
  .catch(function(err) {
    console.error(err.message);
  });
```

- In this small example, no major advantage to using promises
- However, as soon as callbacks become nested, then promises quickly become cleaner and simpler approach

Promises - Function Objects/Chaining

Standard Function Objects readSuccess & readFail

install success and fail methods into promise directly (chaining)

```
const fs = require("fs").promises;

const readSuccess = function(text) {
  try {
    var obj = JSON.parse(text);
    console.log(obj);
  } catch (err) {
    console.error(err.message);
  }
};

const readFail = function(err) {
  console.error(err.message);
};

fs.readFile("users.json")
  .then(readSuccess)
  .catch(readFail);
```

Promises - Function Objects/Chaining

```
fs.readFile("users.json")
  .then(function(text){
    try {
      var obj = JSON.parse(text);
      console.log(obj);
    } catch (err) {
      console.error(err.message);
    }
  })
  .catch(function(err) {
    console.error(err.message);
  });
```

=

```
const fs = require("fs").promises;

const readSuccess = function(text) {
  try {
    var obj = JSON.parse(text);
    console.log(obj);
  } catch (err) {
    console.error(err.message);
  }
};

const readFail = function(err) {
  console.error(err.message);
};

fs.readFile("users.json")
  .then(readSuccess)
  .catch(readFail);
```

Making asynchronous programming easier with async and await

More recent additions to the JavaScript language are async functions and the `await` keyword, part of the so-called ECMAScript 2017 JavaScript edition (see ECMAScript Next support in Mozilla). These features basically act as syntactic sugar on top of promises, making asynchronous code easier to write and to read afterwards. They make async code look more like old-school synchronous code, so they're well worth learning. This article gives you what you need to know.

https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Async_await

async Keyword

- When put in front of a function declaration it turns it into an async function.
- An async function is a function which knows how to expect the possibility of the await keyword being used to invoke asynchronous code.
- The async keyword is added to functions to tell them to return a promise rather than directly returning the value.

await Keyword

- The real advantage of async functions becomes apparent when you combine it with the await keyword.
- This can be put in front of any async promise-based function to pause your code on that line until the promise fulfills, then return the resulting value.
- In the meantime, other code that may be waiting for a chance to execute gets to do so.

Synchronous

```
function readTheFile() {
  const text = fs.readFileSync("users.json")
  var obj = JSON.parse(text);
  console.log(obj);
}

readTheFile();
```

- When readFileSync is called, the entire environment is ‘blocked’
- No other operations can take place until file is open and read.

Promises

```
fs.readFile("users.json")
  .then(text => {
     var obj = JSON.parse(text);
     console.log(obj);
  });
```

- When readFileSync is called, program can continue until ‘then’ is triggered
- Environment can remain active, other aspects of the program can proceed

Async/Await

```
async function readTheFile() {
  const text = await fs.readFile("users.json");
  var obj = JSON.parse(text);
  console.log(obj);
}

readTheFile();
```

- An alternative syntax to Promises
- Reads in a more linear manner
- Is ‘Non-Blocking’ like Promises/ Callbacks

Async/Await

- Similiar in structure to the sync version

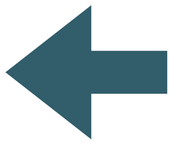
```
async function readTheFile() {
  const text = await fs.readFile("users.json");
  var obj = JSON.parse(text);
  console.log(obj);
}

readTheFile();
```

```
function readTheFile() {
  const text = fs.readFileSync("users.json")
  var obj = JSON.parse(text);
  console.log(obj);
}

readTheFile();
```

- But async version does not ‘block’ the main thread



Async/Await

Synchronous Style

Asynchronous Behaviour

```
const fs = require("fs").promises;

async function readTheFile() {
  try {
    const text = await fs.readFile("users.json");
    var obj = JSON.parse(text);
    console.log(obj);
  } catch (err) {
    console.error(err.message);
  }
}

readTheFile();
```

Promises, Callbacks, async/await

Full Stack Web Development