Intro to Async & Promises

Nick Whyte | @nickw444 | @nickw444

About Me

- Frontend Engineer / Technical Lead @ Canva
- Graduated UNSW in 2016 (Computer Science)
- COMP2041 student in 2014
- COMP2041 tutor in 2015

What is Async

- **synchronous**: Operations can only occur one at a time.
- **asynchronous**: Multiple operations can occur at the same time. Programmers can parallelise their program.

Concurrency models

- Coroutines / Cooperative Multitasking
- Threads / Preemptive Multitasking
- Event Driven

https://www.cse.unsw.edu.au/~cs9242/18/lectures/02-threadsevents.pdf

Threads / Preemptive Multitasking

- A thread can be preempted, so mutex's and semaphores required to guard critical sections & variables.
- Typical multi-tasking model found in many languages like Java and Python.

https://www.cse.unsw.edu.au/~cs9242/18/lectures/02-threadsevents.pdf

Co-routines / Cooperative Multitasking

- co-routines can cooperatively "yield" to other co-routines.
- "yield" saves the state of co-routine A, and resumes B's state from it's previous yield point.
- No preemption between yields, so no need for mutex's or semaphores to guard critical sections.

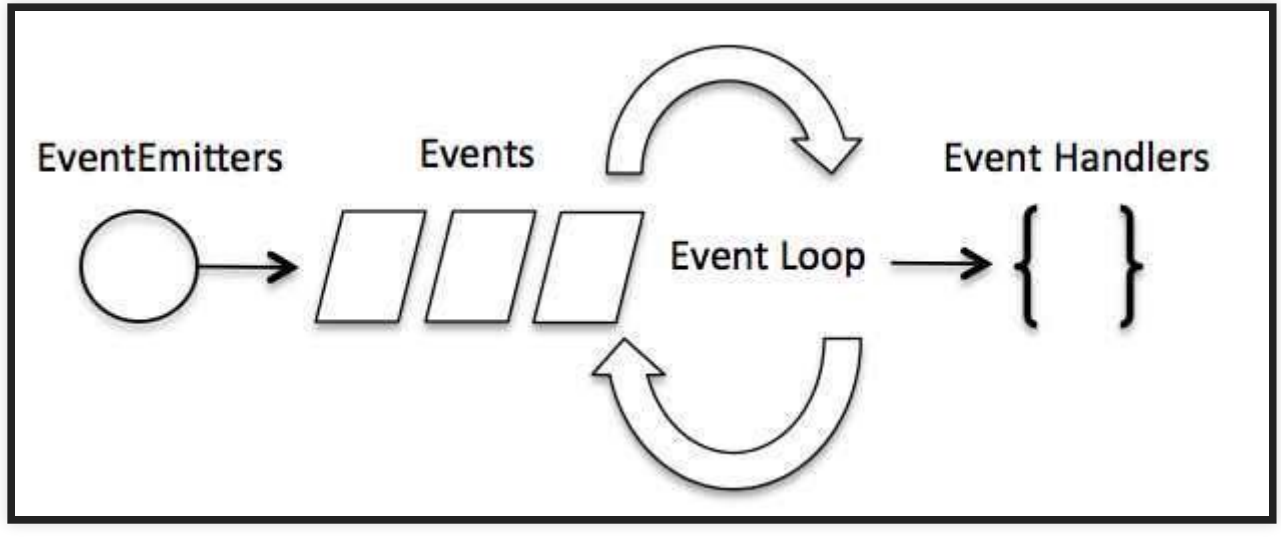
https://www.cse.unsw.edu.au/~cs9242/18/lectures/02-threadsevents.pdf

Event Driven

- External entities generate (post) events. (i.e. button click)
- Event loop waits for events and calls an appropriate event handler.
- Event handler is a function that runs to completion and returns to the event loop.
- No preemption, so no need for mutex's or semaphores
- Cannot parallelise computational workloads ☹️

https://www.cse.unsw.edu.au/~cs9242/18/lectures/02-threadsevents.pdf

Event Driven



Javascript Concurrency Model

- Javascript is single threaded, event driven
- Unable to parallelise computational tasks
- Each handler is run to completion before a new task is started

```
while (queue.waitForMessage()) {
  queue.processNextMessage();
}
```

- A typical Javascript program will be IO Bound, rather than CPU bound, which means the event driven model is appropriate

https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop#Event_loop

Event Loop Demo

callbacks ⇒ {}

What is a callback

- A function that is called when the pending work has completed
- Can use either a named function or an anonymous lamdba:

```
fs.readFile('foo.txt', (result, err) => {
  // Do something with result here
})
```

```
function onComplete(result, err) {
  // Do something with result here
}
fs.readFile('foo.txt', onComplete)
```

Why Do We Use Callbacks?

- Because of the event loop
- Functions cannot block whilst they wait for a result otherwise important tasks on the queue will be delayed
- Therefore they must register a completion handler: A *callback*

```
fs.readFile('foo.txt', (result, err) => {
  if (err) {
    console.error('Something went wrong:', err)
    return;
  }

  // Do something with result here
})
```

Callback Gotchas

try-catch with callbacks

```
function doThing() {
  throw new Error('Something went wrong');
}

window.setTimeout(() => doThing(), 1000)
```

```
Error: Something went wrong
```

try-catch with callbacks

```
function doThing() {
  throw new Error('Something went wrong');
}
try {
  window.setTimeout(() => doThing(), 1000)
} catch (e) {
  console.log('Caught error on click:', e);
}
```

try-catch with callbacks

```
function doThing() {
  throw new Error('Something went wrong');
}
try {
  window.setTimeout(() => doThing(), 1000)
} catch (e) {
  console.log('Caught error on click:', e);
}
```

```
Error: Something went wrong
```

try-catch with callbacks

```
function doThing() {
  throw new Error('Something went wrong');
}
try {
  window.setTimeout(() => doThing(), 0)
} catch (e) {
  console.log('Caught error on click:', e);
}
```

try-catch with callbacks

```
function doThing() {
  throw new Error('Something went wrong');
}
try {
  window.setTimeout(() => doThing(), 0)
} catch (e) {
  console.log('Caught error on click:', e);
}
```

```
Error: Something went wrong
```

try-catch with callbacks

```
function doThing() {
  throw new Error('Something went wrong');
}

window.setTimeout(() => {
  try {
    doThing();
  } catch (e) {
    console.log('Caught error on click:', e);
  }
}), 1000)
```

try-catch with callbacks

```
function doThing() {
  throw new Error('Something went wrong');
}

window.setTimeout(() => {
  try {
    doThing();
  } catch (e) {
    console.log('Caught error on click:', e);
  }
}), 1000)
```

```
Caught error on click: Something went wrong
```

Callback Hell

```
getData(function(x){
  getMoreData(x, function(y){
    getMoreData(y, function(z){
      ...
    });
  });
});
```

Resolving Callback Hell

- Keep your code shallow
- Don't nest functions. Give them names and place them at the top level of your program

When Not To Use Callbacks

- When your work is synchronous

When Not To Use Callbacks

```
function sayHello(callback) {
  console.log('Hello!');
  callback();
}

sayHello(() => {
  console.log('We said hello.');
});
```

```
function sayHello() {
  console.log('Hello!');
}

sayHello();
console.log('We said hello.');
```

Promises

What are Promises

- An abstraction around async work giving us access to “future” values.
- A Promise is an object representing the eventual completion or failure of an asynchronous operation.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

Promise Terminology

A Promise can be:

- **Fulfilled**: The action relating to the promise succeeded.
- **Rejected**: The action relating to the promise failed.
- **Pending**: Hasn't yet fulfilled or rejected.
- **Settled**: Has fulfilled or rejected.

https://developers.google.com/web/fundamentals/primers/promises

Using Promises

```
function callback(results, err) {
  if (err) {
    console.error('Something went wrong:', error);
    return;
  }

  console.log('Got the results:', results)
}
getUsers(callback);
```

Using Promises

```
const promise = getUsers();
promise
  .then(results => {
    console.log('Got the results:', results)
  })
  .catch(error => {
    console.error('Something went wrong:', error)
  })
```

.then

```
p.then(onFulfilled[, onRejected]);
```

- Returns a promise
- **onFulfilled**: A function called if the promise is fulfilled. It receives the fulfillment value as an argument.
- **onRejected**: (optional) A function called if the promise is rejected. It receives the rejection reason as an argument.

`.then`

A handler can:

- return a value
- throw an error
- return a promise

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/then#Return_value

`.catch(`

```
p.catch(onRejected);
```

- The catch() method returns a Promise and deals with rejected cases only
- Internally calls `Promise.prototype.then`

Chaining / Composition

```
getUsers()
  .then(users => getUserProfile(users[0]))
  .then(userProfile => downloadUserImage(userProfile.image.url))
  .then(userImage => {
    console.log('Downloaded user profile image')
  })
  .catch(error => {
    console.error('Something went wrong:', error)
  })
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises#Chaining

Chaining / Composition

```
doSomething(result => {
  doSomethingElse(result, newResult => {
    doThirdThing(newResult, finalResult => {
      console.log('Got the final result: ' + finalResult);
    }, handleFailure);
  }, handleFailure);
}, handleFailure);
```

By chaining promises, we avoid *"callback hell"*

```
doSomething()
  .then(result => doSomethingElse(result))
  .then(newResult => doThirdThing(newResult))
  .then(finalResult => {
    console.log('Got the final result: ' + finalResult);
  })
  .catch(handleFailure);
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises#Chaining

Handling Errors

- A promise chain stops if there's an exception and looks down the chain for a catch handler instead

```
doSomething()
  .then(result => doSomethingElse(result))
  .then(newResult => doThirdThing(newResult))
  .then(finalResult => console.log(`Got the final result: ${final
  .catch(failureCallback);
```

```
try {
  const result = syncDoSomething();
  const newResult = syncDoSomethingElse(result);
  const finalResult = syncDoThirdThing(newResult);
  console.log(`Got the final result: ${finalResult}`);
} catch(error) {
  failureCallback(error);
}
```

Handling Errors

- A catch statement can be used to continue the chain after a failure

```
fetchUsers()
  .then(() => {
      throw new Error('Something failed');
      console.log('Do this');
  })
  .catch(() => {
      console.log('Do that');
  })
  .then(() => {
      console.log('Do this, no matter what happened before');
  });
```

```
Do that
Do this, no matter what happened before
```

Creating Promises

```
// returns a Promise object that is rejected with the given reaso
Promise.reject(new Error('Something went wrong!'));
```

```
// returns a Promise object that is resolved with the given value
Promise.resolve('I am resolved!');
```

```
new Promise((resolve, reject) => {
  doSomethingAsync((result, err) => {
    if (err) {
      reject(err);
      return;
    }
    resolve(result);
  })
})
```

Promisify

Sometimes useful to promisify a callback API to support chaining

```
function wait(ms) {
  return new Promise(resolve => {
    setTimeout(resolve, ms);
  });
}

wait(1000)
  .then(() => {
    console.log('Waited for 1 second');
  })
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises#Creating_a_Promise_around_an_old_callback_API

Promise.all()

- Returns a Promise that resolves when all of the promises have settled
- It rejects with the reason of the first promise that rejects.

```
const p = Promise.all([
  doSomething(),
  doSomethingElse()
]);

p.then(results => {
  // results[0] is the result of doSomething()
  // results[1] is the result of doSomethingElse()
})
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/all

Promise.race()

- Returns a promise that resolves or rejects as soon as one of the promises resolves or rejects.
- Resolves or rejects with the value or reason from that promise.

```
const p = Promise.race([
  doSomething(),
  doSomethingElse()
]);

p.then(result => {
  // result is the value of either doSomething or doSomethingElse
  // whichever resolved first.
})
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/race

When not to use Promises

- If you're doing synchronous work
- When you have a callback situation where the callback is designed to be called multiple times
- For situations where the action often does not finish or occur

https://stackoverflow.com/a/37531576

Promise hell

Simple promise chains are best kept flat without nesting, as nesting can be a result of careless composition.

```
fetchBook()
  .then(book => {
    return formatBook(book)
      .then(book => {
        return sendBookToPrinter(book);
      });
  });
```

```
fetchBook()
  .then(book => formatBook(book))
  .then(book => sendBookToPrinter(book));
```

https://medium.com/@pyrolistical/how-to-get-out-of-promise-hell-8c20e0ab0513
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises#Nesting

async-await

- Syntactic sugar around Promises
- Allows you to write async functions more like synchronous functions

async-await

```
doSomething()
  .then(result => doSomethingElse(result))
  .then(newResult => doThirdThing(newResult))
  .then(finalResult => console.log(`Got the final result: ${final
  .catch(failureCallback);
```

```
try {
  const result = await doSomething();
  const newResult = await doSomethingElse(result);
  const finalResult = await doThirdThing(newResult);
  console.log(`Got the final result: ${finalResult}`)
} catch (e) {
  failureCallback(e);
}
```

async-await

- The await keyword is only valid inside async functions

```
async myAsyncFunction() {
  await doSomething();
  await doSomethingElse();
}

const p = myAsyncFunction();
// typeof p === Promise

p
  .then(() => { ... })
  .catch(() => { ... })
```

Cancellation

- You can’t cancel a Promise
- This poses potential issues when using promises for long running tasks
- There are third party libraries that support cancellation (Bluebird)
- If you want cancellation, Promises might not be what you want

Fetching Data / AJAX

Fetching Data / AJAX

- Fetch partial data from the server
- Update content on a page without refreshing it entirely
- Submit user input to the server without POST'ing a form / reloading the page

Fetching Data / AJAX

XHR (sync)

```
console.log('Loading...');

const request = new XMLHttpRequest();

// `false` makes the request synchronous
request.open('GET', 'https://api.ipify.org?format=json', false);
request.send(null);

if (request.status === 200) {
  console.log(request.responseText);
}
```

Run Code

https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest/Synchronous_and_Asynchronous_Requests#Synchronous_request

XHR (async)

```
console.log('Loading...');
const xhr = new XMLHttpRequest();
xhr.open("GET", "https://api.ipify.org?format=json", true);
xhr.onload = function (e) {
  if (xhr.readyState === 4) {
    if (xhr.status === 200) {
      console.log(xhr.responseText);
    }
  }
};
xhr.send(null);
```

Run Code

https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest/Synchronous_and_Asynchronous_Requests#Asynchronous_request

XHR Util with Callback

```
function getData(url, onSuccess, onError) {
  var xhr = new XMLHttpRequest();
  xhr.open('GET', url, true);
  xhr.onload = function(e) {
    if (xhr.readyState === 4) {
      if (xhr.status === 200) {
        onSuccess(xhr.responseText);
      } else {
        onError(new Error(xhr.statusText));
      }
    }
  };
  xhr.onerror = function(e) {
    onError(new Error(xhr.statusText));
  }
  xhr.send(null);
};
```

```
getData('https://mysite.com/foo.json', result => {
  console.log('Got data:', result);
}, error => {
  console.log('An error occurred:', error)
})
```

XHR with Promise

```
function getData(url) {
  return new Promise((resolve, reject) => {
    var xhr = new XMLHttpRequest();
    xhr.open('GET', url, true);
    xhr.onload = function(e) {
      if (xhr.readyState === 4) {
        if (xhr.status === 200) {
          resolve(xhr.responseText);
        } else {
          reject(new Error(xhr.statusText));
        }
      }
    };
    xhr.onerror = function(e) {
      reject(new Error(xhr.statusText));
    }
    xhr.send(null);
  });
};
```

```
try {
  const data = await getData('https://mysite.com/foo.json');
  console.log('Got data:', result);
} catch (error) {
  console.log('An error occurred:', error)
}
```

`fetch()`

- The Fetch API provides an interface for fetching resources
- More powerful and flexible feature set than XMLHttpRequest

https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API

Using `fetch()`

- Similar to our Promisified `XMLHttpRequest` attempt
- Won't reject on HTTP error status (You will need to compose a chain to reject if you need this)
- Call `Response.json()` to deserialize a JSON response

```
fetch('http://example.com/movies.json')
  .then(response => response.json())
  .then(myJson => {
    console.log(myJson);
  });
```

https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch

Single Page Applications (SPA's)

Single Page Applications (SPA's)

- A website that re-renders its content in response to user actions (i.e. clicking a link) without reloading the entire page
- Will make use of `XHR/fetch` to load partial content on user actions
- Easy to build with the right tools (Angular, React + React-Router, Vue)
- Feels more responsive to users

SPA Pros

- Feels more responsive to users - low latency to switch between "pages"
- Explicit split between "frontend" and "backend" architecture"

SPA Cons

- Difficult to optimize for search engines
- Often quite large to load initially. Bundles > 1MB
- Memory leaks are more likely
- Routing and Navigation is difficult to get right

SPA Examples

- Gmail

Frameworks

jQuery (2006 → present)

- One of the first web "frameworks"
- Designed to simplify client side scripting
- Most widely adopted JS library (still plaguing sites today)
- jQuery's syntax is designed to make it easier to navigate a document

https://en.wikipedia.org/wiki/JQuery

Backbone (2010 → 2016)

- Known for being lightweight as it only had a single dependency
- Designed for developing single page applications
- Assisted in keeping various parts of web applications synchronised.

https://en.wikipedia.org/wiki/Backbone.js

Angular/AngularJS (2010 → present)

- Provides a framework for client-side MVC application architectures
- "Magical" DOM data binding
- Angular 2+ versions are simply called Angular. Angular is an incompatible rewrite of AngularJS

React (2013 → present)

- React does not attempt to provide a complete 'application framework'
- Similar to Vue, it is designed specifically for building user interfaces
- Complex React applications require the use of libraries for state management, routing, etc.

https://en.wikipedia.org/wiki/React_(JavaScript_library)

Vue (2014 → present)

- Built to organize and simplify web development.
- Less opinionated that other frameworks (angular) thus easier for developers to pick up.
- Features an incrementally adoptable architecture.
- Advanced features required for complex applications such as routing, state management and build tooling are offered via officially maintained supporting libraries and packages.

https://en.wikipedia.org/wiki/Vue.js

Thanks

Nick Whyte | @nickw444 | @nickw444

p.s. we are looking for summer interns and 2019 graduates!
Please email nick@canva.com if you are interested