Asynchronicity And Networking

Promises

Created by Zain Afzal @zainafzal08 @blockzain

Overview

Client server Interactions

AJAX

Event Loop

Callbacks and XMLHTTP

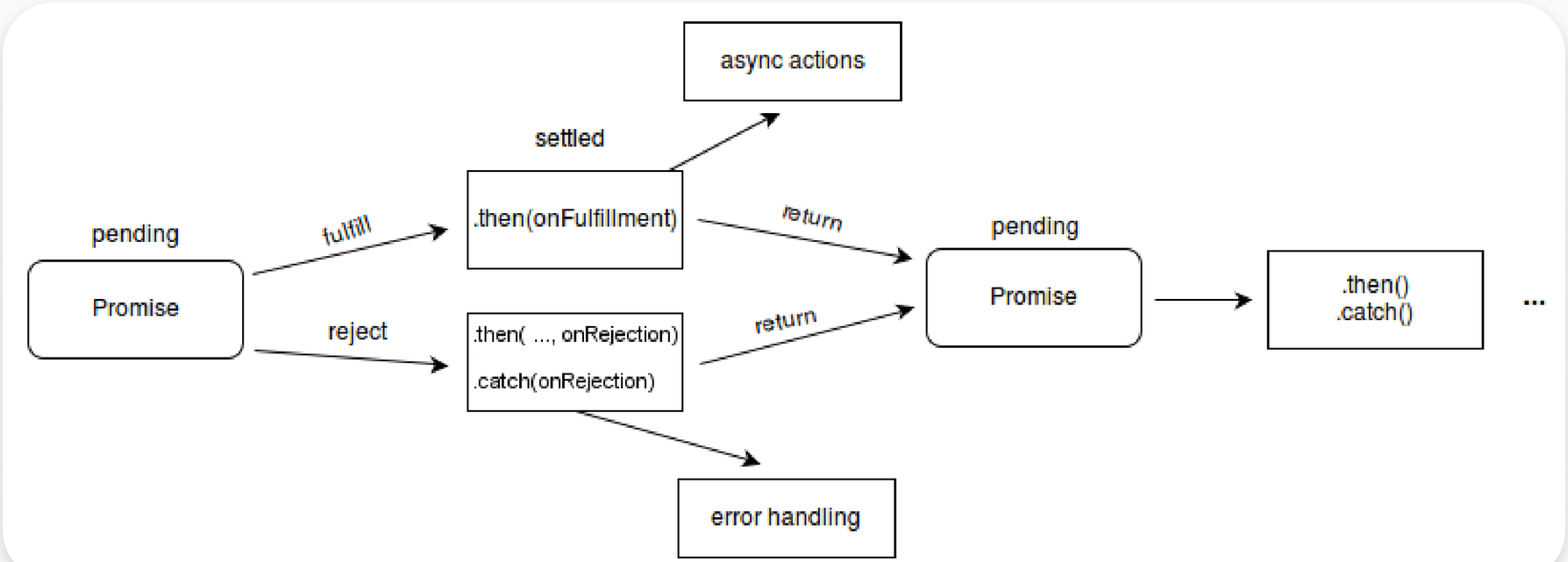
Promises and Fetch

Async Await

```
function getShoppingList() {
    const request = makeNewRequest('/shoppinglist');
    request.onload = (shoppingList) => {
        for (const item of list) {
            const request = makeNewRequest('/buy?item='+item);
            request.onerror = (error) => {
                alert(`couldn\'t buy ${item}`);
            }
            request.send();
        }
    };
    request.onerror = (error) => alert('uh oh');
    request.send();
}
```

XMLHttpRequest and Callback Hell

Using XMLHttpRequest with callbacks has a series of issues, chaining is difficult for one and handling errors requires handling at multiple levels.



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

Promises

Promises are a logical response to the issues of using callbacks to handle asyncronous actions. They are a built in object that **represent the eventual completion (or failure) of an asynchronous operation, and its resulting value.** A promise can be in a number of states, pending, fullfilled (or resolved), rejected and settled.

```
const myPromise = new Promise((resolve, reject) => {
  // Call resolve when it's done.
  // Call reject if something went wrong.
});

myPromise.then(
  () => {
  // This code runs after resolve()
  // is called;
  },
  () => {
  //This code runs after reject is
  // called.
  }
);

myPromise.catch(() => {
  // This code also runs after reject()
  // is called;
});
```

Promises

When you create a promise you simply set up some code that either marks the promise as resolved or rejected. On it’s own this doesn’t seem much better then our callback solution but what’s powerful about promises is **chaining.**

```
getShoppingList()
    .then(buyItems)
    .then(putItemsInPantry)
    .then(pickDinnerRecipe)
    .then(cookDinner)
    .catch(orderTakeout);
```

```
getShoppingList()
  .then(buyItems)
  .then(putItemsInPantry)
  .then(pickDinnerRecipe)
  .then(cookDinner)
  .catch(orderTakeout)
  // This runs regardless of if we
  // cooked dinner or not.
  .then(brushTeeth)
  .then(sleep)
  // This only catches errors from
  // the above 2 then statements.
  .catch(putOnWhiteNoise)
```

Chaining

If within a then() callback you need to wait for another operation you simply return another promise. The next then() callback attached to the first will then wait for the returned promise to complete. Even better errors traverse through the calls until they hit the closest catch statement. Note that after a catch we can keep going in case we have some things we need to do regardless of if the previous step failed or not.

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

const wait20 = wait(10).then(() => wait(10));
const wait30 = wait20.then(() => wait(10));

wait20.then(() => console.log("hello"));
wait30.then(() => console.log("world"));
```

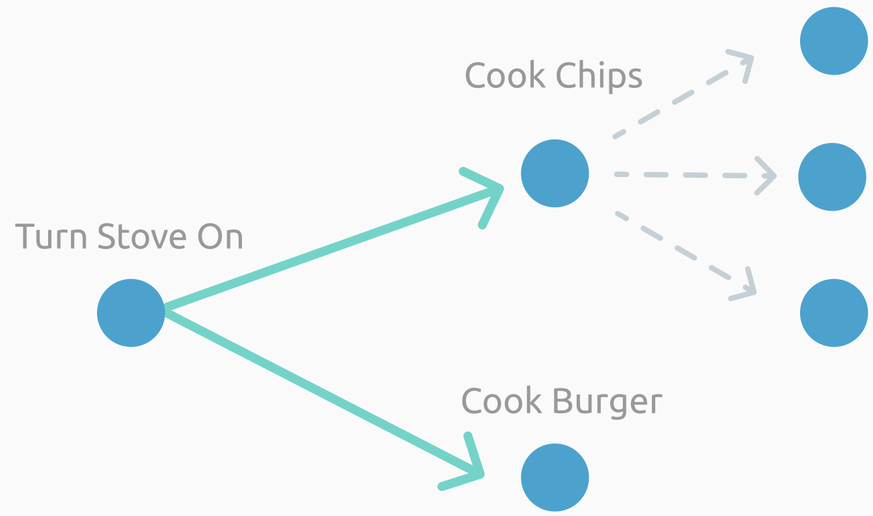
```
20ms hello
30ms world
```

Code example inspired by @darvid7

Chaining

It’s important to remember that .then() creates a new promise which resolves when itself and all the actions before it resolve. This means the original promise and the promise that .then returns are **distinct.** Think of it like a russian nesting doll, each .then wraps the previous promise, creating a new promise with 1 more layer that needs to complete before the promise is completed. This however does not destory or modify the promise it’s wrapping, like a parent nesting doll.

```
1 const stoveOnPromise = turnStoveOn();
2
3 stoveOnPromise
4   .then(cookChips);
5
6 stoveOnPromise
7   .then(cookBurger);
```



Branching

Another interesting consequence of each call of .then returning a distinct promise is the fact that you can attach multiple .then() calls to the same promise, effectively creating a tree of operations that run one layer at a time.

```
1 new Promise(() => {
2   throw new Error('reactor meltdown imminent');
3 }).catch(err => console.log(err));
```

```
> Error: reactor meltdown imminent
```

Error Handling

Any error thrown in a callback passed to a promise, either on the initial constructor, or on a then or on a catch will be converted to a promise rejection.

```
new Promise((_, reject) => reject('oh no!'))
    .catch(() => {
        throw new Error('B RU H');
    })
    .then(() => {
        console.log('foo');
    })
    .catch(e => {
        console.log('bar');
    });
```

Error Handling

What do you think will happen here?

```
new Promise((_, reject) => reject('oh no!'))
    .catch(() => {
        throw new Error('B RU H');
    })
    .then(() => {
        console.log('foo');
    })
    .catch(e => {
        console.log(e.message);
    });
```

Remember throwing an error is silently caught and passed into the reject function.

```
> B RU H
```

Error Handling

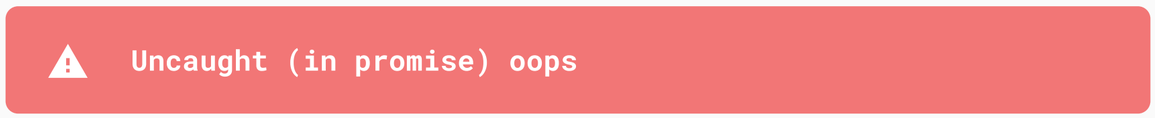
The 2nd catch statement also applies to the nearest catch statement above it. If during the handling of an error, another error occurs it’s passed down to the next handler or if there is no handler throw a top level error as per usual.

```
1 function foo() {
2   try {
3     new Promise((_, reject) => reject('oops'));
4   } catch (err) {
5     console.log(err);
6   }
7 }
8
9 function main() {
10   foo();
11 }
12
13 main();
```

Error Handling

What do you think will happen here?

```
 1 function foo() {
 2   try {
 3     new Promise((_, reject) => reject('oops'));
 4   } catch (err) {
 5     console.log(err);
 6   }
 7 }
 8
 9 function main() {
10   foo();
11 }
12
13 main();
```



Error Handling

When a promise fails and it can’t find a catch it throws a unhandledRejectionError. But it’s worth remembering that once a promise is created it essentially lives at the top level scope. A promise, after being constructed, doesn’t live in the function where it was created. In fact it can’t since the function ends and is popped off the stack before the promise could ever resolve. As such a unhandled error is thrown and can’t be caught, it gets converted to a global error.

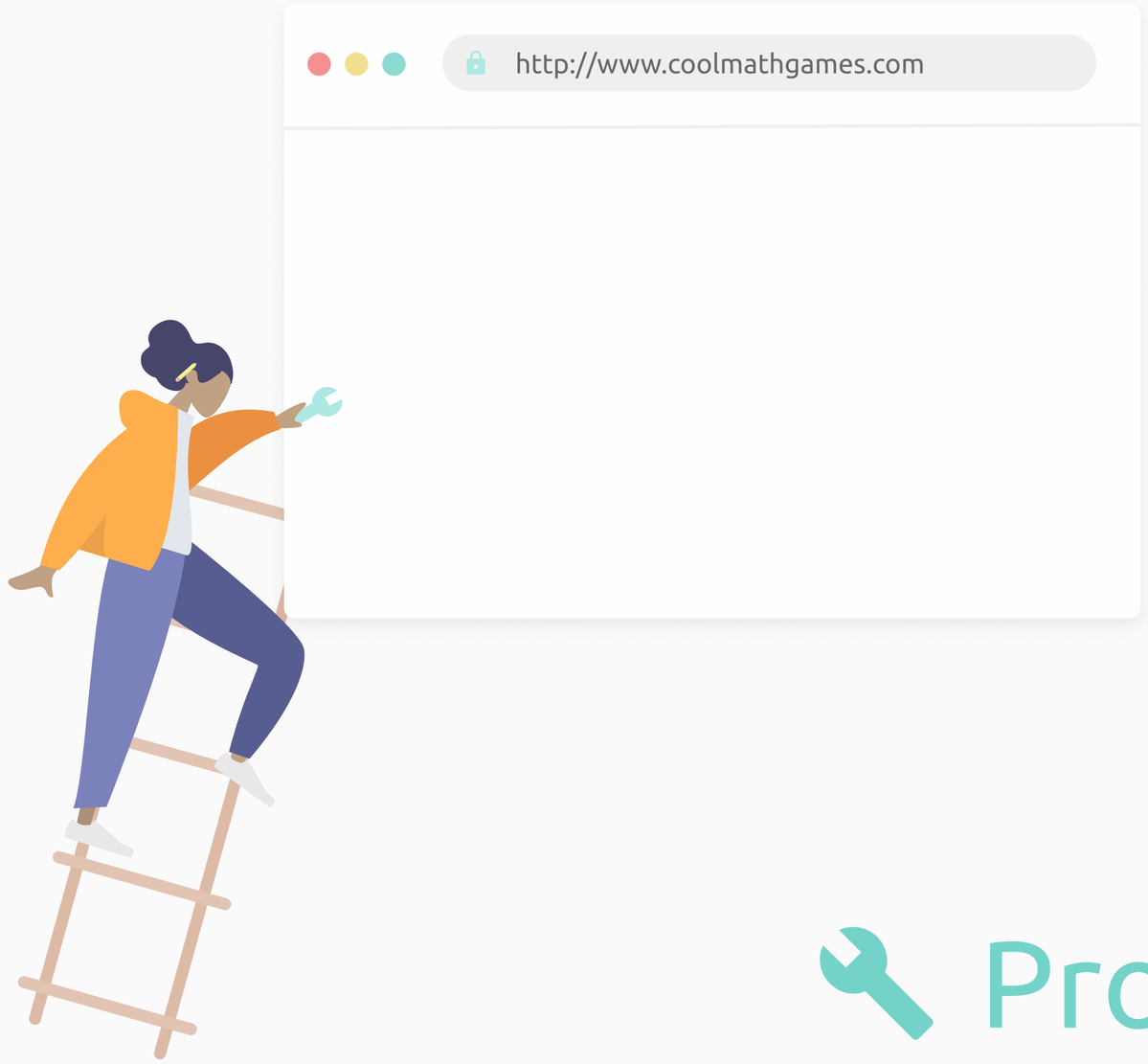
```
1 window.addEventListener('unhandledrejection', function(event) {
2   alert('something went wrong!');
3 });
4
5 new Promise(function() {
6   throw new Error("Whoops!");
7 });
```

⚠ Uncaught (in promise) oops

Error Handling

You can register a handler on window to catch these global failures however so they don’t just print silently to console.

Promises



Promise Demo

main-content/lectures/async-promise-demo

```
class MyThenable {
  then(onFullfill, onReject){
    setTimeout(() => onFullfill(42), 1000);
  }
}

Promise.resolve()
  .then(() => {
    return new MyThenable();
  })
  .then((v) => {
    // Prints 42 after 1s.
    console.log(v);
  })
```

Thenable

A thenable is any object with a “then” function, and promise chains will treat them exactly like family, in fact they call these objects “promise-like”. In the above example the “then” method is called with a resolve and reject function (called onFullfill and onReject in this context) which behave as resolve and reject would. You’ll see thenable objects pop up in areas where the flexibility of having full control over the promise like object is useful.

```
function get(url) {
  const xhr = new XMLHttpRequest();
  xhr.open("https://api.ipify.org?format=json");
  const p = new Promise((resolve, reject) => {
     xhr.onload = resolve;
    xhr.onerror = reject;
  });
  xhr.send();
  return p;
}


```

json() returns a promise which resolves to a json object, r.json().field won't work! Read more

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

```

Fetch

You could wrap XMLHTTP to give you all the nice features of promises or you can just use fetch, it brings the magic of promises by default as well as being a bit less confusing to read/use.

```fetch
fetch(url, {
    // *GET, POST, PUT, DELETE, etc.
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
      // Any other headers.
    },
    // Any data you want to post!
    // Note: body data type must match
    // "Content-Type" header
    body: JSON.stringify(data)
  });
```

Use fetch to do any regular HTTP request

```xhr
const xhr = new XMLHttpRequest();
xhr.open("files.com/thicc_file");
xhr.onprogress = updateProgressBar;
xhr.send();
```

Use XMLHTTPRequest to do large downloads

Fetch

There are cases where you may want to use old school XMLHttpRequest, on older browsers that don’t have fetch, promises can’t be cancelled, a .then() handler only runs once and also fetch doesn’t give you progress events so they arn’t optimal for downloading larger files. Note, you can ask fetch to give you a readable stream if you are ok with a experimental api to help solve some of these issues.

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

```
fetchBook()
  .then(formatBook)
  .then(spellCheckBook)
  .then(sendBookToPrinter);
```

Promise Hell

Promises make is much easier to create flat nested logic but they don’t force you to do so, as such you can get right back into the original problem with callback hell if you arn’t aware. Remember that everytime in a .then() you have additional logic, it might be better left for another .then() statement. An interesting fact is if you don’t return a promise you can still keep chaining! Use this to help keep your code neat and modular.

Challange

Create a simple website to let you view and make posts for any topics.

🔧 Fetch Demo

main-content/lectures/async-fetch-demo

```
let isLoading = true;

Promise.resolve(fakeResponse)
  .then((r) => {
    const contentType = r.headers.get("content-type");
    if (contentType && contentType.includes("application/json")) {
      return response.json();
    }
    throw new TypeError("Oops, we haven't got JSON!");
  })
  .then(handleJson)
  .catch(handleError)
  .finally(() => { isLoading = false; });
```

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

Promise built ins

Some things we haven’t mentioned yet, we can use promise.finally to do some actions regardless of if an error occured or not (note that if a error is raised then() statements are skipped and vice versa is a error isn’t raised) and we can use Promise.resolve and reject to quickly created a resolved or rejected promise (useful for testing!)

```
function handle(f) {
    fruits.push(r)
    if (fruits.length === 4)
        make_fruit_salad(fruits)
}
fruits = []
fetch('myapi/apple').then(r => handle(r))
fetch('myapi/orange').then(r => handle(r))
fetch('myapi/banana').then(r => handle(r))
fetch('myapi/pear').then(r => handle(r))
```

```
fruits = [
    fetch('myapi/apple'),
    fetch('myapi/orange'),
    fetch('myapi/banana'),
    fetch('myapi/pear')
]

Promise.all(fruits)
    .then(fruits => make_fruit_salad(fruits))

```

Promise built ins

Some other useful functions are Promise.all which waits for all promises in a list to complete before resolving. Note that it fails the moment any of the promises fails.

```
function handle(f) {
    if (fruit === null) {
      fruit = f
      eat(f)
    }
}
fruit = null;
fetch('myapi/apple').then(r => handle(r))
fetch('myapi/orange').then(r => handle(r))
fetch('myapi/banana').then(r => handle(r))
fetch('myapi/pear').then(r => handle(r))
```

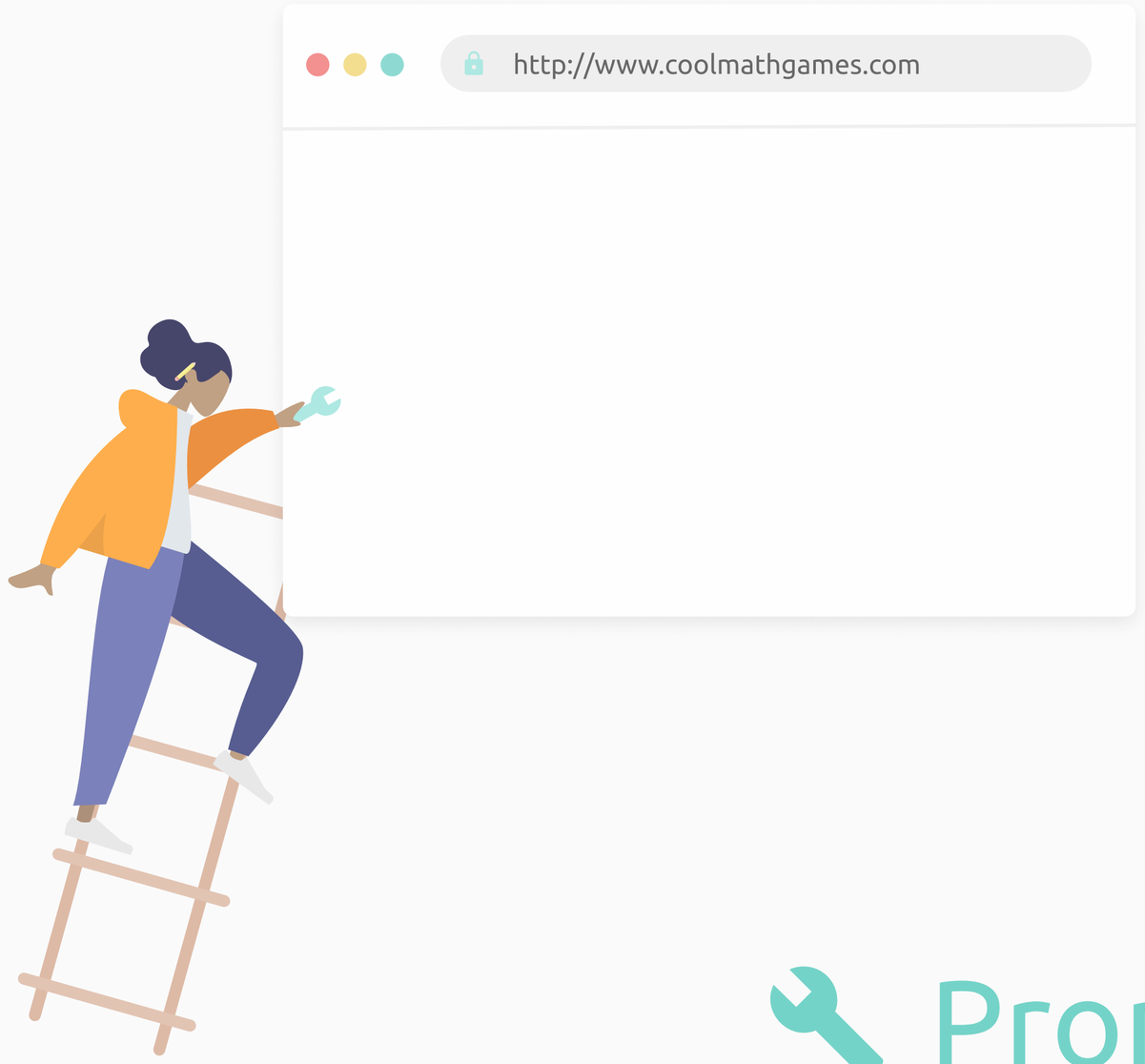
```
fruits = [
    fetch('myapi/apple'),
    fetch('myapi/orange'),
    fetch('myapi/banana'),
    fetch('myapi/pear')
]

Promise.race(fruits)
    .then(fruit => eat(fruits))


```

Promise built ins

Also there is Promise.race which resolves when the fastest promises in it’s list settles. Note that if any promise fails so does Promise.race, you can actually use Promise.any if you expect some to fail and just want to get the first success but note promise.any is **experimental**.

Promise Helpers

Slide 4 of 4



Promise Demo

main-content/lectures/async-promise-helpers-demo