Promises

WE'LL COVER THE FOLLOWING

Promises have been around for a while in JavaScript, but we have never had them in the language itself. In the past we would have had to use a library like `q` or `bluebird`. These libraries follow something called Promises/A+ , it is a standard that outlines how a Promise should behave, and it is the spec that ES6 Promises also implement.

A Promise represents the eventual result of some sort of asynchronous operation. Commonly these are used in conjunction with an HTTP request. However, it could actually just be any sort of event that will take some time to return. Promises can be a few different states. When a promise is created it has a 'pending' state. From there the Promise can have 2 other states, it can be fulfilled(commonly you will hear the word resolved used here), or rejected.

In ES6 the Promise is a constructor that when instantiated will return a new object. This constructor takes what is called an executor, which is a callback function that has two parameters.

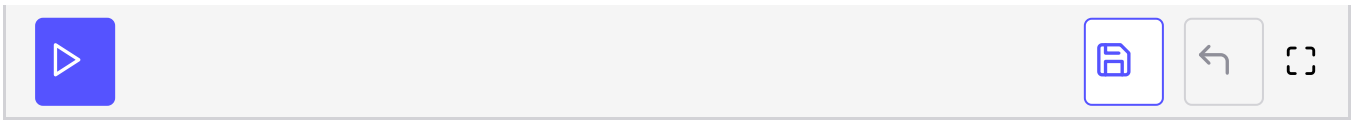
```
const myPromise = new Promise((resolve,reject) => {

});
```

In our executor we are provided with a `resolve` and a `reject` function that allow us to control the promise. Let's go ahead and make our first Promise and then resolve it.

```
const myPromise = new Promise((resolve, reject) => {
    resolve('Good to go');
});
```

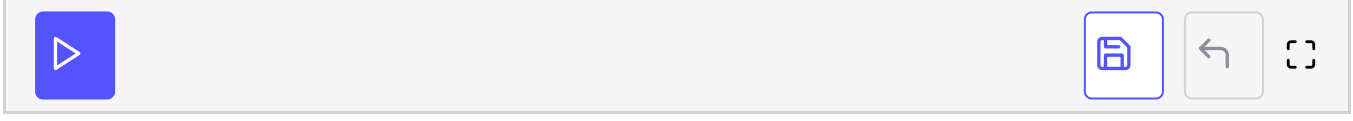
Here we are resolving immediately. We store the Promise on the `myPromise` variable which has an object that has a few methods on it, one of the most important being `.then`. The `.then` method is how we can see what has happened with our Promise, and it takes two callback functions itself. The first is what happens when a promise resolves successfully.

```
const myPromise = new Promise((resolve, reject) => {
    resolve('Good to go');
});

myPromise.then((res) => {
    console.log(res); //Good to go
});
```



The second would be if our Promise was rejected. Let's change our code to see this in action.

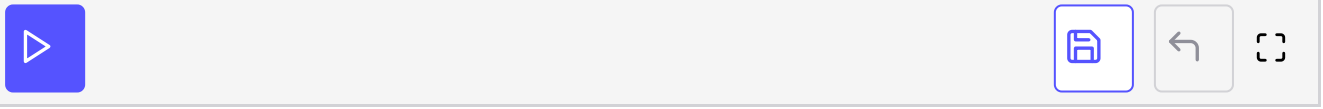
```
const myPromise = new Promise((resolve, reject) => {
    reject('Not good to go');
});

myPromise
  .then((res) => {
  //will not be called
  console.log(res);
}, (err) => {
  //will be called
  console.log(err); //Not good to go
});
```



Here we reject the Promise, and in this case the second callback from the `.then` method will be called.

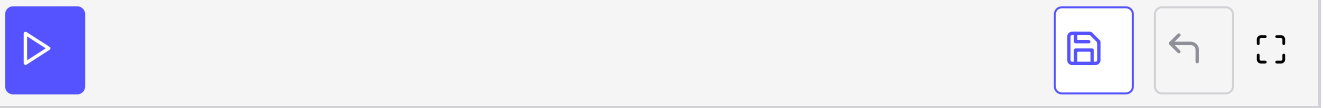
There is an easier way to catch a rejected Promise, and that is with the `.catch` method. We could change the above code to look like this.

```
const myPromise = new Promise((resolve, reject) => {
    reject('Not good to go');

});

myPromise
  .then((res) => {
  //Only if the promise resolved
  console.log(res);
})
  .catch((err) => {
  console.log(err); //Not good to go
});
```



Using the `.catch` method we can handle the rejected promise with ease. Since a promise is all about the eventual effects of some action, lets add a `setTimeout` to simulate that eventuality of an asynchronous event.

```
const myPromise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('Good to go')
  }, 1000)

  setTimeout(() => {
    reject('Uh ohh');
  },1500)
});

myPromise
  .then((res) => {
  console.log(res);
})
  .catch((err) => {
  console.log(err);
});
```



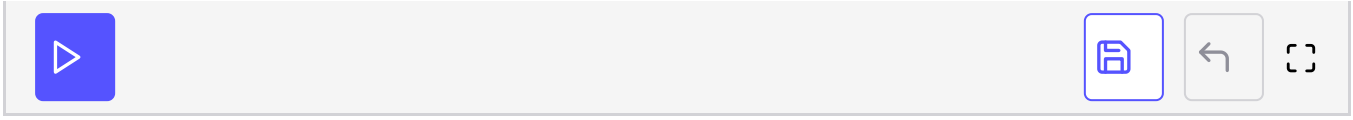
Looking at the above, we have two `setTimeout` functions that will run the `resolve` and `reject` functions at different times. From the code example above the `.then` success will be called since it resolves first. I should point out that once a Promise is either fulfilled or rejected its state can no longer change, so in this case the `Uh ohh` will never be seen.

all()

What about dealing with multiple Promises? Promises in ES6 has a static method called `.all` on it, meaning that we do not have to instantiate a new

Promise object, but rather we can just use the method straight up. First let's create another promise.

```
const myPromise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('Good to go')
  }, 1000)

  setTimeout(() => {
    reject('Uh ohh');
  },1500)
});

const promiseSequel = new Promise((resolve,reject) => {
  setTimeout(() => {
    resolve('Promise 2, the promising');
  },500);

  setTimeout(() => {
    reject('Box office bomb');
  });
});

Promise.all([myPromise,promiseSequel])
  .then((...resolvedData) => {
  console.log(resolvedData);
})
  .catch((...errors) => {
  console.log(errors);
});
```



The `.all` method will take an iterable object (in our case an array) and wait until all of the Promises resolve. If even ONE of them rejects the `.catch` or second callback from `.then` will be called.

race()

The Promise object also has a method called `.race`, this is similar to `.all` however it will only be called once and this is when any of the provided promises either resolves or rejects. So if you just need any data as fast as possible, the `.race` method is for you.

fetch()

Something we should look at is `fetch` so that we can see this in a real world

Something we should look at is `fetch` so that we can see this in a real world situation. Fetch is a new API in the browser that allows us to make HTTP

requests, and it uses ES6 Promises under the hood.

Fetch takes a URL as the first argument and an object as the second that allows us to configure the request. For this example we can just pass the URL.

The fetch function returns a Promise so we can use the `.then` method on it. The value provided to

```
import "isomorphic-fetch"

fetch('http://pokeapi.co/api/v2/pokemon/1/')
  .then((res) => {
  return res.json()
})
  .then((data) => {
  console.log(data)
})
.catch((err) => {
  console.log(err);
});
```