Hello, in this video, I will be introducing the topic of generators, and will be giving an overview on what will be covered in this module. In short, generators are functions that can be paused and resumed. In addition, generators can send output when pausing and

receive input when resuming. Now the reason why generators are such a big deal and

why you should use them is because of how well they work with asynchronous functions, such as fetch. Imagine this. You have a generator function, and inside it you wanna make some asynchronous spec requests. Now you can make the fetch call, pause the generator, wait for

the fetch response to process outside of the generator, and then return the fetch fulfillment value back to the generator before resuming it. This allows for asynchronous code to be written inside the generator synchronously, and that's what's amazing about generators. In this module, I will be covering how to create a generator function, how to pause and restart a generator by iterating through it, and how to send output and receive inputs into a generator.

And lastly, I'll show you how to use generators with asynchronous promises.

**Introduction to Generators**

**What are Generators?**

Generators are functions that can be paused and resumed. Generators can send out values when pausing and take in values when resuming.

**Why are Generators important?**

Generators are important because they allow asynchronous functions to written like normal synchronous functions.

**What are we going to learn?**

* How to create and use generators
* How to send values in and out of generators
* How to use Generators with asynchronous functions

**Using Generators with Asynchronous Functions**

**Creating a Generator Function**

Sample code of a Generator function:

function\* genFunc() {

yield 'a';

yield;

yield 123;

return "finished";

}

**Function\* Keyword**

Generator functions look similar to regular functions, except that they have an asterisk (**\***) after the function keyword. This syntax may look similar to the pointer notation from other languages, but it is unrelated.

Notice how the **function\*** keyword is used the declare a Generator function:

function\* genFunc() { //notice the function\* keyword

}

**Yield Keyword**

The **yield** keyword is used to pause the generator. The **yield** keyword may also be used to receive input and send output from the generator.

Notice how the **yield** keyword is used to pause and send several different types of output from the Generator function:

yield 'a'; //pauses the generator and sends out a value of 'a'

yield; //pauses the generator and sends out an undefined value

yield 123; //pauses the generator and sends out a value of 123

**Return Value**

Generator Functions have an optional **return value**. Omitting the return value is equivalent to returning an undefined value. The **return value** of Generator functions is often left unused.

Notice the **return value** of the Generator function:

return "finished"; //return value of "finished"

### ****Iterating through a Generator Object****

#### ****Creating a Generator Object****

A Generator Object is returned from calling a Generator function. It is important to not confuse Generator Objects with Generator functions.

Notice how a Generator Object is created by calling a Generator function:

function\* genFunc() {

console.log("started");

yield 'a';

console.log("passed first yield");

yield;

console.log("passed second yield");

yield 123;

console.log("passed third yield");

return "finished";

}

var genObject = genFunc(); //creates a generator object called genObject

#### Iterating through a Generator Object with next()

Generator Objects conform to the iterator protocol and may be iterated with the **next()** method.

Generator functions are initially paused and the first call to **next()** starts the Generator function. The Generator function then runs until it hits the first **yield**keyword and then pauses. Subsequent calls to **next()** will resume the Generator function until the next **yield**keyword appears.

The **next()** method returns an object with two properties:

* done - a boolean indicating whether the Generator function has processed all of the **yield** statements or has already returned.
* value - the value associated with the most recent **yield** statement.

Notice how the **next()**method is used to iterate through all of the **yield**statements:

var a = genObject.next(); // Object {value: 'a', done: false}

//console.log("started");

var b = genObject.next(); // Object {value: undefined, done: false}

//console.log("passed first yield");

var c = genObject.next(); // Object {value: 123, done: false}

//console.log("passed second yield");

After all of the **yield**statements have been processed with **next()**, the following **next()** call returns an object with a value property equal to the Generator function **return value** and a done property set to true. If the return statement was omitted from the Generator function then the value property will be undefined. After the the done property is true in one of the returned objects, additional **next()**calls will return objects with an undefined value property and a true doneproperty. **Yield**statements after the return statement are ignored.

Notice how additional calls to **next()**behave:

var d = genObject.next(); // Object {value: "finished", done: true} <-- value property takes the return value of genFunc()

//console.log("passed third yield");

var e = genObject.next(); // Object {value: undefined, done: true} <-- additional next() calls return this

### ****Throwing Errors from within a Generator Function****

If an error is encountered within a Generator function, then the error will be thrown by the **next()**call that encounters the error. The **next()** call that throws the error will return an undefined value and additional yield statements after the error are ignored. Additional **next()** calls after the error will also return undefined values.

Notice the affects of throwing an error within a Generation function:

function\* genFunc() {

yield 'a';

yield 'b';

throw new Error("error thrown by genFunc()!");

yield 'c';

yield 'd';

}

var genObject = genFunc();

try{

var a = genObject.next(); // Object {value: 'a', done: false}

var b = genObject.next(); // Object {value: 'b', done: false}

var c = genObject.next(); // undefined <-- since an uncaught error was thrown, the generator function terminated

//console.log("error thrown by genFunc()!") occurs

var d = genObject.next(); // undefined <-- other yield statements are ignored after the error

}

catch(e){

console.log(e.message);

}

### ****Yielding to other Generators****

#### ****Yield\* Keyword****

The **yield\*** keyword is used to call another Generator function within a Generator function.

Notice how the **yield\***statement is used to call genFuncA() within genFuncB():

function\* genFuncA() {

yield 'a';

yield 'b';

yield 'c';

return "done with genFuncA()!"

}

function\* genFuncB(){

yield 1;

yield\* genFuncA(); // contains iterable [a,b,c]

yield 2;

yield 3;

return "done with genFuncB()!";

}

var genObject = genFuncB();

var a = genObject.next(); //Object {value: 1, done: false}

var b = genObject.next(); //Object {value: 'a', done: false}

var c = genObject.next(); //Object {value: 'b', done: false}

var d = genObject.next(); //Object {value: 'c', done: false}

var e = genObject.next(); //Object {value: 2, done: false}

var f = genObject.next(); //Object {value: 3, done: false}

var g = genObject.next(); //Object {value: "done with genFuncB()!", done: true}

The **yield\***statement does not add the return value of the generator function that it calls to its list of iterables. Instead, the return value may be accessed by the return value of the **yield\***statement.

Notice how the **yield\***genFuncA() statement returns the return value of genFuncA():

function\* genFuncA() {

yield 'a';

yield 'b';

return "done with genFuncA()!"

}

function\* genFuncB(){

yield 1;

var returnVal = yield\* genFuncA(); // contains iterable list [a,b] and returns with value "done with genFuncA()!"

yield returnVal; // returnVal is equal to"done with genFuncA()

yield 2;

return "done with genFuncB()!";

}

var genObject = genFuncB();

var a = genObject.next(); //Object {value: 1, done: false}

var b = genObject.next(); //Object {value: 'a', done: false}

var c = genObject.next(); //Object {value: 'b', done: false}

var d = genObject.next(); //Object {value: "done with genFuncA()!", done: false}

var e = genObject.next(); //Object {value: 2, done: false}

var f = genObject.next(); //Object {value: "done with genFuncB()!", done: true}

The **yield\***statement can be used on any iterable in addition to Generator functions.

Notice how the **yield\*** statement is used to **yield** all of the values of in an array:

function\* genFunc(){

yield 1;

yield\* [2,3,4]; //the array [2,3,4] is iterable

yield 5;

}

var genObject = genFunc();

var a = genObject.next(); //Object {value: 1, done: false}

var b = genObject.next(); //Object {value: 2, done: false}

var c = genObject.next(); //Object {value: 3, done: false}

var d = genObject.next(); //Object {value: 4, done: false}

var e = genObject.next(); //Object {value: 5, done: false}

var f = genObject.next(); //Object {value: undefined, done: true}

**More on Generator Objects**

### ****Sending input using next()****

In addition to iterating through Generator Objects, **next()** can also be used to send values back into Generator functions. This is accomplished by passing a value into the **next()** method call as an argument. The value that is passed into the **next()** method call eventually becomes the return value of the most recent **yield** statement. Since the first **next()** call starts the Generator function, any value that gets passed into it will be ignored.

Notice how the **next()**method call is used to send values back into the Generator function:

function\* genFunc(){

var a = yield;

console.log(a); //a = 1

var b = yield;

console.log(b); //b = 2

var c = yield;

console.log(c); //c = 3

}

var genObject = genFunc();

genObject.next(0); //starts genFunc(), the value inside the next() call is ignored

genObject.next(1); //sends a value of 1 to genFunc()

genObject.next(2); //sends a value of 2 to genFunc()

genObject.next(3); //sends a value of 3 to genFunc()

genObject.next(4); //the value inside next() is ignored because genFunc() has no more yields

The **next()** method can also be used to modify the values sent by the **yield**statement and send them back.

Notice how the **next()** method is used to obtain values from **yield**, modify them, and then send them back:

function\* genFunc(){

var a = yield 'a';

console.log(a); // a = 'a!'

var b = yield 'b';

console.log(b); // b = 'B'

var c = yield 'c';

console.log(c); // c = 'abc'

}

var genObject = genFunc();

var w = genObject.next(); //starts genFunc(), w = Object {value: 'a', done: false}

var x = genObject.next(w.value + '!'); //sends a value of "a!" to genFunc(), x = Object {value: 'b', done: false}

var y = genObject.next(x.value.toUpperCase()); //sends a value of 'B' to genFunc(), y = Object {value: 'c', done: false}

var z = genObject.next(w.value + x.value + y.value); //sends a value of 'abc' to genFunc(), z = Object {value: 'undefined', done: true}

### ****Other Methods to Iterate through Generator Objects****

#### ****For...Of****

Notice how the For...Of statement is used to iterate through a Generator Object:

function\* genFunc(){

yield 'a';

yield;

yield\* [1,2,3];

yield 123;

return "finished";

}

for (var x of genFunc()){ //for...of statement

console.log(x);

}

//Outputs:

//'a'

// undefined

// 1

// 2

// 3

// 123

// <-- return value is not outputted

#### ****Spread Operator (...)****

Notice how the spread operator is used to iterate through a Generator object:

function\* genFunc(){

yield 'a';

yield;

yield\* [1,2,3];

yield 123;

return "finished";

}

var arr = [...genFunc()]; //...spread operator

// arr = ['a',undefined,1,2,3,123]

#### ****Destructuring****

Notice how the destructuring assignment is used to iterate through a Generator object:

function\* genFunc(){

yield 'a';

yield;

yield\* [1,2,3];

yield 123;

return "finished";

}

var [a,b,c,d,e,f,g] = genFunc(); //destructuring assignment

// a = 'a'

// b = undefined

// c = 1

// d = 2

// e = 3

// f = 123

// g = undefined <-- g is undefined because there are no more yields

### ****Return()****

Generator Objects have a **return()**method that terminates the Generator function. **Return()** causes a return statement to be performed at the most recent **yield** statement. The **return()**method takes in one optional variable that is used as the return value of the Generator function. Calling **return(*x*)**will return an object with a value property equal to **x** and a done property of true. After **return()** is called, subsequent **yield**statements in the Generator function are ignored.

Notice how calling **return()** affects the generator function:

function\* genFunc(){

yield 'a';

yield 'b'

yield 'c'

return "finished";

}

var genObject = genFunc();

var a = genObject.next(); // a = Object {value: 'a', done: false}

var b = genObject.return('return() was called'); // b = Object {value: "return() was called", done: true}

var c = genObject.next(); // c = Object {value: undefined, done: true}

### ****Throw()****

Generator Objects have a **throw()** method that causes an error to be thrown at the most recent **yield**statement. The **throw()** method takes in one argument, which is commonly an Error object.

Notice how **throw()** affects the Generator function:

function\* genFunc(){

var a = yield 'a';

console.log(a); // a = 123

var b = yield 'b'; //exception is thrown, function exits

//the code below never occurs because an exception occurred and was uncaught

console.log(b);

var c = yield 'c';

console.log(c);

return "finished!";

}

var genObject = genFunc();

var w = genObject.next(); // w = Object {value: 'a', done: false}, starts generator function

var x = genObject.next(123); // x = Object {value: 'b', done: false}

var y = genObject.throw(new Error("error thrown!")); // thrown() is called, y = undefined

var z = genObject.next('abc'); // z = undefined

### ****Using Generators with Asynchronous Functions****

Generator functions work well with asynchronous functions that return Promises. This is because Generator functions can **yield** a Promise, process the Promise result asynchronously, and then receive the Promise result back. This allows asynchronous code to be written inside generator functions like normal synchronous functions.

Notice how Promises can be written in a synchronous way inside Generator functions:

function\* genFunc(){ //looks synchronously written

var post1title = yield fetch("https://jsonplaceholder.typicode.com/posts/1");

console.log(post1title);   
 //post1title = "sunt aut facere repellat provident occaecati excepturi optio reprehenderit"

var post2title = yield fetch("https://jsonplaceholder.typicode.com/posts/2");

console.log(post2title);  
 //post2title = "qui est esse"

}

var genObject = genFunc(); //creating generator object

var yieldedObject = genObject.next(); //starting generator and returning first yielded object

var promise = yieldedObject.value; //getting promise from value property of the yielded object

promise.then(function(val){ //callback for then() of promise

return val.json(); //getting json stream from fetch response

}).then(function(val){ //chaining another then()

var secondYieldedObject = genObject.next(val.title); //sending title back to generator function

//and receiving second yielded object from generator function

var secondPromise = secondYieldedObject.value; //getting promise from value property of second yielded object

secondPromise.then(function(val){ //callback for then() of promise

return val.json(); //getting json stream from fetch response

}).then(function(val){ //chaining another then()

genObject.next(val.title); //sending back the second title to the generator function

})

})

The code inside the generator function is clean and readable, however all the iterating code below it is a mess. Luckily, there is a recursive method for iterating through promises that will be covered on the next page.

### ****Recursive Method for Iterating through Promises****

A recursive function may be used to iterate through **yielded** Promises and return their fulfillment values back to the Generator function.

Notice how a recursive function is used to handle **yields** to Promises and **yields** to other values in any order:

function run(genFunc){

const genObject= genFunc(); //creating a generator object

function iterate(iteration){ //recursive function to iterate through promises

if(iteration.done) //stop iterating when done and return the final value wrapped in a promise

return Promise.resolve(iteration.value);

return Promise.resolve(iteration.value) //returns a promise with its then() and catch() methods filled

.then(x => iterate(genObject.next(x))) //calls recursive function on the next value to be iterated

.catch(x => iterate(genObject.throw(x))); //throws an error if a rejection is encountered

}

try {

return iterate(genObject.next()); //starts the recursive loop

} catch (ex) {

return Promise.reject(ex); //returns a rejected promise if an exception is caught

}

}

The run() function shown above takes in a Generator function as an argument and uses the recursive iterate() function to process through all of the Generator function's **yield** statements. If a Promise is **yielded**, the fulfillment value of that Promise is sent back to the Generator function. If an integer, string or object is **yielded**, then those values are sent back as is to the Generator function.

Notice how the run() function is used to process a Generator function:

function \*gen(){

var post1Stream = yield fetch("https://jsonplaceholder.typicode.com/posts/1");

var post1 = yield post1Stream.json();

console.log(post1.title);

//post1.title = "sunt aut facere repellat provident occaecati excepturi optio reprehenderit"

var post2Stream = yield fetch("https://jsonplaceholder.typicode.com/posts/2");

var post2 = yield post2Stream.json();

console.log(post2.title);

//post2.title = "qui est esse"

var number = yield 12345;

console.log(number)

//number = 12345

var string = yield "abc";

console.log(string)

//string = "abc"

var obj = yield {id:123,name:"xyz"};

console.log(obj)

//obj = Object {id:123,name:"xyz"}

var a = yield 54434337746;

console.log(a);

return "done";

}

run(gen).then(x => console.log(x)) //logs "done"

.catch(x => console.log(x.message));

If a rejected Promise is **yielded**, the run() method will stop iterating through the Generator function and return a rejected Promise.

Notice how the run() method handles rejected promises:

function \*gen(){

var post1Stream = yield fetch("https://jsonplaceholder.typicode.com/posts/1");

var post1 = yield post1Stream.json();

console.log(post1.title);

//post1.title = "sunt aut facere repellat provident occaecati excepturi optio reprehenderit"

var post2Stream = yield fetch("https://jsonplaceholder.typicode.com/posts/2");

var post2 = yield post2Stream.json();

console.log(post2.title);

//post2.title = "qui est esse"

var error = yield Promise.reject(Error("error message!"));

//error thrown here, generator function terminates

var number = yield 12345;

console.log(number); //doesn't occur because an earlier promise was rejected

return 'done'; //doesn't occur because an earlier promise was rejected

}

run(gen).then(x => console.log(x))

.catch(err => console.log(err.message); //logs "error message!" from the rejected Promise