

Welcome To Advanced NodeJS

Monday, October 12, 2020 10:38 AM

Advanced
Node JS

Assignment01

Monday, October 12, 2020 10:41 AM

- Create a function to find and return all primes in a given min and max range
 - Example find primes between 2 and 200
- Psudo code of isPrime

```
bool isPrime(int x){  
  
    If(x<2)  
        return false;  
  
    for(int i=2;i<x;i++)  
        If(x%i==0)  
            return false;  
  
    return true;  
  
}
```

The common problems

Monday, October 12, 2020 12:15 PM

```
15 function findPrimes(min,max){
16   //what to do with invalid argument
17   if(max<min)
18     return false;
19   let result=[];
20   for(let i=min;i<=max;i++){
21     if(isPrime(i))
22       continue;
23     for(var j=2;j<=i;j++){
24       if(i%j==0)
25         break;
26       if(j==i) //its a prime number
27         result.push(i);
28     }
29   }
30   return result;
31 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

(c) 2020 Microsoft Corporation. All rights reserved.

D:\OneDrive\myworks\corporate\202011-1ng-advnode\nodejsdemos>node "d:\OneDrive\myworks\corporate\202011-1ng-advnode\nodejsdemos\primeapp01.js" findPrimes(2,100) [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]

findPrimes(2,10) [2, 3, 5, 7]

findPrimes(10,2) false

Returning completely different type of values

- Client is forced to check the types

Recommendation!

- If you function returns an array, always return an array, may be an empty array when you have not value to return instead of returning false or null.

Don't return a value to indicate an error. If possible **throw exception or any standard Mechanism to indicate error.**

Loose types?

- Javascript as loose (dynamic) types.
- But to create a consistent API we must adhere to some common denominators

- Example a method may return

```
{
  status: 'success',
  data:[1,2,3,4]
}
```

Or

```
{
  Status:'failed',
  reason:'invalid range'
}
```

Different data

Common denominator

Nodejs is Single threaded Asynchronous Programming model

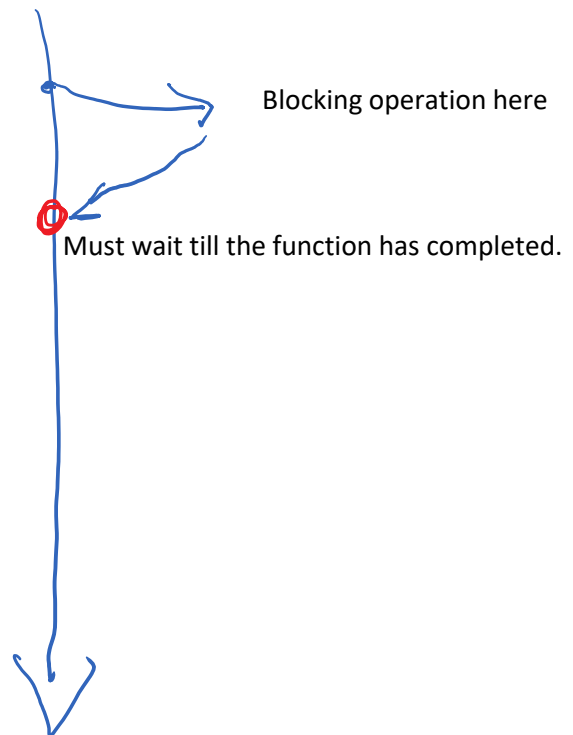
Monday, October 12, 2020 12:30 PM

NodeJS expects your functions to be async by default

- If your function is synchronous for whatever reason, it must be suffixed with the word sync

Note

- Languages like java and C# using async suffix to mark an asynchronous function.
- By default functions are synchronous
- NodeJS expects functions to be async by default.



Javascript Asynchrnous Programming

Monday, October 12, 2020 3:16 PM

- A general paradigm of programming, where we don't need to wait for a function to finish
 - Function returns immediately
 - Continues to work in backgournd
 - Updates the client once it finishes with the help of some kind of call back

Different Types of Asynchrnous Programming Model

1. NodeJS Callback pattern
 - a. Callback is not a new concept
 - b. NodeJS has a special callback syntax for function : `function callback(err,result);`
 - i. **We can use this model anywhere as this is just a pattern and now a NODE JS feature**
 - ii. **Most of the NodeJS API follow the same syntax.**
2. **ES2015 Promises**

Assignment 02

Monday, October 12, 2020 12:52 PM

1. Continue with Assignment01 and make the API asynchronous
2. Use Modular approach by separating business and presentation tier

NodeJS Callback Pattern

Monday, October 12, 2020 1:03 PM

1. NodeJS callback architecture

- Nodejs expects your functions not to return using return keyword
- You pass a callback as the last parameter to your function
- Once function finishes it calls the call back
- The callback should take two parameter in order
 - Err
 - Should specify in case of error
 - Second parameter should be null/undefined
 - Result
 - Err should be null
 - Result should contain the result

```
function findPrimesSync(min,max){  
  
    let result=[];  
  
    return result;  
}
```

Should change to

```
function findPrimes(min,max, cb){  
  
    let result=[];  
    if(success)  
        cb(null, result); //success  
    else  
        cb('invalid input'); //error  
}
```

```
function findPrimes(min, max, cb) {  
    setTimeout(() => {  
        if (min >= max)  
            cb(new Error(`Invalid Range(${min}-${max})`)); //result is undefined  
        else {  
            let primes = [];  
            for (let i = min; i < max; i++)  
                isPrime(i, (err, result) => {  
                    if (result)  
                        primes.push(i);  
                });  
            cb(null, primes); //first parameter null indicates success  
        }  
    }, 2); //just to simulate that job may take long time.  
}
```

Simulates a long running process

- Is running synchronously as one big chunk of code.
- Once you start, you end only after searching everything
- Not giving any other job time to work
- This is called **selfish** programming

Cooperative Worker Pattern

- A code should allow other codes to work by taking a break
- This should allow vital UI updates and other short worker to complete

How to implement co-operative worker in our code

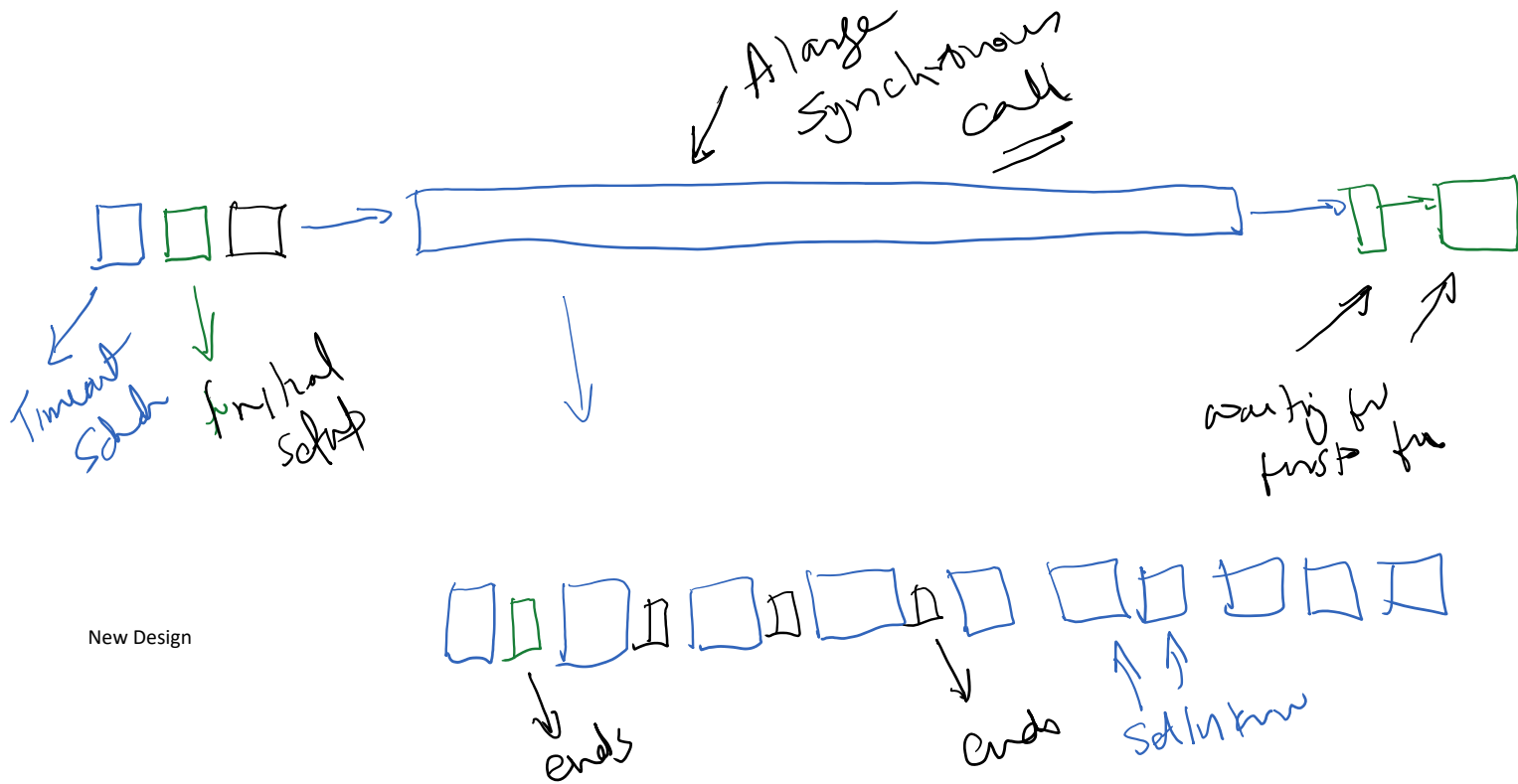
- Say we are finding all primes between 2 and 500000
- We may take a short break of say 10ms after every 1000 iteration.

Assignment03

Tuesday, October 13, 2020 10:43 AM

Cooperative Async Pattern

Monday, October 12, 2020 12:52 PM



ES2015 Promises

Monday, October 12, 2020 3:19 PM

- It is not a NodeJS feature but available in general in all javascript programming
- **Evolved much later**
- **NodeJS was already using its own model of programming**
- Many Nodejs libraries are now slowly moving to Promise rather than node callbacks

A Promise

- It's a built in ES2015 (Javascript feature)
- **Promise is an object that promises to get some result in future**
 - Promise also take a callback with two parameters
 - These two parameters are again call backs
 1. To call when success
 2. To call when failed
- Promise to get a result asynchronously by calling another function

Promise says let me run this code and I will let you know when we are ready

```
let promise= new Promise( function_that_will_give_you_a_result );
```

```
function function_that_will_give_you_a_result( fnResolve, fnReject ){  
  ...  
  if(success)  
    fnResolve( result); //call when you completed successfully  
  else  
    fnReject(err_details); //call this function when you fail  
}
```

This is your business logic

Creating an api — callback vs Promise

```
function findPrimes( min, max, cb ){  
  //business logic  
  ...  
  if(success)  
    cb( null, result);  
  else  
    cb(err_details);  
  //This function returns nothing  
}
```

```
function findPrimes( min, max ){  
  let promise=new Promise( (resolve,reject) =>{  
    //business logic here  
    if(success)  
      resolve( result);  
    else  
      reject(err_details);  
  }  
  return promise;  
}
```

No callback passed.

We handle promise once returned

Consuming The Asynchronous operations

```
//callback example  
findPrimes( 2, 100, (err,primes) =>{  
  if(err){  
    console.log('err',err); //on failure  
  } else{  
    Console.log('primes', primes.length); //on success  
  }  
});  
  
//we are free to do whatever we want  
//the callback will be called sometimes in future  
//same callback will get both err and result
```

```
//promise based design  
  
//function doesn't return result. It returns a future promise  
let promise= findPrimes(2,100);  
  
//we can set for future when it completes  
//if promise is resolved successfully  
promise.then( primes=> console.log('primes', primes.length);  
  
//if promise is rejected because of error  
promise.catch( err => console.log( 'err', err);  
  
//we can do whatever we want to do. then() and catch() will
```

execute asynchronously when promise is resolved/rejected in future.

//this code will execute immediately.

Promises can
Be chained

```
findPrimes(2,100)
  .then(primes=> console.log(primes))
  .catch(err=>console.log(err);
```

Nested Promise Problem

```
return new Promise((resolve, reject) => {
  factorial(n)
    .then(fn => {
      factorial(n-r)
        .then(fn_r => {
          factorial(r)
            .then(fr => {
              let result = fn / fn_r / fr;
              resolve(result);
            }).catch(reject);
        }).catch(reject);
      })
    .catch(reject);
});
```

Nested calls

1. Calculate factorial n
2. Calculate factorial of n-r
3. Calculate factorial of r
4. Use the first 3 calculation to calculate combination

- Can you see the sequence in nested promise?

This calculation depends on all the three

Async - Await Keywords

- Since Promise is a javascript feature, javascript has defined a set of keywords that makes working with Promise easy and straight forward.
- **await** is a javascript keyword that automatically resolves the promise and give you resolved result rather than promise
 - Remember this result will not come immediately but sometimes in future
- When you use **await**, the rejection is thrown as an exception that can be handled using standard **catch** keyword
- The function is actually waiting for resolved/rejected, but will finish immediately asynchronously
 - It will execute the code later.

Manual Promise Resolution

```
function testFactorial(n){
  let p = factorial(n); //it returns a promise

  //wait for promise to complete and get resolved result
  p.then( fn => console.log(fn));

  //if promise is rejected you get rejection message
  p.catch( err=> console.log('err',err) );
}
```

Using await

```
async function testFactorial(n){
```

```
  try{
    //await will wait for promise resolution.
    let fn = await factorial(n); //taking n*100ms

    //next piece of code is what you would write in then, to be executed in future
    console.log('result is ',fn); //typically what you write in then
  }
```

```
  catch(err){ //rejection is handled in catch
    console.log('err',err); //what you write in .catch()
  }
}
```

1. Looks like this code is synchronous. But actually there may be long gap between these two lines
2. This code may run in future but the function will return immediately

- Function having await must be marked **async**
- An async function always returns a **Promise implicitly**

Anything that follows await will be executed later and therefore this function creates a Promise and returns immediately

```

let combination=(n,r)=>{
  //factorial(n)/factorial(n-r)/factorial(r)
  return new Promise((resolve,reject)=>{
    factorial(n)
    .then(fn=>{
      factorial(n-r)
      .then(fn_r=>{
        factorial(r)
        .then(fr=>{
          let result= fn/fn_r/fr;
          resolve(result);
        }).catch(reject);
      }).catch(reject);
    }).catch(reject);
  });
});

```

```

async function comibnation(n,r){
  let fn= await factorial(n);
  let fn_r=await factorial(n-r);
  let fr=await factorial(r);
  let c= fn/fn_r/fr;
  return c;
}

```

1. Awaits (resolves then) and gets you resolved result fn
 - a. But this will happen in future. So it is just a promise
2. Second will execute once the first promise is resolved.
 - a. It is a promise against a promise.
 - b. It is also future tense
- 3.

What is this returning

- Since an async function always returns a promise
 - We can always use it with then() and catch() if we need

await must always be written inside an async function

- You can't write await in global
- Constructor of a class can't be marked async
 - You can't await inside a constructor
 - You can use standard then(),catch()

- It appears that this function is returning a number
- But this number depends on other calculation which are based on promises
- **Here we are telling that we will return this value to you in future**
- This function is returning a **Promise** that will have this value

Understanding Promises

Tuesday, October 13, 2020 9:59 AM

```
function combination(n,r){
  let fn = factorial(n);
  let fn_r = factorial(n - r);
  let fr = factorial(r);
  var comb='waiting for the result...';

  Promise.all([fn,fn_r,fr]) //when all promises are fulfilled (resolved/rejected)
    .then((result) => {
      //result[0] is output of promise fn
      console.log(result[0], result[1], result[2]);

      //we will reach here in apporx 1400ms for comibination(7,2):
      comb = (result[0] / result[1] / result[2]);
    })
    .catch(function(err){
      reject("combination Error: " + err);
    });

  //we reach here immediately without waiting for promise to be fulfilled.
  console.log("Calculate Factorial: " + comb);
}

combination(7, 2);
```

Will be evaluated sometimes in future

We reach here in present, immediately long before the calculations are done.

To calculate the comination we need another calcuation.

```
//let us make a mega promise which is a promise of all promises
let megaCombProm = new Promise(function(resolve,reject){

  return Promise.all([fn,fn_r,fr]) //when all promises are fulfilled (resolved/rejected)
    .then((result) => {
      //result[0] is output of promise fn
      console.log(result[0], result[1], result[2]);

      //we will reach here in apporx 1400ms for comibination(7,2);
      comb = (result[0] / result[1] / result[2]);
      //we must mark the promise resolved.

      resolve(comb); //which promise are we resolving?
    })
    .catch(function(err){
      reject("combination Error: " + err);
    });

});

megaCombProm.then(function (comb){
  console.log("Calculate Factorial: " + comb);
}).catch(function (err) {
  console.log("Calculate Factorial Error: " + err);
});
```

Promise to calculate the combination when other promises are fulfilled

We don't need another promise to wrap this promise!

```
14 async function combination(n,r){
15   // try{
16   let fn = factorial(n);
17   let fn_r = factorial(n - r);
18   let fr = factorial(r);
19   var comb='waiting for the result...';
20   let result=await Promise.all([fn,fn_r,fr]);
21   //any exception is automatically wrapped in reject()
22   comb = (result[0] / result[1] / result[2]);
23   return comb; // internally resolve(comb)
24   // } catch(err){
25   //   console.log('err',err);
26   // }
27 }
28
29 combination(7, 2).then(console.log).catch(console.log);
30
31
32
33
34
35
36
37
38
```

```
14 function combination(n,r){
15   let fn = factorial(n);
16   let fn_r = factorial(n - r);
17   let fr = factorial(r);
18   var comb='waiting for the result...';
19   //This promise is a promise to calculate combination
20   //when factorial promises are fulfilled.
21   return new Promise((resolve,reject)=>{
22     return Promise.all([fn,fn_r,fr]) //when all promises a
23     .then((result) => {
24       //we will reach here in apporx 1400ms for comb
25       comb = (result[0] / result[1] / result[2]);
26       resolve(comb);
27     })
28     .catch(function(err){ //you must manually catch
29       reject(err); //and re-reject it
30     });
31   });
32 }
33
34 combination(7, 2).then(console.log).catch(console.log);
35 console.log('waiting for the combination...');
36
37 //combination(7, -2).then(console.log).catch(console.log);
38
```

If an inner promise is rejected

- You must write catch()
- If you don't want to handle rejection you still must
 - Write a catch
 - **Re-reject it**

```
36 combination(-7, 2).then(console.log).catch(console.log);
37
38
36 console.log('waiting for the combination...');
37
38 //combination(7, -2).then(console.log).catch(console.log);
```

Async await benefits

1. Code looks sequential.
2. Return is automatically translated to resolve
 - a. If no return is specified end of function is resolve
3. Any rejection is an exception thrown.
 - a. You don't have to handle the exception if you don't need
 - b. If you don't write try catch, it is automatically re-rejected.

Assignment 04

Monday, October 12, 2020 3:41 PM

- Convert findPrimes from callback to Promise model
- Write the test application

Assignment05

Monday, October 12, 2020 4:32 PM

Create a long running factorial function.

- Psudo code for factorial

```
int factorial(int n){
    if(n<0) //error

    let fn=1;

    while(n>1)
        fn*=n--;

    return fn;
}
```

Assume factorial is a long running task and needs $n \cdot 100$ ms to complete

1. Create an asynchronous factorial function that returns in $n \cdot 100$ ms.
 - a. It should return a promise
2. Use the factorial function to calculate combination(n, r); pseudocode for combination is

```
int combination(int n, int r){
    int fn=factorial(n);

    int fn_r=factorial(n-r);

    int fr=factorial(r);

    return fn/fn_r/fr;
}
```

Comination will not have any delays programmed.
It will be delayed because of factorial

Assignment06

Tuesday, October 13, 2020 10:41 AM

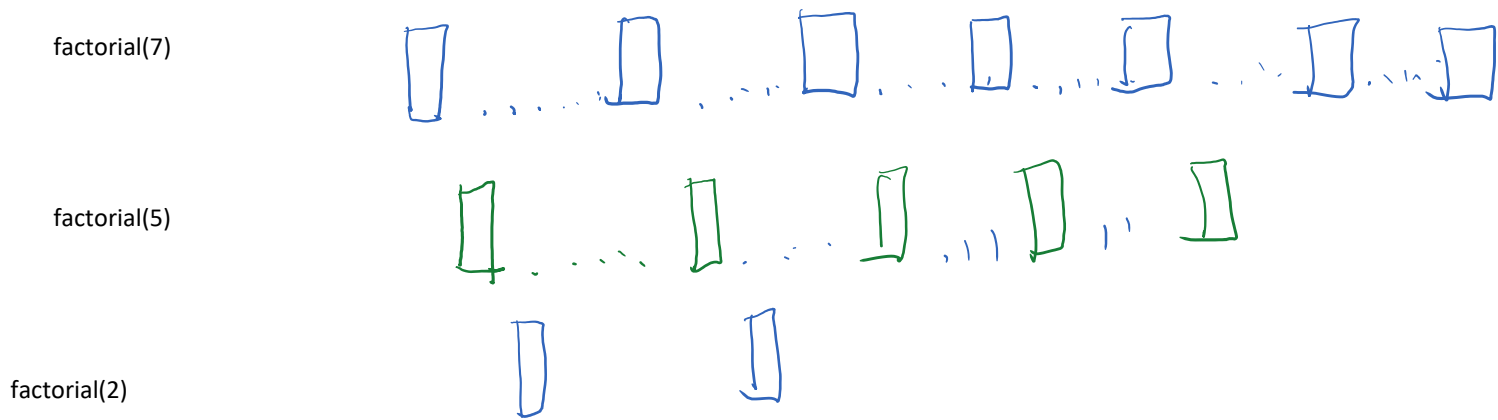
Convert the factorial function given below to a cooperative function

- It should still take $n \times 100\text{ms}$ to complete successfully
- It should take 100ms if it fails

```
let factorial=(number)=>{
  return new Promise((resolve,reject)=>{
    setTimeout(()=>{
      if(number<0)
        reject('negative numbers do not have factorial');
      let f=1;
      while(number>0)
        f*=number--;
      resolve(f);
    }, (number>0?number:1)*100);
  });
};
```

How async code works

Tuesday, October 13, 2020 11:31 AM



Convert Normal Call to Promise

Tuesday, October 13, 2020 11:46 AM

```
lib > JS utils.js > sleep
1
2
3 async function sleep(ms){
4
5   return new Promise(resolve=>{
6     setTimeout(resolve, ms); //this promise will be resolve
7   });
8 }
9
10
11 module.exports = {sleep};
```

```
lib > JS math.js > factorial
49
50
51 async function factorial(number){
52
53   await utils.sleep(100);
54   if(number<0)
55     throw `negative numbers don't have factorial ${numb
56   let factorial=1;
57
58   while(number>1){
59     await utils.sleep(100); //called at an interval of
60     factorial*=number--;
61   }
62
63   return factorial; //resolve
64
65 }
66
67
68
69 let combination(n,r)=1/factorial(n)/factorial(r)/factorial(n-r);
```

A Normal callback like sleep can be converted to a Promise
By this conversion we get an opportunity to utilize async-await
Features of JavaScript

The code looks more sequential now.
Now you can convert your sequential logic easily
To async logic

Handle Large Data

Tuesday, October 13, 2020 11:57 AM

Let us revisit our logic to find all primes between 2-500,000

- It takes **roughly** ~44 seconds complete
- It returns a **array** of ~41K+ primes

Use cases -- what will you do after getting 41K primes?

- What are the possible usage of these 41K values?
 - Display all values
 - Save all values to disk
 - Send values across network
 - Calculate the sum of those values
 - Find First 1000 primes ending with 7 eg--> 7,17,37,47,67...
- Think instead of searching for primes, you have searched for products on Amazon or Google
 - Display a list of values
 - Select one of those values

Important Consideration!

- In which of the use cases do you need all those values together?
 - Most of these cases needs values one by one.
- Are you sure you will use all the values?
 - After a google/amazon search that returns 100 pages of results, how many pages you actually see?
 - What

Problem

- We may never use the entire data set generated.
- If we use entire dataset we still process **one information at a time**
- **We can't use the first prime number till we have calculated all the 41K+ prime number**
 - Can't I use results in smaller chunk and not wait for complete calculation.

Handling Large Data Options

Tuesday, October 13, 2020 12:29 PM

We can apply different techniques

Two important techniques

1. ES2015 generator.
 - a. It is like java iterator or c# enumerators
 - b.

2. Nodejs Events

Generators

Tuesday, October 13, 2020 12:30 PM

- Javascript has the concept of a generator like C# and Python.
- A generator is based on a new keyword **yield**
- **yield** looks like **return** but works differently

Return statement

```
function getResult(){  
    return 1; // returns 1 and exist the program  
    return 2; //unreachable code  
    return 3; // unreachable code  
}
```

```
console.log(getResult()); //1  
console.log(getResult()); //1  
console.log(getResult()); //1
```

Return statement

//A function that has **yield**, must have **"**"** prefix

```
function *getResult(){  
    yield 1; // returns 1 and exist the program  
    yied 2; //unreachable code  
    yield 3; // unreachable code  
}
```

let x= getResult(); //you get a result which is not 1

```
15 console.log('testing yield...');  
16 function *getValues(){  
17     yield 1;  
18     yield 2;  
19     yield 3;  
20 }  
21  
22 let x=getValues(); //returns a generator  
23  
24 console.log('x',x);  
25  
26 console.log('x.next()',x.next()); //returns value: first yield, done: false suggests there may be more values  
27  
28 console.log('x.next()',x.next()); //returns value: second yield, done: false suggests there may be more values  
29  
30 console.log('x.next()',x.next()); //returns value: third yield, done: false suggests there may be more values  
31  
32 console.log('x.next()',x.next()); //returns value: undefined, done: true as we have gone past the last yield
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
getResult() 1  
testing yield...  
x Object [Generator] {}  
x.next() { value: 1, done: false }  
x.next() { value: 2, done: false }  
x.next() { value: 3, done: false }  
x.next() { value: undefined, done: true }
```

D:\OneDrive\myworks\corporate\202011-lng-advnode\nodejsdemos>

```
eld03.js X
ield03.js > [0] range

let range= function *(min,max) {

  console.log('starting the range...');

  for(let x=min; x<max; x++){
    console.log('yeielding ',x);
    yield x;
  }

  console.log('end of range...');
}

let g= range(0,3); //generates 0,1,2

console.log('g',g); //note range function hasn't executed any code yet.

console.log('reaches first yield',g.next()); //here all codes till first yield execute, but no further

//notice we have not completed loop yet. if we don't call next further, no further calculation will happen.

console.log('reaches second yield',g.next()); //executes code till next yield and then wait for another next call

console.log('reaches last yield',g.next()); //this will encounter our last yield, but program hasn't finished yet.

console.log('reaches the end of code',g.next()); //executes the rest of the code to realize that there is no more yield pending.

console.log('once you are past the last line of the code');

console.log('end of code reached by earlier call, so no action here',g.next()); //no more execution as you already gone past last line of
```

Generated Values can easily be stored in an array we need them together using loop or spread operator.

```
primeapp09.js > ...
1
2 let {primeRange} =require('./lib/primeutils3');
3
4 //what if I need array of primes
5 //... spreads any iterator/generator as individual which are collected in a list
6 let primeList= [ ... primeRange(2,100)];
7
8 console.log(primeList);
```

```
primeapp09.js
1
2 let {primeRange} =require('./lib/primeutils3');
3
4 let last=0;
5 let count=0;
6 for(let prime of primeRange(2,100)){
7   last=prime;
8   count++;
9   if(count==10)
10    break;
11 }
12 console.log('prime$(count) is $(last)');
13
14

primeutils3.js
lib> # primeutils3.js > primeRange
85
86
87 > function promisedPrimes(min, max) {
137
138
139 function * primeRange(min,max){
140
141   for(let i=min;i<max;i++){
142     if(!isPrimeSync(i)){
143       console.log('prime is ',i);
144       yield i;
145     }
146   }
147
148
149
150
151
152 module.exports = {
```

Once I break no further code is executed in generator function

Problem

- If we stop to call next() of a generator, generator code will not execute further
- **BUT IT WILL NOT EXIT EITHER.**
 - **THE GENERATOR FUNCTION WILL REMAIN SUSPENDED**
 - **ALL RESOURCES AND MEMORY ALLOCATED TO IT WILL ALIVE**

Solution -- communicating to generator using next()

- Generator is actually a two way communication!
- We can supply a value to generator function using the call to **next**
- This value is obtained by taking a return from the yield call.



```
let range= function *(min,max) {
  console.log('starting the range...');
  for(let x=min; x<max; x++){
    console.log('yeielding ',x);
    let clientToken=yield x;
    if(clientToken && clientToken.kill)
      break;
  }
  console.log('end of range...');
}

let gen=range(1,15);
let x=gen.next();
while(!x.done){
  console.log(x.value);
  if(x.value==5){
    gen.next({kill:true});
    break;
  }
  x=gen.next();
}
```

Parameter pass to next()

Can be collected by generator from yield statement.

You may pass signals like

- Stop
- Skip
- Reset()
- Start

In our example signal is a call to terminate the generator function

- Once the function terminates it releases all the resources

NodeJS Events

Tuesday, October 13, 2020 2:31 PM

- NodeJS has an event mechanism. You code can send information in small chunks to the caller using event rather than return.

Events vs Promises

How are they similar

- An async function may return either Promise or a Events
- User handle the promise in **then()/catch()** and they can listen to events in **on()**

```
function primePromise(){
  return new Promise(...){
    resolve(result_as_bulk);
  };
}

primePromise.then(result_as_build=>doSomething(result_as_bulk))
  .catch(...);
```

```
function primeEvents(){
  let event=new EventEmitter();
  ...
  ...
  event.emit( result_chunk);
  return event;
}

primeEvents().on( event_chunk => so_something(event_chunk);
```

How are they different

- Frequency of call**
 - Promise is resolved only once** and returns the entire data in one go.
 - Not great for large amount data
 - Client must wait till entire data is ready
 - Events can be triggered multiple times**
 - You can send data in small unit multiple times
 - You can use fetch and emit loop
 - In our example you can emit each prime number one by one
- Type of Signals**
 - Promise had two fixed types** — resolve and reject
 - We can't specify what is resolved if there are different type of elements resolved
 - Events has no fixed types**
 - They can define any number of **custom events** and send different data with each of them.
 - There is no separate **reject** equivalent. If error can be considered as a type
- Type of object**
 - Promise is a ES2015 object available to all javascript programs**
 - EventEmitter is a nodejs object which is part of event-emitter module**

Note:

EventEmitter is present in module **event-emitter**

You need to require it

```
function process( ... data){
  let event=new EventEmitter();

  If(data.length==0)
    event.emit('error', 'no data supplied'); //sends error

  for(let value in data){
    event.emit('processing', value); //sends processing
    let result=process(value);
    event.emit('processed', value, result); //sends processed
  }

  event.emit('done'); //sends a done signal

  return event;
}

process(1,2,3,4)
  .on('error', msg=>{})
  .on('processing', value=>{})
  .on('processed', (value,result)=>{})
  .on('done',()=>{});
```

Assignment 07

Tuesday, October 13, 2020 2:52 PM

- Create a function **fetchPrimes** that should be an event based model
 - Should return error as an event
 - The function should take a task id
 - Should return each prime number as they are found with format {id: 1, index:1, prime:2}, {id:2, index=2, prime:3}
 - Should return the progress as an event {id:1, progress:12} <--12% progress
 - Should return completed event
- Write the application to test the events