

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     }
24     for(var j=2;j<=i;j++){
25       if(i%j==0)
26         break;
27       if(j==i) //its a prime number
28         result.push(i);
29     }
30   }
31   return result;
}
```

Terminal Output:

```
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

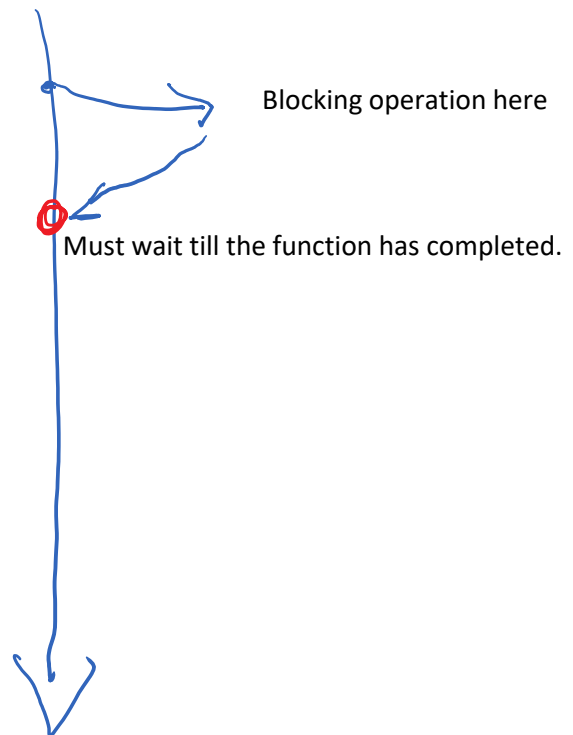
Monday, October 12, 2020 12:30 PM

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

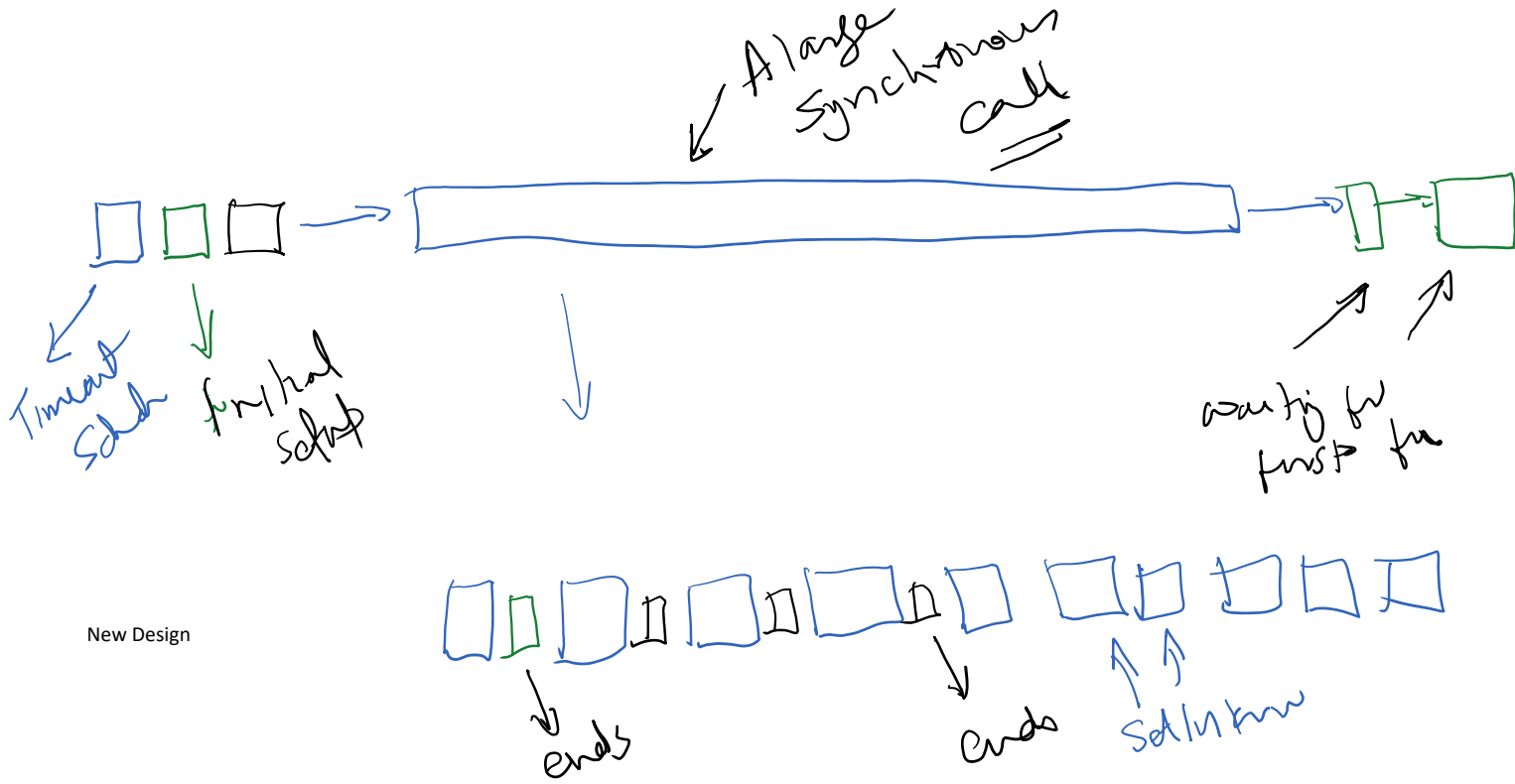
- Say we are finding all primes between 2 and 500000
- We may take a short break of say 10ms after every 1000 iteration.

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 $n \times 100$ ms to complete

1. Create an asynchronous factorial function that returns in $n \times 100$ 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

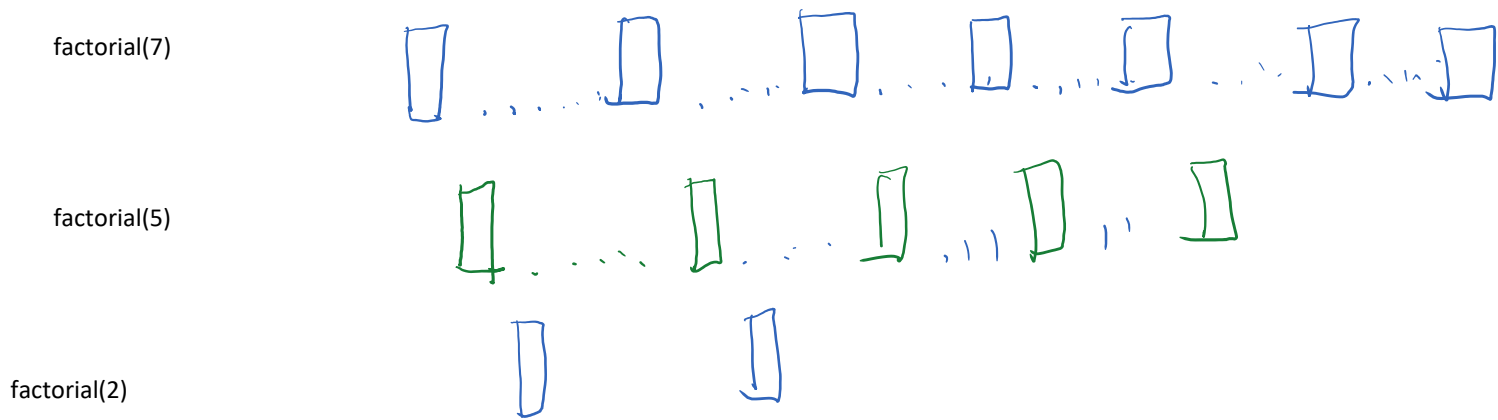
Convert the factorial function given below to a cooperative function

- 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);

```

VS Code interface showing two files:

- 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

```
- primeutils3.js**:


```

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 module.exports = {
152

```

Handwritten note with a green arrow pointing from the `yield i;` line in `primeutils3.js` to the `primeRange(2,100)` call in `primeapp09.js`:

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',()=>{});
```

```
function primeEvents(){
  Let event=new EventEmitter();
```

```

...
...
event.emit( result_chunk);
return event;
}

```

Event has already been emitted.

You get event object only when all processing have finished.
It's like inviting you to an event that has ended.

```
function primeEvents(){
```

```
  let event=new EventEmitter();
```

```
  ...
```

```
  setTimeout(()=>{
```

```
    event.emit( result_chunk);
  },0);
```

```
  return event;
```

```
}
```

```
function fetchUrl(url){
```

```
  let event=new EventEmitter();
```

```
  request.get(url, (err,data)=>{
    if(!err)
      event.emit('response', data);
  });
```

```
  return event;
```

```
}
```

Event must be emitted from within some callback that will execute
Sometimes in future after the event object is made available to the client
That callback may be a timed callback or external request callback.

External request callback--

- Talking to OS (file read write)
- Database calls
- Networks calls
- Interprocess communication.

Natural time delays

This event shall be emitted in future

Long after

We returned the event object

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

```
let EventEmitter = require('events');

function fetchPrimes(min,max,id){
  let event=new EventEmitter();
  event.emit('start', 'fetching has started...');
  return event;
}
```

1 let {fetchPrimes} =require('./lib/primeutils3');
2
3
4 //before returning the event, it has been emitted
5 let e=fetchPrimes(2,100)
6
7 //we are trying wait for an event which has already been trigge
8 e.on('start',console.log);

Here is the Event is emitted before it is given to client and client got any change to Listen to it.
Provide client the event object and let them register before events are emitted.

```
let EventEmitter = require('events');

function fetchPrimes(min,max,id){
  let event=new EventEmitter();
  setTimeout(()=>{
    event.emit('start', 'fetching has started...');
  },1)
  return event;
}
```

1 let {fetchPrimes} =require('./lib/primeutils3');
2
3
4 //before returning the event, it has been emitted
5 let e=fetchPrimes(2,100)
6
7 //we are trying wait for an event which has already been trigge
8 e.on('start',console.log);

Step 1 to 4 happens immediately
Step 5 happens after a delay of 1ms
By Now we are ready to handle the event
Step 6 is handling event whenever they are emitted

Assignment 08

Tuesday, October 13, 2020 5:26 PM

- All fetchPrime functions to get aborted on the client request
- Request could be sent through the event emitter

Assignement 09

Tuesday, October 13, 2020 5:27 PM

- Create a js program to read a file and display the progress bar while it is being read
 - Verify if all the bytes are read
 - Display necessary stats about the file
-
- Also create a file copy function using createReadStream and createWriteStream

NodeJS Streams

Wednesday, October 14, 2020 10:18 AM

- NodeJS supports the concept of streams.
- There are three broad type of streams
 - Readable Stream
 - Writable Stream
 - Transform Stream
- Each Stream is typically (like an interface) having a set of
 1. **Standard methods** that should be present in the stream object
 - **A standard set of events** that the stream object may emit.
 - Standard events mean events with a
 - Particular name
 - Particular payload (data that is sent with particular event)

Readable Stream

- A Stream from which we can read the data
 - createReadStream returns stream of data from a file
- It contains
 - Methods
 - read()
 - Read the data from stream
 - Generally done when it is readable
 - Pause()
 - ◆ Pause the reading
 - Resume()
 - ◆ Resume the reading
 - close()
 - Events
 - 'data'
 - Tells some data is available for reading
 - 'end'
 - Tells we have reached the end of our stream
 - 'error'
 - Informs about the error
 - 'close'
 - Stream has been closed
 - 'readable'
 - Stream is ready to be read

Readable Streams

Events

- data
- end
- error
- close
- readable

Functions

- pipe(), unpipe()
- read(), unshift(), resume()
- pause(), isPaused()
- setEncoding()

Writable Streams

Events

- drain
- finish
- error
- close
- pipe/unpipe

Functions

- write()
- end()
- cork(), uncork()
- setDefaultEncoding()



Readable Streams

Events

- data
- end
- error
- close
- readable

Functions

- pipe(), unpipe()
- read(), unshift(), resume()
- pause(), isPaused()
- setEncoding()

Writable Streams

Events

- drain
- finish
- error
- close
- pipe/unpipe

Functions

- write()
- end()
- cork(), uncork()
- setDefaultEncoding()



Writeable Stream

Events

- **drain**
 - We have consumed the data earlier supplied
 - It has been written
 - We are ready for more data

Communication between ReadStream and WriteStream

Readable Streams

Events

- data
- end
- error
- close
- readable

Functions

- pipe(), unpipe()
- read(), unshift(), resume()
- pause(), isPaused()
- setEncoding()

Writable Streams

Events

- drain
- finish
- error
- close
- pipe/unpipe

Functions

- write()
- end()
- cork(), uncork()
- setDefaultEncoding()

Stream Dance

Pipe()

- The **Stream dance** can be automated by using the function **pipe()** on **ReadableStream** which takes a **WritableStream** as a parameter

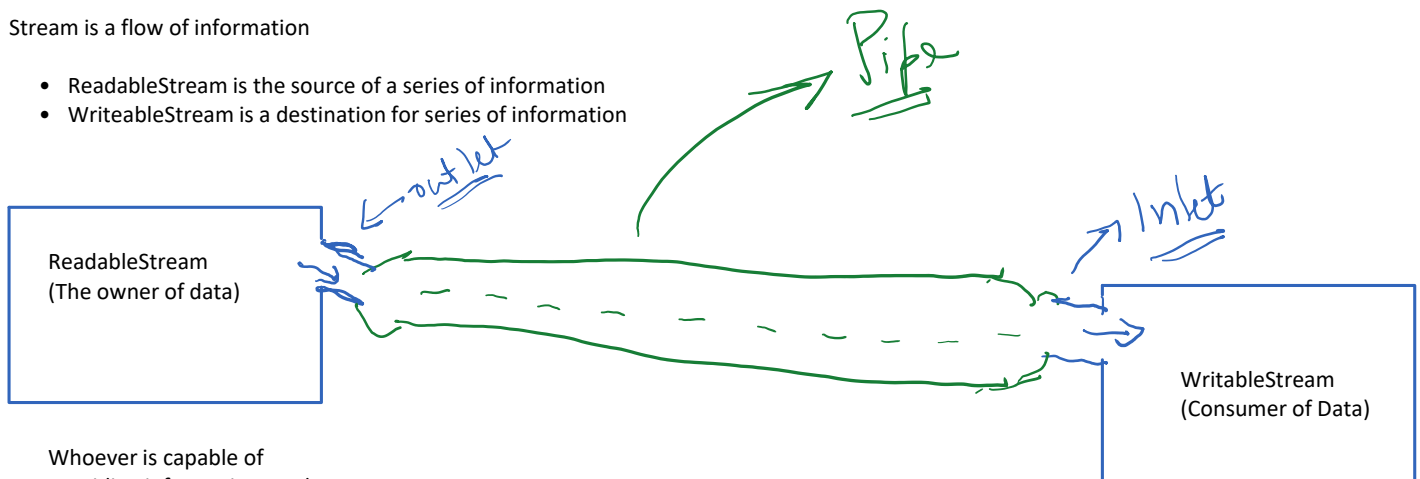
What is a Stream

Wednesday, October 14, 2020

12:21 PM

Stream is a flow of information

- ReadableStream is the source of a series of information
- WritableStream is a destination for series of information



Whoever is capable of
Providing information can be
Considered as a ReadableStream

Examples:

1. File
2. Network
3. **Computed Data Source**
4. Standard Input Device

Example

1. File
2. Network
3. Standard output device

Can I consider a process returning a series (like primes) be a ReadableStream?

YES

To be a Stream instead of returning custom events, we need to have standard events like data and end

How to create custom Readable Streams

1. Create your own type that extends Readable type
 - a. Chain constructor call
 - b. Copy prototype


```

let PrintStream=function(min,max){
  //Javascript Inheritance Step 1 -- chain constructor
  Readable.call(this); //chain the constructor to the super

  //user initialization here

}

//copy the prototype of Readable type to PrintStream type, so th
//PrintStream
util.inherits(PrintStream, Readable); //PrintStream gets all me

```

2. Define `_read()` method to emit
 - a. 'data'
 - b. 'end'
 - c. 'error'
3. You may translate your custom events to standard events

```

PrintStream.prototype._read = function() {
  let self=this; //generally this will be lost in nested callb

  fetchPrimes(this.min,this.max)
  //my api sends 'PRIME', but Readable is supposed send '
  .on('PRIME',data=>{
    //create a buffer from the json data you have
    let buffer= Buffer.from(JSON.stringify(data));
    self.emit('data',buffer);
  })
  .on('FINISHED',()=>{
    self.emit('end');
  })
  .on('ERROR',(error)=>{
    let buffer=Buffer.from(JSON.stringify(error));
    self.emit('error',buffer);
  });
};

```

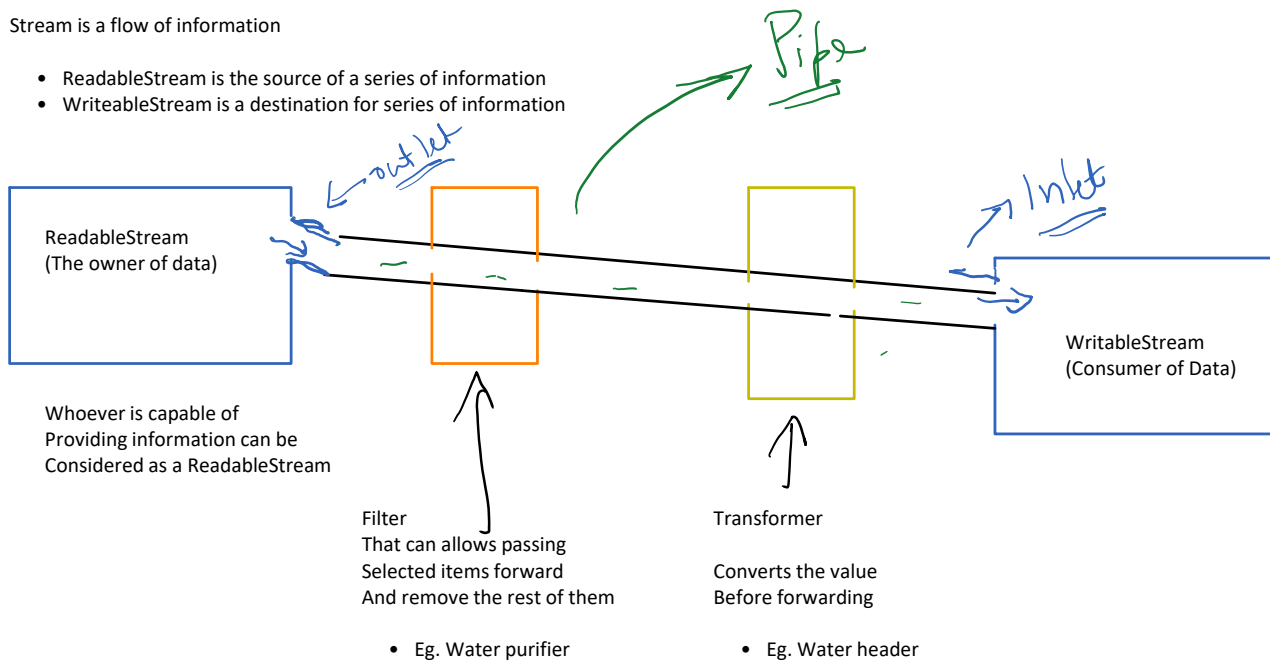
1. Define `_read` that should eventually emit
 - a. data
 - b. end
 - c. Error
2. Send data as buffer and not as plain data
3. Translate custom events to standard events

Transform Stream

Wednesday, October 14, 2020 12:21 PM

Stream is a flow of information

- ReadableStream is the source of a series of information
- WritableStream is a destination for series of information



TransformStream

- They have properties of both Readable and Writeable
- They can receive the data using WritableStream and forward data as RedableStream
- They can be added to pipe making a chain

e.g.

getEmployeees()

```
.pipe(filter(emp=> emp.dept=='training'))  
.pipe(filter(emp=>emp.salary>100000))  
.pipe(converter(convertToXml))  
.pipe(converter(zipCompress))  
.pipe(fs.createWriteStream())
```

1. Get a list of all employess
2. Pipe it to a filter that takes employees from training department
3. Pipe to another filter that takes employee with a min salary
4. Covert data to xml
5. Compress xml
6. Write to a file

Convert to Transform

```

let Converter=function(convertFunction){
    //Fixed Step 1:chain the constructor
    Transform.call(this);
    this.convertFunction=convertFunction; //this function will actually be used over the stream
}

//Fixed Step 2: inherits
util.inherits(Converter, Transform);

//Fixed Step 3: overwrite the required method
Converter.prototype._transform = function(chunk, enc, cb){

    let original=chunk.toString() ; //convert buffer into data

    let covertValue= this.convertFunction(original); //covert input data to desired type

    let outputBuffer= Buffer.from(covertValue.toString()); //create a new buffer

    this.push(outputBuffer); //send it to the client

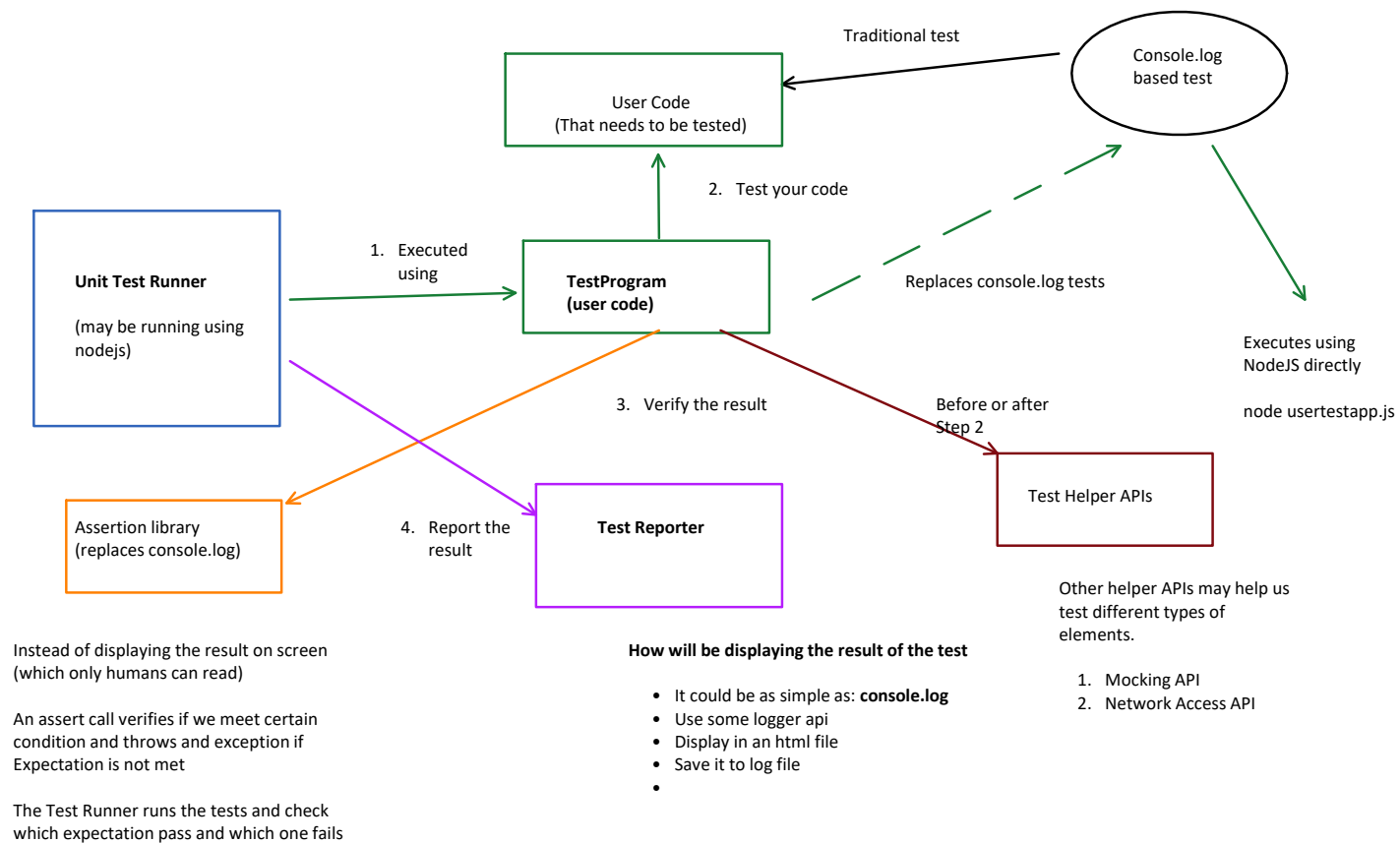
    cb(); //inform the system that convert is complete
};

```

N

Unit Testing Component

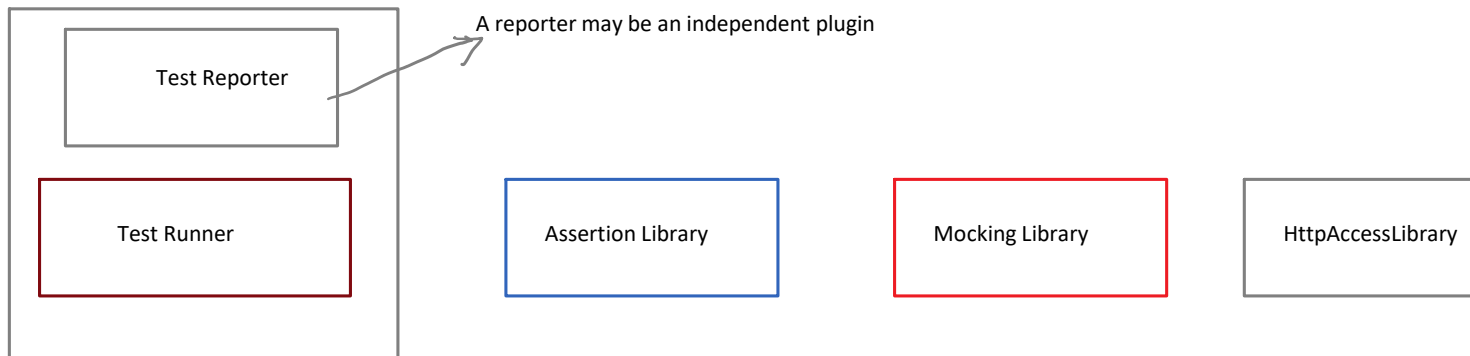
Wednesday, October 14, 2020 2:23 PM



Unit Testing framework

Wednesday, October 14, 2020 2:44 PM

- Provides one or more re-usable components required for unit testing
- All these components may not be part of the same product
- We may use different products at different levels



Generally a test runner includes

- The runner
- The Reporter (customizable)

Popular Product in this domain

- Mocha
- Karma
- Jest

Popular products

- Nodejs builtin assertions
- Chai
- Should
- Jasmine
- Jest

Popular products

- Sinon
- should
- Jest

Popular Product

- supertest

Note:

- A Testing framework may come with more than one builtin elements
 - Example Jest
 - Is a test runner but also includes
 - OsAssertion library
 - Mocking library
 - Snaptho library
- Most test runners can work with most of the other libraries
 - Example
 - Jest can also use assertions from Chai/Should
 - Can use mocking using sinon
 - supertest

Async testing with mocha

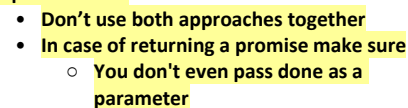
Wednesday, October 14, 2020 2:52 PM

- Initially **it()** completes before the callback is called.
 - When it completes no Assertion error has occurred
 - So test is marked **pass**

Eventually after a delay, the callback completes and assertion fails.
This crashes the runtime engine if it is still running

```
JS prime.js JS findPrimes-callback.test.js X
tests > lib > findPrimes-callback.test.js describe(findPrimes async function tests) callback
21 it('should wait for long running async function', ()=>{
22
23   findPrimes(1,500000,(err,primes)=>{
24
25     //we will reach here after a long time
26     //mocha doesn't wait this long
27     primes.should.be.an('array').of.length.equal(5000)
28     console.log('test has finished');
29   });
30
31   //this function ends with any error
32   //so mocha things test passed without waiting for
33   //assertions to happen
34
35   //assert.fail('fails as test does not wait for async f
36
37 });
38
39
40
41 it('should fail for invalid range', ()=>{
42
43   findPrimes(10,2,err=>{
44
45     err.message.should.contain('Invalid Range');
46     err.range.min.should.be.equal(10);
47   });
48
49 });
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575

```



Unit Testing in NodeJS Page 39

Using chai-as-promised plugin to handle promises

1. Get npm package

`npm install --save-dev chai-as-promised`

2. Use chai-as-promised with chai

`require('chai').use(require('chai-as-promised'));`

```
it('should reject invalid range', async()=>{  
  promisedPrimes(10,1)  
    .should  
    .eventually  
    .be.rejectedWith('Invalid Range')  
    // .and.have.property('range',{min:10,max:1});  
    .then(err=>{  
      err.range.min.should.equal(10);  
      err.range.max.should.equal(1);  
    })  
});
```

- `.eventually` will internally await
- `.reject/.rejectedWith` will handle **catch**
 - Returns a **resolved** promise containing the error value
- You will handle **then()** and not **catch**

```
it('should return 4 primes under 10', async()=>{  
  promisedPrimes(1,10)  
    .should  
    .eventually //await for promise to resolve  
    .be.an('array').with.members([2,3,5,7]);  
});
```

- `eventually` awaits for the call