

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 }
```

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

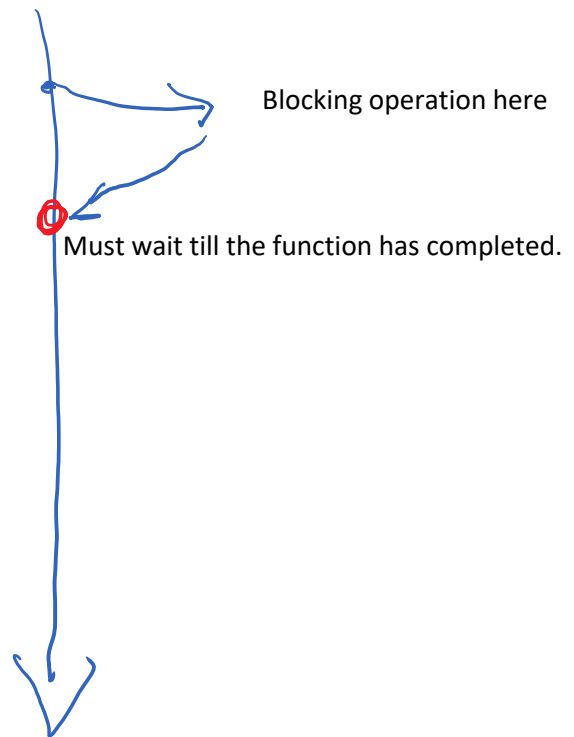
Monday, October 12, 2020 12:30 PM

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

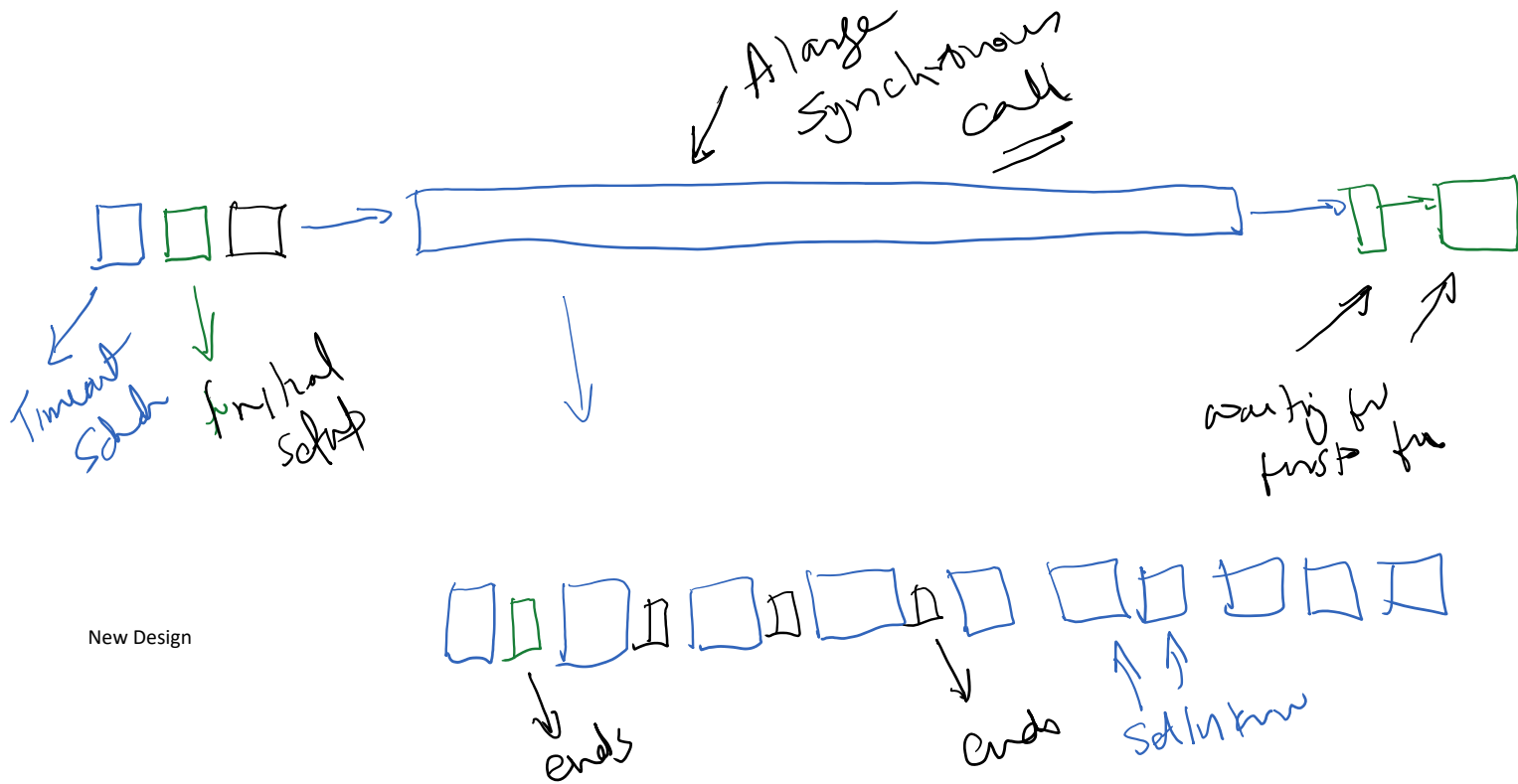
- Say we are finding all primes between 2 and 500000
- We may take a short break of say 10ms after every 1000 iteration.

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);
    }).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

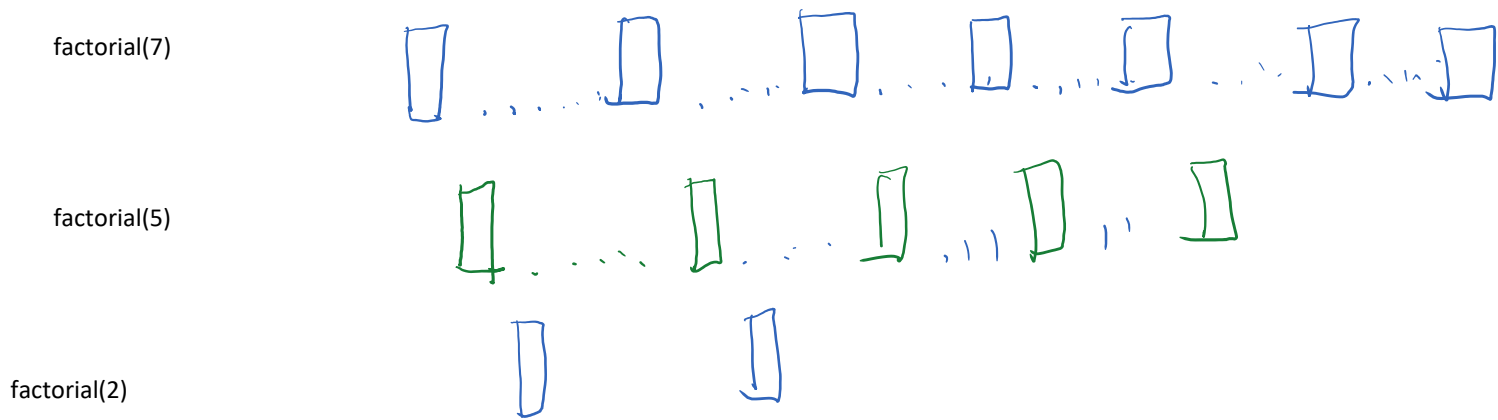
Convert the factorial function given below to a cooperative function

- 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(a,n)=1/factorial(a)/factorial(n-a)/factorial(a-n);
```

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
yield03.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 green text: "Once I break no further code is executed in generator function" with an arrow pointing from the `yield i;` line in `primeutils3.js` to the `prime` variable in the `for` loop of `primeapp09.js`.

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

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

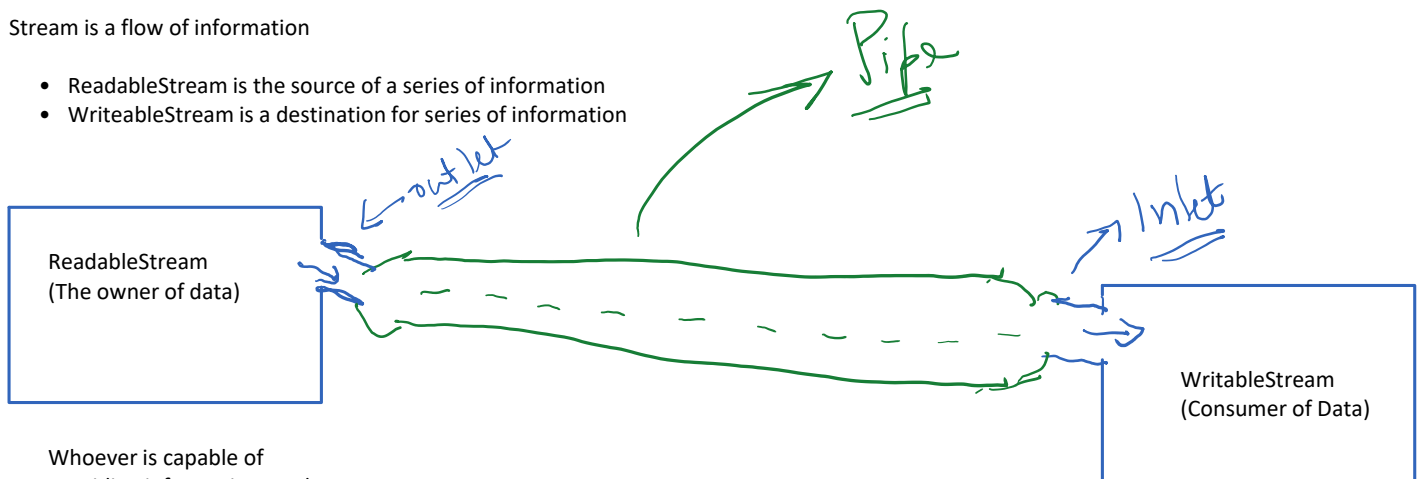
What is a Stream

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);
  });
};

```

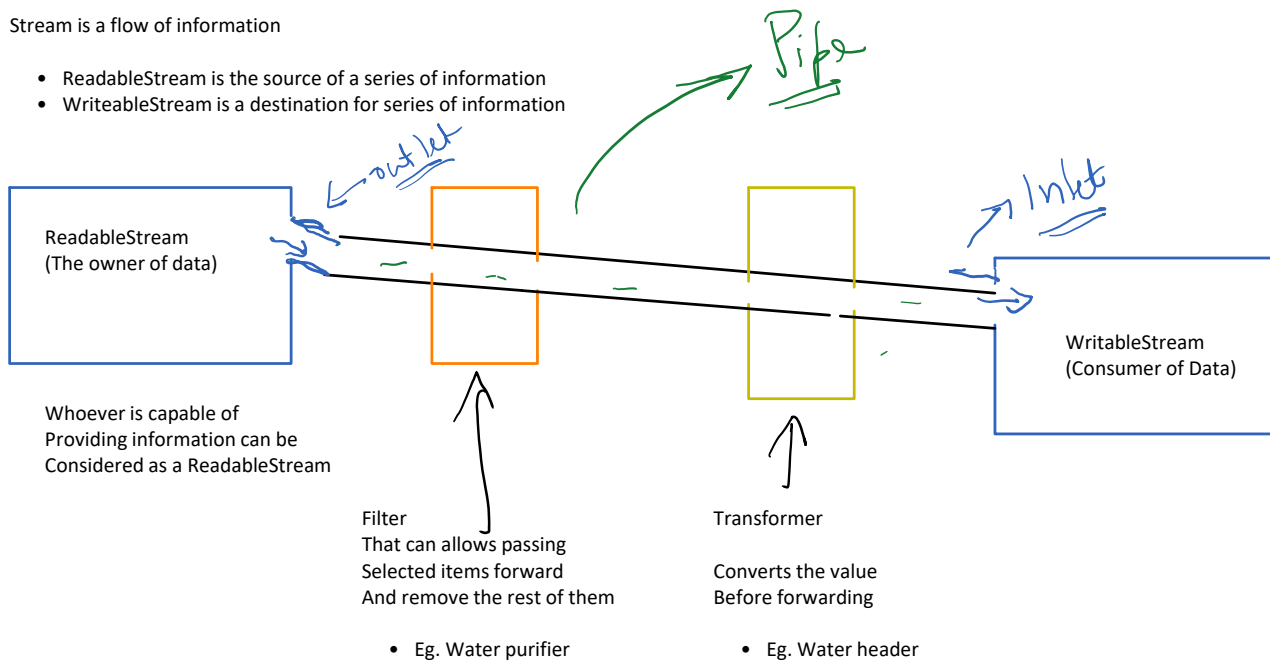
1. Define `_read` that should eventually emit
 - a. data
 - b. end
 - c. Error
2. Send data as buffer and not as plain data
3. Translate custom events to standard events

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

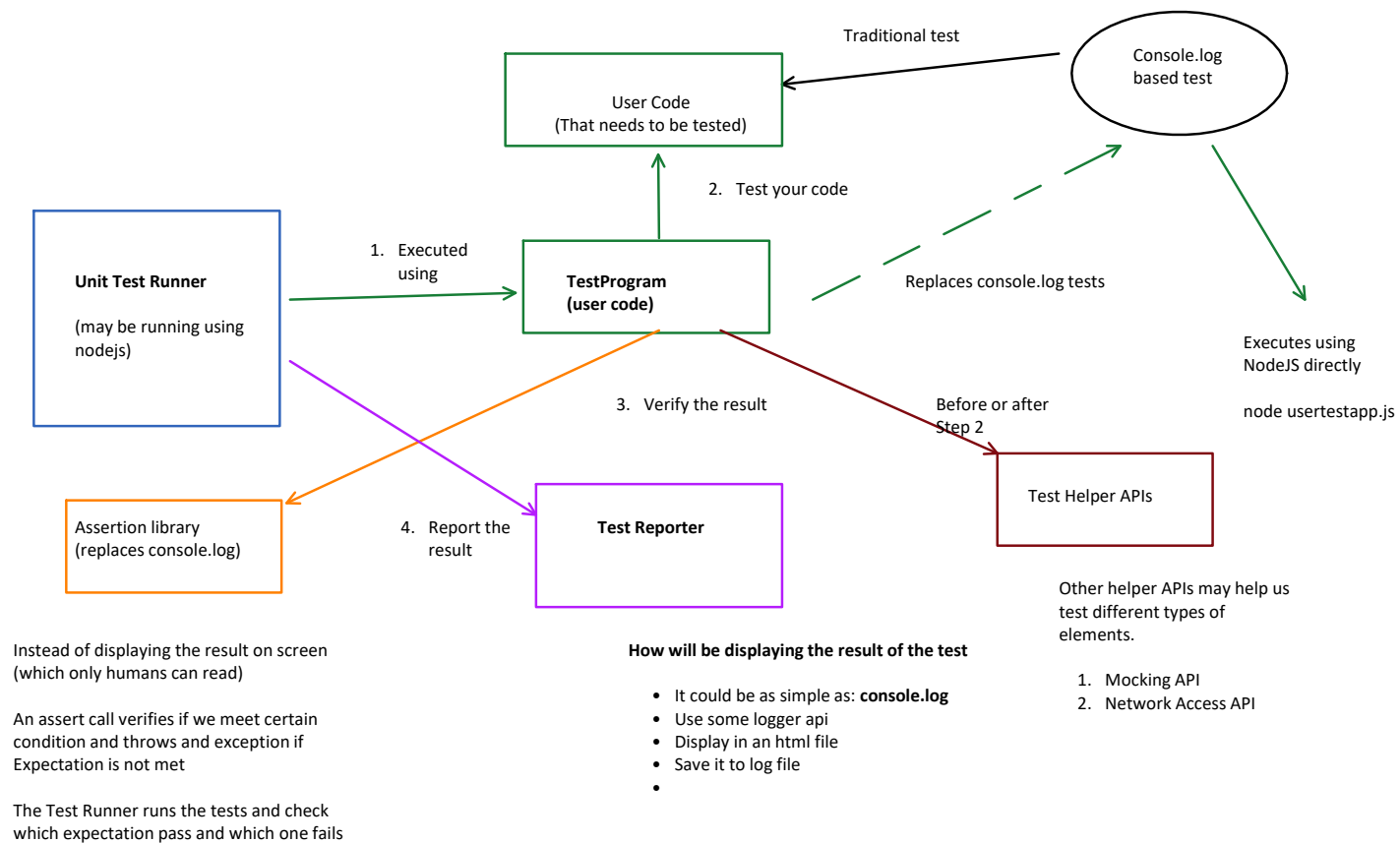
    cb(); //inform the system that convert is complete
};

```

N

Unit Testing Component

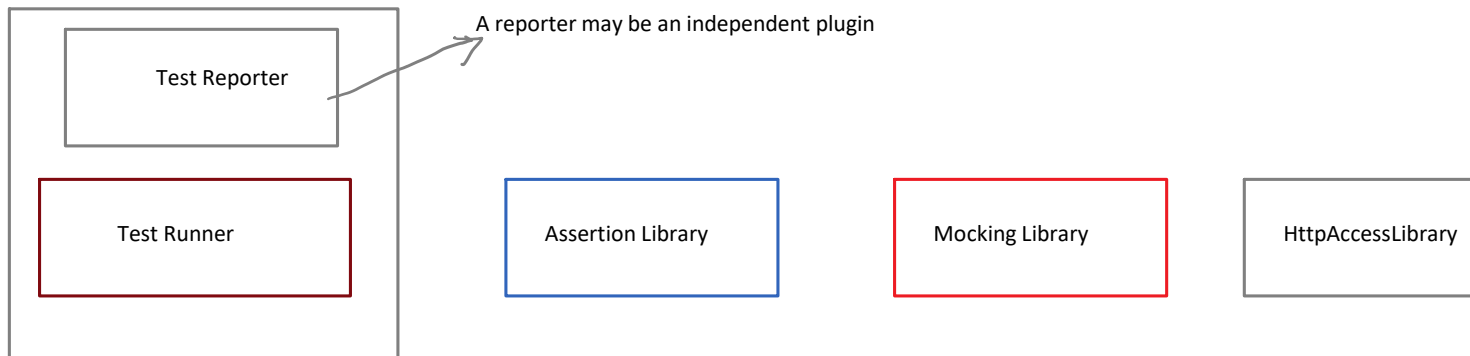
Wednesday, October 14, 2020 2:23 PM



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

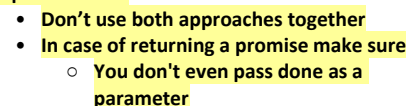
```
21 describe('findPrimes async function tests', () => {
22   it('should wait for long running async function', () => {
23     findPrimes(1, 500000, (err, primes) => {
24       //we will reach here after a long time
25       //mocha doesn't wait this long
26       primes.should.be.an('array').of.length.equal(5000)
27       console.log('test has finished');
28     });
29     //this function ends with any error
30     //so mocha things test passed without waiting for
31     //assertions to happen
32     //assert.fail('fails as test does not wait for async f
33   });
34   it('should fail for invalid range', () => {
35     findPrimes(10, 2, err => {
36       err.message.should.contain('Invalid Range');
37       err.range.min.should.be.equal(10);
38     });
39   });
40 });
```

```
findPrimes async function tests
  ✓ should have null error for valid range
  ✓ should wait for long running async function
  ✓ should fail for invalid range
3 passing (14ms)

AssertionError: expected [ Array(41538) ] to equal 5000
    at D:\OneDrive\myworks\corporate\202011-lng-advnode\unittestdemos\tests\lib\findP
    at D:\OneDrive\myworks\corporate\202011-lng-advnode\unittestdemos\tests\lib\findP
    at Timeout.<anonymous> (D:\OneDrive\myworks\corporate\202011-lng-advnode\unitest
    at processTimers (internal/timers.js:492:7) {
  showDiff: true,
  actual: [
    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, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151,
    157, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223,
    227, 229, 233, 239, 241, 251, 257, 263, 269, 271, 277, 281,
    283, 293, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359,
    367, 373, 379, 383, 389, 397, 401, 409, 419, 421, 431, 433,
    439, 443, 449, 457, 461, 463, 467, 479, 487, 491, 499, 503,
    509, 521, 523, 541,
    ... 41438 more items
  ],
  expected: 5000
}
npm ERR! Test failed. See above for more details.
```

Solution

- You can pass a special callback called **done()** to your test function.
- If **done()** is apssed it should be invoked after your async function completes
- done()** notifies testing framework that test is complete
- Test runners waits for a **fixed time frame** for done() to be called after which they time out



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);  
    })  
});
```

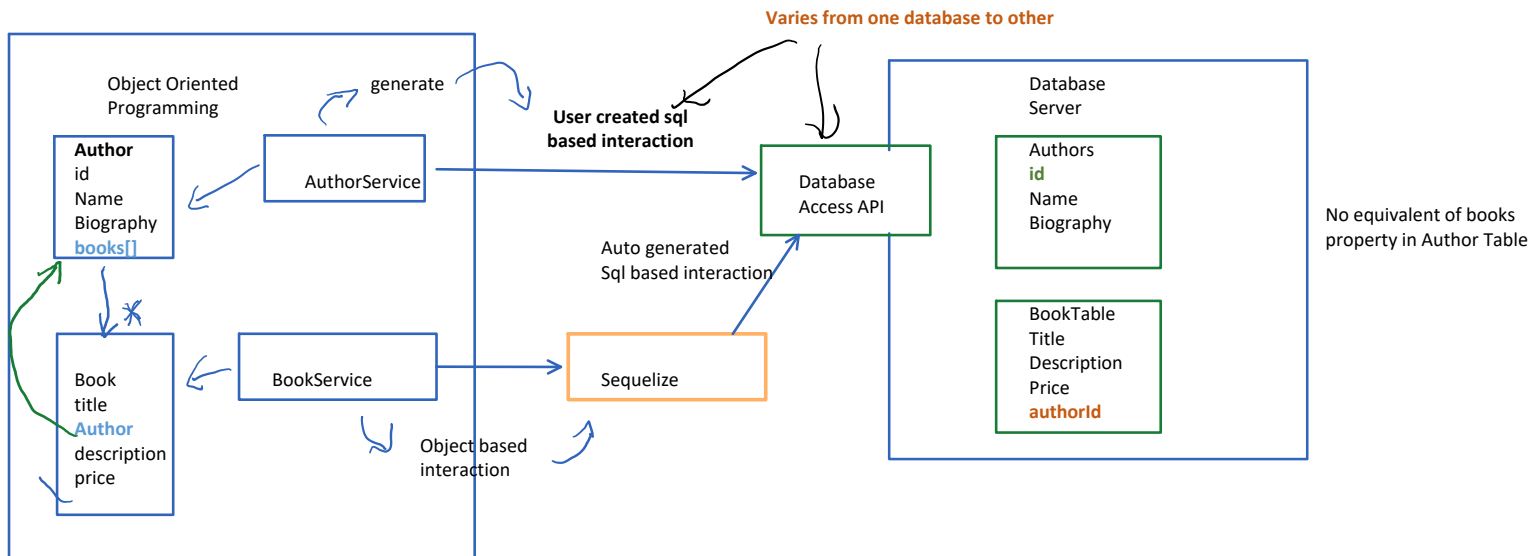
- `.eventually` will internally await
- `.reject/.rejectedWith` will handle **catch**
 - Returns a **resolved** promise containing the error value
- You will handle **then()** and not **catch**

```
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

ORM (Object Relation Mapping)

Thursday, October 15, 2020 11:55 AM



Sequelize Supports

- MySQL
- Postgress
- Sqlite

ORM's Responsibility

- Map between objects and tables
- **Auto generated queries**
- **Database agnostic design**
- Auto fetch data
- Handle other issues
 - Change management
 - Caching of Record
 - Change detection

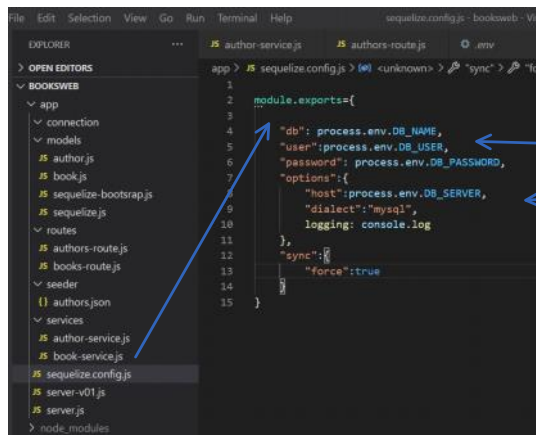
Sequelize Manual

<https://sequelize.org/master/manual/validations-and-constraints.html>

Enabling Sequelize For our Project

Thursday, October 15, 2020 3:41 PM

1. Add NPM package for Sequelize --> `npm install --save sequelize`
2. Add NPM package for underlying database
 - a. `npm install --save mysql2 <-- mysql`
 - b. `npm install --save sqlite3 <-- sqlite database`
3. Create configuration file for data connection
 - a. `./app/sequelize.js`

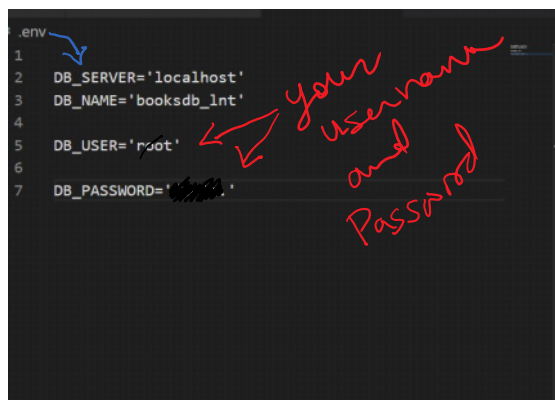


```
1 module.exports={
2   'db': process.env.DB_NAME,
3   'user': process.env.DB_USER,
4   'password': process.env.DB_PASSWORD,
5   'options':{
6     'host': process.env.DB_SERVER,
7     'dialect': 'mysql',
8     'logging': console.log
9   },
10  'sync':{
11    'force': true
12  }
13 }
```

Generally it would contain parameters for

- Credentials to connect to database
- Metadata for database connection
 - Server name
 - Dialect of database
 - Logging or not
- Sync options
 - Force etc

4. Move sensitive information to environment variable
 - a. You may use **dotenv** package (`npm install --save dotenv`) for this purpose
 - b. Create a `.env` file and add details



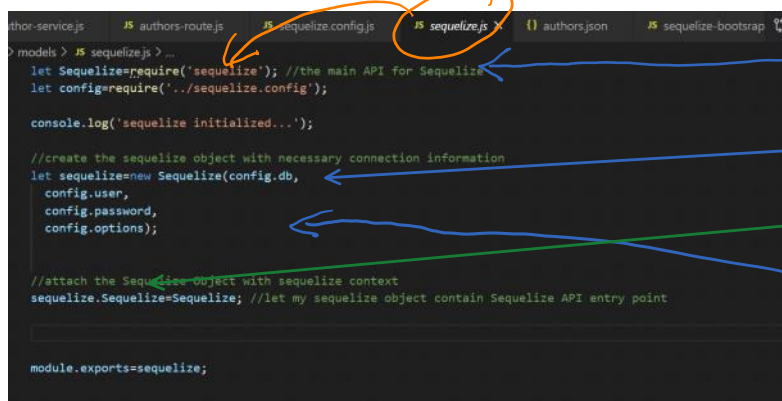
```
1 DB_SERVER='localhost'
2 DB_NAME='booksdb_int'
3 DB_USER='root'
4 DB_PASSWORD='your username and Password'
```

Important!

- **Never commit .env to version control.**
- **Add it to .gitignore**

5. Create file for holding sequelize object for you project

This will be the single point of contact between your application and database server



```
1 let Sequelize=require('sequelize'); //the main API for Sequelize
2 let config=require('../sequelize.config');
3
4 console.log('sequelize initialized...');
5
6 //create the sequelize object with necessary connection information
7 let sequelize=new Sequelize(config.db,
8   config.user,
9   config.password,
10  config.options);
11
12 //attach the Sequelize-Object with sequelize context
13 sequelize.Sequelize=Sequelize; //let my sequelize object contain Sequelize API entry point
14
15 module.exports=sequelize;
```

- Get main **sequelize** package
- Create Your own **sequelize context**.
 - This object is the single point of contact between your app and database
- Attach Sequelize api to your context
 - This is optional and convinience
 - You don't need to require two files
 - Just access the main api using **sequelize.Sequelize**
- Access details using configuration

Note:

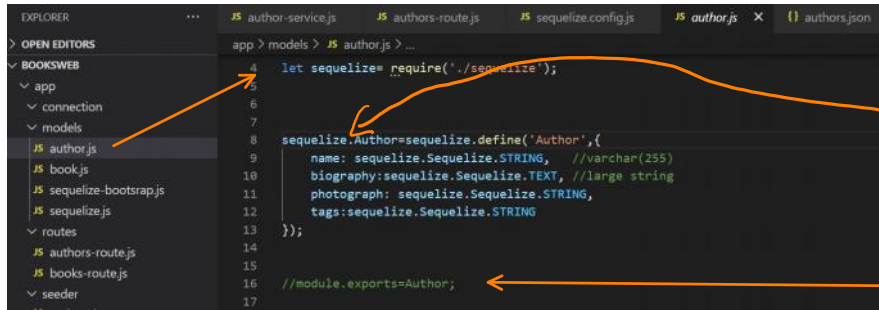
```
module.exports=sequelize;
```

Note:

This is the module you will be using every where

6. Define you model definition files

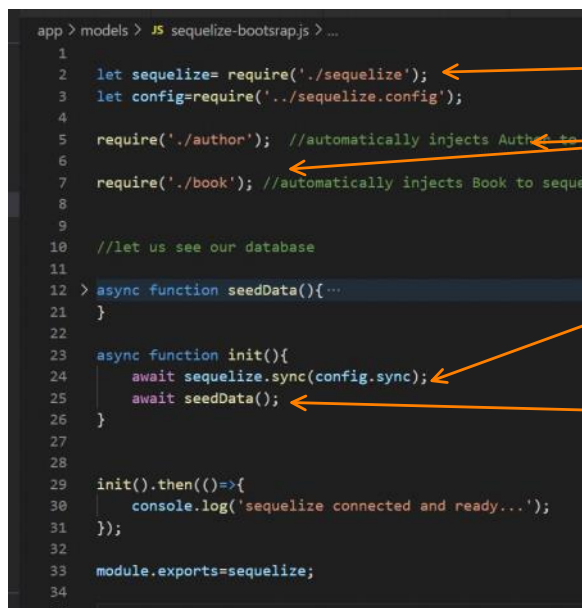
- We will create a model definition for one object per file
-



```
1 let sequelize= require('./sequelize');
2
3
4 sequelize.Author=sequelize.define('Author',{
5   name: sequelize.Sequelize.STRING, //varchar(255)
6   biography: sequelize.Sequelize.TEXT, //large string
7   photograph: sequelize.Sequelize.STRING,
8   tags: sequelize.Sequelize.STRING
9 });
10
11 //module.exports=Author;
```

1. Get Your **sequelize context** to define the model
2. Use **sequelize.define()** to define your model
3. Access Sequelize data types using Sequelize API attached in your context
4. **Optional but RECOMMENDED**
 - a. Attach your new definition to your context object
 - b. This way you **don't need to** export this definition from current module

7. Define a bootstrap module



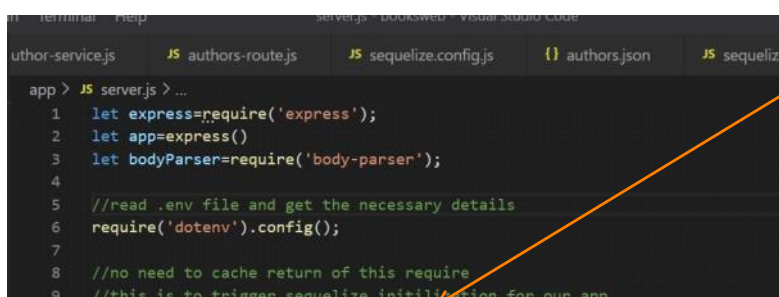
```
1 let sequelize= require('./sequelize');
2 let config=require('./sequelize.config');
3
4 require('./author'); //automatically injects Author to sequelize
5 require('./book'); //automatically injects Book to sequelize
6
7 //let us see our database
8
9
10 > async function seedData(){...
11 }
12
13
14 async function init(){
15   await sequelize.sync(config.sync);
16   await seedData();
17 }
18
19
20 init().then(()=>{
21   console.log('sequelize connected and ready...');
22 });
23
24 module.exports=sequelize;
```

1. Initialize the sequelize context for our application
2. Attach the model definitions by simply including them
 - a. The require() doesn't need any reference
3. Sync the context to the database
4. Optional Steps
 - a. Seed the database
 - b. Define Associations between the Models

Note:

Bootstrap is a one time action that should be added to your main.js

You don't need to access this value here. Require is just called once



```
1 let express=require('express');
2 let app=express()
3 let bodyParser=require('body-parser');
4
5 //read .env file and get the necessary details
6 require('dotenv').config();
7
8 //no need to cache return of this require
9 //this is to trigger sequelize initialization for our app
```

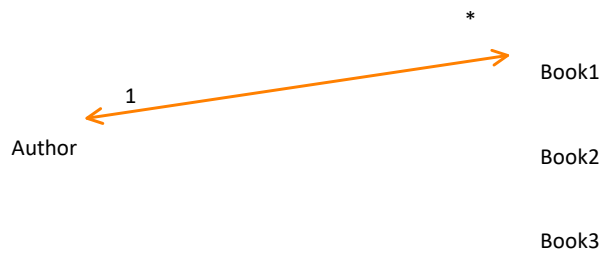
```
5 //read .env file and get the necessary details
6 require('dotenv').config();
7
8 //no need to cache return of this require
9 //this is to trigger sequelize initialization for our app
10 require('./models/sequelize-bootstrap'); //will initialize sequelize
11
```

9. Now we can access our Sequelize context and model in our services layer

```
JS author-service.js X JS authors-route.js JS sequelize.config.js {} authors.json JS sequelize-bootstrap.js JS se
app > services > JS author-service.js > ...
1 let sequelize= require('./models/sequelize');
2
3
4
5 class AuthorService{
6
7   async addAuthor(author){
8
9     let result=await sequelize.Author.create(author); //may throw an exception
10    return result;
11  }
12
13
```

Associations

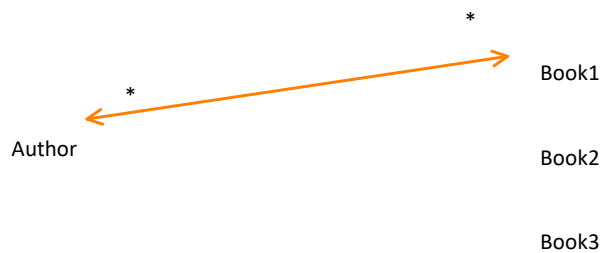
Thursday, October 15, 2020 5:12 PM



Book.belongsTo(Author)..... A Book can have only one Author

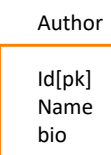
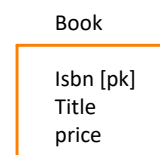
Author.hasMany(Book)..... A Author may write multiple books

Many to Many



Book.hasMany(Author)..... A Book can have only one Author

Author.hasMany(Book)..... A Author may write multiple books



AuthorBook

authorId[FK]
isbn [FK]

Isbn	authorId
1234	1
1234	2
2345	2
5555	1

Migrations

Friday, October 16, 2020 9:37 AM

- Your model changes over a period of time
 - You bring new ideas
 - Our website needs
 - Reviews for the book
 - There will be guests and member reviewing it
 - Members may have favorite book or book shelf
 - We may add additional details to existing entities
 - Author
 - May include email or website
 - List of awards
 - Book may include
 - Rating
 - Reviews
 - website

Sequelize Migration

- Semi-automatic
 - You have tools to perform partial jobs
 - You have to walk the rest of it

Challenge

- We will have purge our database (**force:true**)
- We may need to revert back the changes
 - We need to have a log of the changes done
- We must have version control on our database
- We may need to save
- In production, that may often mean loss of data
 - You can't use **force:true** on production
- Can be done
 - Manually
 - Automatically

Sequelize-cli (command line interface)

Friday, October 16, 2020 9:43 AM

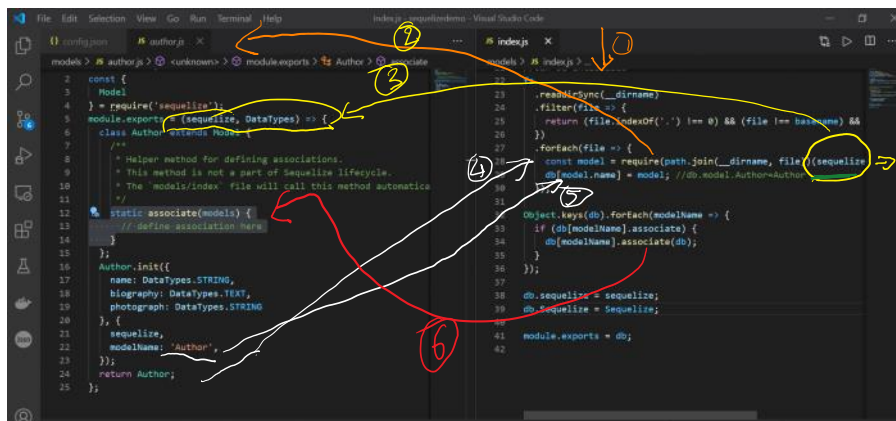
- A utility to help you develop application faster using a command line interface
- Can generate some boilerplate code for you
 - Eg.
 - Model File
 - Configuration File
 - Bootstrap file (from our example)
 - **Migration codes**
 - Seeder

Install the cli

```
c:\> npm install --global sequelize-cli
```

Sequelize model and index.js

Friday, October 16, 2020 10:10 AM



1. Index.js is required
2. It loads all the model definition modules from models folder
 - a. It imports the default function that takes two parameters
 - i. Our sequelize context object
 - ii. Sequelize DataTypes to create definition
3. It invokes the imported function with required parameters
4. The model definition function should create the model object and return it
5. We attach the model object to our sequelize context by name
6. It checks if the association function is present, then it is invoked
 - a. Association functions are invoked after all models have been loaded
 - b. Association functions should contain relationship of current object with the other object

Memory Leaks

Friday, October 16, 2020 11:34 AM

- **Javascript** works on the **Garbage Collection Mechanism** like java/.NET
 - Unlike C++ you can't explicitly deallocate memories
 - This is supposed to avoid problems of c/c++ like explicit memory deallocation
 - Memory leak
 - Dangling references
 - References that refers to memory that had been freed

How garbage collector works in Javascript

- A garbage collector frees memories that are no more used by the user
- **How does it know what memory is no more used**
 - A memory that doesn't have a reference is considered as un-needed and a candidate for garbage collection.
- Any memory that has a reference will not be garbage collected no matter we are planning to use it or not
 - **This is the primary reason for Memory Leak**

NodeJs follows mark and sweep Algorithm

- NodeJs garbage collector works in two phases
 1. It **Marks** every object that is being referred
 - i. It starts from the root object (global in nodejs and window in browser based object)
 - ii. An object which is referred by global object is a live object
 - iii. Any object referred by any live object is also a live object
 - iv. Garbage collectors makes a graph all object connected to global or a child or global to any depth.
 2. Once it recognizes live objects, it **sweeps** remaining object
 - i. Every object that is not picked up as the live object is considered dead
 - ii. Memory of those objects can be freed easily by garbage collector
 - 1) **These objects don't cause any problem**

Why garbage collector is important

- When we start running out of memory, garbage collector is invoked to mark and sweep
 - Or system is comparatively idle, garbage collector may preemptively run to avoid running when we are busy.
- Garbage collector will block the thread and our application can't perform its usual till garbage collector finishes
 - This is to ensure that references are updated before you use them
- If garbage collectors runs very often or takes more time our service would be unavailable all those period.

What is a memory Leak

- **A Memory leak is a live object which user doesn't need and**
 - forgot to remove references from it
 - **Or holding on to it assuming I may need it at sometime**
 - **Not realizing cost of holding is more than cost of re-fetching at sometime**

Types of Memory Leak

Friday, October 16, 2020 11:51 AM

1. Accidental Memory Leaks

- When you have a reference and you don't realize you have a reference
- You don't understand that such activities may lead to leak
- Needs you to understand what may lead to this type leak.**

2. Caching

- Holding on to data which you may actually need in future
- But cost of retaining this data in memory is lot higher than getting it back from source or secondary storage
- You may need to optimize your algorithm.**

3. Leaks in Third Party API and Tools we are Using

- NodeJS 8.1 garbage collector had a memory leak issue.
- It is Fixed now.

How to handle leaks

- We can't (shouldn't) control Garbage collection process
- We can however proactively mark objects that are garbage.
 - To mark a garbage we must unmark (de-reference) object we no more need.

1. Accidental Memory Leaks

A. Global/Long leaving references to a short spanned object

- An object that is supposed to live for a short time (do its work and die) has been attached to a long living object.
- Now this object will stick on as long as the object holding a reference to it is leaving

- Avoid global variables**
- Don't holder references to variables you don't need in future**
- As soon as objects use is over make sure you set the reference to null or undefined.**

```
let primes=findPrimesSync(2,10);
```

```
console.log(primes);
```

```
//I don't need primes any more  
primes=null; //mark it free.
```

Any variable referred explicitly defining using var/let/const is automatically created in the global scope

B. Global Scope is tricky

- Global scope is tricky in nodejs

```
function getAuthorById(id){  
  author= await authorService.getByld(id); //← missing var/let/const makes author a global variable  
  return author;  
}
```

Problem

- var/let/const is missing**

Solution

- Always using "use strict" meta on the top your js page
 - It causes such reference to be considered error
- Prefer ES2015 let keyword over var
 - Let is private to the scope

C. Closure

- Closure is essentially
 - A function that defines another function and returns it to the client.

```
function random( min,max ) {  
  let seed=new Date();  
  let r=new Random(seed);  
  let apiKey=getApiKey(); //never used by closure  
  function get(){  
    return Math.floor(r*next())*(max-min+1)+min;  
  }  
  return get;  
}  
  
let dice=random(1,6); //dice game  
let coin=random(0,1); //returns a coin  
  
for(let x=1;x<=10;x++){  
  console.log('dice', dice()); //random between 1-6  
  Consooe.log('toss', coin?'head':'tail'); //coin toss  
}
```

- The two get functions inside random are two different instances of function and not reference to same function

- Both random function remembers all variables available to them (min,max,seed,r). In this case there will be two copies of these variables in memory one for each instance of multiply function

Note:

A callback is not a closure

- In a closure, as long as the inner function (random) is live all variable available to it via its out closer will also remain live
- In a non-closure situation the variables min,max, seed, r which are argument and local variables to function is a short lived object and the reference is removed from the memory as soon as the random function call finishes execution.
 - The associated object will be eventually swiped out
- Since in current example inner multiply is live, it maintains a reference to all objects that were available to it when it was created. They will remain live till my closure function is live.
 - So both of my random function holds on to objects like
 - Date() ← a large object when we needed only a numeric
 - Random()
 - Even variables not referred by closure will still be live
 - Latest V8 engine is likely to optimize and remove those variables which are not being referred by closure

- The Garbage collector is improving every day

D. Events

- While the idea would apply to all types callback, it is more prevalent with events. (Why?)
- Any method added to **on** of an event will stick there for a long time (like forever)
- Any object or memory attached to that function will also be alive as long as the event emitter itself is alive
- If the EventEmitter lives past the last event it would ever fire, it would still be consider a live object.

```
let primeEvents=[];
function testPrime(min,max){
  let primes=[];
  let primeEvent =fetchPrimes(2,500000);
  primeEvent.on('prime', (value)=>{primes.push(value)});
  primeEvent.on('end', ()=>{});
  primeEvents.push(primeEvent);
}
```

- primeEvents will live for ever
- The on functions will also live for ever
- On function uses primes[], that will also live for ever
 - We have effectively created a closure here

- Promises and callbacks have a single execution and generally their life cycle ends there.
- Events because they run for a long time it is often easy to forget to clear them
- We may face a problem if a reference to a Promise is stored and it lives on even after fulfilment

Solution

- When using events
 1. Avoid caching the event objects
 2. Unregister the listener when it is no more required by calling event.removeListener
 3. Set event to null when you don't need it

```
let primeEvents=[];
function testPrime(min,max){
  let primes=[];
  let primeEvent =fetchPrimes(2,500000);
  primeEvent.on('prime', (value)=>{primes.push(value)});
  primeEvent.on('end', ()=>{
    primeEvent.removeListener('on', function_to_remove);
    primeEvent.removeListener('end', fuction_to_remove);
    primeEvent=null; //event object is cleared
    primeEvents=primeEvents.filter(e=>e!==primeEvent); //remove the event from
    events collection
  });
  primeEvents.push(primeEvent);
}
```

E. setTimeout/setInterval

- We must clear setTimeout() and setInterval() when they are done by calling clearTimeout() and clearInterval()
- If not cleared
 1. They remain live in javascript engine
 2. Any function passed to them can be a potential closure

2. Caching

- Caching is a common technique which we use to avoid re-fetching data from its source and increase the performance
- As per the definition caching keeps objects live in memory that may be (but not certainly) used in future
- They will not be garbage collected.
- A large amount of the data is a potential memory leak
- **If not used properly can be the number 1 reason of your memory leaks**

```
let _isPrimeSync=(number)=>{
  if(number<2) return false;
  if(number==2) return true;
  for(let i=2;i<number;i++)
    if(number%i===0) return false;
  return true;
}
```

```
function isPrimeSync( number ){
  let cache={};
  return ()=>{
    if(cache[number])
      return cache[number];
  }
```

Advantage!

- We are caching every calculation for future
- We will not calculate isPrime of any number for a second

```

let cache={};

return ()=>{
  if(cache[number])
    return cache[number];
  let result=_isPrimeSync(number);
  cache[number]=result;
  return result;
}

```

```

isPrime(29); //calculated and cached cache{29:true}
isPrime(24); //calculated and cached cache{29:true, 24:false}
isPrime(29); //no calculation required, return cache[29];

```

Solution

- Solution is not, not to cache, but to cache in an optimized manner
- Also try to preemptively clear the cache and set a limit to maximum number of items you would keep
 - You may chose to delete the older items to push new one
 - You may store a requested count each cache item and chose to remove items from the cache with least requested count
 - This way popular requests will always be in cache and not popular once shall be purged from time to time.

Optimum Solution

```

let _isPrimeSync=(number)=>{
  if(number<2) return false;
  if(number==2) return true;
  for(let i=2;i<number;i++)
    if(number%i===0) return false;
  return true;
}

function isPrimeSync( number ){

  let cache={}; //cache of primes only
  let maxValueChecked=0;
  let cacheSize=0;

  return ()=>{

    if(cacheSize>=maxCacheAllowed){
      purgeAllItemsWithRequestCountLessThan(10); //personal garbage collector
    }

    if(number <= maxValueChecked){
      if(cache[number]){ //cache contains the number. It must be prime
        cache[number]+=1; //one more request done for this one
        cacheSize++;
        return true;
      } else{
        return false; //we checked for this thing earlier and it was not prime
      }
    } else{
      let result=_isPrimeSync(number);
      maxValueChecked=number;
      if(result)
        cache[number]=1; //one request
      return result; //true/false
    }

  }

}

isPrime(29); //calculated and cached cache{29:true}
isPrime(24); //calculated and cached cache{29:true, 24:false}
isPrime(29); //no calculation required, return cache[29];

```

- We are caching every calculation for future
- We will not calculate isPrime of any number for a second time

Problem

- We are caching every single value.
- So if I use this function to find all primes between 5 and 5Lac it will cache 5Lac value in dictionary
- We may not refer to all those values ever again

- Store only primes
- Remember what was the highest number you ever checked
- If a new request comes
 1. The value is in cache
 - i. Increment request count
 - ii. Return true
 2. Value is not in cache but lesser than last check
 - i. Value is not prime as it has been tested earlier but not cached
 3. We never reached this limite (first ever request)
 - i. Update the maxValueTest
 - ii. If it is prime
 - 1) Cache it
 - 2) Mark request count as 1
 - iii. If it is not prime
 - 1) Ignore
 - 2) We have already include maxValueTest