Welcome To Advanced NodeJS

Monday, October 12, 2020 10:38 AM

Advanced

Node Js

Assignment01

Monday, October 12, 2020 10:41 AM

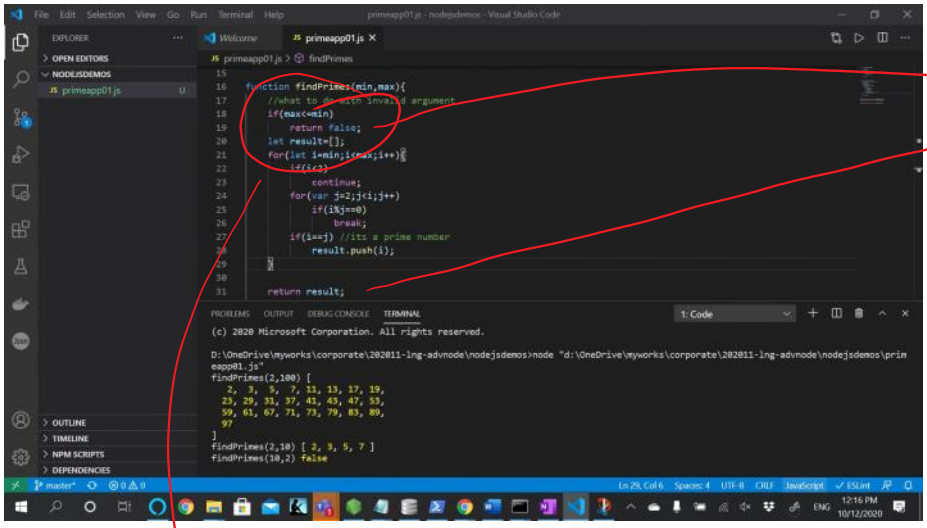
- Create a function to find and return all primes in a given min and max range
  - Example find primes between 2 and 200

- 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



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 denomniators

- Example a method may return

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

Or

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

Different data

Common denom

Nodejs is Single threaded Asynchronous Programming model

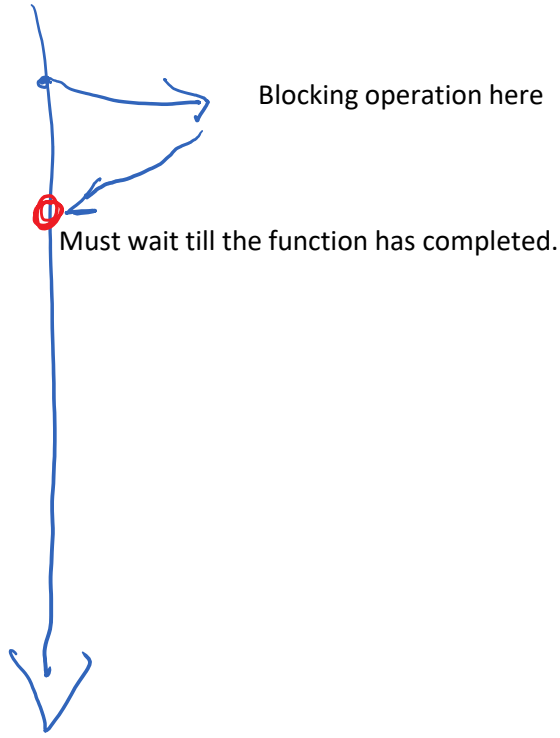
Monday, October 12, 2020 12:30 PM

NodeJS expects your functions to be async by default

- If you 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 by 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 specificy 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.

}
```

- 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

Simulates a long running process

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

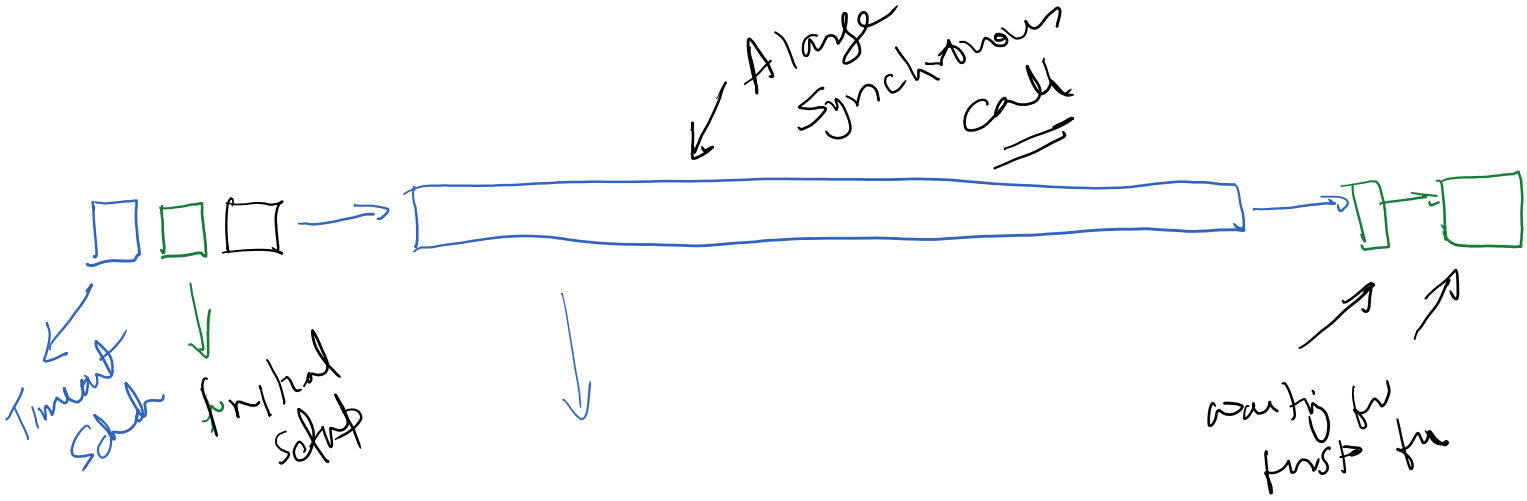
- Say we are finding all primes between 2 and 500000
- We may take a short break of say 10ms after every 1000 iteration.

Assignment03

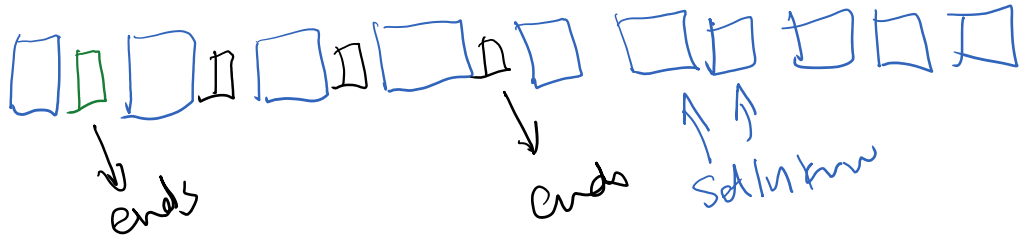
Tuesday, October 13, 2020 10:43 AM

Cooperative Async Pattern

Monday, October 12, 2020 12:52 PM



New Design



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 bulit 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
}
```

No callback passed.

```
function findPrimes(   min, max ){

    let promise=new Promise( (resolve,reject) =>{
        //business logic here
        if(success)
            resolve(  result);
        else
            reject(err_details);

    }
    return promise;
}
```

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', promes.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 asynchrnously 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 asynchrnously
  - 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) );


}
```

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

Using await

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

```
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()
    }

}
```

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 returninig

- 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

- 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()**

Understanding Promises

Tuesday, October 13, 2020 9:59 AM

```javascript
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.

Promise to calculate the combination when other promises are fulfilled

```javascript
//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);
});
```

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

```javascript
JS combination-sumit05.js > combination
14  async function combination(n,r){
15
16      // try{
17
18          let fn = factorial(n);
19          let fn_r = factorial(n - r);
20          let fr = factorial(r);
21          var comb='waiting for the result...';
22
23          let result=await    Promise.all([fn,fn_r,fr]);
24
25          //any exception is autmatically wrapped in reject()
26          comb=  comb = (result[0] / result[1] / result[2]);
27
28          return comb; // internally resolve(comb)
29
30      // } catch(err){
31      //     console.log('err',err);
32      // }
33
34
35  }
36
37  combination(-7, 2).then(console.log).catch(console.log);
38
```

```javascript
JS combination-sumit04.js > combination
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
25                  //we will reach here in apporx 1400ms for comib
26                  comb = (result[0] / result[1] / result[2]);
27                  resolve(comb);
28
29              })
30              // .catch(function(err){   //you must manually catc
31              //     reject(err);         //and re-reject it
32              // });
33      });
34  }
35  combination(-7, 2).then(console.log).catch(console.log);
36  console.log('waiting for the combination...');
37
38  //combination(7, -2).then(console.log).catch(console.log);
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL 1: Code

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**

```
combination(-7, 2).then(console.log).catch(console.log);
```

```
console.log('waiting for the combination...');

//combination(7, -2).then(console.log).catch(console.log);
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL 1: Code

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*100 ms to complete

1. Create an asynchrnous factorial function that returns in n*100 ms.
   a. It should return a promise

2. Use the factorial function to calculate comination(n,r); psudocode 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*100ms 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);
    });
};
```

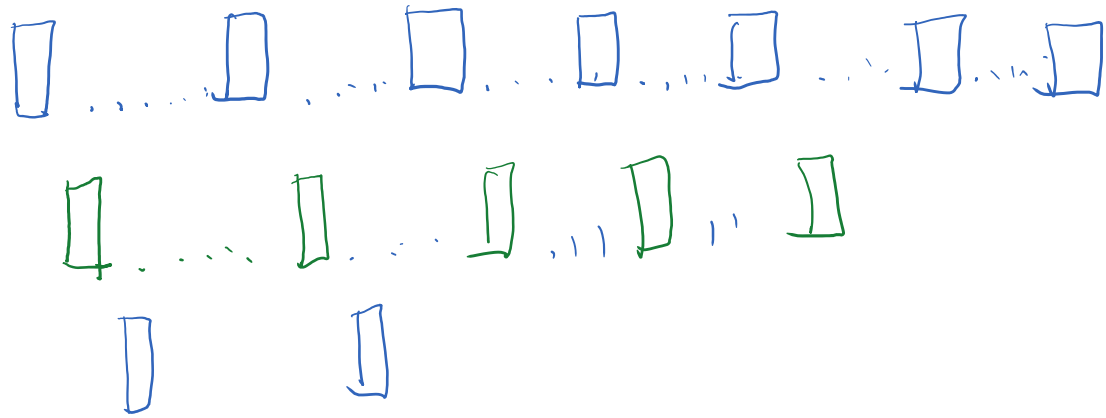
How async code works

Tuesday, October 13, 2020 11:31 AM

factorial(7)

factorial(5)

factorial(2)



Convert Normal Call to Promise

Tuesday, October 13, 2020 11:46 AM

lib > JS utils.js > sleep

```js
async function sleep(ms){

    return new Promise(resolve=>{
        setTimeout(resolve, ms); //this promise will be resolve
    });
}


module.exports = {sleep};
```

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

lib > JS math.js > factorial

```js
async function factorial(number){

        await utils.sleep(100);
        if(number<0)
            throw `negative numbers don't have factorial ${numb
        let factorial=1;

        while(number>1){
            await utils.sleep(100);  //called at an interval of
            factorial*=number--;
        }

        return factorial; //resolve

}
```

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 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.**

Hadling Large Data Options

Tuesday, October 13, 2020 12:29 PM

We can apply different technqiues

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

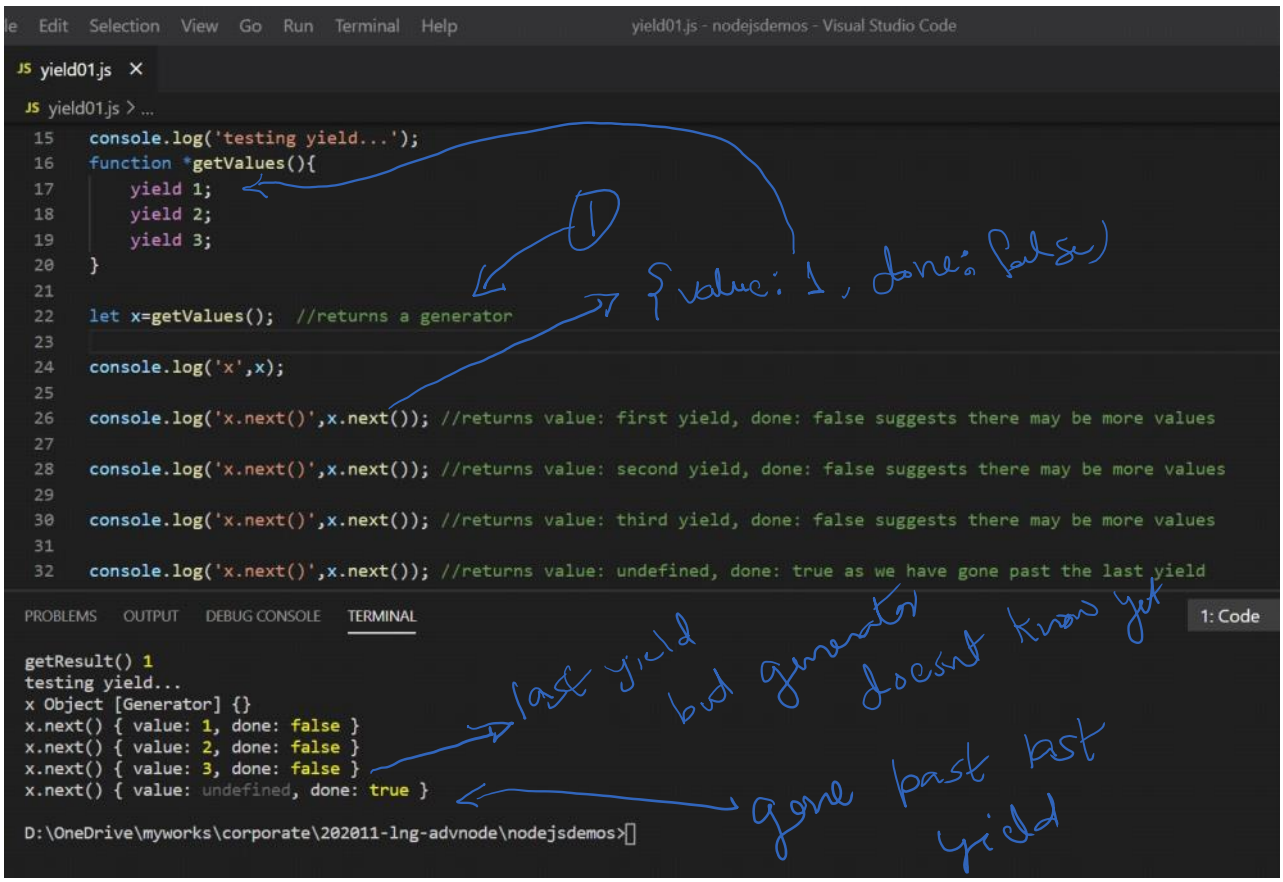
- Javascript has the concept of a generator like C# and Python.
- A generator is based on a new keyword **yield**
- **yield** looks like **return** but works differently

Return statement

```
function getResults(){

        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

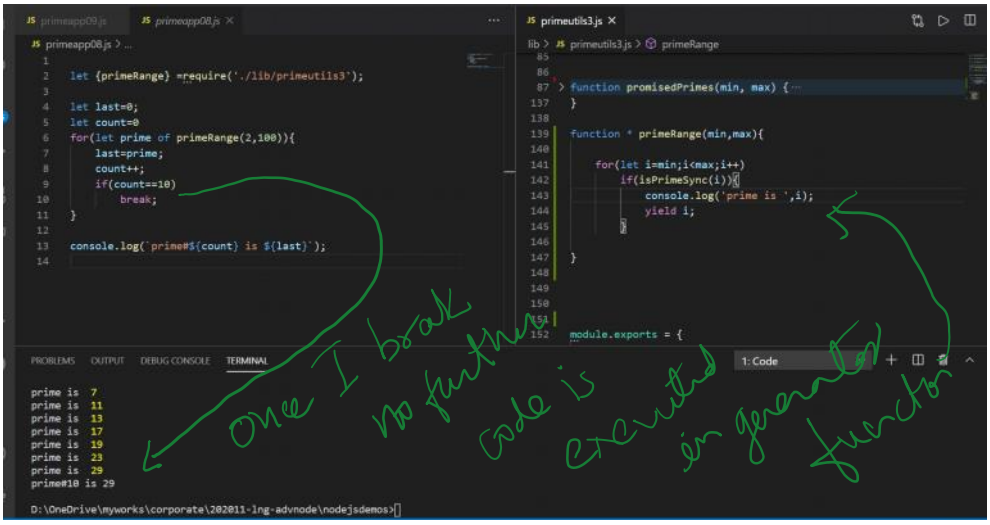
```
//A function that has yield, must have "*" prefix

function * getResults(){

        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
```



```
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.

```
let {primeRange} =require('./lib/primeutils3');

//what if I need array of primes
//... spreads any iterator/generator as individual which are collected in a list
let primeList= [ ... primeRange(2,100)];

console.log(primeList);
```



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 Singals**
   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;
}
```

```
let {fetchPrimes} =require('./lib/primeutils3');


//before returning the event, it has been emitted
let e=fetchPrimes(2,100)

//we are trying wait for an event which has already been trigge
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;
}
```

```
let {fetchPrimes} =require('./lib/primeutils3');


//before returning the event, it has been emitted
let e=fetchPrimes(2,100)

//we are trying wait for an event which has already been trigge
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
  - Writeable 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
  - createReadableStream 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

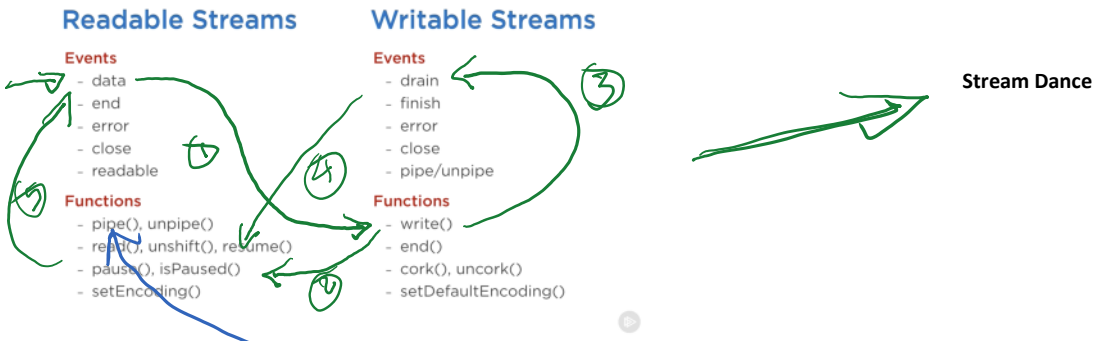
Functions
- write()
- end()
- cork(), uncork()
- setDefaultEncoding()

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



Pipe()

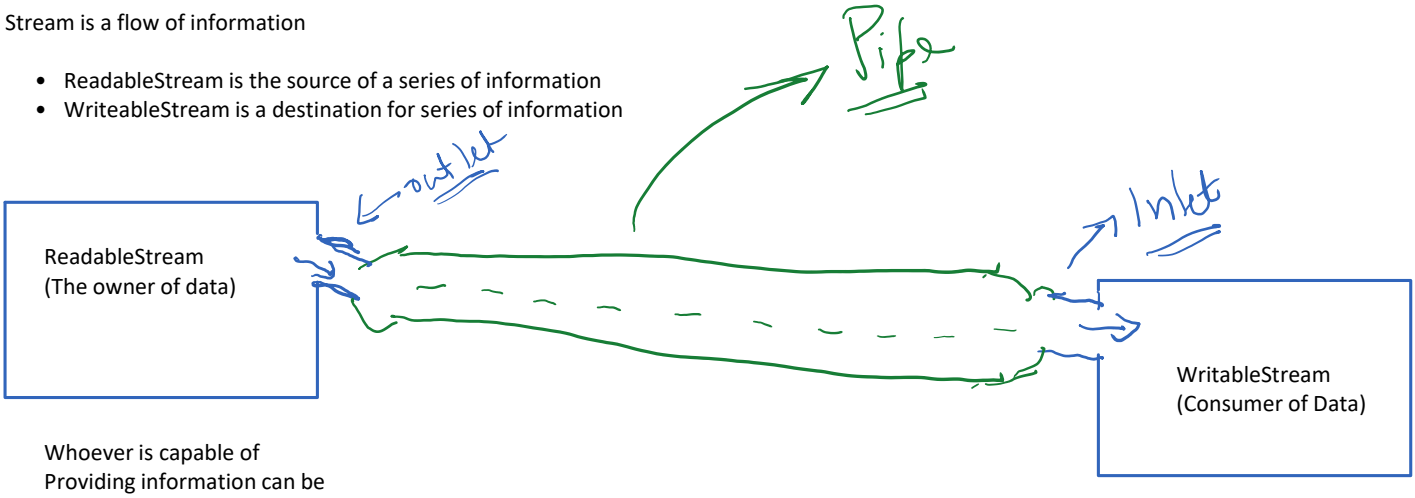
- The **Stream dance** can be automated by using the function **pipe()** on **ReadableStream** which takes a **WritableStream as a parameter**

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
- WriteableStream is a destination for series of information



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

Example
1. File
2. Network
3. Standard output device

Examples:

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

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

    //user initilazation here


}

//copy the prototype of Redable 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

```javascript
PrintStream.properties._read =function(){

    let self=this; //generall 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);
        });
```

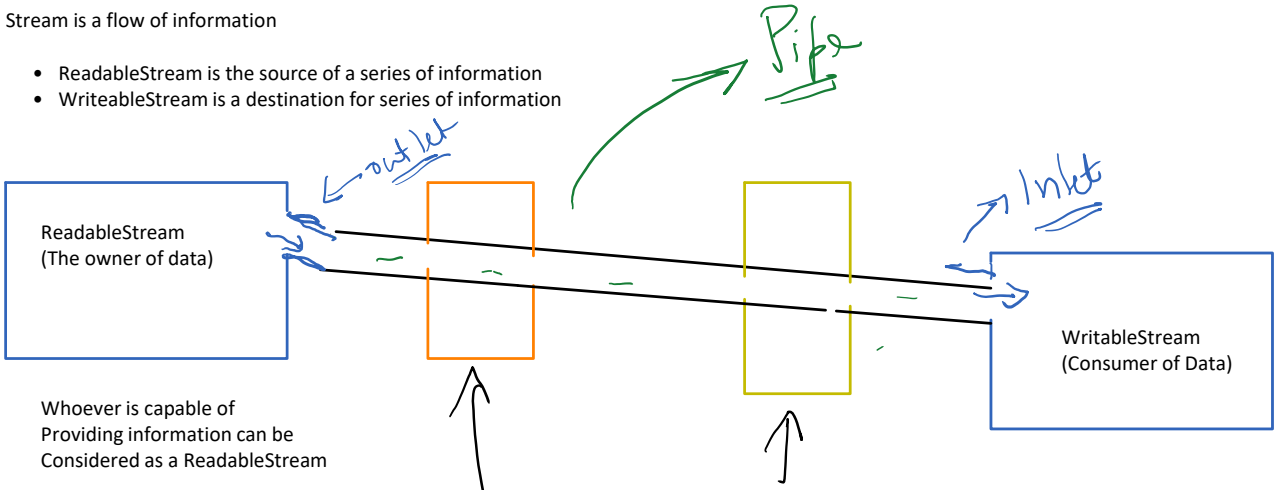
1. Define _read that should eventually emit
    a. data
    b. end
    c. Error
2. Send data as buffer and not as plain data
3. Translate custom events to standard events

Transform Stream

Wednesday, October 14, 2020 12:21 PM

Stream is a flow of information

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



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

Filter
That can allows passing
Selected items forward
And remove the rest of them

- Eg. Water purifier

Transformer

Converts the value
Before forwarding

- Eg. Water header

TransformStream

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

e.g.

```
getEmployeess()
        .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

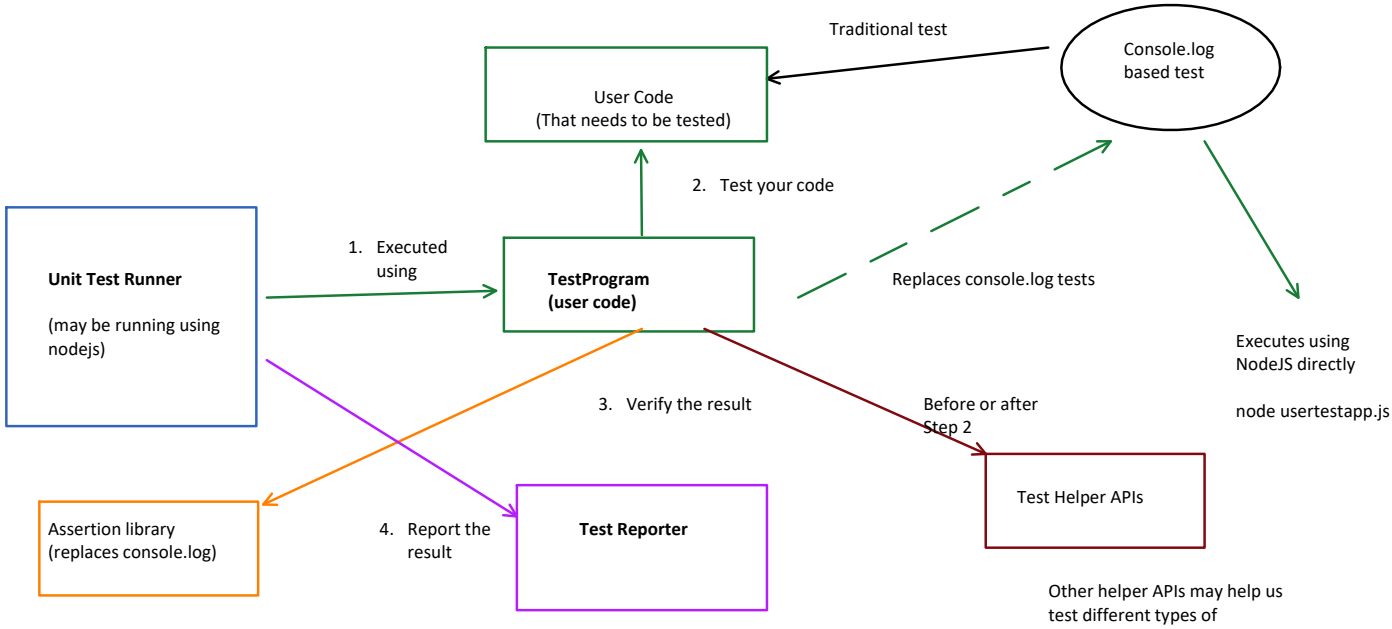
Convert to Transform

```
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 covertedValue= this.convertFunction(original); //covert input data to desired type

    let outputBuffer= Buffer.from(covertedValue.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



Other helper APIs may help us test different types of elements.

1. Mocking API
2. Network Access API

Instead of displaying the result on screen (which only humans can read)

An assert call verifies if we meet certain condition and throws and exception if Expectation is not met

The Test Runner runs the tests and check which expectation pass and which one fails

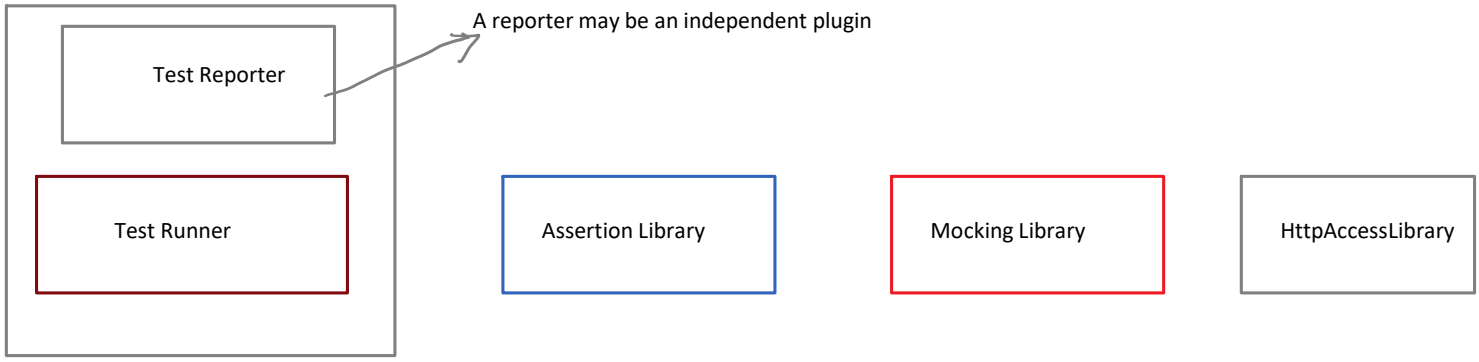
How will be displaying the result of the test

- It could be as simple as: **console.log**
- Use some logger api
- Display in an html file
- Save it to log file
-

Unit Testing framework

Wednesday, October 14, 2020 2:44 PM

- Provides one or more re-usables compoent required for unit Testing
- All these components may not be part of the same product
- We may use different protects at different level



Generally a test runner includes

- The runner
- The Reporter (customizable)

Popular Prouduct 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**
    - **Snapstho 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 sinion**
    - **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

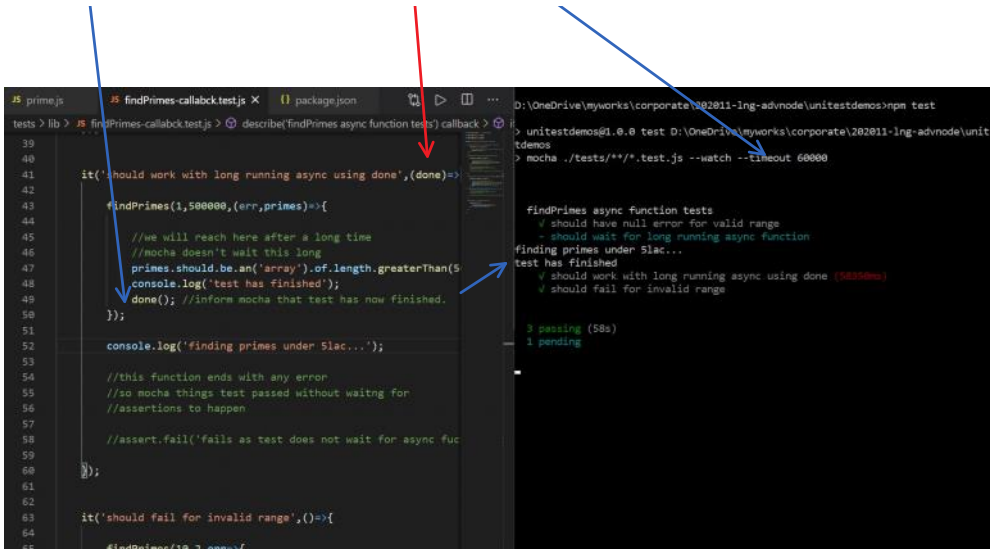
```
prime.js   findPrimes-callabck.test.js
tests > lib > findPrimes-callabck.test.js > describe('findPrimes async function tests') callback

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

```
PROBLEMS   TERMINAL   ...   2: Code
D:\OneDrive\myworks\corporate\202011-lng-advnode\unitestdemos>
```

```
  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)


D:\OneDrive\myworks\corporate\202011-lng-advnode\unitestdemos\node_modules\mocha\lib
\runner.js:906
    throw err;
    ^
AssertionError: expected [ Array(41538) ] to equal 5000
    at D:\OneDrive\myworks\corporate\202011-lng-advnode\unitestdemos\tests\lib\findP
rimes-callabck.test.js:28:52
    at Timeout.<anonymous> (D:\OneDrive\myworks\corporate\202011-lng-advnode\unitest
demos\api\lib\prime.js:72:24)
    at listOnTimeout (internal/timers.js:549:17)
    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



Testing Promises

```
describe('promisedPrimesTest',function(){

    describe('testing as promise',()=>{

        it('should return 4 primes under 10',(done)=>{
            promisedPrimes(1,10)
                .then(primes=>{
                    primes.should.be.an('array').of.length(4).and.have.members([2,3,5,7]);
                    done();
                });
        });

        it('should return 25 primes under 100',()=>{
            return promisedPrimes(1,100)
                .then(primes=>{
                    primes.should.be.an('array').of.length(25);

                });
        });
```

There are two approaches

1. Use done like any other callback function
2. Return a Promise which will automatically be awaited by test runner.
    a. No need of done

Important Note

- **Don't use both approaches together**
- **In case of returning a promise make sure**
    - **You don't even pass done as a parameter**

Testing Promise using async await

1. Write the test function as **async**
2. **await** for the call
3. Don't use done
    a. Don't pass the done parameter

```
describe('testing as async',()=>{

    it('should return 4 primes under 10', async ()=>{

        let primes=await promisedPrimes(1,10);

        primes.should.be.an('array').of.length(4);

        //done();

    });
```

Using chai-as-promised plugin to handle promises

1. Get npm package

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

2. **Use chai-as-promsied 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);

        })

});
```

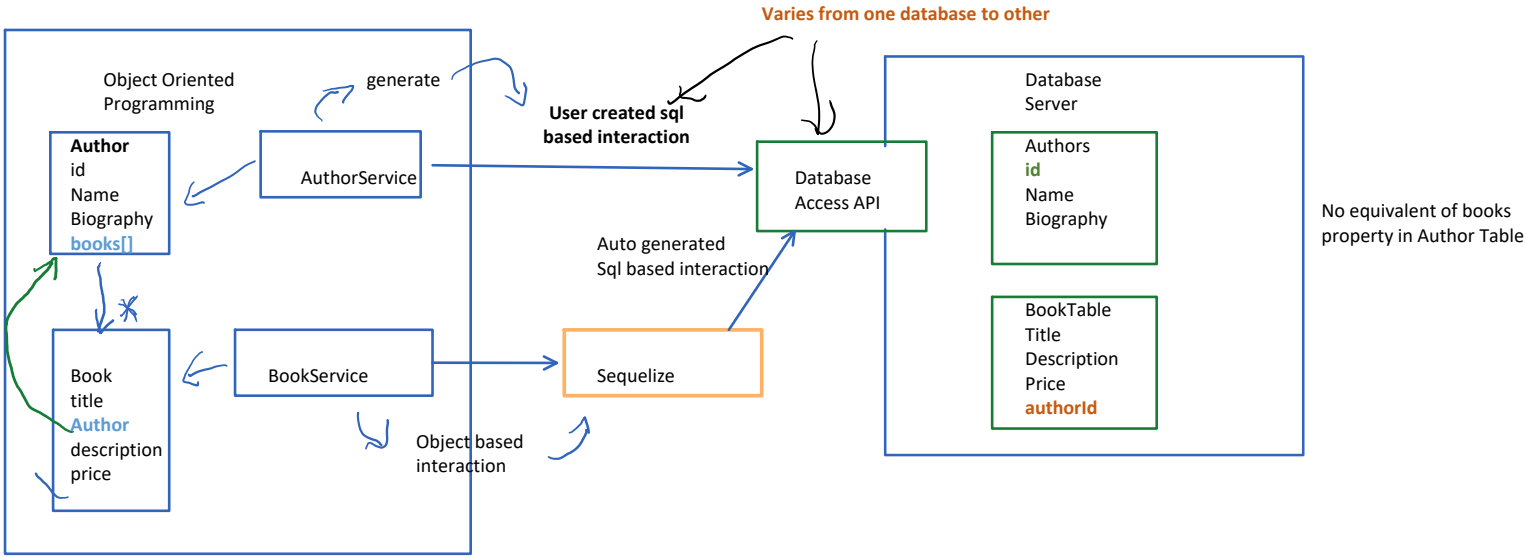
- **.eventaully** will internally await
- **.reject/.rejectedWith** will handle **cach**
  - Returns a **resolved** promise containing the error value
- You will handle **then()** and not catch

```
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

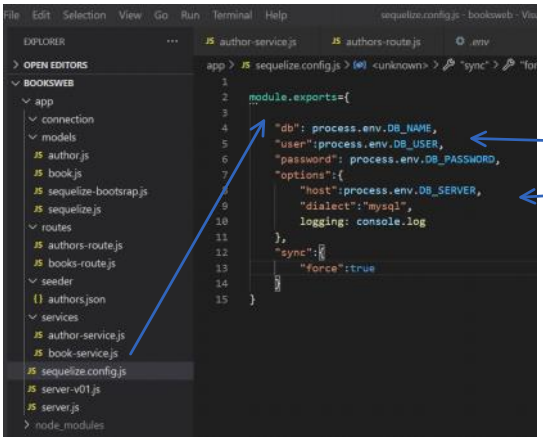
Sequalize Manual

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

Enabling Sequelize For our Project

Thursday, October 15, 2020 3:41 PM

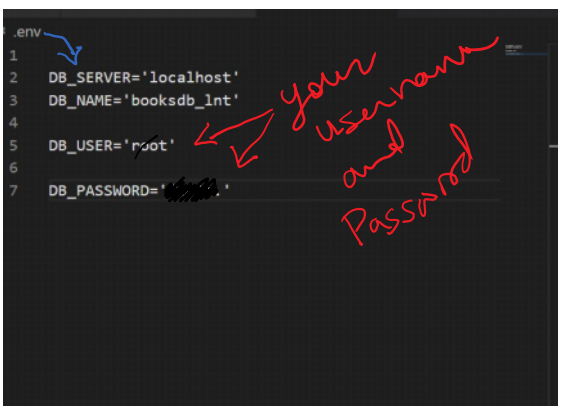
1. Add NPM package for Sequelize --> **npm install --save sequelize**
2. Add NPM package for underlying database
   a. **npm install --save mysql2 <— mysql**
   b. **npm install --save sqlite3 <— sqlite database**
3. Create configuration file for data connection
   a. ./app/sequelize.js



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 sensetive information to environment variable
   a. You may use **dotenv packag (npm install --save dotenv)** for this purpose
   b. Create a .env file and add details

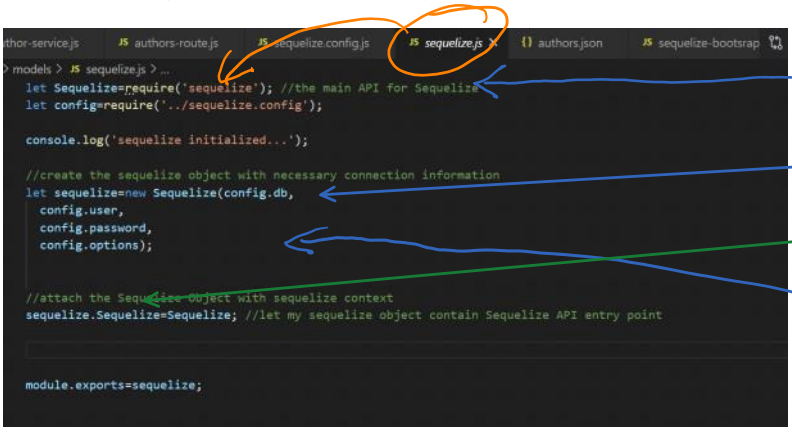


Important!

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

5. Create file for holding sequalize object for you project

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



- Get main **sequelize** package
- Create Your own **sequalize context.**
  - This object is the single point of contact between your app and database
- Attach Sequealize 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
-

```
let sequelize= require('./sequelize');


sequelize.Author=sequelize.define('Author',{
    name: sequelize.Sequelize.STRING,    //varchar(255)
    biography:sequelize.Sequelize.TEXT, //large string
    photograph: sequelize.Sequelize.STRING,
    tags:sequelize.Sequelize.STRING
});


//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

```
let sequelize= require('./sequelize');
let config=require('../sequelize.config');

require('./author');  //automatically injects Author to

require('./book'); //automatically injects Book to seque


//let us see our database

async function seedData(){ ...
}

async function init(){
    await sequelize.sync(config.sync);
    await seedData();
}


init().then(()=>{
    console.log('sequelize connected and ready...');
});

module.exports=sequelize;
```

1. Initialize the sequelize context for our appliction
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

```
let express=require('express');
let app=express()
let bodyParser=require('body-parser');

//read .env file and get the necessary details
require('dotenv').config();

//no need to cache return of this require
//this is to trigger sequelize initilization for our app
```

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

```
require('dotenv').config();

//no need to cache return of this require
//this is to trigger sequelize initilization for our app
require('./models/sequelize-bootsrap'); //will initialize sequelize
```

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

author-service.js | authors-route.js | sequelize.config.js | authors.json | sequelize-bootsrap.js

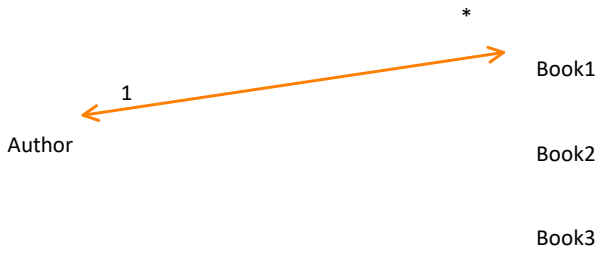
app > services > author-service.js > ...

```
let sequelize= require('../models/sequelize');


class AuthorService{

    async addAuthor(author){

        let result=await sequelize.Author.create(author); //may throw an exception
        return result;

    }

```

Model

Associations

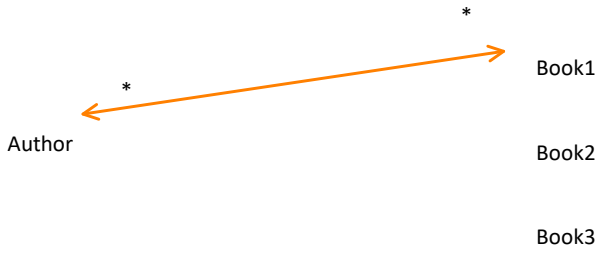
Thursday, October 15, 2020 5:12 PM



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

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

Many to Many



Book

Isbn [pk]
Title
price

Author

Id[pk]
Name
bio

AuthorBook

authorId[FK]
Isbn [FK]

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

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

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

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 Migration

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

Sequelize-cli (command line interface)

Friday, October 16, 2020 9:43 AM

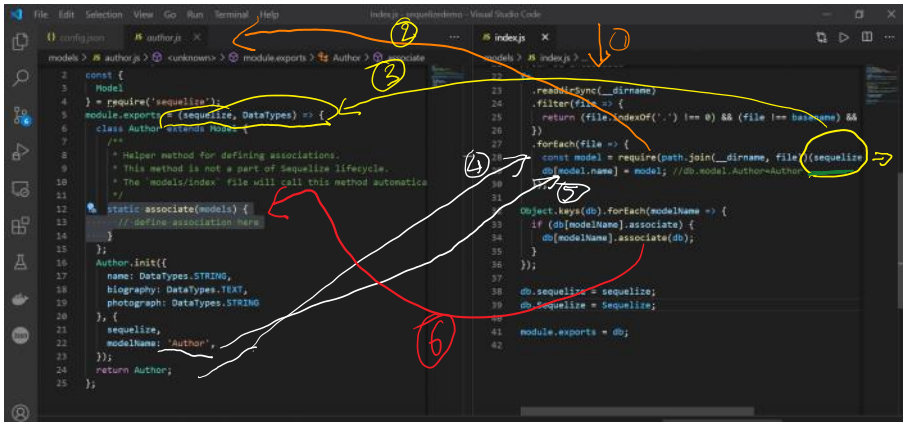
- A utility to help you develop application faster using a command line interface
- Can generate some boilerplate code for you
  - Eg.
    - Model File
    - Configuration File
    - Bootstrap file (from our example)
    - **Migration codes**
    - Seeder

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 funciton is present, then it is invoked
   a. Association functions are invoked after all models have been loaded
   b. Assoication 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 dealloc 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
   a. When you have a reference and you don't realize you have a reference
   b. You don't understand that such activities may lead to leak
   c. **Needs you to understand what may lead to this type leak.**
2. Caching
   a. Holding on to data which you may actually need in future
   b. But cost of retaining this data in memory is lot higher than getting it back from source or secondary storage
   c. **You may need to optimize your algorithm.**
3. Leaks in Third Party API and Tools we are Using
   a. NodeJS 8.1 garbage collector had a memory leak issue.
   b. 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

a. An object that is supposed to live for a short time (do its work and die) has been attached to a long living object.
b. 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.
```

B. Global Scope is tricky

a. Global scope is tricky in nodejs

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

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

Problem

- **var/let/const is missing**

Solution

1. Always using "use strict" meta on the top your js page
   a. It causes such reference to be considered error
2. Prefer ES2015 let keyword over var
   a. 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

- 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 callfinishes 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

- 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

- **The Garbage collector is improving every day**

D. Events

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

- 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

```
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

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.on('end',()=>{
            primeEvent.removeListener('on', function_to_remove);
            primeEvent.removeListner('end', fucntion 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

a. We must clear setTimeOut() and setInterval() when they are done by calling clearTimeOut() and clearInterval()
b. 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;

    }
}
```

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

```
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];
```

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

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];
```

- Store only primes
- Remember what was the hightest 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

Common Use cases

Friday, October 16, 2020 12:55 PM

A Garbage is not be a memory leak if Garbage Collector knows, it's a memory leak

```
app.get('/demo01',async(req,res)=>{
    let garbage=new Array(10000000); //Its a garbage but not a memory leak.
                                     //Its a local variable and reference is lost after function call.
    res.send('created let garbage=new Array(10000000)');
});

app.get('/demo02',async(req,res)=>{
    //It's again not a memory although added to req object which is long lived
    //After every request, once it over,
    //express server marks the request object as dead by removing all references
    req.garbage=new array(10000000);
    res.send('added gargbage to request object');
});
```

Thanks to the careful design of express API, it removes references from request response objects once the work is over!

After call is over request object is a garbage. So we are adding garbage to garbage

```
let primeService =require('./service/prime-service');
app.get('/primes/:min/:max',async (req,res)=>{

    let {min,max}=req.params;

    req.primes=[];
    //console.log(`finding primes ${min}-${max}`);
    primeService
    .submitPrimeRequest(min,max)
        .on('PRIME',data=>{
            //console.log('data',data);
            req.primes.push(data.prime);
        })
        .on('FINISHED',()=>{
            //console.log('finished');
            res.json(req.primes);
        });
});
```

- Our event handler is using
  - Req object
  - Res object
- Although express would release the req,res object
  - They would stay in memory as long as event is live
- Event object is connected to prime service object
- Prime service object is a global object
- So all these object is always retained.

*

Detecting the Leak

Friday, October 16, 2020 4:18 PM

Heap Dump

- We can use chrome://inspect tool to inspect the memory of node process
- To inspect you should start node process using

```
node  --inspect app.js
```

If your process is already running and you can't stop it (such as production)

- You can attach a debugger later using

```
kill -s SIGUSR1 nodejs-pid
```

- **Kill doesn't kill the node process. It simply attaches a special signal SIGUSR1**
- **This signal is a nodejs singal to enter in debug mode**

We can do programmetic Heap Dump based on an end point to at a fixed interval using **heap-dump module**

```
let heapdump=require('heapdump');

app.get('/heapdump',async(req,res)=>{

    heapdump.writeSnapshot((err,filename)=>{
        if(!err){
            console.log('heap dump written to ',filename);
            res.send('head dump written to '+filename);
        }
        else
            res.send('error saving heapdump');
    });

});
```

Note:

- You may run these codes at a certain setInterval
- You may check if heap usage is going up you may
  - Heapdump
  - Send alert email to administrator
  - etc

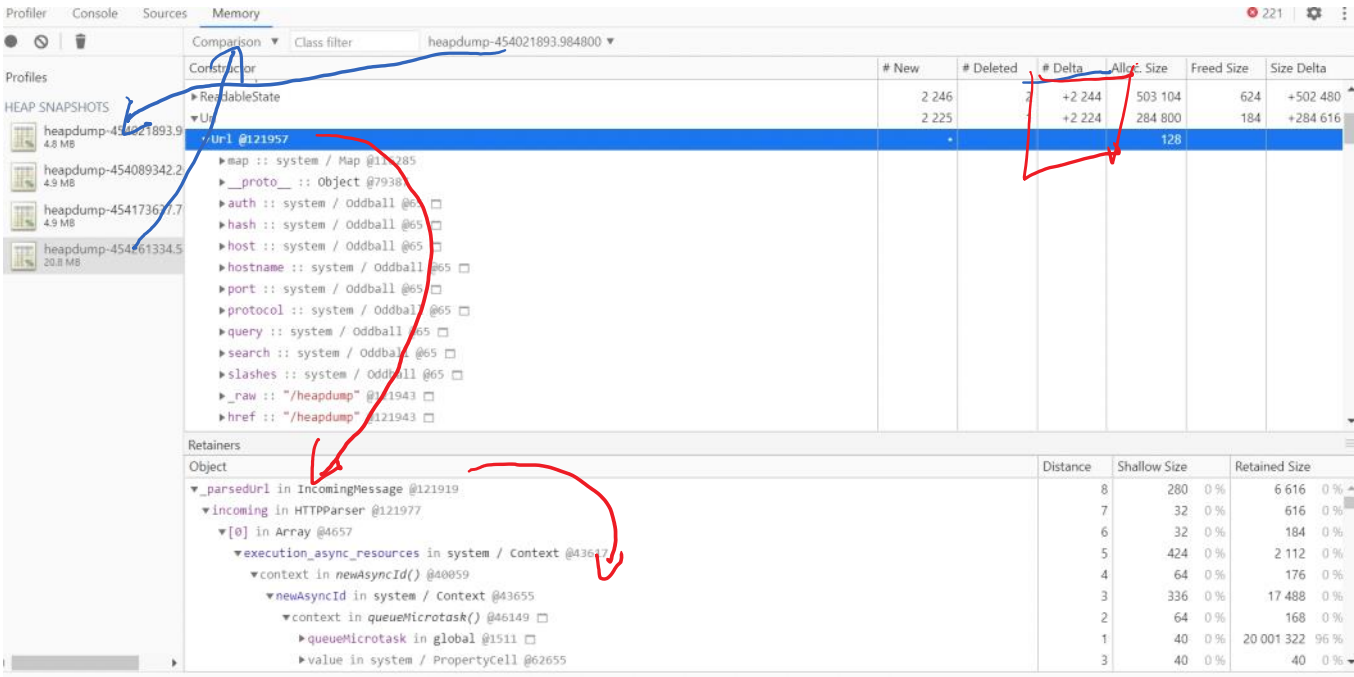
Use process.memoryUsage() can help you see the memory usage

```
app.get('/memory',async(req,res)=>{

    let memory=process.memoryUsage();
    await res.json(memory);

});
```

Fixing

Friday, October 16, 2020 5:02 PM

- We can analyse Heapdump to Identify what has increased memory consumption
- Once we locate the url or the objects which are increasing we may fix our logic



- Locate the Problem URL/Object/EventEmitters and it can give you some idea of what may be wrong in your code
- Fix your code

Patching/Improvising

Friday, October 16, 2020 5:06 PM

- We can watch the **process.memoryUsage()** at a timed interval
- Once we realize that memory usage is going beyond the limit we can
  - HeapDump
  - Create A Log
  - **Restart the process.**

PM2

- Run as a cluster

```
pm2 start app.js -i max
```

- Restart at certain memory level

```
pm2 start big-array.js --max-memory-restart 20M
```

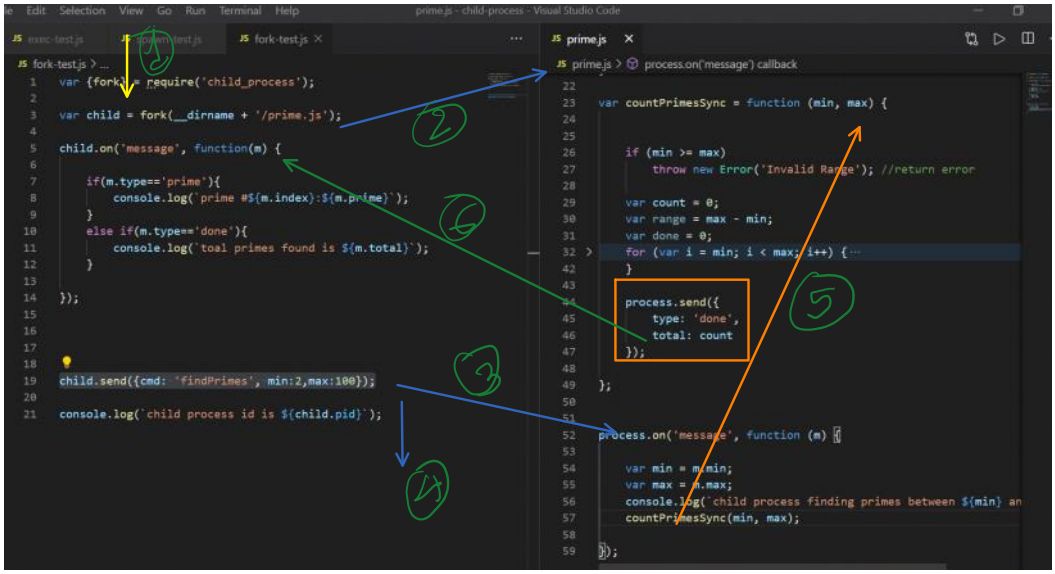
From <https://pm2.keymetrics.io/docs/usage/process-management/#max-memory-restart>

Important Note

- This can be an effective step to avoid latency due to memory leak
- It is specially useful when you are running your application in the cluster mode
- If you are using tools like **PM2** you can set **PM2** to restart the application once it reaches certain memory use level
- **PM2** can also run your application in cluster mode.

Fork

Friday, October 16, 2020 5:28 PM



1. Create a child process
   a. It runs separately without blocking the parent process
   b. You can send and receive message from the child process
2. Child process starts
3. You send you signal to the child process to calcuate
4. Parent process is free to do whatever they want
5. Child process calculates the result
6. Child process sends the result using send signal
7. Both child and Parent process can wait for message using event mechanism

cluster

Friday, October 16, 2020 5:44 PM

C:\windows\system32\cmd.exe - node cluster.js

```
servied  by 5860: total requests 21
servied  by 33164: total requests 22
servied  by 10724: total requests 35
servied  by 10364: total requests 25
servied  by 42192: total requests 25
servied  by 18740: total requests 26
servied  by 44328: total requests 24
servied  by 4304: total requests 21
servied  by 33164: total requests 23
servied  by 10724: total requests 36
servied  by 10364: total requests 26
servied  by 5860: total requests 22
servied  by 42192: total requests 26
servied  by 18740: total requests 27
servied  by 44328: total requests 25
servied  by 4304: total requests 22
servied  by 10724: total requests 37
servied  by 33164: total requests 24
servied  by 10364: total requests 27
servied  by 18740: total requests 28
servied  by 10364: total requests 28
Worker 18740 died with code: 1, and signal: null
Starting a new worker
Worker 47900 is online
Process 47900 is listening to all incoming requests
```

When a task is killed

A new task can be spawned

Spawing a Python (or anyother) process

Friday, October 16, 2020 5:53 PM

```
router.get('/spawn',(req,res)=>{
    let min= req.query.min || 2;
    let max= req.query.max||100;
    let output='';
    if(min>max){
        res.status(400).send({error:'invalid range. min should be less than max', min,max});
    }
    else{
        let python=spawn('python',['primeagent.py',`${min}`,`${max}`]);
        python.stdout
        .on('data',data=>{
            let d=data.toString();
            console.log(d);
            output+=d;
        })
        .on('end', ()=>{
            console.log('end');
            res.send(output);
        });
```

You can spawn a child process

- Supply name of program and an array of command line arugment
- You can access the child process stdout/stdin

Collect out put on 'data' event

Sent the output to the client on end event

Memory Management

Friday, October 16, 2020 5:53 PM

```
router.get('/spawn',(req,res)=>{
    let min= req.query.min || 2;
    let max= req.query.max||100;
    let output='';
    if(min>max){
        res.status(400).send({error:'invalid range. min should be less than max', min,max});
    }
    else{
        let python=spawn('python',['primeagent.py',`${min}`,`${max}`]);
        python.stdout
        .on('data',data=>{
            let d=data.toString();
            console.log(d);
            output+=d;
        })
        .on('end', ()=>{
            console.log('end');
            res.send(output);
        });
```

- Here we are collecting output over a period from 'data' event and then we send to client on end event
- While this may or may not be a memory leak, it certainly
  - Takes a lot of server memory even for the time being
  - Client must wait till the entire data is collected in server memory before sent to the client

```
else{
    let python=spawn('python',['primeagent.py',`${min}`,`${max}`]);
    // python.stdout
    // .on('data',data=>{
    //     let d=data.toString();
    //     console.log(d);
    //     output+=d;
    // })
    // .on('end', ()=>{
    //     console.log('end');
    //     res.send(output);
    // });
    python.stdout.pipe(res);  //pipe the response from the spawned task directly to client. you may also pi
}
```

Direct piping data from spawned or other streams to response can avoid

- Waiting for collecting the data
- Large temporary storage