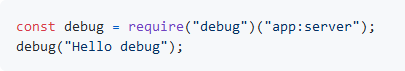
**Debugging e inspect**

**Haciendo debugging**

Para aprovechar por completo la funcionalidad de debugging que implementa Express, lo que recomiendo es cambiar todos los console.log por debug haciendo uso de un namespace de la siguiente forma:



De esta manera si ejecutamos nuestra aplicación con el comando DEBUG=app:\* node index.js nos mostrará los diferentes logs.

De esta manera si ejecutamos nuestra aplicación con el comando DEBUG=app:\* node index.js nos mostrará los diferentes logs.

Los namespaces que recomiendo son los siguientes:

* **app:server** para todo lo relacionado con el inicio del servidor como el mensaje Listening on http://localshost
* **app:db** para todo lo relacionado con logs de las bases de datos, inicialización y ejecución de scripts.
* **app:error**+++++++++++ para todo lo relacionado con errores en nuestra aplicación.

Nótese que esta convención es opcional, es decir, tu puedes seleccionar cualquier namespace. Lo más importante es que sea el mismo que se pasará en la opción **DEBUG**.

Express.js por defecto ya trae unos logs de debugging por defecto los podemos activar mediante la variable de entorno DEBUG=express:\*.

Por lo que recomiendo los scripts en nuestro archivo package.json de la siguiente manera:



### Ejecutando el modo inspect en desarrollo

El módulo inspect de Node.js nos permite ejecutar un ambiente para hacer debugging de código haciendo uso de la consola de desarrolladores de Google. Para ejecutarlo en modo desarrollo con nodemon basta con agregar el flag --inspect por lo que recomiendo el siguiente script en nuestro archivo package.json



**Tests**

**Creación de test para nuestros endpoints**

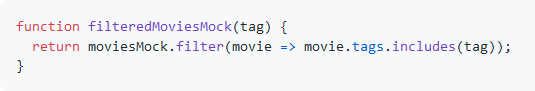
En esté modulo aprenderemos a crear test con nodejs, la primer suite de test que vamos a crear es para nuestros endpoints que están ubicados a la capa de rutas, **su única responsabilidad de test es probar que le llegan los datos y devuelven los datos correspondientes**, vamos a ver como se hace en el código.

1. Lo primero es instalar nuestras dependencias para los test npm i -D mocha supertest sinon proxyquire

* **mocha**: es quien nos ayuda a correr los test.
* **supertest**: es una utilidad que nos ayuda a levantar un servidor temporal.
* **sinon** es una utilidad que nos ayuda a hacer mocks para test
* **proxyquire**: es una utilidad que nos ayuda a inyectar los mocks cuando requiramos los paquetes.

1. Antes de proceder con los test vamos a hacer una pequeña modificación en movies mocks. No solo teniendo estos mocks que nos van a hacer útiles para los test, voy a crear una pequeña utilidad, la cual nos ayudará a crear las películas filtradas.

Como nuestro objetivo es hacer test de las rutas, nosotros nunca vamos a hacer que llegue hasta los servicios. Solo vamos a testear las rutas, y eso lo vamos a hacer con **sinon** y **proxyquire**.

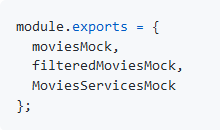
Luego vamos a crear un Mock de nuestros servicios, lo que hace esté mock es que cada vez que llamemos a getMovies() va a retornar la promesa que sería el Mock de las películas.



Y vamos a hacer un mock de método createMovie, y va a retornar la primera película de nuestro MoviesMock



Y exportamos ambas utilidades:



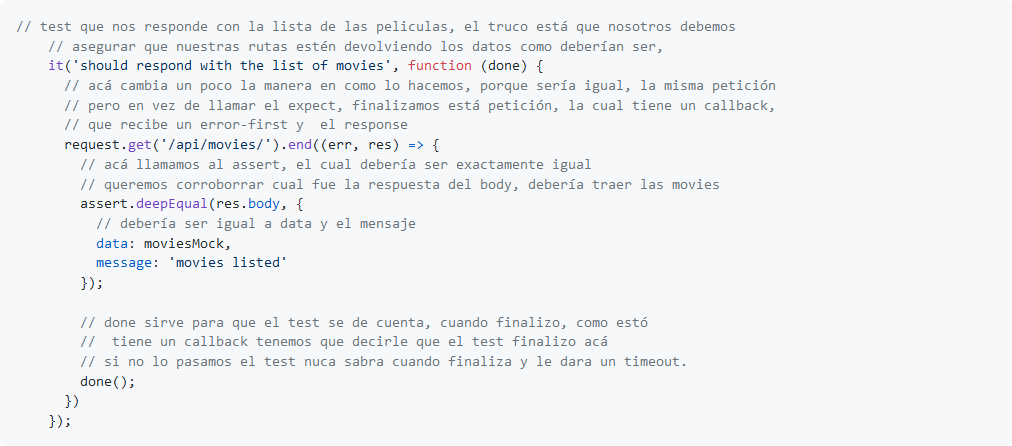
1. El próximo va a ser en nuestra carpeta de utilidades, es crear un test server, esté es un server que su única misión va a ser levantar un server para pruebas. **Los Test siempre se deben correr independientes del server original** y más porque en esté caso no queremos levantar todo lo que tienen nuestro server, solo queremos un server muy pequeño que nos ayude a probar.



Con esto ya tenemos lo necesario para empezar a crear nuestros test, entonces vamos a crear una carpeta que se llamé test y en este caso como queremos hacer test a las rutas se va a llamar routes.movies.test.js, es buena práctica que esté archivo finalice en **.test.js**

Ahora vamos a hacer una pequeña prueba corriendo esté test, para ello lo que vamos a hacer es configurar un script donde podamos correr los test.

Esté test llamará a mocha y como está en la carpeta de test y los archivos finalizan en **.test.js** debería ser capaz de leerlos sin problema, y si nos damos cuenta nuestro test está pasando sin problema.

Vamos a escribir otro test, esté test que vamos a escribir, es el test que nos va edificar que responda con la lista de películas.

Si volvemos a correr nuestros test y funcionará bien si los mensajes de los test son iguales a de las rutas.

Los Test son muy importante hacerlos en el código, porque si en un futuro hacen cambios, el test nunca se rompan, por ejemplo si en un futuro por alguna razón, llegarán y sin ninguna intención, cambiarán el mensaje que sería un error, si corremos nuestros test nos van a fallar y nos va a decir que no es exactamente igual.

**Los test son una buena manera de poder asegurar que nuestro código tiene calidad y que si en el futuro hay cambios, no se va a poder romper nuestro código**.

**Agregando un comando coverage**

Los Test son muy importantes, pero también es muy importante asegurarnos que estamos probando todos los caminos de una funcionalidad a la hora de hacer test, un comando de **coverage** nos permite identificar en donde estamos fallando y como los podemos corregir.

Para correr nuestro reporte de coverage, lo primero que tenemos que hacer es instalar una nueva herramienta que se llama **nyc**. npm i -D nyc **nyc**: hace parte de una herramienta llamada [istabul](https://istanbul.js.org/). Luego lo que necesitamos crear es nuestro comando el packages.json

{

"scripts": {

// El coverast se hace a partir de los test que hallamos creado. con esto

// estamos aplicando el coverast sobre nuestro comando de test

"cover": "nyc npm run test",

// Estás herramientas nos ayudan a crear diferentes reportes, en esté caso

// quiero que me abra el reporte en mi navegador.

"report": "nyc report --reported=html && open coverage/index.html"

},

}

Estás herramientas nos ayudan a generar diferentes reportes, sean para **environment** de integración continua o sea para nuestros ojos, en esté caso vamos a crear un reporte que va a estar en html.

Lo otro que debemos hacer es configurar como queremos hacer nuestro coverage, para ello nos vamos al final de nuestro package.json, y decimos que para ncy lo que queremos que testee es:

{

"nyc": {

"all": true,

"include": ["routes", "services", "lib", "utils"]

}

}

Esto es muy importante porque nos ayuda a visualizar el nivel de cobertura de nuestra aplicación, en el equipo se puede definir un **mínimo aceptable**, **la recomendación** de mínimo aceptable **es entre 60 y el 80%** porque tiene poco sentido obsesionarse por obtener el 100%, porque **es mucho más importante crear producto que simplemente estar creando test**.

En esté modulo podemos visualizar:

* Como crear test para nuestros endpoints, servicios y utilidades.
* Agregando un comando para coverage
* Debugging inspect

**Hacer test puede ser mucho más complicado que incluso estar escribiendo código pero te recomiendo que tengas mucha paciencia**.