UTN.BA
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL BUENOS AIRES

Centro de e-Learning

Diplomatura en programación web full stack con React JS

Centro de e-Learning SCEU UTN - BA.
Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

Módulo 4: NodeJS Intermedio

Unidad 4: Testeo y deploy

Presentación:

En esta unidad abordamos los conceptos fundamentales sobre la etapa de testeo, tan importante para garantizar el funcionamiento y la calidad de la aplicación que se desarrolle. Una vez establecidas las bases, utilizamos algunas herramientas que facilitan y automatizan el proceso de testing. Serán Mocha, Chai y Nock.

Objetivos:

Que los participantes:

Sean capaces de realizar el testeo y deploy de su aplicación Express.

Bloques temáticos:

1. Introducción al concepto de testeo
2. Formas de testear NodeJS
3. Concepto de deploy
4. Preparación de un proyecto NodeJS para el deploy
5. Alternativas de servidores para deploy
6. Ejemplo
7. Práctica

Consignas para el aprendizaje colaborativo

En esta Unidad los participantes se encontrarán con diferentes tipos de actividades que, en el marco de los fundamentos del MEC*, los referenciarán a tres comunidades de aprendizaje, que pondremos en funcionamiento en esta instancia de formación, a los efectos de aprovecharlas pedagógicamente:

- Los foros proactivos asociados a cada una de las unidades.
- La Web 2.0.
- Los contextos de desempeño de los participantes.

Es importante que todos los participantes realicen algunas de las actividades sugeridas y compartan en los foros los resultados obtenidos.

Además, también se propondrán reflexiones, notas especiales y vinculaciones a bibliografía y sitios web.

El carácter constructivista y colaborativo del MEC nos exige que todas las actividades realizadas por los participantes sean compartidas en los foros.

** El MEC es el modelo de E-learning colaborativo de nuestro Centro.*

Tomen nota:

Las actividades son opcionales y pueden realizarse en forma individual, pero siempre es deseable que se las realice en equipo, con la finalidad de estimular y favorecer el trabajo colaborativo y el aprendizaje entre pares. Tenga en cuenta que, si bien las actividades son opcionales, su realización es de vital importancia para el logro de los objetivos de aprendizaje de esta instancia de formación. Si su tiempo no le permite realizar todas las actividades, por lo menos realice alguna, es fundamental que lo haga. Si cada uno de los participantes realiza alguna, el foro, que es una instancia clave en este tipo de cursos, tendrá una actividad muy enriquecedora.

Asimismo, también tengan en cuenta cuando trabajen en la Web, que en ella hay de todo, cosas excelentes, muy buenas, buenas, regulares, malas y muy malas. Por eso, es necesario aplicar filtros críticos para que las investigaciones y búsquedas se encaminen a la excelencia. Si tienen dudas con alguno de los datos recolectados, no dejen de consultar al profesor-tutor. También aprovechen en el foro proactivo las opiniones de sus compañeros de curso y colegas.

1. Introducción al concepto de testeo

Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence. – Edsger Dijkstra	*El testing de programas puede ser una forma muy efectiva de mostrar la presencia de errores, pero es desesperanzadoramente inadecuado para mostrar su ausencia.* – Edsger Dijkstra

Definición de testing

El testing es una de las etapas del ciclo de vida de los sistemas de información y es sumamente importante!

El testing es el proceso de validar y verificar que un programa de software o aplicación cumpla con los requerimientos, funcione como se espera y pueda ser implementado.

Como dijera Dijkstra, el testing será una vía para encontrar los errores en el sistema que realicemos pero NO servirá para garantizar la ausencia de ellos. De hecho, un buen desarrollador debería estar bastante preocupado si al probar el sistema no encuentra algunos cuantos errores.

Por regla general, sistema sin errores, sistema que no ha sido probado o que habiendo sido probado, no se probó correctamente. Por lo tanto, durante el testeo buscaremos encontrar errores, fallos, respuestas incorrectas o inesperadas.

Definiciones

Validación: ¿estamos construyendo el producto correcto? Consiste en corroborar si el programa satisface las expectativas del cliente/usuario.

Verificación: ¿estamos construyendo el producto correctamente? Consiste en comprobar que el programa respeta su especificación.

Casos de testeo: descripción de una prueba atómica, por medio de parámetros de entrada (datos de entrada), condiciones de ejecución (necesarias para preparar el programa para las pruebas) y resultados esperados.

Set de testeo: conjunto de casos de testeo.

Bug: Característica no deseada del programa desde el punto de vista del usuario. Puede ser producido por uno o varios defectos del programa.

Errores: una equivocación que conduce a un resultado incorrecto. Generalmente causada por los desarrolladores, son fallas humanas.

Fracasos: resultados inesperados en la ejecución de un programa.

Incidente: condición que viola la especificación de un programa.

Tips

➔ El test debe ser reproducible
➔ No debe haber auto-testeo
➔ Probar casos para condiciones válidas e inválidas

Diseño de casos de testeo

Documentación de caso de testeo

Lo que caracteriza a la documentación de un caso de testeo es que existe una entrada conocida y una salida esperada que se formulen previo a que se ejecute la prueba.

Estructura de los casos de pruebas

Introducción/visión general: Contiene información general acerca de los Casos de Prueba.

Identificador: Es un identificador único para futuras referencias, por ejemplo, mientras se describe un defecto encontrado.

Caso de prueba dueño/creador: Es el nombre del analista o diseñador de pruebas, quien ha desarrollado pruebas o es responsable de su desarrollo.

Versión: La actual definición del caso de prueba.

Nombre: El caso de prueba debe tener un título entendible por personas, para la fácil comprensión del propósito del caso de prueba y su campo de aplicación.

Identificador de requerimientos: el cual está incluido en el caso de prueba. También aquí puede ser un identificador de casos de uso o de especificación funcional.

Propósito: Contiene una breve descripción del propósito de la prueba, y la funcionalidad que chequea.

Dependencias: Indica qué otros subsistemas están involucrados y en qué grado.

Actividades de los casos de prueba

Ambiente de prueba/configuración: Contiene información acerca de la configuración del hardware o software en el cual se ejecutará el caso de prueba.

Inicialización: Describe acciones, que deben ser ejecutadas antes de que los casos de prueba se hayan inicializado. Por ejemplo, debemos abrir algún archivo.

Finalización: Describe acciones, que deben ser ejecutadas después de realizado el caso de prueba. Por ejemplo si el caso de prueba estropea la base de datos, el analista debe restaurarla antes de que otro caso de prueba sea ejecutado.

Acciones: Pasos a realizar para completar la prueba.

Descripción de los **datos de entrada**

Resultados

Salida esperada: Contiene una descripción de lo que el analista debería ver tras haber completado todos los pasos de la prueba.

Salida obtenida: Contiene una breve descripción de lo que el analista encuentra después de que los pasos de prueba se hayan completado.

Resultado: Indica el resultado cualitativo de la ejecución del caso de prueba, a menudo con un Correcto/Fallido.

Severidad: Indica el impacto del defecto en el sistema: Grave, Mayor, Normal, Menor.

Evidencia: En los casos que aplica, contiene información, bien un link al print de pantalla (screenshot), bien una consulta a una base de datos, bien el contenido de un fichero de trazas, donde se evidencia la salida obtenida.

Seguimiento: Si un caso de prueba falla, frecuentemente la referencia al defecto implicado se debe enumerar en esta columna. Contiene el código correlativo del defecto, a menudo corresponde al código del sistema de tracking de bugs que se esté usando.

Estado: Indica si el caso de prueba está: No iniciado, En curso, o terminado.

Diferentes tipos de testeo

Unitarios: pruebas individuales de cada módulo/componente del sistema.

De Integración: se prueba justamente la integración de los módulos/componentes. El foco está puesto en el enlace entre módulos/componentes. La "salida" de un módulo que alimenta la "entrada" de otro. Estos testeos suelen ser un poco más complejos. Deberían realizarse una vez que hayan sido aprobados los test unitarios de los respectivos módulos intervinientes, caso contrario, un error dentro de un módulo provocaría una falla en el testeo de integración y podría resultar difícil detectar el punto de error.

Funcionales: pruebas realizadas desde fuera del sistema (black box). Aquí se prueba una función de las que el sistema ofrece, no nos concentramos en el cómo lo hace sino en que lo haga.

End-To-End (e2e): prueba de la aplicación en un entorno similar al real. Aquí se prueba todo el sistema, ya lo hemos concluido y antes de entregarlo realizamos las pruebas generales para asegurarnos que su funcionamiento es correcto.

Aceptación: prueba del usuario para verificar que el sistema cumple con los requerimientos solicitados. Este tipo de pruebas se realizan con un documento de aceptación donde se especifica los requerimientos realizados previamente por el cliente.

2. Formas de testear NodeJS

Herramientas de testeo

Mocha

Framework JS de testeo simple y flexible para Node JS

Es una aplicación que facilita el testeo del sistema. Se “programa” un set de testeos y Mocha ejecutará cada caso automáticamente e informará sobre el resultado de los mismos.

Consideraciones

- Los testeos corren de manera serial (uno luego de otro)
- Incorpora reportes
- Incluye las excepciones no capturadas en el caso de testeo correspondiente
- Reporte de coverage (cobertura)
- Permite correr tests de cumplen una RegExp (Expresión Regular)
- Podemos usar Promises
- Permite la utilización de bibliotecas de assertions como por ejemplo, Chai

Instalación

```
npm install --save-dev mocha
```

Chai

Es una biblioteca para assertions (indicar los resultados esperados). Las palabras claves son:

expect, assert, should

Instalación

```
npm install --save-dev chai
```

Una vez instalados ambos paquetes, comenzamos con la forma de trabajar con estas herramientas!

Código de testeo

```
var nombre = 'Juan';
var assert = require('chai').assert
var expect = require('chai').expect;
var should = require('chai').should(); // Aquí llamamos a una función

assert.equal(nombre, 'Juan'); // El resultado correcto es que nombre sea Juan
expect(nombre).to.equal('Juan'); // Se espera que nombre sea igual a Juan
nombre.should.equal('Juan'); // El nombre debería ser Juan
```

Algunas estructura

Al codificar se pueden usar cualquiera de las opciones indicadas (assert, expect, should), a continuación utilizaremos expect pero es sólo una cuestión de gustos.

```
expect(obj).to.be.a('<tipo de datos>');
expect(obj).to.equal('<valor>');
expect(obj).to.have.length(<longitud>);
expect(obj).to.have.property('<nombre prop>');
```

Enlaces

Detengámonos en la lectura “fluida” (en inglés) que podemos hacer de cada una de las sentencias anteriore. Esto se logra por los “enlaces” que son palabras sin ningún tipo de funcionalidad pero que son proveídas por las herramientas para facilitar la lectura del código.

Los enlaces disponibles son:

- to
- be
- been
- is
- that
- which
- and
- has
- have
- with
- at
- of
- same

Ejemplo:

```
expect(obj).to.have.length(<longitud>);
```

to.have son 2 enlaces que se utilizan juntos, siempre separados por un punto (.)

Palabras reservadas con función

Algunas funciones que nos serán de utilidad:

.not()	niega la respuesta
.true()	espera el valor verdadero
.false()	espera el valor falso
.null()	espera el valor null
.undefined()	espera el valor undefined
.NaN()	espera el valor NaN (not a number)
.above(x)	espera un valor menor al pasado
.below(x)	espera un valor mayor al pasado
.within(x, y)	espera que el valor esté entre el mínimo y máximo pasado

Nock

Es una biblioteca para realizar mock (simuladores) de peticiones HTTP.

Es de uso muy sencillo y nos permite hacer mock de servidor HTTP remoto

Instalación

```
npm --save-dev nock
```

Sintaxis:

```
var nock = require('nock');
var mockServer = nock('<server>').get('<url>')
    .reply(<http status>, <json de respuesta>);
```

Mocha con Chai

Modularización - Estructura de carpeta

Para los test vamos a crear una carpeta a fin de mantener ordenado el trabajo.

Entonces, dentro de la carpeta app vamos a crear la carpeta test:

app
  + test: carpeta en la cual incluímos todos los tests

Estructura básica de un archivo de testeo

```
var expect = require('chai').expect;

describe('Nombre que identifique los testeos', function() {
    it('Testeo 1', function() { });
```

```
    it('Testeo 2', function() { });
});
```

Dentro de cada estructura *it('nombre testeo', function() { })* se debe escribir el código del testeo a realizar.

```
var expect = require('chai').expect;

describe('Prueba de un velador', function() {
    it('Caso 1: Funcionamiento correcto del velador', function(velador)
{
        // Código para probar que un velador funcione
    });

          it('Caso  2:  Funcionamiento  incorrecto  a  110  volts',
function(velador) {
        // Código para probar que el velador no funciona con 110 volts
    });
});
```

Implementación de un testeo en Mocha con Chai

Luego de instalarlo y crear la carpeta test...

1. Crear el/los archivos de testeo. Por ejemplo test.js (dentro de la carpeta de testeo)

2. Con el editor que utilice abrir el archivo test.js y escribir el código de testeo que deberá ser algo similar a lo siguiente:

```
var assert = require('assert');
describe('Array', function() {
  describe('#indexOf()', function() {
    it('should return -1 when the value is not present', function() {
```

```
      assert.equal([1,2,3].indexOf(4), -1);
    });
  });
});
```

3. Editar el archivo package.json y agregar las siguientes líneas para crear un script del testeo y que sea más fácil su ejecución

```
"scripts": {
    "test": "mocha --exit"
  }
```

4. Correr el testeo

```
$ npm test
```

Nock

Ejemplo para crear un mock de servidor HTTP:

```
var nock = require('nock'); // Indica que se utilizará nock
var mockServer = nock('https://www.google.com') // Indica la url del
servidor a emular, puede ser cualquiera
                    .get('/?q=mercadolibre') // La forma que tendrá la
petición
                  .query({q: 'mercadolibre'})
                  .reply(200,  // respuesta
                      {    results: [
                              {title: 'Sitio ML 1'},
                              {title: 'Sitio ML 2'}
                          ]
                      }
                  );
```

Testeo de funciones asincrónicas

En los casos que los testeos se deban realizar sobre funciones asincrónicas (que retornan una Promise, por ejemplo), podemos agregar un parámetro (callback) al caso de testeo para indicar que el mismo finalizó y su resultado del mismo

```
const chai = require('chai');
const nock = require('nock');

const assert = require('chai').assert;
const expect = require('chai').expect;
require('chai').should();


function funcionAsyncOk() {
   return new Promise((resolve, reject) => {
       resolve('OK');
   })
}

function funcionAsyncConError() {
   return new Promise( (resolve, reject) => {
       reject(new Error('No se pudo hacer la operación'));
   })
}

it('Prueba que pasa ok', (done) => {
   funcionAsyncOk().then(rta => {
       done(); // <- Pasa por aquí
   }).catch(e => {
       done(e);
   })
})

it('Prueba que falla', (done) => {
   funcionAsyncConError().then(rta => {
       done();
   }).catch(e => {
          done(e); // <- Pasa por aquí. Al llamar a done con un parámetro estamos
indicando que falló la operación
   })
})
```

El primer testeo, ‘Prueba que pasa ok’ se recibe el parámetro done (callback) el cual es llamado tanto si la función se ejecuta bien, como si la misma se ejecuta con error. En caso de producirse un error (que falle el testeo) a la callback se le pasa un parámetro (el error).

En el .then de la Promise se llama a done() porque la función se ejecutó de la manera esperada.

En el .catch de la Promise se llama a done(e) porque la función no se ejecutó de la manera esperada.

Preparación del entorno de testeo y liberación

Cuando estamos creando los casos de testeo, es posible que necesitemos preparar el entorno antes de realizar cada prueba. Podemos crear datos necesarios, conexiones, configuraciones, etc. Todo este código podemos agruparlo dentro de una función especial “before” la cual recibe como parámetro la función que debe ejecutarse antes de iniciar el conjunto de casos de testeo.

También podemos ejecutar código antes de iniciar cada caso de testeo en particular, para este caso tenemos la función “beforeEach”. Veamos un ejemplo de cómo podemos usar ambas.

```
const chai = require('chai');

const assert = chai.assert;

describe('Casos de testeo', () => {

   before( () => {
       console.log('Al iniciar casos de testeo')
   })

   beforeEach(() => {
       console.log('Antes de cada caso de testeo')
   })

   it('Caso 1', () => {
       assert(true, 'True es true');
   })

   it('Caso 2', () => {
       assert(true, 'True es true');
   })

   it('Caso 3', () => {
       assert(true, 'True es true');
   })

})
```

Y la correspondiente salida por consola de la ejecución de las pruebas.

```
  Casos de testeo
Al iniciar casos de testeo
Antes de cada caso de testeo
    ✓ Caso 1
Antes de cada caso de testeo
    ✓ Caso 2
Antes de cada caso de testeo
    ✓ Caso 3


  3 passing (10ms)
```

Como podemos apreciar:

- El código de la función before() es llamado una única vez para este conjunto de casos de testeo
- El código de la función beforeEach() es llamado antes de iniciar cada caso de testeo

Al igual que disponemos de funciones que se ejecutan antes de los casos de testeo, tenemos la posibilidad de ejecutar funciones luego de la ejecución de cada caso de testeo, o del conjunto de casos de testeo.

Veamos el ejemplo anterior, ampliado para estos casos

```
const chai = require('chai');
const assert = chai.assert;

describe('Casos de testeo', () => {
   before( () => {
       console.log('Al iniciar casos de testeo')
   })
   beforeEach(() => {
       console.log('Antes de cada caso de testeo')
   })
   after( () => {
       console.log('Ejecutado al final de todos los testeos')
   })
   afterEach( () => {
       console.log('Ejecutado al finalizar cada caso')
   })
```

```
    it('Caso 1', () => {
        assert(true, 'True es true');
    })
    it('Caso 2', () => {
        assert(true, 'True es true');
    })
    it('Caso 3', () => {
        assert(true, 'True es true');
    })
})
```

Y la correspondiente salida por pantalla al ejecutar este nuevo caso de testeo.

```
  Casos de testeo
Al iniciar casos de testeo
Antes de cada caso de testeo
    ✓ Caso 1
Ejecutado al finalizar cada caso
Antes de cada caso de testeo
    ✓ Caso 2
Ejecutado al finalizar cada caso
Antes de cada caso de testeo
    ✓ Caso 3
Ejecutado al finalizar cada caso
Ejecutado al final de todos los testeos


  3 passing (10ms)
```

Las funciones before() y beforeEach() son útiles para inicializar datos necesarios por todos los testeos del conjunto. Mientras que las funciones after() y afterEach() son útiles para liberar los datos o recursos que hayamos utilizado en nuestros testeos.

Algunos ejemplos en los cuales podemos usar estas funciones son:

- Conexión/desconexión con una base de datos
- Inicialización/borrado de datos en una base de datos
- Asignación/liberación de recursos

Testeo de una aplicación Express

Al momento de realizar las pruebas en nuestra aplicación también tenemos la posibilidad de realizar pruebas sobre Express, emulando peticiones del cliente y analizando la respuesta.

Para poder realizar peticiones a la aplicación Express debemos incorporar un nuevo paquete a nuestras dependencias de desarrollo chai-http

```
npm install --save-dev chai-http
```

Para poder probar nuestra aplicación, debemos realizar una serie de pasos que a continuación veremos.

1. Modificar nuestra aplicación (en la cual utilizamos Express) para que exporte el servidor

```
const server = app.listen(3000, function () {
   console.log('Iniciando la aplicación en http://localhost:3000 ');
});
module.exports = server;
```

2. Definir los casos de testeo y que en los mismos se utilice el módulo de la aplicación Express. En el ejemplo siguiente el archivo de la aplicación Express tiene como nombre “index.js” y se encuentra en la carpeta ../

   Al iniciar el conjunto de casos de testeo se define la variable server con el servidor express que se inicia al realizar el require

   Al finalizar los casos de testeo, se detiene el servidor Express

```
describe('Conjunto de testeos', () => {
   var server;
   before(function () {
       server = require('../index');
   });
   after(function (done) {
       server.close();
       done();
   });
   ...
})
```

3. Realizar peticiones al servidor Express. Para ello llamamos a chai.request(server) y le indicamos que tipo de petición deseamos realizar get, post, put, etc. Luego en la respuesta (res) obtenemos la respuesta de Express, en la cual podemos verificar el status code de la respuesta, el cuerpo (body) o las comprobaciones que deseemos realizar sobre la misma.

```
   it('Obtener pagina principal', (done) => {
       chai.request(server).get('/').end( (err, res) => {
           res.should.have.status(200);
           done();
       })
   })
```

3. Concepto de deploy

Deploy o despliegue de una aplicación es el proceso por el cual una aplicación pasa a estar productiva.

Ahora que tenemos nuestra aplicación lista para publicar en nuestro servidor de Internet, debemos realizar unos pasos previos para poder publicar la misma definitivamente.

Los pasos principales que debemos realizar para publicar nuestra aplicación son:

1. Validar que pase todos los casos de testeo de manera correcta
2. Verificar la seguridad de nuestra aplicación
3. Hacer los cambios necesarios para el entorno productivo
4. Subir la aplicación al servidor productivo
5. Iniciar la aplicación en el servidor productivo
6. Verificar el funcionamiento de la aplicación

4. Preparación de un proyecto NodeJS para el deploy

Validar casos de testeo

La manera más sencilla de verificar que nuestra aplicación pase los testeos es correr los mismos de forma manual, por línea de comandos, y en caso que todos los testeos hayan pasado de manera correcta, podemos garantizar que la aplicación funciona correctamente (para los casos de testeo que hayamos definido)

Existen herramientas de automatización de disparo de los tests y despliegue de la aplicación, las mismas se encuentran fuera de los alcances de este curso, pero recomendamos analizar las mismas y su funcionamiento, ya que se obtendrán ventajas significativas con su uso.

Estas herramientas son las llamadas CI (Continuous Integration / Integración continua) y entre las más populares podemos destacar:

- Jenkins: https://jenkins.io
- Travis CI: https://travis-ci.com
- Hudson CI: http://hudson-ci.org
- Bamboo: https://www.atlassian.com/software/bamboo
- TeamCity: https://www.jetbrains.com/teamcity/
- GitLab CI: https://about.gitlab.com/product/continuous-integration/

Verificar la seguridad

La seguridad en nuestras aplicaciones es esencial, debemos proteger nuestra aplicación ante ataques maliciosos. Es importante poder proteger todo el trabajo que hemos hecho, y la seguridad de nuestros usuarios que nos confían sus datos.

Existen varias herramientas que ayudan a validar la seguridad de nuestra aplicación, las cuales ampliaremos en la sección 2 de esta Unidad.

Hacer los cambios necesarios para el entorno productivo

Existen algunos cambios, aunque sea mínimos, que debemos realizar en nuestra aplicación para poder publicarla en otro equipo. Entre los cambios más comunes podemos destacar los siguientes:

- Cambiar la configuración de los accesos a los datos (base de datos)
- Cambiar la configuración de los equipos contra los cuales nos conectamos (en caso de ser necesario)
- Cambiar la configuración de los directorios de los cuales depende nuestra aplicación y se encuentran por fuera de la misma (ej: si accedemos a un directorio particular del equipo, por fuera del directorio de nuestra aplicación)

Para facilitar el cambio de configuración entre el equipo en el cual se desarrolla y el servidor en el cual correrá la aplicación podemos utilizar paquetes que nos facilitarán el trabajo. Uno de estos paquetes es dotenv el cual permite usar un archivo de configuración con nombre ".env". Este archivo se debe encontrar en el directorio raíz de nuestro proyecto y no debemos incorporarlo en el control de versiones (Ej: GIT). Al no incorporarlo en el control de versiones nos aseguramos que no sobre-escribiremos la configuración de ningún equipo en el cual se corra la aplicación.

Comencemos instalando el paquete

```
npm install --save dotenv
```

Una vez instalado, debemos crear el archivo .env el cual contiene toda la configuración de nuestro proyecto en el formato clave=valor

```
PUERTO=3000
DB_HOST="localhost"
DB_COLLECTION="curso_nodejs_intermedio"
MEMCACHED_SERVER="localhost"
MEMCACHED_SECRET="clave-muy-secreta-01-memcached"
```

Cada equipo en el cual se correrá la aplicación deberá tener su propio archivo .env con sus parámetros particulares dependiendo de la configuración.

En nuestro proyecto debemos inicializar dotenv. Para ello importamos el paquete e inicializamos el módulo llamando al método config.

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

Para usar la configuración del archivo .env solo debemos agregar process.env.<nombre de la clave>. Por ejemplo para usar el PUERTO en la inicialización del servidor Express

```
app.listen(process.env.PUERTO, function () {
   console.log(`Iniciando la aplicación en http://localhost:${process.env.PUERTO}`);
});
```

Para la configuración especificada en el ejemplo se sustituirá en tiempo de ejecución por

```
app.listen(3000, function () {
   console.log(`Iniciando la aplicación en http://localhost:3000`);
});
```

Es una práctica habitual el incorporar en el control de versiones un archivo env_example o similar, el cual contenga los parámetros que acepta el archivo de configuración (aquellos parámetros que luego utilizamos en nuestra aplicación). Para que al momento de descargar el código en una nueva computadora, se copie el archivo env_example en .env y se modifique .env para que contenga los parámetros de configuración para el equipo en el cual se está corriendo la aplicación.

Puedes acceder al código de ejemplo en el repositorio https://github.com/cursos-utn/nodejs-intermedio/tree/m2_u3_configuracion en el cual se encuentra el archivo env_example de referencia. Recuerda que al momento de descargar el código en tu computadora, deberás crear el archivo .env con los parámetros de configuración que se encuentran en el archivo env_example (adaptandolo a tu equipo)

Subir la aplicación al servidor productivo

Una vez que ya tenemos nuestra aplicación lista para subir al servidor, queda el paso crucial que es su subida propiamente dicho e inicio. Existen diferentes métodos para subir la aplicación a un servidor, algunos manuales y otros automatizados.

Por medio de los procesos manuales, debemos realizar una conexión con el servidor en el cual vamos a desplegar nuestra aplicación (por FTP, SFTP o cualquier otro método de conexión que disponga el servidor), seleccionar la carpeta en la cual correrá nuestra aplicación, y subir todos los archivos de nuestra aplicación a dicho equipo. Este es un proceso manual, que nos llevará tiempo y que debemos tener especial cuidado para garantizar la correcta subida de toda la aplicación.

Para el proceso de subida automatizado, existen muchas herramientas que nos pueden ayudar en esta tarea, que es repetitiva (cada vez que debemos actualizar nuestra aplicación en el servidor, debemos realizar esta tarea). Las herramientas de Integración Continua que mencionamos anteriormente, generalmente disponen de herramientas para poder subir la aplicación al servidor de manera automática.

Cuando veamos la sección 3 Process Managers, veremos que NodeJS ya cuenta con una herramienta que nos puede ayudar en este proceso.

Es importante recordar que no es necesario copiar la carpeta node_modules, siempre y cuando hayamos utilizado el archivo package.json para manejar las dependencias de nuestra aplicación. Solo debemos ejecutar el siguiente comando en el servidor, para descargar todas las dependencias de nuestra aplicación:

```
npm install
```

Automáticamente descarga todas las dependencias de nuestra aplicación en el directorio node_modules de nuestro servidor.

Iniciar la aplicación en el servidor productivo

Con nuestra aplicación ya lista para comenzar a funcionar, solo nos resta iniciar la misma. El inicio de la aplicación se puede realizar de forma manual por línea de comandos (suponiendo que nuestro archivo principal es el app.js)

```
node app.js
```

Debemos prestar especial atención a que si por algún motivo nuestra aplicación se cierra inesperadamente, la misma dejará de funcionar. En la sección 3 de Process Managers, veremos cómo podemos hacer para que ante una finalización inesperada de la aplicación, la misma se inicie automáticamente.

Verificar el funcionamiento de la aplicación

Debemos acceder por medio de nuestro navegador a la aplicación, en el servidor, para verificar que la misma haya iniciado correctamente y poder garantizar su funcionamiento. En caso que se haya producido algún error en los pasos anteriores, es muy probable que aquí nos encontremos con que nuestra aplicación muestra un mensaje de error, o presenta algún comportamiento inesperado. De ser ese el caso, debemos volver a verificar todos los pasos que hemos realizado para poder detectar en cual se produjo el error y solucionarlo.

Seguridad

Antes de publicar nuestra aplicación, debemos verificar la seguridad de la misma. Recomendamos verificar los siguientes pasos antes de subir nuestra aplicación:

1. Verificar que estamos utilizando versiones actuales de los módulos (dependencias de nuestra aplicación)
2. Usar HTTPS en lo posible, para garantizar que las conexiones son seguras
3. Usar cookies de manera segura (intentar no utilizar el nombre predeterminado)
4. Verificar que las dependencias sean seguras
5. Asegurar nuestra aplicación con paquetes especiales (Ej: Helmet)
6. Verificar avisos de seguridad

Verificación de seguridad en dependencias

Para verificar si las dependencias de nuestro proyecto son seguras (no tienen vulnerabilidades) podemos ejecutar el comando

```
npm audit
```

El cual nos indicará las vulnerabilidades encontradas en nuestro proyecto (con respecto a dependencias).

El mismo npm nos permite solucionar los inconvenientes de dependencias por medio de una simple línea de comando

```
npm audit fix
```

Helmet

Es un paquete que podemos instalar como dependencia de nuestra aplicación, el cual nos permite asegurar las aplicaciones que utilizan Express, modificando varios de los encabezados HTTP que usa Express de manera predeterminada.

¿Que hace Helmet?

Helmet es un paquete que actúa como middleware para nuestras aplicaciones Express, el cual provee modificadores al comportamiento predeterminado de Express.

Los encabezados que modifica Helmet en Express son:

- **Content security policy**: permite definir en qué sitios y contenido debe confiar el browser.
- **DNS Prefetch Control**: no permite que se resuelvan los nombres de dominio a IPs antes de tiempo (haciendo prefetch a los enlaces de un sitio web)
- **Frameguard**: indica al navegador que la página no debe ser utilizada dentro de un IFrame.
- **HPKP - HTTP Public Pinning**: se envia al browser la clave pública del certificado (HTTPS) para que el pueda evaluar si el certificado ha sido vulnerado.
- **HSTS - Strict Transport Security**: le indica al browser que siempre utilice la versión HTTPS del sitio web, y nunca visite la versión HTTP (si existe).
- **IE No Open**: previene a versiones viejas de Internet Explorer que abran un archivo HTML descargado en el contexto de la aplicación.
- **No cache**: le indica al browser que no realice caché de los archivos. Esto previene el uso de archivos viejos (ej: JavaScript desactualizados).
- **Don't Sniff Mime Type**: Indica al browser que no intente detectar el MIME Type del contenido. Ej: evita que un archivo con extensión JPG que contiene código JS sea ejecutado
- **Referrer Policy**: Le indica al browser que no incluya el campo Referer (para evitar que detecten desde donde se realizan los enlaces)
- **XSS Filter**: Intenta prevenir un único tipo de ataque XSS. El browser no debería ejecutar el código dentro de <script> si concuerda con el query string

Instalación de Helmet

La instalación de Helmet es muy sencilla, como todos los paquetes de NodeJS.

```
npm install helmet --save
```

Uso de Helmet

Al igual que la instalación, su uso es bastante sencillo. Una vez que hemos instalado la dependencia, solo resta incluir helmet en nuestro proyecto (en el código de nuestro archivo inicial)

```
var helmet = require('helmet');

app.use(helmet());
```

Debemos indicar que incluímos módulo helmet y luego indicarle a Express que use dicho módulo como un middleware. Helmet automáticamente se encargará del resto.

5. Alternativas de servidores para deploy

Entre las alternativas actuales (al momento de preparar este documento) más populares se encuentran:

Heroku: https://www.heroku.com/
AWS: https://aws.amazon.com/
GCP: https://cloud.google.com/

y muchas más que podrán consultar en Internet

6. Ejemplo

Testeo

El velador

Carlos ha comprado un velador y en la casa de electrodomésticos le sugirieron probarlo apenas llegara a su casa, ya que el plazo máximo para devolverlo es de 48hs.

Apenas llega a su casa, Carlos abre la caja del velador, tomó las instrucciones y se dispone a leerlas para evaluar la forma de probarlo.

En sus instrucciones de uso se indica que debe ser conectado a 220 volt de corriente alterna y previamente ajustar una lamparita de máximo 100 W. La lamparita mínima sugerida es de 20 W. El tipo de toma es de 3 patas planas, estilo americano

Carlos decide entonces, escribir algunos casos de testeo para asegurarse de que su nuevo velador es lo que esperaba y funciona correctamente.

Caso de prueba 1: El velador funciona?

Pantalla/Módulo/Caso de uso: Encender el velador
Objetivo de la prueba: Encender el velador con una lamparita de 50 watt y el velador conectado a 220 volt
Prerrequisitos de la prueba:

- Contar con el velador
- Contar con la lamparita
- Que haya corriente eléctrica en un toma de 3 patas planas

Procedimiento

- Apoyar el velador en una superficie plana
- Enroscar la lamparita en el velador
- Enchufar el velador en el tomacorriente
- Presionar el interruptor del velador

Resultados esperados:

La lámpara se enciende

Resultado obtenido:

De acuerdo a lo esperado

Observaciones: ---

Resultado de la prueba: aprobado

Deploy

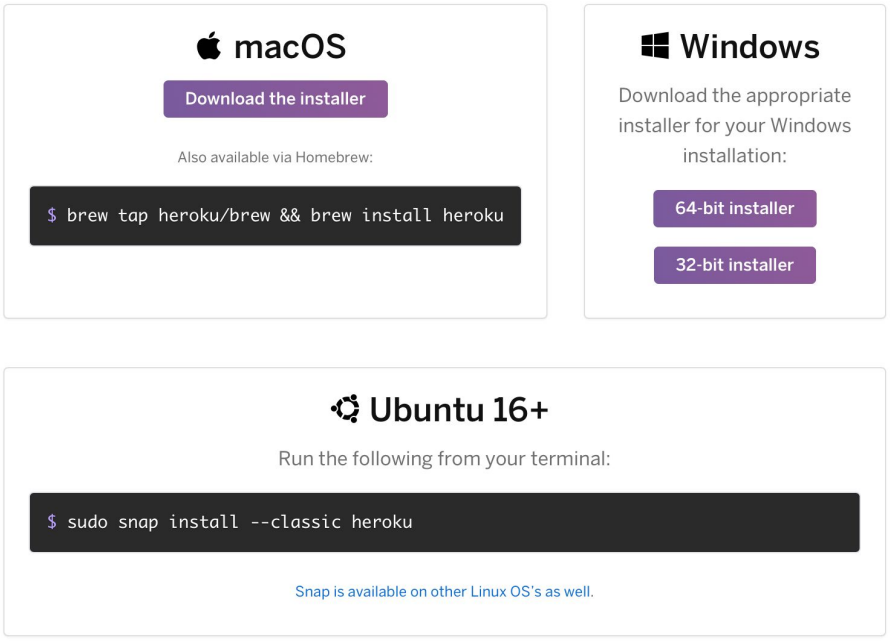
Despliegue en Heroku

Es una plataforma como servicio de computación en la nube (SAAS) que soporta distintos lenguajes de programación, entre ellos NodeJS.

Instalación del cliente heroku

Para poder utilizar heroku debemos instalar un cliente (el cual se utiliza por medio de línea de comando) en nuestra computadora. Debemos ingresar a https://devcenter.heroku.com/articles/heroku-cli#other-installation-methods

Y seleccionar el método que se adapte a nuestro sistema operativo



Verificación que el cliente funciona correctamente

Como primer paso debemos verificar que el cliente de heroku se encuentra instalado y funcionando, para ello ejecutamos en la línea de comandos (cmd, símbolo del sistema, terminal)

```
heroku –version
```

Asociar el cliente a nuestra cuenta

Debemos asociar el cliente de heroku con nuestra cuenta, para ello ejecutamos

```
heroku login
```

Adaptar el proyecto a heroku

Debemos realizar algunos cambios en nuestro proyecto para que el mismo pueda correr en heroku.

Debemos verificar que nuestro archivo package.json contenga un script de start, y que el mismo inicie la aplicación.

```
"scripts": {
  "start": "node index.js",
  "test": "echo \"Error: no test specified\" && exit 1"
},
```

Debemos agregar una sección engines, en la cual le indiquemos con que versión de NodeJS deberá correr nuestra aplicación cuando la despleguemos en los servidores de Heroku.

```
"engines": {
    "node": "10.x"
}
```

También deberemos modificar nuestro archivo principal para permitir que Heroku defina en qué puerto escuchará nuestra aplicación.

```
app.listen(process.env.PORT || 3000, () => {
    console.log('App escuchando en puerto 3000')
})
```

La configuración del puerto es interna a Heroku y es necesaria por la forma en al cual despliega las aplicaciones. Independientemente del puerto en el cual sea asignada nuestra aplicación, accederemos a ella de manera tradicional sin necesidad de especificar un puerto.

Desplegar la aplicación

Para el despliegue de la aplicación debemos seguir una serie de pasos sencillos.

En primer lugar, debemos incorporar nuestro código al sistema de control de versiones GIT.

```
git add .

git commit -m "Proyecto en GIT"
```

Debemos indicarle a heroku que vamos a desplegar este proyecto en su plataforma

```
heroku create
```

Y realizamos el despliegue final realizando un push (prestar especial atención a la sintaxis en la cual se indica heroku en vez de origin (como muchas veces se utiliza en git)

```
git push heroku master
```

Al finalizar la operación nos mostrará la URL de la aplicación en un formato

```
https://<nombre-app>.herokuapp.com/
```

Dicha URL es en la cual la aplicación se encuentra funcionando. De esta forma ya tenemos todo listo para usar nuestra aplicación.

Logs para detección de errores

Podemos acceder a los últimos logs que genera nuestra aplicación por medio del siguiente comando

```
heroku logs –tail
```

Esto nos permitirá ver los últimos logs y de esta forma detectar problemas o conocer el funcionamiento de nuestra aplicación.

7. Práctica

Testeo

En el repositorio de Github:
(https://github.com/cursos-utn/nodejs-intermedio/tree/m4_u2_testeo) se encuentra todo el código utilizado en este módulo y las modificaciones en index.js para que se puedan realizar las pruebas a Express.

Recuerda que el código de las pruebas se encuentra en la carpeta “test”

Deploy

Realizar el deploy de una aplicación NodeJs en Heroku o similar.

Bibliografía utilizada y sugerida

Conceptos básicos de pruebas de software:

https://es.scribd.com/document/249375104/Conceptos-Basicos-de-Pruebas-de-Software

Casos de Prueba. https://es.wikipedia.org/wiki/Caso_de_prueba

Express Tutorial Part 7: Deploying to production. Recuperado de:

https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/deployment

Maximiliano Cristiá. Introducción al Testing de Software. Cátedra Ingeniería de Software. Facultad de Ciencias Exactas, Ingeniería y Agrimensura.Universidad Nacional de Rosario. Noviembre de 2009 https://www.fceia.unr.edu.ar/ingsoft/testing-intro-a.pdf

Sitio oficial de Helmet. https://helmetjs.github.io/

Sitio oficial de Chai: http://www.chaijs.com/

Sitio oficial de Mocha. https://github.com/mochajs/mocha

Sitio oficial de Nock: https://github.com/nock/nock

Lo que vimos:

Testeo y deploy.

Lo que viene:

En el próximo módulo comenzamos con ReactJS!.