

# Trabajo Final

*Guía paso a paso*

Fecha de entrega: 30/12/2020

## Contenido

Ejercicio Final – Paso a paso .....	2
Consigna .....	2
Paso 1 - Iniciar un proyecto con npm.....	2
Paso 2 – Estructura de Carpetas .....	3
Paso 3 – Servidor con Express.....	3
Paso 4 – Configurando Handlebars .....	3
Paso 5 – Configurando las rutas.....	4
Paso 6 – Creando las vistas de handlebars .....	5
Paso 7 – Corremos el servidor y testeamos que todo funcione .....	5
Paso 8 – Configuramos la posibilidad de crear partials .....	5
Paso 9 – Creamos un partial para el header .....	6
Paso 10 – Customización.....	6
Paso 11 – Eligiendo un servicio .....	8
Paso 12 – Creando un custom module que fetchee el servicio elegido .....	8
Paso 13 – Ejecutamos nuestro custom module .....	9
Paso 14 – Integrando el servicio .....	9
Paso 15 – Dinamizando la Web.....	10
Paso 16 – Página de Contacto .....	10

## Ejercicio Final – Paso a paso

### Consigna

Vamos a crear la página principal y la página de contacto/formulario de un negocio a elección, creando una web dinámica. La idea es que elijan una temática que tenga un servicio gratuito y expuesto (como Weather Stack).

Vamos a crear nuestra web del lado del servidor y vamos a utilizar html, css y javascript del lado del cliente, así como node, handlebars y npm del lado del servidor.

Los requisitos son:

1. Se deberá crear un servidor con express
2. Se deberán crear las rutas en express
3. Se trabajará con handlebars como motor de template
4. Deberá haber como mínimo dos páginas: contacto e index.
5. El index.hbs tiene que contar **como mínimo** los siguientes partials
  - a. Header
  - b. Footer
6. El index.hbs se tiene que construir de manera dinámica, es decir que
  - a. Si hay muchos **productos**, los mismos deberán venir de un servicio y el hbs se construirá mediante un `{{#each}}`
  - b. Si hay muchos **artistas**, los mismos deberán venir de un servicio y el hbs se construirá mediante un `{{#each}}`
  - c. Si hay muchos **posteos**, los mismos deberán venir de un servicio y el hbs se construirá mediante un `{{#each}}`
7. La página de contacto/formulario deberá contener un formulario que intractúe con el BackEnd y al “submitear” el formulario, la información será recibida en una ruta /formulario, mediante el request.

Será validada tanto en el FrontEnd como en el BackEnd con la librería validator. Para esta instancia, con imprimir los datos en el BackEnd mediante un console.log, es suficiente, aunque suma si le agregan funcionalidad.

En caso de no encontrar un servicio o no poder integrarlo, la página deberá crearse de manera dinámica mediante un array de objetos “hardcodeado” que creen ustedes en el app.js

Si bien es mucha información, este ejercicio será mediante un paso a paso para que no se pierdan en el camino, y que el ejercicio final sirva como repaso de todo lo aprendido.

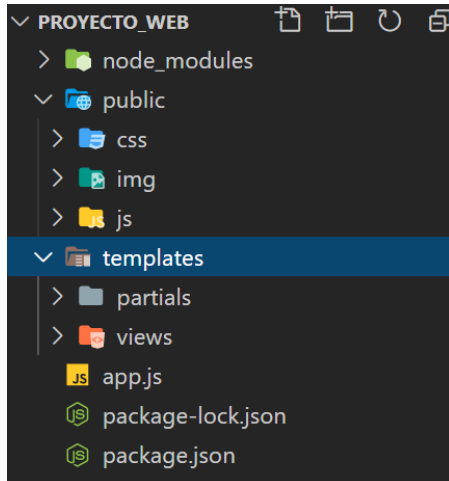
### Paso 1 - Iniciar un proyecto con npm

1. Creamos una carpeta nueva que se llame “proyecto\_web”
2. Creamos un archivo app.js
3. Ejecutamos el comando *npm init*.

## Paso 2 – Estructura de Carpetas

1. Creamos una carpeta llamada “templates”
  - a. Creamos una subcarpeta llamada views
  - b. Creamos una subcarpeta llamada partials
2. Creamos una carpeta llamada “public”
  - a. Creamos las subcarpetas css, js e img

Todo junto:



## Paso 3 – Servidor con Express

1. Instalamos express a nuestro proyecto con el comando *npm i express*
2. Lo requerimos e inicializamos el servidor

```
const express = require('express');
const app = express();
app.listen(3000, () => { console.log('La conexión fue exitosa') });
```

Con estas líneas, ya tenemos nuestro servidor andando.

## Paso 4 – Configurando Handlebars

1. Instalamos hbs ejecutando el comando *npm i hbs*
2. Le decimos a nuestra app, que utilice el view engine ‘hbs’ (handlebars).  
`app.set('view engine', 'hbs');`
3. Para que esto funcione, tenemos que decirle a nuestra app dónde van a estar las views y para eso tenemos que crearnos un path hacia la carpeta de views.
4. Requerimos el core-module “path”  
`const path = require('path');`
5. Creamos el path que vaya hasta las views  
`const viewsPath = path.join(__dirname, 'templates/views');`
6. Seteamos este path a nuestra app  
`app.set('views', viewsPath);`
7. Todo junto:

```
const path = require('path');
const express = require('express');
const app = express();

const viewsPath = path.join(__dirname, 'templates/views');
app.set('view engine', 'hbs');
app.set('views', viewsPath);

app.listen(3000, () => { console.log('La conexión fue exitosa') });
```

## Paso 5 – Configurando las rutas

1. Creamos la ruta para la página principal

```
app.get('/', (req, res) => {
  res.render('index')
});
```

*Recordemos que como estamos trabajando con handlebars, utilizamos el `res.render` y no el `res.send`.*

2. Creamos la ruta para la página de contacto.

```
app.get('/contacto', (req, res) => {
  res.render('contacto');
});
```

Todo junto:

```
const path = require('path');
const express = require('express');
const app = express();

const viewsPath = path.join(__dirname, 'templates/views');
app.set('view engine', 'hbs');
app.set('views', viewsPath);

app.get('/', (req, res) => {
  res.render('index');
});

app.get('/contacto', (req, res) => {
  res.render('contacto');
});

app.listen(3000, () => { console.log('La conexión fue exitosa') });
```

## Paso 6 – Creando las vistas de handlebars

Vamos a crear las vistas, dentro de la carpeta views.

1. Creamos el index.hbs
  - a. Le agregamos la estructura HTML básica
  - b. Le agregamos un h2 que diga “Productos WEB”
2. Creamos el contacto.hbs
  - a. Repetimos el proceso anterior, agregando un h2 que diga “Contacto”.

```
index.hbs X contacto.hbs
templates > views > index.hbs > html
1 <!DOCTYPE html>
2 <html lang="en">
3
4 <head>
5   <meta charset="UTF-8">
6   <meta name="viewport" content="width=device-width, initial-scale=1.0">
7   <title>Document</title>
8 </head>
9
10 <body>
11   <h2>Productos Web</h2>
12 </body>
13
14 </html>
```

## Paso 7 – Corremos el servidor y testeamos que todo funcione

En caso de que algo no funcione, lean los errores, no desesperen, googleen, intenten de nuevo, repitan paso por paso, chequeen sintaxis, y pregunten por slack!

## Paso 8 – Configuramos la posibilidad de crear partials

1. Requerimos hbs

```
const hbs = require('hbs');
```
2. Creamos una ruta para los partials

```
const partialPath = path.join(__dirname, 'templates/partial');
```
3. Utilizamos el método registerPartials de hbs

```
hbs.registerPartials(partialPath);
```

Todo junto:

```
const path = require('path');
const hbs = require('hbs');
const express = require('express');
const app = express();
```

```
const viewsPath = path.join(__dirname, 'templates/views');
const partialPath = path.join(__dirname, 'templates/partials');

app.set('view engine', 'hbs');
app.set('views', viewsPath);
hbs.registerPartials(partialPath);

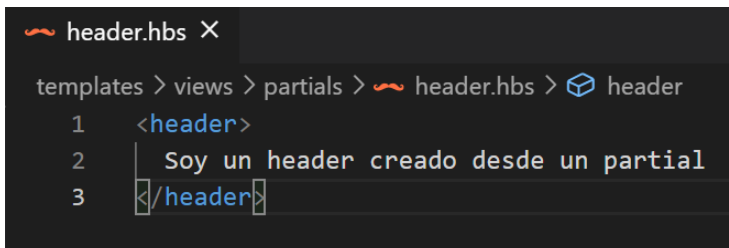
app.get('/', (req, res) => {
  res.render('index');
});

app.get('/contacto', (req, res) => {
  res.render('contacto');
});

app.listen(3000, () => { console.log('La conexión fue exitosa') });
```

## Paso 9 – Creamos un partial para el header

1. Creamos el header.hbs dentro de la subcarpeta partials.



```
header.hbs X
templates > views > partials > header.hbs > header
1 <header>
2   Soy un header creado desde un partial
3 </header>
```

2. Lo integramos al index

```
{{>header}}
```

Corremos y testeamos: Recordar restartear el servidor.

## Paso 10 – Customización

1. Vamos a crear un archivo style.css (dentro de la carpeta css) y un archivo main.js (dentro de la carpeta js). Ambos dentro de la carpeta public.
2. Para que express reconozca que vamos a trabajar con archivos estáticos y poder indicarle dónde está el directorio, tenemos que hacer dos cosas:
  - a. Crear un public path
 

```
const publicPath = path.join(__dirname, 'public');
```
  - b. Registrar a express como un servidor de archivos estáticos
 

```
app.use(express.static(publicPath))
```
3. Vamos a linkear el js y css a nuestras vistas .hbs. Para este paso **vamos a pensar como si que nuestros hbs estuvieran dentro de la carpeta public.** (Esto es lo que pasa en última instancia “por detrás de escena” cuando el hbs se convierte en html)

Todo junto:

```

const path = require('path');
const hbs = require('hbs');
const express = require('express');
const app = express();

const viewsPath = path.join(__dirname, 'templates/views');
const partialPath = path.join(__dirname, 'templates/partials');
const publicPath = path.join(__dirname, 'public');

app.use(express.static(publicPath))
app.set('view engine', 'hbs');
app.set('views', viewsPath);
hbs.registerPartials(partialPath);

app.get('/', (req, res) => {
  res.render('index');
});

app.get('/contacto', (req, res) => {
  res.render('contacto');
});

app.listen(3000, () => { console.log('La conexión fue exitosa') });

```

El linkeo del index.hbs

```

<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <link rel="stylesheet" href="css/style.css">
  <title>Document</title>
</head>

<body>
  {{>header}}
  <h2>Productos Web</h2>
  <script src="js/main.js"></script>
</body>

</html>

```

Queda en ustedes estilar las páginas, crear nuevos partials e integrar bootstrap, agregarle un favicon, cambiar el title, escribir información, etc, etc, etc. El diseño tiene que ser interesante, propio de un e-commerce.

## Paso 11 – Eligiendo un servicio

Para encontrar el servicio que les guste, hay que googlear. Sin embargo, en esta página hay un gran listado de APIS para analizar.

<https://any-api.com/>

Para el ejemplo, voy a utilizar una Fake API:

<https://jsonplaceholder.typicode.com/>

## Paso 12 – Creando un custom module que fetchee el servicio elegido

Para utilizar mi servicio, y que en el app.js solo quede lo relevante a las rutas y configuración del proyecto, vamos a crear un custom module

1. Creamos un archivo js que se llame (en mi caso) postService.js
2. Nos instalamos la librería request (porque sabemos que vamos a tener que hacer un request http) mediante el comando *npm i postman-request*
3. Lo requiero en mi custom module.

```
const request = require('postman-request');
```

4. Me creo una función encargada de fetchear la data

```
const getPosts = () => {};
```

5. Exporto esa función en forma de objeto (esto es opcional, express exporta una función, pero particularmente prefiero tenerlo en un objeto por si en día de mañana escala)

```
module.exports = {  
  getPosts  
}
```

6. Dentro de mi función, utilizo (habiendo leído la documentación) el request para fetchear la data de mi servicio. Me creo una constante url con la ruta del servicio elegido:

```
const url = 'https://jsonplaceholder.typicode.com/posts';
```

```
const getPosts = () => {  
  request({url, json:true}, (error, response) => {  
  
    });  
};
```

7. Hago una validación

Todo junto:

```
const request = require('postman-request');  
const url = 'https://jsonplaceholder.typicode.com/posts';  
  
const getPosts = () => {
```



```

request({ url, json: true }, (error, response) => {
  if (error) {
    console.log('Ocurrió un error', error);
  } else {
    console.log(response.body);
  }
});
});

module.exports = {
  getPosts
}

```

### Paso 13 – Ejecutamos nuestro custom module

Para ver si funciona lo planteado, simplemente corremos en la terminal *node postService.js* habiendo ejecutado la función en algún lado, y vemos qué nos devuelve.

```
getPosts();
```

Si nos devuelve información, vamos por buen camino.

### Paso 14 – Integrando el servicio

Este paso queda a producción de ustedes, pero sin embargo les doy un par de pistas. Probablemente, como están haciendo un request http, tarde. Y si algo tarda, sabemos que entraba el mundo de los callbacks. Lo que van a tener que hacer es:

1. Requerir su custom module en el app.js  

```
const posts = require('./postService');
```
2. Ejecutar la función de su custom module (que trae data de su servicio) y pasarle un callback para cuando esté disponible la información.
3. En el servicio, ejecutar el callback cuando la información esté disponible (ya sea de manera exitosa o con un error)
4. Cuando está el callback disponible, se creará la página de manera dinámica con los resultados.

Todo junto en el app.js

```

app.get('/', (req, res) => {
  posts.getPosts((err, response) => {
    if (err) {
      return res.send(err);
    }
    res.render('index', {
      response,
    });
  });
})

```

```
});
```

Acá en este ejemplo queda la lógica por definir del servicio que están utilizando ustedes. Lo importante es que la respuesta se ejecute en forma de callback y esa respuesta la puedan mandar al momento en que se crea la web.

### Paso 15 – Dinamizando la Web

En su index.hbs probablemente estén recibiendo un array de objetos o un objeto, ahora es cuestión de iterarlo con

```
{{#each people}}  
    {{this}}  
{{/each}}
```

Es muy probable que se encuentren con algunos problemas

1. ¿Cómo recorrer un array de objetos y sus propiedades dentro de handlebars?
  - a. Google, mucho Google
  - b. NUNCA olviden que están en un entorno de JS, pueden alterar la respuesta y guardarla de la forma en que necesiten.

Ejemplito Google: <https://stackoverflow.com/questions/9058774/handlebars-mustache-is-there-a-built-in-way-to-loop-through-the-properties-of>

### Paso 16 – Página de Contacto

Esta queda para ustedes. Con todo el paso a paso de arriba, no es tan complejo. El objetivo de este paso es que:

1. Usen fetch del lado del front
2. Consuman un servicio que crearon ustedes desde el backend
3. Hagan al menos una validación con validator, del lado del Back End.

Si el formulario tiene alguna acción, mejor (algo así como lo que vimos con weatherStack, que podíamos buscar info). Si no tiene ninguna acción, al menos hagan un console log desde el BackEnd así cierra el flujo entero.

Bases de datos y angular, al ser introducciones, no se evaluará.

¡Éxitos!