



Curso de Back-End con Node.js (Inicial)

Clase 07



Temario



Temario

- Promesas.
- Async/Await.
- MySQL y Node.
- Librería `mysql2`:
 - Escapar valores.
 - Consultas con *placeholders*.
 - Funcionalidades útiles.
- Variables de entorno.



Promesas

(Este es un tema de ES6)



Promesas

Las promesas (en inglés: *promises*) fueron creadas para **dar consistencia** y **estandarizar** el manejo de operaciones **asíncronas**.

Hasta ahora, la forma más usual de tener comportamiento **asíncrono** en JavaScript era a través de los famosos ***callbacks***, los cuales funcionan bien pero la forma de usarlos depende mucho de cada implementación. Por ejemplo, cada librería puede implementar *callbacks* de forma diferente y eso implica tener que apoyarse mucho en cada documentación, lo cual no es ideal. Además, era fácil caer en un ***Callback Hell***.

Documentación:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise.



Ejemplos de *callbacks* en jQuery

Si bien en este curso no se está usando la librería jQuery, se la usará como ejemplo en esta diapositiva.

Para los que no saben, jQuery es una librería JavaScript creada en el año 2006 y que fue, y en cierta manera sigue siendo, sumamente popular.

jQuery incluye una función asíncrona llamada *ajax* que permite hacer llamados HTTP. Los creadores de la librería decidieron implementar dicha función de tal forma que reciba como parámetro un objeto con un método llamado `success`, que es una función *callback* que se ejecutará cuando la llamada finalice exitosamente.

Si bien esta sintaxis propuesta por los creadores de jQuery funciona, la misma no sigue ningún estándar. Por ejemplo, en lugar de llamarle `success` al método, le podrían haber puesto `successful` u `ok`. Otras librerías podrían haber elegido otras nomenclaturas y/o sintaxis. Justamente *este tipo de situaciones son las que se quieren evitar con las promesas*.

```
$.ajax({
  url: "https://swapi.dev/api/people/1/?format=json",
  success: function(datosObtenidos) {
    console.log("Nombre del personaje: " + datosObtenidos.name);
  }
});
```

Promesas (cont)

Una promesa es algo que sucederá en algún **momento futuro**.

Es un **objeto** que representa la eventual completitud (o falla) de una operación asíncrona, y el resultado de dicha operación.

El uso de promesas se resume a dos acciones:

1. **Crear** una promesa para exponer una funcionalidad asíncrona. Esto generalmente lo hace el creador de la librería o paquete que quiere exponer la funcionalidad. No suele ser algo que tengamos que hacer nosotros.
2. **Consumir** una promesa ya creada. 🙌 Lo que comúnmente hacemos.



Promesas (Promise): Creación (1/2)

Sintaxis:

```
const promesa = new Promise((resolve, reject) => {  
    // Hacer algo, probablemente asíncrono...  
  
    if (/* Si todo salió bien */) {  
        resolve("¡Funcionó! El resultado es...");  
    } else {  
        reject(Error("Hubo un error"));  
    }  
});
```

⚠ En general, esto no es algo que debemos hacer nosotros. Esto lo suele hacer el creador de la librería o paquete que desea exponer una funcionalidad asíncrona.



Promesas (Promise): Creación (2/2)

Ejemplo:

```
const usuarioPromesa = new Promise((resolve, reject) => {  
  const usuario = obtenerUsuarioPorId(2); // Llamada a BD.  
  if (usuario) {  
    resolve(usuario);  
  } else {  
    reject(Error("Hubo un error. No se pudieron obtener los datos"));  
  }  
});
```

⚠ En general, esto no es algo que debemos hacer nosotros. Esto lo suele hacer el creador de la librería o paquete que desea exponer una funcionalidad asíncrona.



Promesas (Promise): Consumo (1/2)

Ejemplo:

```
usuarioPromesa
  .then(usuario => console.log(usuario))
  .catch(error => console.log(error));
```

A cualquier operación que retorne una promesa al ejecutarse, se le puede concatenar un `.then()` y un `catch()`. Todas las promesas se comportan igual, por lo que la sintaxis no depende del creador de la librería o paquete.

Esto otorga **consistencia** para consumir resultados de operaciones asíncronas.



Promesas (Promise): Consumo (2/2)

Otro ejemplo:

```
fetch("https://swapi.dev/api/people/1/?format=json")  
  .then(respuesta => respuesta.json())  
  .then(personaje => console.log(personaje))  
  .catch(error => console.log(error));
```

👉 Nota: El método `fetch` es algo nuevo que reemplaza a `XMLHttpRequest` (que es la forma en la que tradicionalmente se realizaban las llamadas AJAX). Pero no es un concepto de ES6.

Este método, recibe como argumento una ruta (*path*) y retorna una `Promise`. A su vez, el método `respuesta.json()` retorna otra `Promise`, por lo que fue necesario encadenar otro `then`. Se pueden encadenar tantos `then` como sea necesario, con la gran ventaja de que quedan ordenados uno abajo del otro, en lugar de producir un *Callback Hell* como sucedía antes. El método `catch` se ejecuta en caso de que haya ocurrido algún error en el camino.



Async/Await



Async/Await (1/2)

A la hora de trabajar con **promesas**, en lugar de usar la sintaxis **then/catch** (ES6) se puede usar la sintaxis **async/await** (ES7). Son equivalentes.

Ahora, el ejemplo de la diapositiva anterior (`fetch`), que se escribió con `then/catch`, se puede re-escribir de la siguiente manera con `async/await`.

```
async function obtenerPersonajeDeStarWars() {  
  const respuesta = await fetch("https://swapi.dev/api/people/1/?format=json");  
  const personaje = await respuesta.json();  
  console.log(personaje);  
}  
  
obtenerPersonajeDeStarWars();
```

Tener en cuenta que si se desea "atrapar" ("catchear") un posible error que surja en durante la operación, en el caso de usar `async/await`, habría que agregar un `try/catch`. Ver la siguiente diapositiva.



Async/Await (2/2)

Ejemplo completo incluyendo try/catch:

```
async function obtenerPersonajeDeStarWars() {  
  try {  
    const respuesta = await fetch("https://swapi.dev/api/people/1/?format=json");  
    const personaje = await respuesta.json();  
    console.log("Personaje", personaje);  
  } catch (error) {  
    console.log("Ocurrió un error en el fetch", error);  
  }  
}
```

Como con todo parámetro, el nombre de parámetro `error` es arbitrario. Se le podría haber puesto `e` o `err`.



MySQL y Node



MySQL y Node (1/3)

En la clase anterior hablamos sobre Bases de Datos, particularmente sobre bases de datos **relacionales** o SQL. Y además vimos un motor de base de datos llamado MySQL.

A continuación veremos cómo interactuar con MySQL desde una aplicación Node.

Lo primero que vamos a hacer es instalar un módulo (paquete) llamado `mysql2` (Docs: <https://github.com/sidorares/node-mysql2>).

```
npm install mysql2
```

⚠ No confundir este módulo con uno muy similar llamado `mysql` (sin el "2"), el cual puede ocasionar [este problema](#).



MySQL y Node (2/3)

Luego hay que importar el módulo y crear una **conexión a la BD**:

```
const mysql = require("mysql2");

const connection = mysql.createConnection({
  host: "localhost",
  user: "root",
  password: "root",
  database: "hack_academy_db",
});

connection.connect(function (err) {
  if (err) throw err;
  console.log("¡Nos conectamos a la BD!");
});
```

⚠ Recordar que para poder conectarnos a la BD no sólo debe estar instalado MySQL sino que también debe estar corriendo.

⚠ Si están usando MySQL v8, podrían llegar a tener un problema de autenticación al tratar de conectarse. Para solucionar el problema, leer [este artículo](#).

⚠ Recordar que la base de datos `hack_academy_db` debe estar creada previamente.



MySQL y Node (3/3)

Luego de conectarse a la BD, podrán hacer **consultas** (*queries*):

```
connection.query("SELECT * FROM clients", function (err, clients) {  
  
    if (err) throw err;  
  
    console.log(clients);  
  
});  
  
connection.end();
```

Se pueden hacer llamadas adicionales a la BD usando una misma conexión. Lo importante es acordarse de cerrar la conexión luego usarla.

De hecho, se suele recomendar:

- * Abrir una conexión lo más tarde posible.
- * Cerrar una conexión lo más temprano posible.

Las conexiones suelen ser limitadas y costosas.



mysql2/promise



mysql2/promise

Afortunadamente, el módulo `mysql2` permite usar **promesas** 🎉 en lugar de *callbacks*:

```
const mysql = require("mysql2/promise");  
  
const connection = await mysql.createConnection({  
  host: "localhost",  
  user: "root",  
  password: "root",  
  database: "hack_academy_db",  
});  
  
const [results, fields] = await connection.execute("SELECT * FROM clients");
```

⚠️ Es necesario cambiar la importación del módulo.

fields contiene datos extra sobre los resultados, si los hubiese.



Librería `mysql2`

Escapar valores



Escapar valores (1/3)

⚠ Hay que tener **mucho cuidado** a la hora de hacer consultas a una base de datos usando valores que vinieron, por ejemplo, de un formulario HTML.

Veamos el siguiente ejemplo.

Supongamos que recibimos por una URL el `id` de un usuario, por ejemplo, el número 7, para luego realizar esta consulta:

```
const userId = req.query.id; // Dato que vino de "afuera".  
  
const sql = "SELECT * FROM users WHERE nombre = " + userId;
```

A priori, parece un código inofensivo. La idea es obtener los datos del usuario número 7 y luego mostrarlos.



Escapar valores (2/3)

Supongamos ahora que en lugar de recibir el número 7, se recibe el string “38 OR 1=1”. Es decir:

```
const userId = req.query.id; // userId = "38 OR 1=1"
```

Con este simple cambio, ahora la consulta pasa a ser:

```
"SELECT * FROM users WHERE user_id = 38 OR 1=1"
```

“1=1” siempre es true, por lo cual ahora la consulta retorna el listado completo de usuarios. 😱

Esto es lo que se conoce como una SQL Injection.



Escapar valores (3/3)

Moraleja: nunca se debe confiar en los datos que llegan al Back-End.

Siempre hay que realizar validaciones y en el caso de tener que hacer una consulta SQL utilizando valores que llegaron “desde afuera” es necesario “escaparlos” previamente con el fin de **evitar** Inyecciones SQL.

Para eso, el módulo `mysql2` trae un método llamado `escape`:

```
const userId = req.query.id; // Dato que vino de “afuera”.  
  
const sql = "SELECT * FROM users WHERE user_id = " + connection.escape(userId);
```

Para escapar el nombre de una columna o base de datos, se utiliza el método `escapeId` en lugar de `escape`.



Librería `mysql2`

Consultas con Placeholders



Consultas con Placeholders

Una forma muy cómoda de realizar consultas (*queries*) es usando *placeholders* usando el símbolo de "?". Ejemplo:

```
const { firstname, lastname } = req.query; // Datos que vinieron de "afuera".

const [results, fields] = await connection.execute(
  "INSERT INTO users (firstname, lastname) VALUES(?,?)",
  [firstname, lastname]
);

console.log(results);
```

👉 Una gran ventaja de este método es que los valores se “escapan” automáticamente.



Librería `mysql2`

Funcionalidades útiles



Obtener el `id` de un registro insertado

```
const sqlString = `
  INSERT INTO users (firstname, lastname)
  VALUES ("María", "López")
`;

const [results, fields] = await connection.execute(sqlString);

console.log(results.insertId);
```

Para que esto funcione, la tabla `users` debe tener un campo auto-incremental, generalmente la columna `id`. El valor automático que se generó para dicho campo queda disponible en el atributo `insertId`.



Obtener cantidad de filas afectadas

```
const sqlString = `DELETE FROM users WHERE lastname = "Pérez"`;  
  
const [results, fields] = await connection.execute(sqlString);  
  
console.log(`Se borraron ${results.affectedRows} filas`);
```

`affectedRows` retorna el número de filas afectadas en un INSERT, UPDATE o DELETE.



Obtener cantidad de filas cambiadas

```
const sqlString = `
  UPDATE users
  SET firstname = "Pablo"
  WHERE lastname = "Gómez";
`;

const [results, fields] = await connection.execute(sqlString);

console.log(`Se actualizaron ${results.changedRows} filas`);
```

`changedRows` es distinto de `affectedRows` porque sólo cuenta las filas que efectivamente cambiaron. Por ejemplo, si ya había una fila con el usuario “Pablo Gómez”, esa fila no se cambió.



Variables de Entorno



Variables de Entorno (1/6)

Toda aplicación se ejecuta en cierto **entorno** (*environment*) como, por ejemplo:

- La PC de uno de los desarrolladores del equipo.
- Un servidor de testing.
- Un servidor de producción.
- etc.

Existen **variables** que dependen de dicho **entorno** como, por ejemplo:

- Usuario y contraseña de la base de datos.
- Puerto donde corre la aplicación de Express.
- Puerto donde corre la base de datos.
- etc.

Ejemplo: La contraseña de la base de datos en producción no es la misma que la contraseña que tiene uno de los desarrolladores en su máquina local.

Variables de Entorno (2/6)

A las variables que dependen del entorno, se las denomina “**Variables de Entorno**” y, que dependen del entorno, sus valores no deberían formar parte del código fuente y por lo tanto no deberían guardarse en el repositorio Git.

👉 Los variables de entorno jamás se deben compartir ni quedar públicas.

🤔 Entonces... ¿dónde colocamos las variables de entorno?



Variables de Entorno (3/6)

Node.js tiene un objeto global llamado `process.env` que justamente tiene el propósito de guardar variables de entorno.

¿Pero cómo agregamos datos a dicho objeto sin que queden formando parte del código fuente?

Para esto existe la librería [dotenv](#) 🎉, la cual se instala con el siguiente comando:

```
npm i dotenv
```



Variables de Entorno (4/6)

Luego de instalar la librería **dotenv**, se deberá crear un archivo llamado `.env` en la raíz del proyecto.

En dicho archivo se crearán las variables de entorno. Ejemplo:

Archivo / `.env`

```
APP_PORT=3000
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=hack_database
DB_USERNAME=root
DB_PASSWORD=root
```

Cada entorno tendrá su propio archivo `.env`, el cual nunca deberá incluirse en el repositorio.



Variables de Entorno (5/6)

Para hacer uso de las variables de entorno, definidas en el archivo `.env` deberán hacer lo siguiente.

En la primer línea del el archivo `index.js` (o `server.js`) requerir la librería `dotenv`, la cual se encarga de levantar las variables definidas en el archivo `.env` y colocarlas dentro de `process.env`.

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

Luego, en el lugar de la aplicación donde precisen acceder a una variable de entorno deberán escribir `process.env` seguido del nombre la variable, tal como se definió en el archivo `.env`:

```
process.env.DB_DATABASE
```

Variables de Entorno (6/6)

Importante:

- El archivo `.env` jamás se deberá compartir en el repositorio Git. Para ello se deberá colocar una línea `“.env”` dentro del archivo `.gitignore`.
En caso contrario se podría exponer información confidencial de nuestro sistema (ej: contraseñas).
- Cuando se trabaja en equipo, una buena práctica es compartir (en el repositorio Git) un archivo `.env.example` con valores de ejemplo de forma tal que otros desarrolladores puedan conocer cuáles son las variables que se deben configurar, pero sin exponer datos confidenciales.



Ejercicio 1

Ejercicio 1

En este ejercicio se hará un **CRUD** de usuarios.

Esto implicará hacer:

- C – Create.
- R – Read.
- U – Update.
- D – Delete.

Listado de Usuarios

Id	Nombre	Apellido	Edad	Acciones
1	María	López	27	Eliminar
2	Rodrigo	Fernández	52	Eliminar
3	Ana	Sánchez	34	Eliminar
4	Juan	Fagúndez	44	Eliminar
5	Pedro	Rodríguez	18	Eliminar
6	Sofía	Jiménez	31	Eliminar
7	Pablo	Gutiérrez	17	Eliminar
8	Victoria	Gómez	31	Eliminar
9	José	Gómez	20	Eliminar
10	María Laura	Jiménez	30	Eliminar
11	Juan Pablo	Rodríguez	46	Eliminar
12	Inés	Vázquez	29	Eliminar

[Ir a la página de Inicio](#)

Crear nuevo usuario

Nombre

Apellido

Edad

[Guardar](#)

Modificar usuario

Id

Nombre

Apellido

Edad

[Modificar](#)



Ejercicio 1 (cont)

1. Crear una carpeta llamada `ejercicio_mysql`.
2. Inicializar un proyecto con el comando `npm init -y`. Verificar que se haya creado el archivo `package.json`.
3. Crear un archivo: `index.js`.
4. Instalar los módulos `express`, `mysql2` y `nunjucks`.
5. Crear una base de datos MySQL llamada `ha_ejercicio_mysql`.
6. Crear una tabla llamada `users`.



Ejercicio 1 (cont)

7. Crear las siguientes columnas:

- `id` (BIGINT, auto-incremental, Primary Key).
- `firstname` (VARCHAR, 100, Not Null).
- `lastname` (VARCHAR, 100, Not Null).
- `age` (INT).

8. Insertar en la tabla el contenido de [este gist](#).



Ejercicio 1 (cont)

9. Crear las siguientes rutas (*endpoints*):

- [GET] <http://localhost:3000>.
- [GET] <http://localhost:3000/usuarios>.
- [POST] <http://localhost:3000/usuarios>.
- [GET] <http://localhost:3000/usuarios/eliminar/:id>.
- [POST] <http://localhost:3000/usuarios/modificar>.