Ya sabemos como callback hell dificulta nuestra vida cuando hablamos de manutención y complejidad de código, como en este ejemplo abajo, donde tenemos varias funciones auxiliares una dentro de la otra para ejecutar el movimiento de un personaje:

**moverPersonaje**(‘100’, ‘**Izquierda**’, **function**() {

**moverPersonaje**(‘800’, ‘**Derecha**’, **function**() {

**moverPersonaje**(‘200’, ‘**Izquierda**’, **function**() {

**moverPersonaje**(‘10’, ‘**Derecha**’, **function**() {

**moverPersonaje**(‘60’, ‘**Izquierda**’, **function**() {

})

})

})

})

})

Callback hell termina siendo necesario en esos casos ya que de esa manera garantizamos que la segunda función solamente será disparada una vez que la primera es concluida, y así sigue, ya que en ese escenario de ejemplo estamos trabajando con funciones asíncronas para mover el personaje.

Vimos también que una alternativa para dejar el código más “limpio” es utilizar una **Promise**. Con el regreso de un objeto de promesa, podemos encadenar el .then() garantizando la secuencia de la ejecución.

**moverPersonaje**(‘100’, ‘**Izquierda**’)

.**then**(() => **moverPersonaje**(‘800’, ‘**Derecha**’))

.**then**(() => **moverPersonaje**(‘200’, ‘**Izquierda**’))

.**then**(() => **moverPersonaje**(‘10’, ‘**Derecha**’ ))

.**then**(() => **moverPersonaje**(‘60’, ‘**Izquierda**’ ))

Ese escenario donde hacemos varias requisiciones que son dependientes una de la otra es muy común, y en ese escenario podemos hacer el uso del método .all de la **Promise**. Pasando cada una de las funciones dentro de un arreglo como argumento de la Promise.all podemos ejecutar todas sus funciones en orden sin la necesidad de encadenar varios .then().

Promise.all([

moverPersonaje(‘100’, ‘Izquierda’),

moverPersonaje(‘800’, ‘Derecha’),

moverPersonaje(‘200’, ‘Izquierda’),

moverPersonaje(10, ‘Izquierda’),

moverPersonaje(‘60’, ‘Izquierda’)

])

.then(...)

El Promise.all ejecutará todas las llamadas en el orden y regresará una respuesta que podrá ser utilizada en el .then.

Json-server está listado como dependiente del package.json, que es el archivo donde encontramos todas las bibliotecas externas que estamos usando en nuestro proyecto. Al hacer npm install dentro de la carpeta **admin**, que es la carpeta del proyecto (veremos esta parte en el video) empieza la descarga de json-server automáticamente.

El comando json-server --watch db.json que usamos en la clase necesita ser ejecutado dentro de la carpeta del proyecto de **admin**, caso contrario este comando no será reconocido.

**Otra información importante es que para funcionar localmente, al invés de ejecutar el comando json-server --watch db.json, como lo hacemos en clase, tenemos que ejecutar este comando con npx adelante**:

**npx json-server --watch db.json**

En el caso de que quieras utilizar el comando sin el npx adelante es necesario instalar json-server de manera global:

npm install -g json-serverCOPIA EL CÓDIGO

Después, para subir el servidor, ejecutamos json-server --watch db.json, como lo hacemos en clase.

BROWSER-SYNC (libreria que simula un servidor)

npm install –g browser-sync

correrla: browser-sync

browser-sync start –-server --file . --host --port 5000 --startPath screens/lista\_cliente.html

LIBRERIA UUID (generador id aleatorio)

<https://cdnjs.com/libraries/uuid/8.3.2>