DESARROLLO WEB

REQUISITOS:

- Visual Studio Code
- Git
- Postman
- Saber buscar en google xd

TEMAS:

- Manejo de Git
- Front end Básico
- Back end Avanzado

GIT

Instalar: https://git-scm.com/

Buscar en la documentación:

- Qué es
- Cómo funciona
- Cómo ayuda al control de código
- Concepto de repositorios, pull request, ramas, comandos más frecuentes (git clone, git add, git commit, git push, git pull, git branch)

Teniendo estos conceptos más o menos claros hacer lo siguiente:

1. Crear un repo en https://github.com/ (con el nombre que quieras, algo que identifique es un proyecto de Front)
2. Clonar repo en tu pc y abrirlo en el vs code.
3. Comenzamos a trabajar dentro del repo vacío. En la consola del vs code ejecutar "***npm init -y***" para inicializar el proyecto, se debería crear el archivo package.json
4. Comandos a ejecutar en la consola para subir los cambios de tu repositorio local al repositorio remoto de github:
    a. "***git add .***" -> con esto añadimos los archivos que queremos subir, el punto al final del comando indica que vamos a subir todos los archivos que hayan sufrido cambios. En nuestro caso que solo tenemos creado el package.json también podríamos indicar que solo queremos subir ese archivo, haciendo "***git add package.json***"

b. "***git commit -m "initial commit"*** " -> con esto confirmamos los archivos que acabamos de guardar en el comando anterior. El "**-m**" nos permite añadir un comentario breve de los cambios que se hicieron. (Este comentario aparecerá en el repo de github).
c. "***git push***" -> con esto subimos los cambios al repo remoto. Es probable que cuando pushees por primera vez te tire un error porque falta especificar la rama a la cual queres pushear. En este caso la consola te va a dar el comando adecuado, que normalmente suele ser: "git push --set-upstream origin <nombreRama>"
d. Cómo verificar que los cambios se pushearon? Ir al repo de github y verificar que en los archivos este el package.json

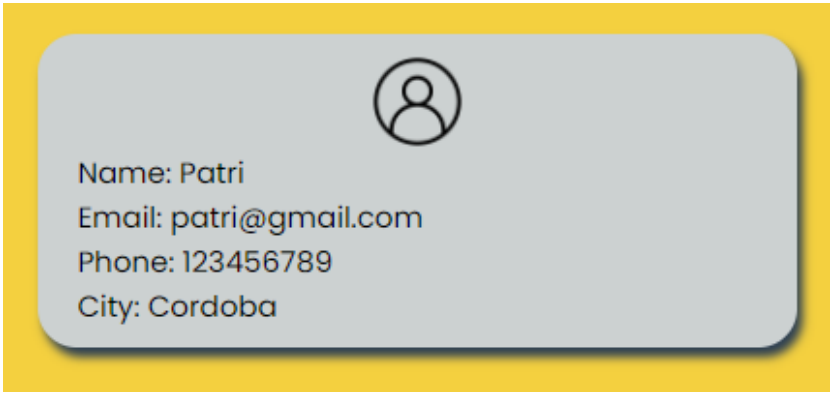
FRONT END

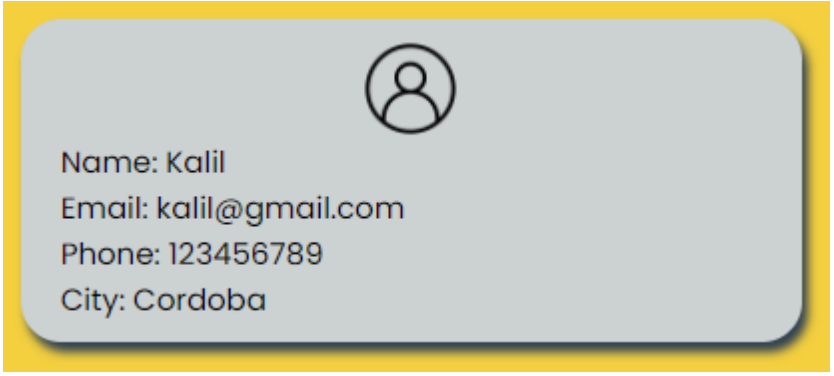
Continuamos trabajando en el repo y vamos a armar una página web básica que muestre información.
Antes de hacer lo siguiente se debe tener un conocimiento básico de html, css y js.
Dentro del repo vamos a crear 3 archivos:

- "**index.html**" donde va a estar la estructura de la pag.
- "**index.css**" donde van a estar los estilos de la pag.
- "**index.js**" donde va a estar la lógica y lo que le da vida a la pag.

La página web va a consistir en mostrar información de varios usuarios. Cada uno tendrá su tarjeta en la cual se mostrará información como: una imagen default, nombre, email, teléfono y ciudad.

Resultado esperado:





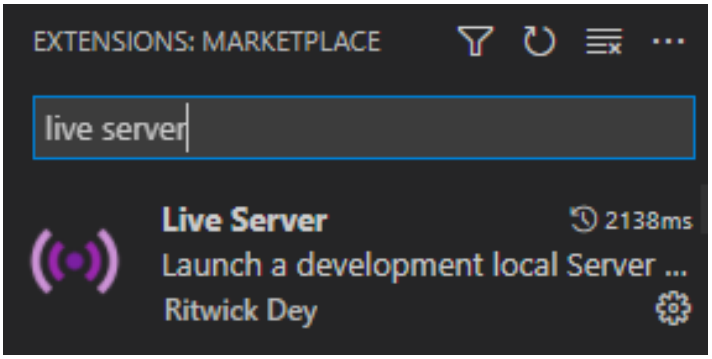
* El estilo de la pag manejalo como quieras (colores, fuente de texto, tamaños). Lo que importa es mostrar el mismo tipo de información de cada usuario en su respectiva tarjeta.

Si te sirve te dejo la dirección de la imagen de usuario que uso en cada tarjeta:

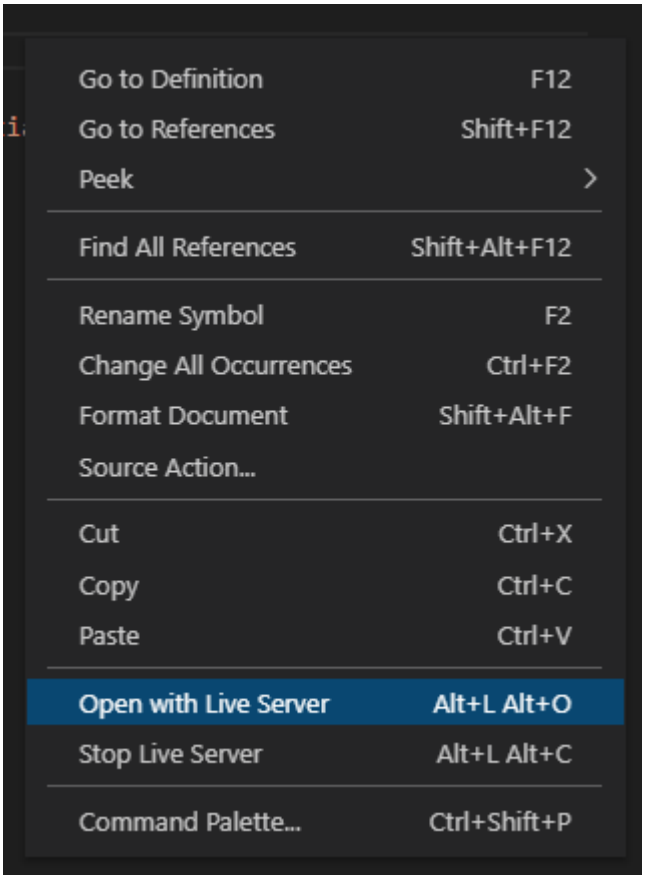
```
https://i.pinimg.com/originals/0c/3b/3a/0c3b3adb1a7530892e55ef36d3be6cb8.png
```

Documentación que te puede servir:
https://developer.mozilla.org/es/docs/Web/API/Document/createElement
https://www.w3schools.com/jsref/prop_html_id.asp
https://developer.mozilla.org/es/docs/Web/API/Element/innerHTML
https://www.w3schools.com/css/css_grid.asp

Te recomiendo además instalar una extensión en el vs code que se llama “Live Server”. Esto hace que cada vez que guardas los cambios en el código, también se impactan en la página web.



Esta extensión la inicializas haciendo click derecho en el código de tu archivo .html y seleccionando la opción “Open with Live Server”. Se te debería abrir en el navegador una nueva pestaña con un puerto local. Y ahí podrías ir viendo en tiempo real cómo impactan los cambios de tu código.



Una vez realizada la interfaz que muestra a todos los usuarios sólo quedaría que subas a github los cambios.

BACK END

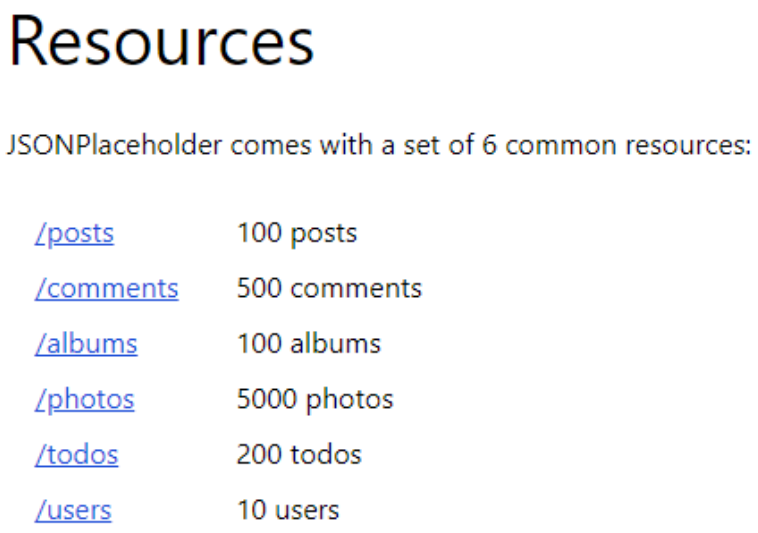
Antes de empezar con el back te recomiendo repasar algunos conceptos:

- API
- JSON
- Promesas en JS
- Request a una API
- Endpoints
- HTTP Methods
- HTTP Status Codes

También para esta parte vas a necesitar tener instalado Postman (software que simula peticiones a una API) y NodeJS (framework). Para esta parte del curso vamos a todas nuestras peticiones/llamadas a la siguiente API:

```
https://jsonplaceholder.typicode.com
```

Esta es una API que tiene varios endpoints y que en c/u nos devuelve diferente información



Nosotros nos vamos a estar enfocando en el endpoint ***/users***. Que devuelve información de 10 diferentes usuarios.
Lo primero que vamos a hacer es realizar en Postman una petición a ese endpoint.

Acá hay una guía de cómo se hacen requests en Postman: https://learning.postman.com/docs/getting-started/sending-the-first-request/

Si todo salió bien Postman nos debería arrojar la data en formato JSON de los 10 usuarios.

Request en JS

Lo próximo que sigue es integrar estos conocimientos en el Front End que armamos en la sección anterior.

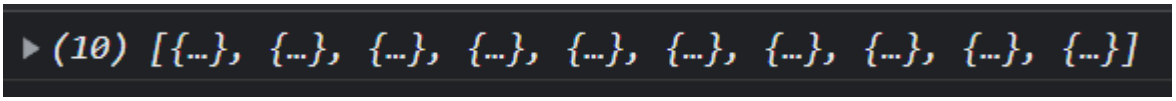
El Front que tenemos hasta ahora es bastante limitado, es estático y poco escalable, ya que mostramos un número fijo de usuarios, y si quisiéramos agregar algún usuario deberíamos agregar otra tarjeta más a nuestro código html. Tal vez el problema no se nota mucho ya que mostramos muy pocos usuarios, pero si en vez de 6, tuviésemos 1000 usuarios para mostrar, esto se hace inviable.

El objetivo ahora es convertirlo en una página dinámica que muestra una cantidad variable de usuarios dependiendo de la información que obtenga.

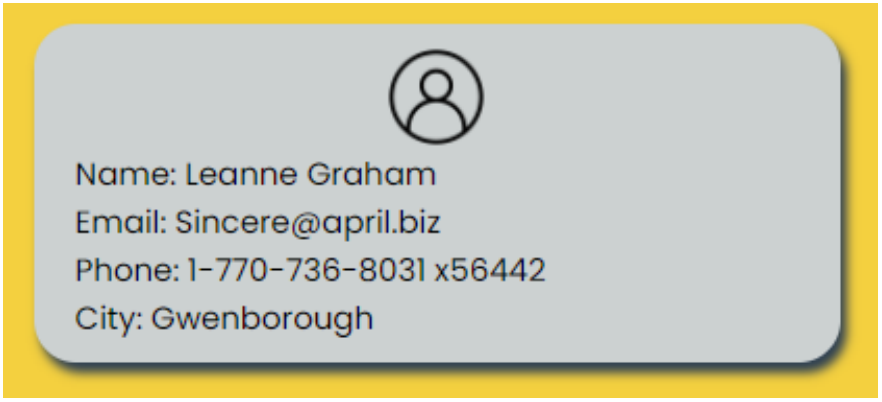
En el código JS vamos a realizar una petición a la API de usuarios. Vamos a obtener un Array de usuarios. Y con esta lista vamos a crear una tarjeta por cada usuario obtenido.

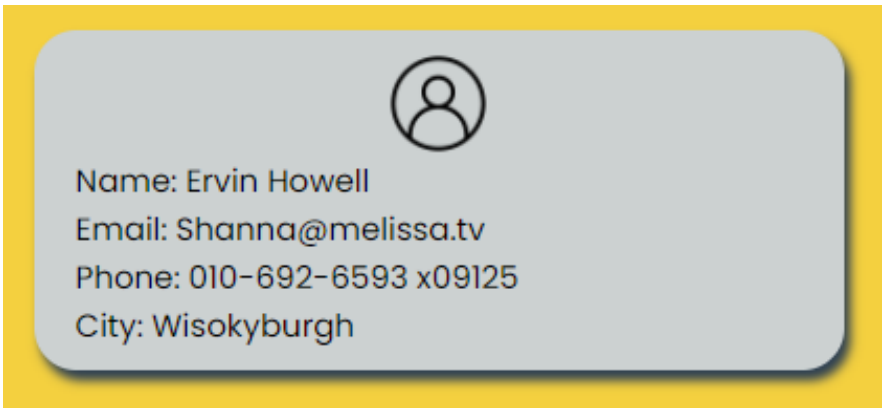
Documentación básica para realizar peticiones en JS: https://developer.mozilla.org/es/docs/Web/API/Fetch_API/Using_Fetch

Resultado esperado de la petición: Array con 10 elementos



Resultado esperado en la página: (ahora ya con los datos traídos de la API)





Construcción de API propia

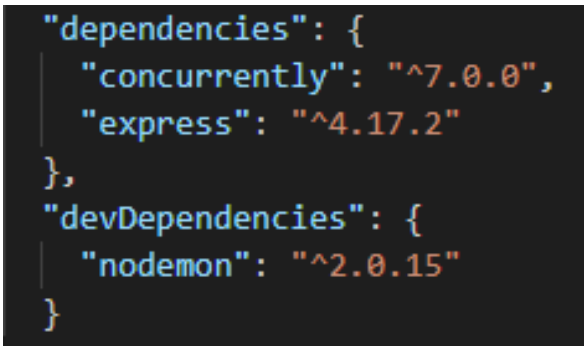
Para construir nuestra propia API vamos a hacer un nuevo repositorio. Repetir los pasos del comienzo (Clonar repo, npm init para inicializar el proyecto, etc).

Vamos a necesitar instalar los siguientes paquetes para el proyecto:

- **express** -> infraestructura que nos permiten construir aplicaciones web en NodeJS. ("***npm i express***" para instalarlo como dependencia)

- **nodemon** -> herramienta que reinicia nuestra aplicación cada vez que se guardan los cambios. ("***npm i nodemon --save-dev***" para instalarlo como herramienta de desarrollo)
- **concurrently** -> nos ayuda en la ejecución de la app con nodemon ("***npm i concurrently***" para instalarlo)

Verificamos que estos paquetes se hayan instalado yendo al package.json y viendo si están dichos paquetes en "dependencies" o "devDependencies" según corresponda.

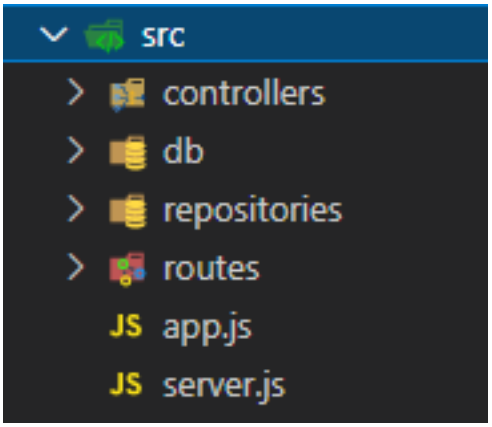


Además vamos a modificar dentro del package.json el script para iniciar la app para que la iniciemos ya con nodemon. Debe quedar de esta forma:

Con todo esto configurado empezamos con la creación del servidor:

1. Creamos la carpeta "**src**". Acá va a estar todo nuestro código. A partir de ahora vamos a trabajar dentro de esta carpeta creada.

La estructura que vamos a manejar es la siguiente:



- **app.js** -> acá vamos a importar la librería “express” la cual nos va a permitir crear una instancia de la nueva App y definimos el puerto en el cual va a correr la misma. También en este archivo se realiza el “Ruteo”, es decir definir las rutas que va a utilizar la App.
- **server.js** -> acá vamos a importar la instancia creada en “app.js” y el puerto definido. Es decir, este archivo es el que hace que la App comience a correr.
- **db/** -> esta carpeta va a simular una base de datos. Acá van a estar los archivos JSON con los cuales vamos a trabajar. Yo te voy a mandar estos archivos.
- **repositories/** -> en esta carpeta vamos a trabajar la lógica de la base de datos. Es decir, vamos a manipular los datos de los archivos JSON a nuestro antojo.
- **controllers/** -> esta carpeta funciona como un pasamanos. Vamos a obtener la información de los repositorios y vamos a construir los mensajes apropiados que va a devolver el servidor para cada caso.
- **routes/** -> en esta carpeta vamos a unir la data de los controllers a un determinado endpoint.

2. Creamos el archivo “**app.js**”, el cual va a contener lo siguiente:

```
const express = require('express');
const app = express();
const PORT = process.env.PORT || 5000;

app.use(express.json());
app.use(express.urlencoded({extended: true}));

app.use('/', require('./routes/cervezas'));

module.exports = {
    app,
    PORT
};
```

3. Creamos el archivo “**server.js**”, el cual va a contener lo siguiente:

```
const { app, PORT } = require('./app');

app.listen(PORT, () => {

    console.log(`Server running at PORT: ${PORT}`);

});
```

4. Vas a situar los archivos JSON que te mando dentro de la carpeta **db/**

5. Dentro de **repositories/** vamos a crear el archivo “**cervezas.js**” (Vamos a trabajar con info de cervezas para el ejemplo). Acá va un ejemplo de cómo puede lucir este archivo repositorio:

```
const cervezas = require("../db/cervezas.json");

const getAll = () => {
    return cervezas;
}

const getById = (id) => {
    const cerveza = cervezas.find(beer =>
        beer.id == id
    );

    return cerveza;
}

module.exports = {
    getAll,
    getById
}
```

Como dijimos antes, en los archivos repositorios se trabaja la lógica de la base de datos. Entonces lo primero que hacemos es obtener la data de cervezas de nuestro archivo JSON. Luego creamos dos funciones:

- getAll() -> esta función va a devolver todas las cervezas encontradas.
- getById(id) -> esta función sirve para buscar una determinada cerveza por su ID. Recibe un **id** como parámetro y va a realizar una búsqueda entre todas las cervezas que tengo y va a devolver la cerveza cuyo ID coincida con el parámetro recibido. Si no tenés muy claro lo que hace el "**.find()**" te dejo esta docu: https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Array/find

6. Dentro de **controllers/** vamos a crear el archivo "**cervezas.js**". Va el ejemplo:

```
const cervezasRepository = require("../repositories/cervezas");

const getAll = (req, res) => {
    try {
        const cervezas = cervezasRepository.getAll();

        return res.status(200).json(cervezas);
    }
    catch(err) {
        return res.status(500).json({
            msg: "Internal Server Error"
        })
    }
}

const getById = (req, res) => {
    try {
        const id = req.params.id;
        const cerveza = cervezasRepository.getById(id);

        if(cerveza === undefined) {
            return res.status(404).json({
                msg: "Beer Not Found"
            })
        }

        return res.status(200).json(cerveza);
    }
    catch(err) {
        return res.status(500).json({
            msg: "Internal Server Error"
        })
    }
}

module.exports = {
    getAll,
    getById,
}
```

Como mencionamos anteriormente los archivos controllers funcionan como un pasamanos donde simplemente se obtiene la info del archivo repositorio y se construyen las respuestas del servidor. Antes de seguir te recomiendo tener en claro cómo funciona la estructura **try-catch**.
Vamos a tener dos funciones:

- getAll() -> en esta función llamamos al getAll() del repositorio para obtener todas las cervezas y construimos la respuesta en caso de éxito. Para esto (y por convención) utilizamos "**res**", el cual va a tener un **200** como "status code" y la respuesta como tal ("**res.json()"**) va a ser la data obtenida del repositorio.
- getById() -> en esta función lo primero que hacemos es obtener el **id** que nos va a llegar como parámetro del request. Luego el procedimiento es el mismo que en la función anterior. Llamamos al getById() del repositorio y le pasamos el **id** que obtuvimos del request. Ahora generamos dos tipos de respuestas. Una para cuando no se encontró ninguna cerveza con ese **id**, la cual tiene como status code un **404** y un mensaje informativo. Y la otra en caso de éxito, devolvemos un **200** y la cerveza encontrada.
  NOTA: en ambas funciones tenemos el bloque **catch** donde manejamos el mismo error para cuando hay un fallo en el servidor, devolviendo un 500 y un mensaje informativo.

7. Dentro de **routes/** vamos a crear el archivo "**cervezas.js**". Va el ejemplo:

```
const { Router } = require('express');
const router = new Router();
const cervezasController = require("../controllers/cervezas");

// GET requests
router.get('/cervezas', cervezasController.getAll);
router.get('/cervezas/:id', cervezasController.getById);

module.exports = router;
```

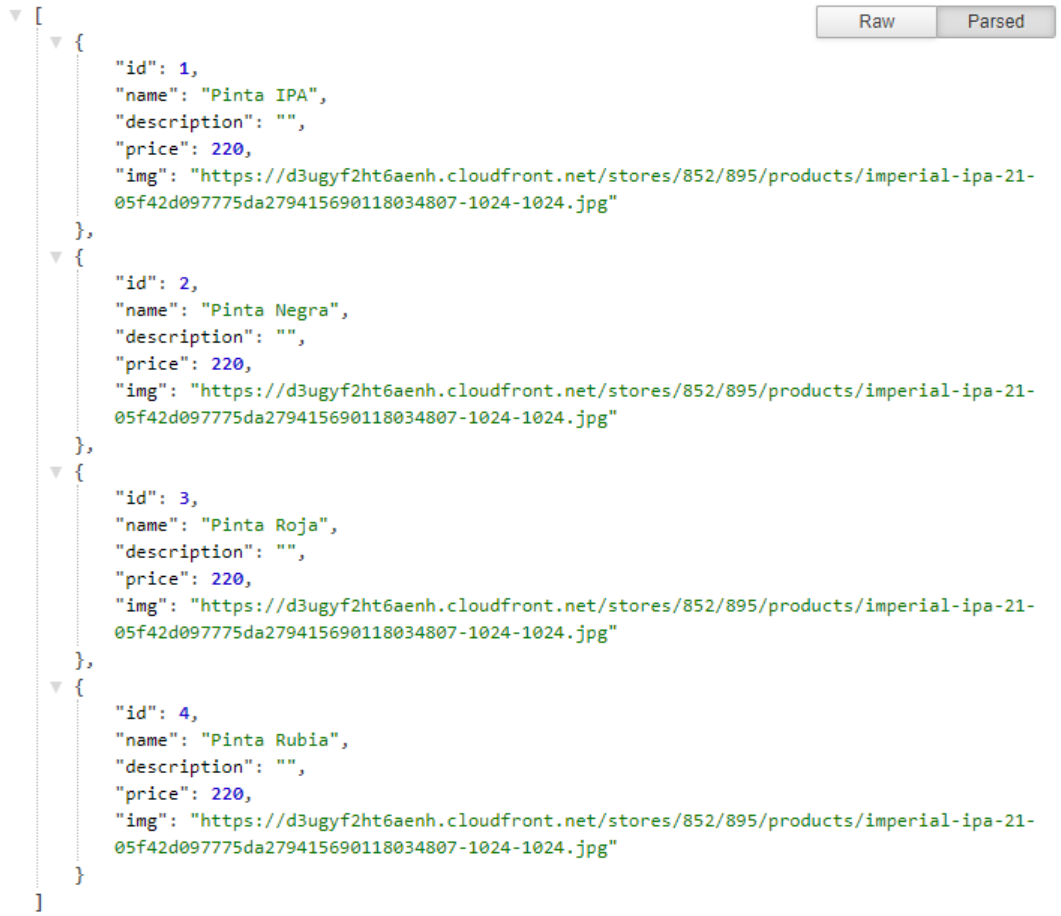
8. Como dijimos, los archivos routes sirven para asociar la data a un endpoint. Lo primero que hacemos es obtener el objeto "Router" que ya viene predefinido en express y luego creamos una instancia del router.
   Después llamamos al controller para poder acceder a sus funciones.
   Finalmente creamos los endpoints. En este caso vamos a usar un GET para acceder a estos endpoints (llamamos a la función .get de la instancia router creada).
   El primer endpoint va a ser "**/cervezas**" y va a devolver toda la info que obtenemos en la función getAll() del controller (es decir, la lista entera de cervezas).
   El segundo endpoint va a ser "**/cervezas/:id**", el símbolo de los dos puntos indica que el id es un parámetro del request, se escribe así para generalizar porque en ese parámetro puede venir cualquier valor. Entonces un request real a este endpoint luciría así: ***URL/cervezas/2*** (así indicamos que queremos obtener la cerveza con id = 2)

9. Nótese que una vez creada la ruta la debemos asociar a la App. Revisar imagen del **punto 2** (línea 8):

```
app.use('/', require('./routes/cervezas'));
```

Con esto decimos que la App va a usar a partir de su raíz (‘/’) el contenido de nuestro archivo “/routes/cervezas”.

10. Para verificar que nuestra Aplicación Web funciona como queremos vamos a hacer **npm start** en la consola. Deberíamos ver en la consola algo como:
Y ahora vamos a verificar que nuestros endpoints estén andando y devolviendo la info adecuada: Vamos a ir a nuestro navegador y vamos a ingresar a:
    - http://localhost:5000/cervezas -> deberíamos obtener la lista de todas las cervezas.



- http://localhost:5000/cervezas/1 -> deberíamos obtener sólo la cerveza con id = 1.

11. Si llegaste acá ya podes decir que construiste tu propia API. Ahora te queda generar nuevos endpoints (así como hicimos para cervezas), pero ahora para: botellas, milanesas, pizzas, tragos. Acordate que vas a tener la lista de cada uno de estos productos en los JSON que te paso.

    NOTA: Para el ejemplo de cervezas hicimos solamente dos endpoints: una búsqueda general y una búsqueda por id. Pero para los demás productos podes trabajar la info como vos quieras, podes filtrar también por Nombre del producto, por un rango de precio, etc.

12. Por último queda subir todos los cambios que hiciste al repo de github y ya estas para aplicar a google pa.

Cualquier duda que tengas durante el todo el proceso contactate conmigo y vemos de armar una meet o algo para revisarlo.