



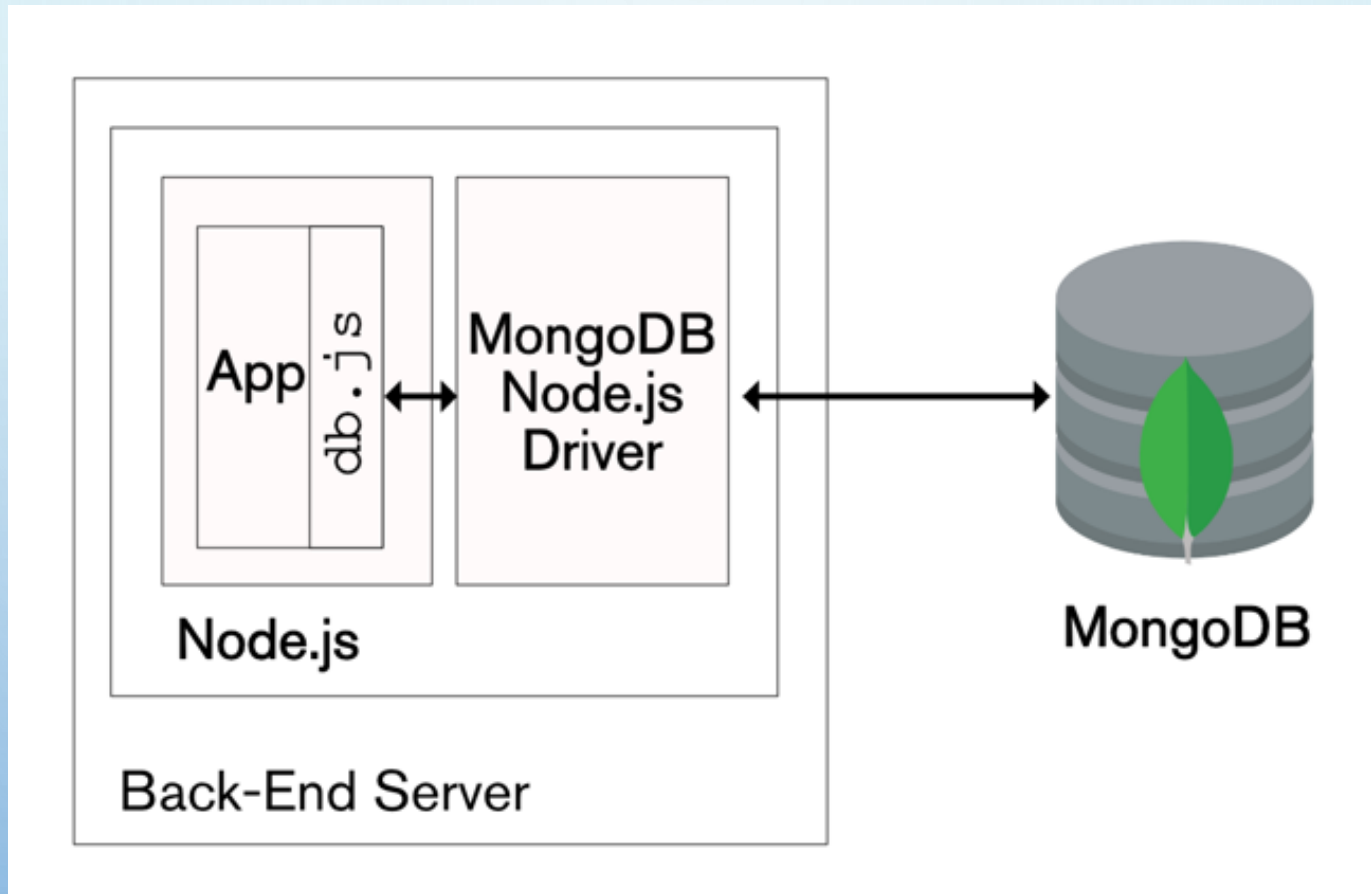
UTN.BA

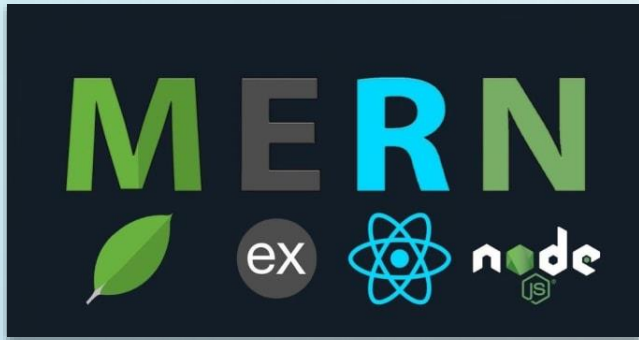
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL BUENOS AIRES

MONGODB Y NODE.JS

- Conectarse a una base de datos MongoDB a través de Node.js.
- Utilizar mongoose para definir esquemas, modelos e interactuar con la base.
- Realizar un CRUD utilizando mongoose

MongoDB con Node.js



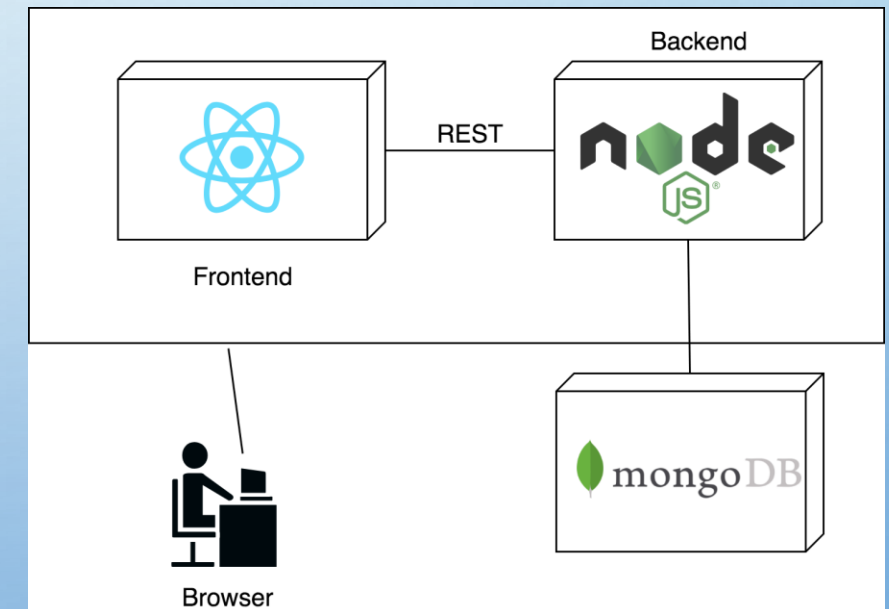
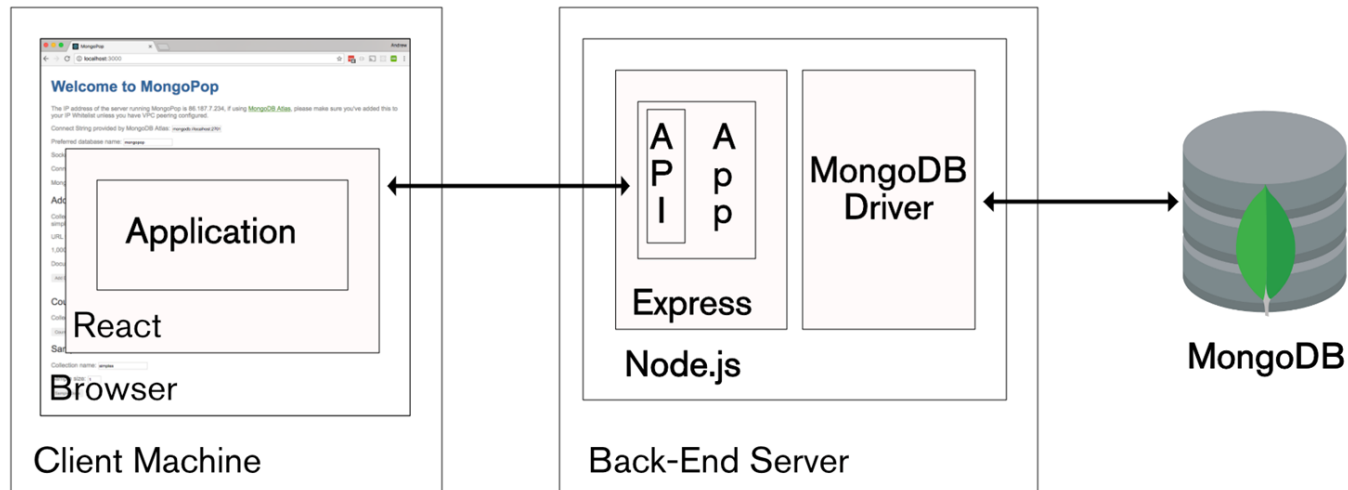
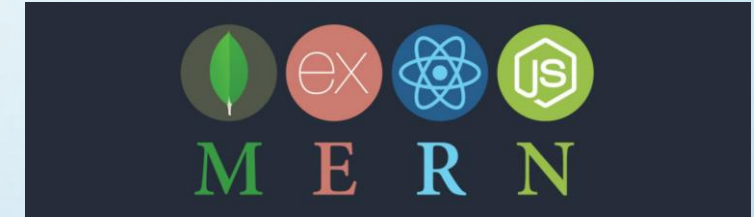
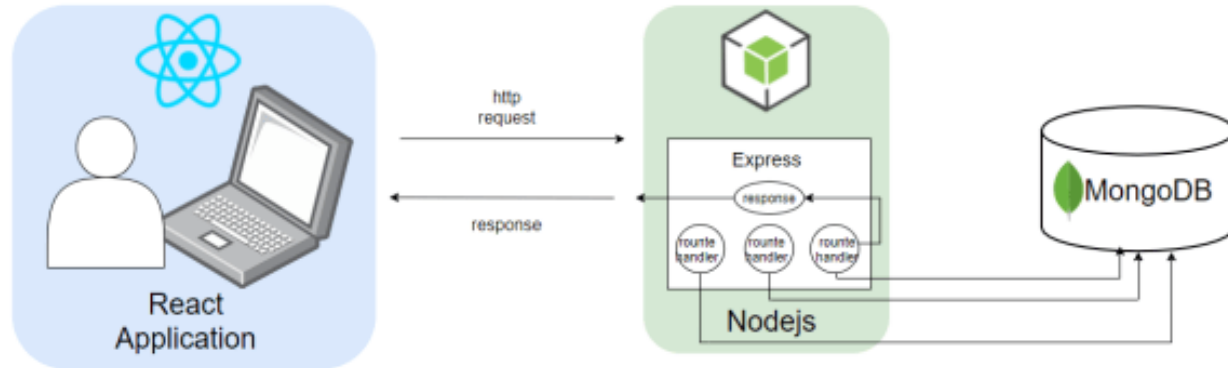


MongoDB en MERN Stack

Recordemos que **MongoDB** constituye una de las herramientas recomendadas de uso en el MERN Stack:

- **MongoDB** es un base de datos NoSQL está orientada a documentos.
- **Express** es una infraestructura de aplicaciones web Node.js
- **React JS** es una biblioteca para crear componentes de interfaz de usuario.
- **Node.js** es un entorno de ejecución para JavaScript que puede permitirle ejecutar JavaScript fuera del navegador, por ejemplo del lado servidor.

MERN Stack Esquemas



Mongoose

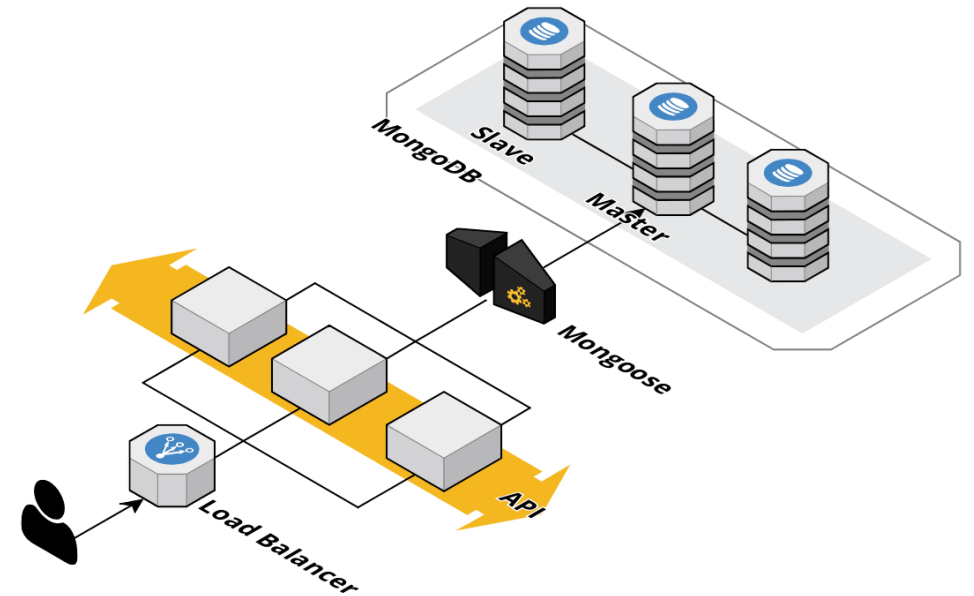
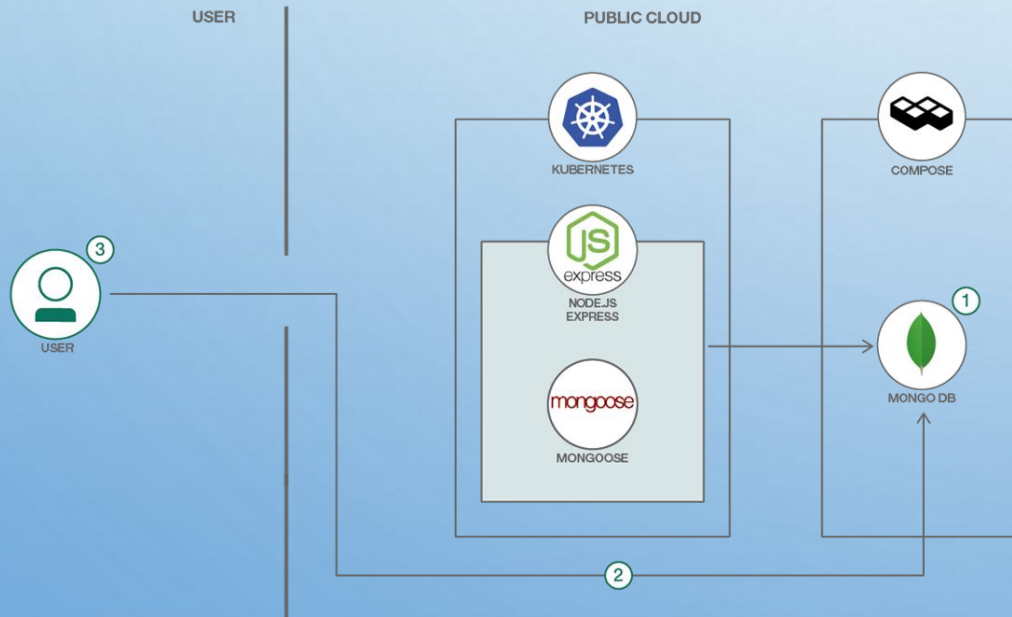
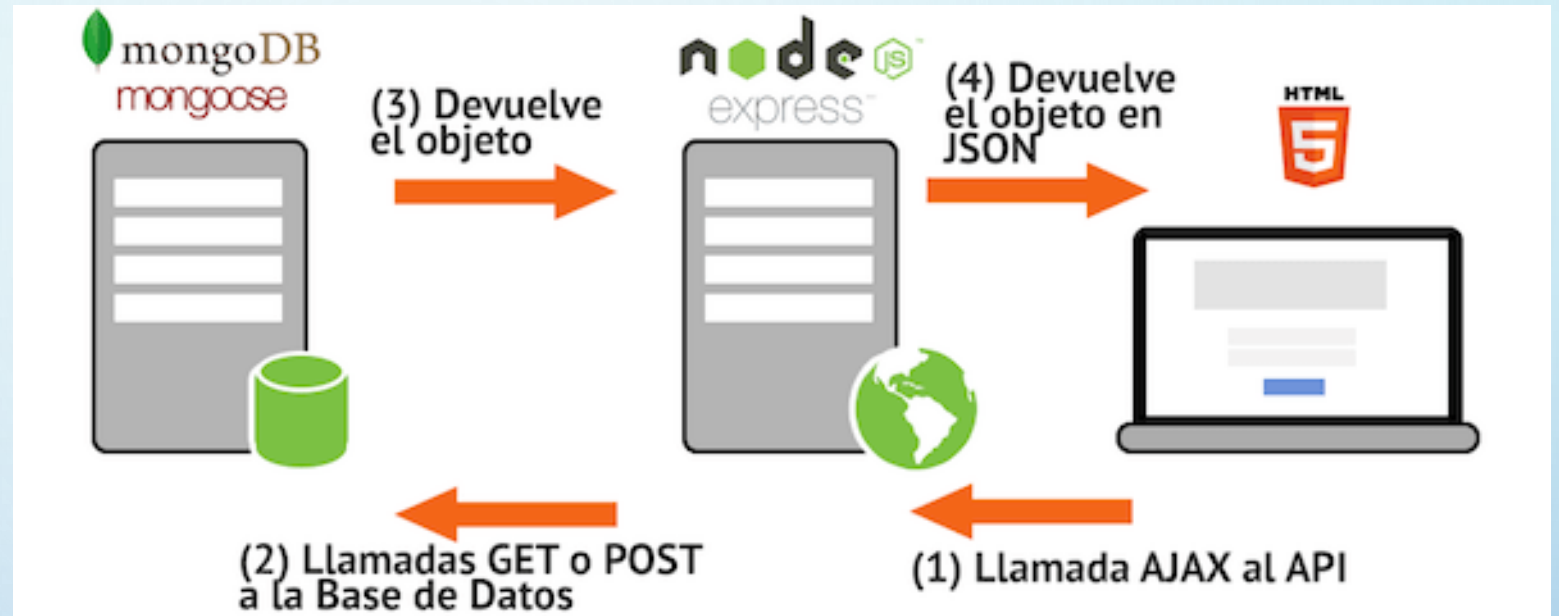
mongoose

elegant **mongodb** object modeling for **node.js**

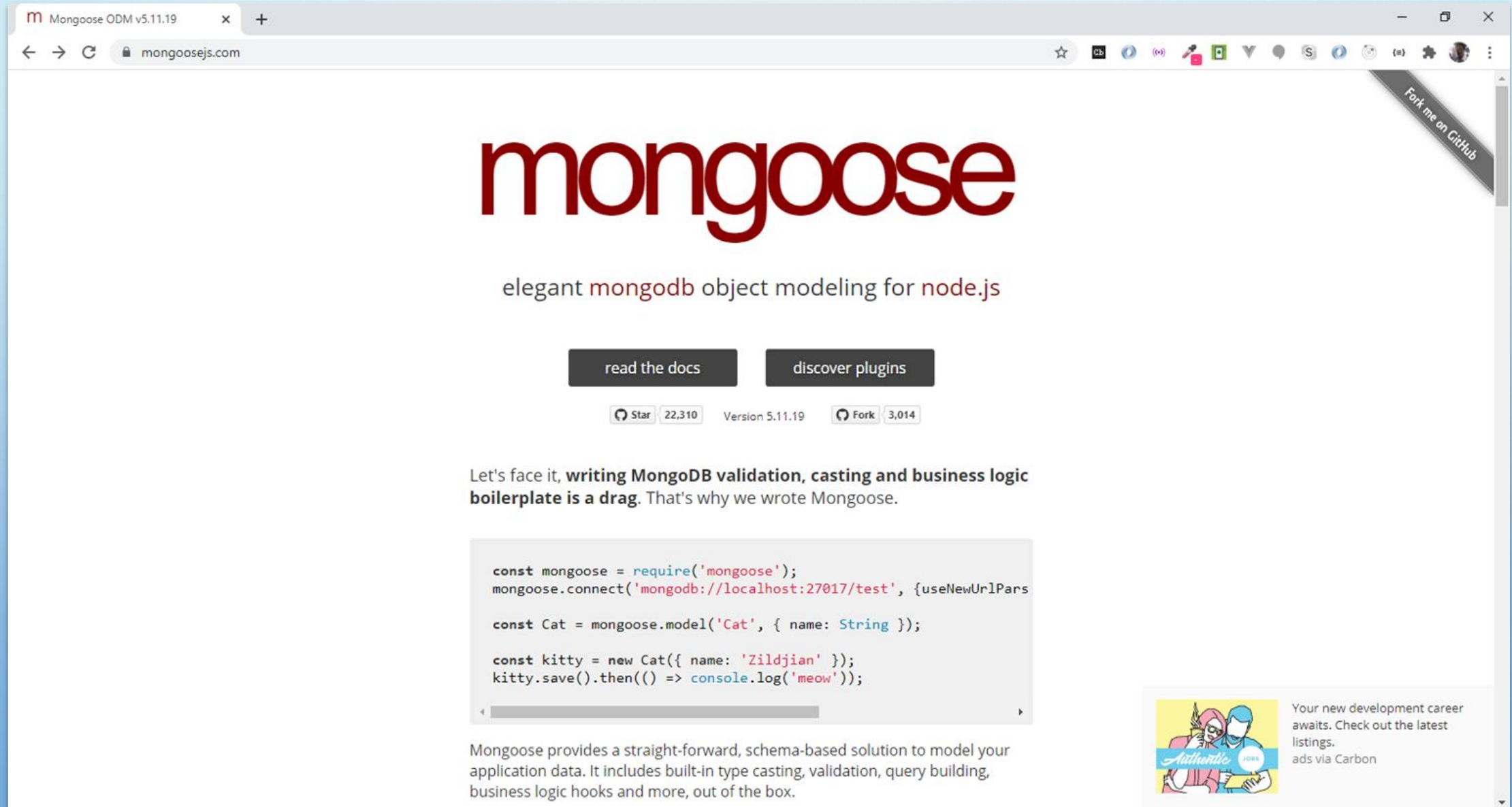
¿Qué es Mongoose?

- Mongoose es una **dependencia Javascript** que realiza la **conexión** a la **instancia** de **MongoDB**
- Pero la magia real del módulo Mongoose es la **habilidad** para **definir** un **esquema del documento**.
- *MongoDB* usa colecciones para almacenar múltiples documentos, los cuales no necesitan tener la misma estructura.
- Cuando tratamos con objetos es necesario que los documentos sean algo parecido. En este punto nos ayudan los esquemas y modelos de **Mongoose**.

Esquemas Mongoose



Website oficial: <https://mongoosejs.com/>



The screenshot shows the official website of Mongoose ODM. The browser's address bar displays 'mongoosejs.com'. The page features the 'mongoose' logo in a large, dark red font. Below the logo, the tagline 'elegant mongodb object modeling for node.js' is centered. Two dark buttons, 'read the docs' and 'discover plugins', are positioned below the tagline. A GitHub repository statistics bar shows 22,310 stars and 3,014 forks for version 5.11.19. A diagonal banner in the top right corner says 'Fork me on GitHub'. The main text block states: 'Let's face it, writing MongoDB validation, casting and business logic boilerplate is a drag. That's why we wrote Mongoose.' Below this is a code block with a light gray background containing JavaScript code for connecting to MongoDB and creating a model. At the bottom, a paragraph describes Mongoose as a schema-based solution for modeling application data. An advertisement for 'Authentic JOBS' is visible in the bottom right corner.

Mongoose ODM v5.11.19

mongoosejs.com

mongoose

elegant **mongodb** object modeling for **node.js**

[read the docs](#) [discover plugins](#)

Star 22,310 Version 5.11.19 Fork 3,014

Let's face it, **writing MongoDB validation, casting and business logic boilerplate is a drag.** That's why we wrote Mongoose.

```
const mongoose = require('mongoose');
mongoose.connect('mongodb://localhost:27017/test', {useNewUrlParser: true});

const Cat = mongoose.model('Cat', { name: String });

const kitty = new Cat({ name: 'Zildjian' });
kitty.save().then(() => console.log('meow'));
```

Mongoose provides a straight-forward, schema-based solution to model your application data. It includes built-in type casting, validation, query building, business logic hooks and more, out of the box.

Authentic JOBS

Your new development career awaits. Check out the latest listings.
ads via Carbon



Mongoose: Schema y Model

- Mongoose usa un **objeto Schema** para definir una lista de **propiedades del documento**, cada una con su propio tipo y características para forzar la estructura del documento.
- Después de especificar un esquema deberemos definir un **Modelo constructor** para así poder crear instancias de los documentos de MongoDB

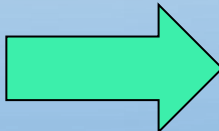
```
const Schema = mongoose.Schema;
```

```
const employeeSchema = new Schema({  
  name : { type : String, required : true, max : [127, "Max Length is 127  
characters"] },  
  age : { type : Number, required : true},  
  salary : Number,  
  designation : { type : String, required : true}  
});
```

```
module.exports = mongoose.model('Employee', employeeSchema);
```



Schema y Model : Validaciones

- Mongoose es un **Object Document Mapper (ODM)**. Esto significa que permite definir objetos con un **esquema fuertemente tipado** que se asigna a un documento MongoDB.
 - Mongoose proporciona una amplia cantidad de funcionalidades para crear y trabajar con esquemas.
 - Actualmente contiene ocho **SchemaTypes** definidos para una propiedad
- 
- *String (Cadena)*
 - *Number (Número)*
 - *Date (Fecha)*
 - *Buffer*
 - *Boolean (Booleano)*
 - *Mixed (Mixto)*
 - *ObjectId*
 - *Array (Matriz)*



Schema y Model : Validaciones

- **Cada tipo de dato permite especificar:**
 - Un valor predeterminado
 - Una función de validación personalizada
 - La indicación de campo requerido
 - Una función get que le permite manipular los datos antes de que se devuelva como un objeto
 - Una función de conjunto que le permite manipular los datos antes de guardarlos en la base de datos
 - Crear índices para permitir que los datos se obtengan más rápido



Schema y Model : Validaciones

Además de estas opciones comunes, ciertos tipos de datos permiten **personalizar** cómo se almacenan y recuperan los datos de la base de datos.

Por ejemplo, un **String** especifica **opciones adicionales**:

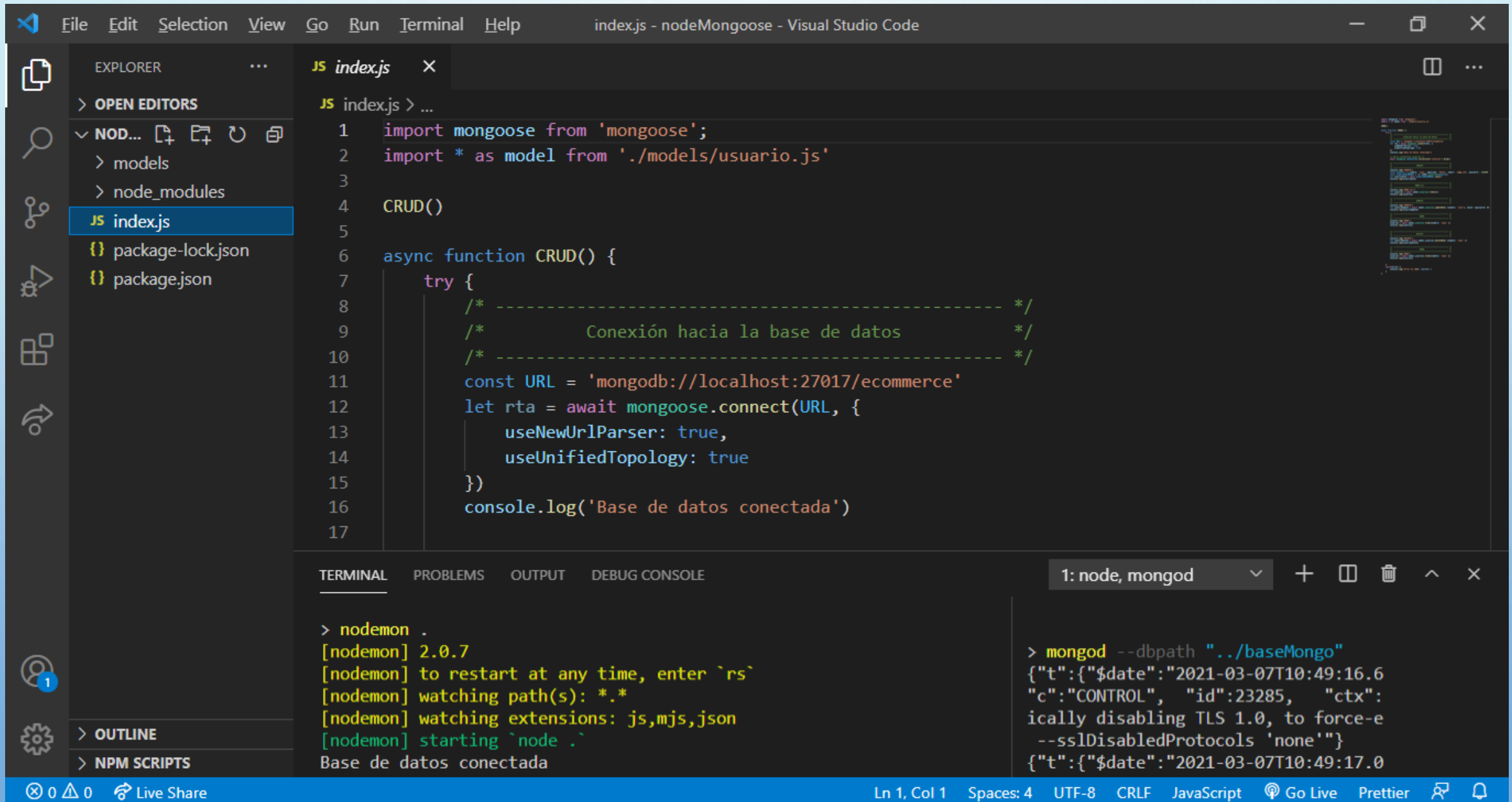
- Convertir en minúsculas y a mayúsculas
- Recortar datos antes de guardar
- Una expresión regular que puede limitar los datos que se pueden guardar durante el proceso de validación
- Una enumeración que puede definir una lista de cadenas que son válidas

Integrando Mongoose en un proyecto Node.js

Configuración del proyecto: pasos a seguir

1. Creamos un proyecto Node.js con **npm init -y**
2. Instalamos la dependencia mongoose con **npm i mongoose**
3. Describimos nuestro modelo de datos (Schema + Model) con las validaciones necesarias.
4. Levantamos el motor de base de datos MongoDB.
5. Creamos la función de conexión mediante mongoose, con las opciones configuradas.
6. Con mongoose realizamos las operaciones CRUD hacia MongoDB: Read, Create, Update y Delete.
7. Mostramos consultas con distintos filtros de Query y con el uso de projection, funciones sort, limit y skip

Mongoose: Conexión hacia la base de datos



Visual Studio Code interface showing a project named 'nodeMongoose' with the file 'index.js' open in the editor.

The Explorer sidebar shows the file structure:

- EXPLORED
- OPEN EDITORS
- NOD... (expanded)
- models
- node_modules
- index.js (selected)
- package-lock.json
- package.json

The Editor shows the code for 'index.js':

```
1 import mongoose from 'mongoose';
2 import * as model from './models/usuario.js'
3
4 CRUD()
5
6 async function CRUD() {
7   try {
8     /* ----- */
9     /*           Conexión hacia la base de datos           */
10    /* ----- */
11    const URL = 'mongodb://localhost:27017/ecommerce'
12    let rta = await mongoose.connect(URL, {
13      useNewUrlParser: true,
14      useUnifiedTopology: true
15    })
16    console.log('Base de datos conectada')
17  }
```

The Terminal shows the command 'nodemon .' being executed, and the output indicates that the database is connected:

```
> nodemon .
[nodemon] 2.0.7
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node .`
Base de datos conectada
```

The Output window shows the command 'mongod --dbpath ../baseMongo' being executed, and the output shows the MongoDB server starting:

```
> mongod --dbpath ../baseMongo
{"t":{"$date":"2021-03-07T10:49:16.6
"c":"CONTROL", "id":23285, "ctx":
ically disabling TLS 1.0, to force-e
--sslDisabledProtocols 'none'"}
{"t":{"$date":"2021-03-07T10:49:17.0
```

The status bar at the bottom shows: Ln 1, Col 1 Spaces: 4 UTF-8 CRLF JavaScript Go Live Prettier

Mongoose: Modelo de datos

The image shows a Visual Studio Code editor window with the file `usuario.js` open. The Explorer sidebar on the left shows the project structure, including `models` and `usuario.js`. The main editor displays the following JavaScript code:

```
1 import mongoose from 'mongoose';
2
3 const usuariosCollection = 'usuarios';
4
5 const UsuarioSchema = new mongoose.Schema({
6   nombre: {type: String, require: true, max: 100},
7   apellido: {type: String, require: true, max: 100},
8   email: {type: String, require: true, max: 100},
9   usuario: {type: String, require: true, max: 100},
10  password: {type: Number, require: true}
11 })
12
13 export const usuarios = mongoose.model(usuariosCollection, UsuarioSchema);
14
```

The bottom of the screen features a terminal panel with the following output:

```
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node .`
Base de datos conectada
```

On the right side of the terminal, a MongoDB command is shown:

```
> mongod --dbpath "../baseMongo"
{"t":{"$date":"2021-03-07T10:49:16.6"}}
```

The status bar at the bottom indicates the current file is `Ln 1, Col 1`, with 4 spaces, UTF-8 encoding, CRLF line endings, and JavaScript language mode. It also shows the Prettier formatter is active.

Mongoose: CREATE / READ ALL

The screenshot displays the Visual Studio Code interface with a project named 'nodeMongoose'. The Explorer sidebar on the left shows the file structure, including 'index.js', 'package-lock.json', and 'package.json'. The main editor window shows the content of 'index.js', which contains two functions: 'CREATE' and 'READ all'. The 'CREATE' function logs the operation, creates a user object, and saves it to the database. The 'READ all' function logs the operation and retrieves all users from the database. The Terminal at the bottom shows the output of the 'CREATE' function, displaying the user object. The Output window on the right shows the command 'mongod --dbpath ../baseMongo' and its output, indicating that MongoDB is running.

```
index.js - nodeMongoose - Visual Studio Code
```

EXPLORER

- > OPEN EDITORS
- > NOD...
 - > models
 - > node_modules
 - JS index.js
 - { } package-lock.json
 - { } package.json
- > OUTLINE
- > NPM SCRIPTS

JS index.js

```
20
21
22  /* ----- */
23  /*             CREATE             */
24  /* ----- */
25  console.log('CREATE')
26  const usuario = { nombre: 'Juan', apellido: 'Perez', email: 'jp@g.com', password : 123456 }
27  const usuarioSaveModel = new model.usuarios(usuario);
28  let usuarioSave = await usuarioSaveModel.save()
29  console.log(usuarioSave)
30
31  /* ----- */
32  /*             READ all             */
33  /* ----- */
34  console.log('READ all')
35  let usuarios = await model.usuarios.find({})
36  console.log(usuarios)
37
38  /* ----- */
```

TERMINAL

```
CREATE
{
  _id: 6044d9bec530193c6034529c,
  nombre: 'Juan',
  apellido: 'Perez',
  email: 'jp@g.com',
  password: 123456,
}
```

1: node, mongod

```
> mongod --dbpath ../baseMongo
{"t":{"$date":"2021-03-07T10:49:16.6
"c":"CONTROL", "id":23285, "ctx":
ically disabling TLS 1.0, to force-e
--sslDisabledProtocols 'none'}}
{"t":{"$date":"2021-03-07T10:49:17.0
```

Ln 1, Col 1 Spaces: 4 UTF-8 CRLF JavaScript Go Live Prettier

Mongoose: DELETE/ READ

The image shows a Visual Studio Code editor window with the file `index.js` open. The file contains a Mongoose CRUD operation for deleting and reading a user. The code is as follows:

```
52  /* ----- */
53  /*           DELETE           */
54  /* ----- */
55  console.log('DELETE')
56  let usuarioDelete = await model.usuarios.deleteOne( {nombre: 'Juan' })
57  console.log(usuarioDelete)
58
59  /* ----- */
60  /*           READ           */
61  /* ----- */
62  console.log('READ')
63  usuarios = await model.usuarios.find({nombre: 'Juan' })
64  console.log(usuarios)
65
66
67  }
68  catch(error) {
69    console.log(`Error en CRUD: ${error}`)
70  }
```

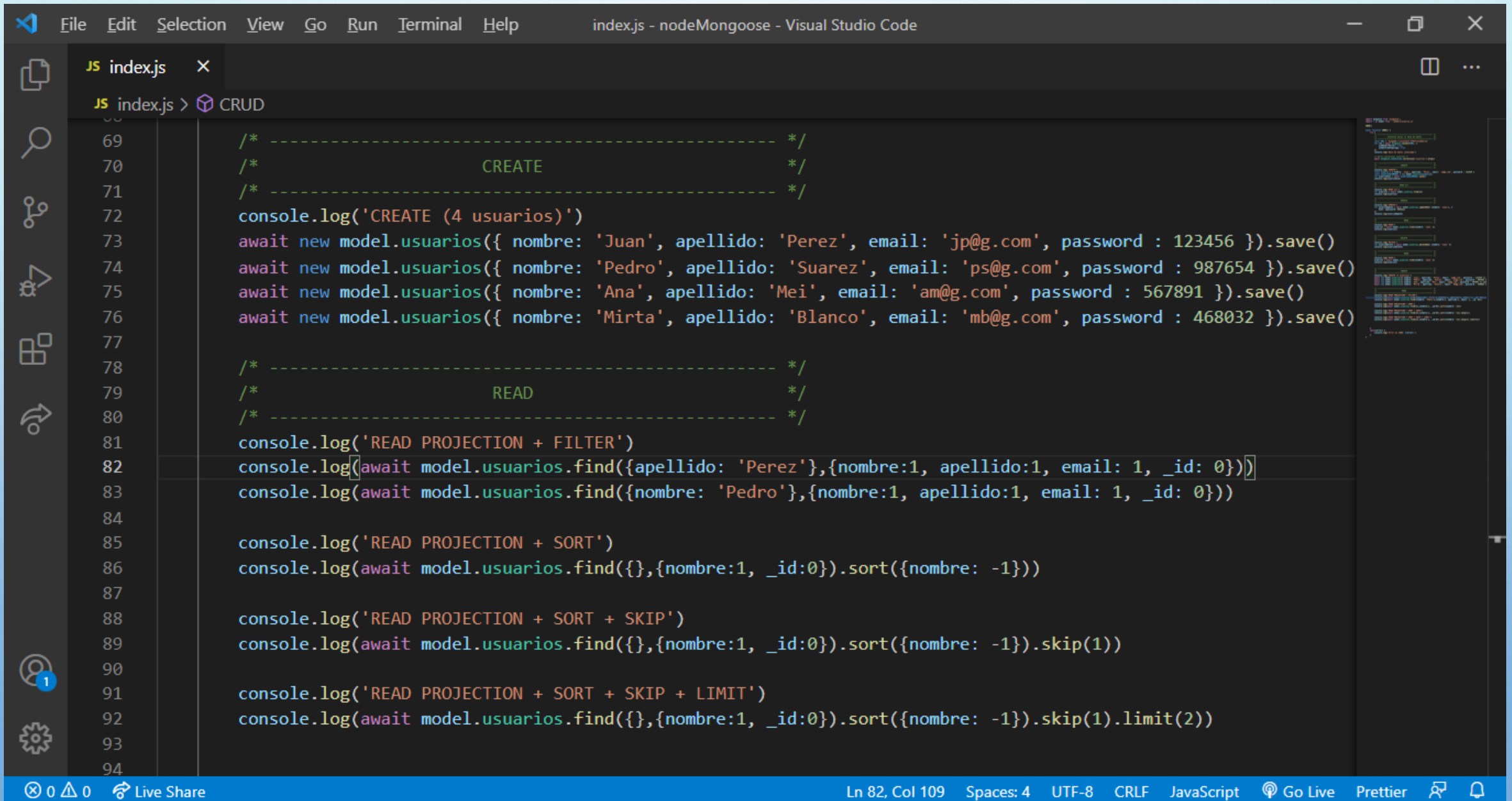
The terminal output shows the results of the operations:

```
DELETE
{ n: 1, ok: 1, deletedCount: 1 }
READ
[]
```

The MongoDB command being executed is:

```
> mongod --dbpath "../baseMongo"
{"t":{"$date":"2021-03-07T10:49:16.6
"c":"CONTROL", "id":23285, "ctx":
ically disabling TLS 1.0, to force-e
--sslDisabledProtocols 'none'"}
}
```

Mongoose: READ PROJECTION + SORT + SKIP + LIMIT



```
index.js - nodeMongoose - Visual Studio Code

JS index.js x
JS index.js > CRUD

69  /* ----- */
70  /*             CREATE             */
71  /* ----- */
72  console.log('CREATE (4 usuarios)')
73  await new model.usuarios({ nombre: 'Juan', apellido: 'Perez', email: 'jp@g.com', password: 123456 }).save()
74  await new model.usuarios({ nombre: 'Pedro', apellido: 'Suarez', email: 'ps@g.com', password: 987654 }).save()
75  await new model.usuarios({ nombre: 'Ana', apellido: 'Mei', email: 'am@g.com', password: 567891 }).save()
76  await new model.usuarios({ nombre: 'Mirta', apellido: 'Blanco', email: 'mb@g.com', password: 468032 }).save()
77
78  /* ----- */
79  /*             READ             */
80  /* ----- */
81  console.log('READ PROJECTION + FILTER')
82  console.log(await model.usuarios.find({apellido: 'Perez'}, {nombre:1, apellido:1, email: 1, _id: 0}))
83  console.log(await model.usuarios.find({nombre: 'Pedro'}, {nombre:1, apellido:1, email: 1, _id: 0}))
84
85  console.log('READ PROJECTION + SORT')
86  console.log(await model.usuarios.find({}, {nombre:1, _id:0}).sort({nombre: -1}))
87
88  console.log('READ PROJECTION + SORT + SKIP')
89  console.log(await model.usuarios.find({}, {nombre:1, _id:0}).sort({nombre: -1}).skip(1))
90
91  console.log('READ PROJECTION + SORT + SKIP + LIMIT')
92  console.log(await model.usuarios.find({}, {nombre:1, _id:0}).sort({nombre: -1}).skip(1).limit(2))
93
94
```

Ln 82, Col 109 Spaces: 4 UTF-8 CRLF JavaScript Go Live Prettier

READ PROJECTION + SORT + SKIP + LIMIT : Salida a consola

Cmder

DELETE

```
{ n: 1, ok: 1, deletedCount: 1 }
```

READ

```
[]
```

CREATE (4 usuarios)

READ PROJECTION + FILTER

```
[ { nombre: 'Juan', apellido: 'Perez', email: 'jp@g.com' } ]
```

```
[ { nombre: 'Pedro', apellido: 'Suarez', email: 'ps@g.com' } ]
```

READ PROJECTION + SORT

```
[
```

```
  { nombre: 'Pedro' },
```

```
  { nombre: 'Mirta' },
```

```
  { nombre: 'Juan' },
```

```
  { nombre: 'Ana' }]
```

READ PROJECTION + SORT + SKIP

```
[ { nombre: 'Mirta' }, { nombre: 'Juan' }, { nombre: 'Ana' } ]
```

READ PROJECTION + SORT + SKIP + LIMIT

```
[ { nombre: 'Mirta' }, { nombre: 'Juan' } ]
```

node.exe

Search

Typescript





- TypeScript es un **superset** de Javascript. Es decir, un lenguaje que está **construido encima** del propio **Javascript**
- Agrega nuevas características al lenguaje Javascript como tipado estático y objetos basados en clases, ofreciendo nuevas herramientas para los programadores.
- TypeScript se puede usar en sustitución de Javascript, aunque para ello es necesario el paso adicional de la **transpilación** del código: una operación por la cual el código TypeScript es compilado/traducido a código Javascript estándar.

Tipado estático

Tipado estático

YO SOY *STATIC TYPED*, SI UNA VARIABLE ES DE UN TIPO DE DATO, NO SE LE PUEDE CAMBIAR EL TIPO Y PUNTO.



YO SOY *DYNAMIC TYPED*, SI UNA VARIABLE ES DE UN TIPO DE DATO, SI TE ARREPENTÍS PODÉS CAMBIARLE EL TIPO, ¡TODO BIEN!



Tipado estático

- *Tipado*: es una propiedad de las variables que indica **qué valores se pueden guardar** en ellas y **qué operaciones se pueden ejecutar**.
- A diferencia de Javascript, que es *Dynamic Type*, Typescript es exactamente lo opuesto, es *Static Type*
- El tipo de las variables se va a **chequear en tiempo de compilación** (antes de que se ejecute tu código)
- El tipado en Typescript **es opcional**.

Ejemplo Tipado estático

La característica más importante de TypeScript es la de agregar tipado estático a las variables.

```
//boolean
let isDone: boolean = false;

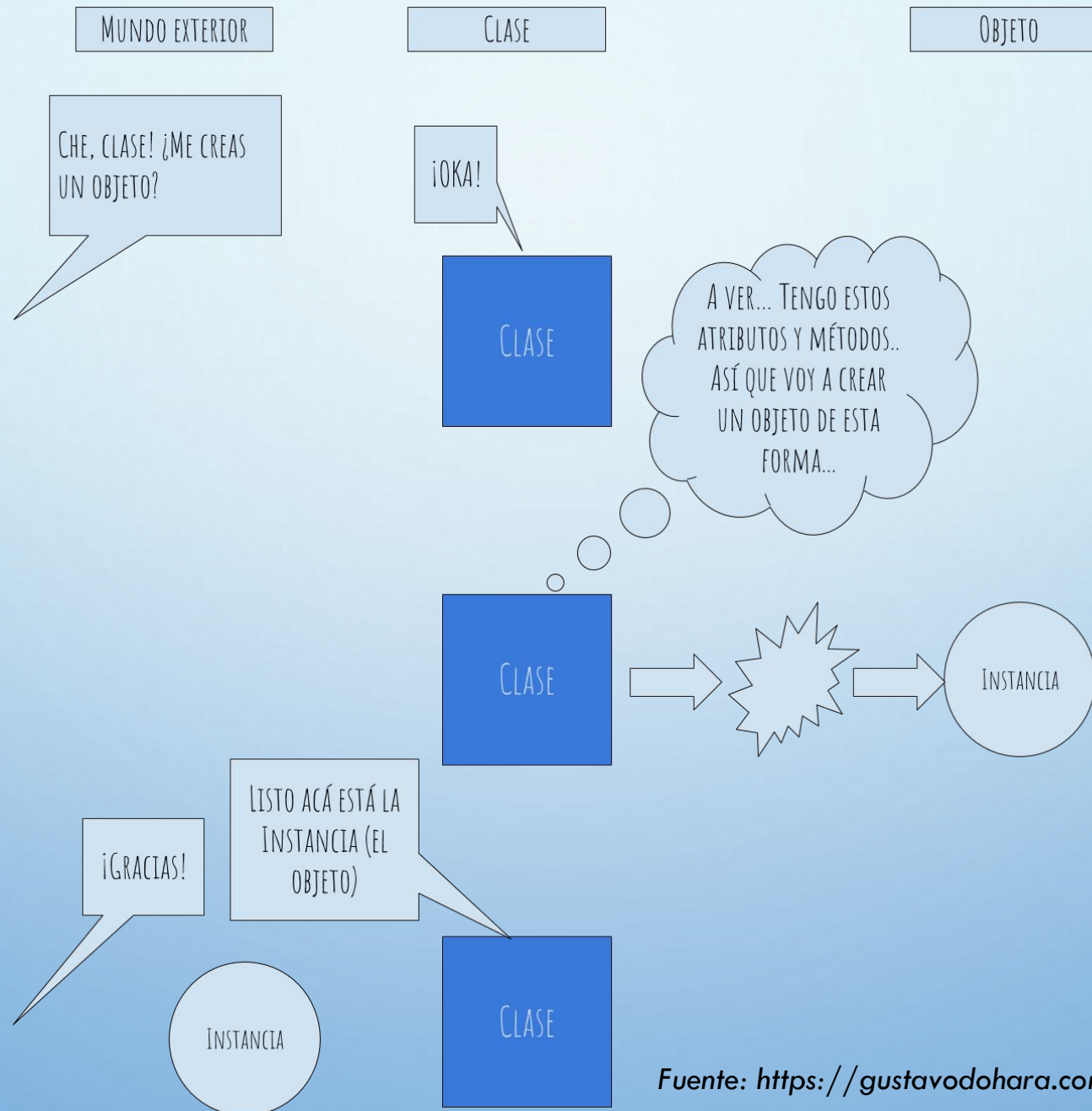
//number
let decimal: number = 6;
let hex: number = 0xf00d;
let binary: number = 0b1010;
let octal: number = 0o744;

//string
let color: string = "blue";
color = "red";

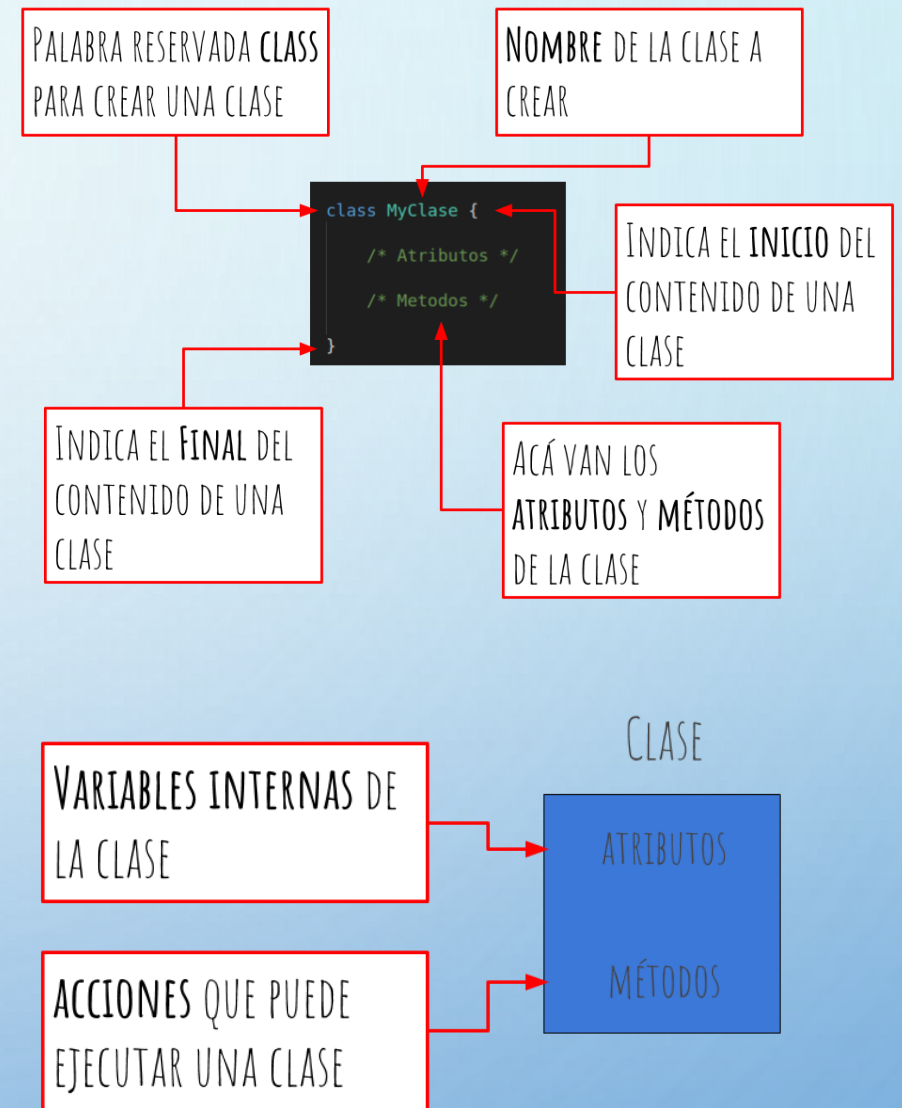
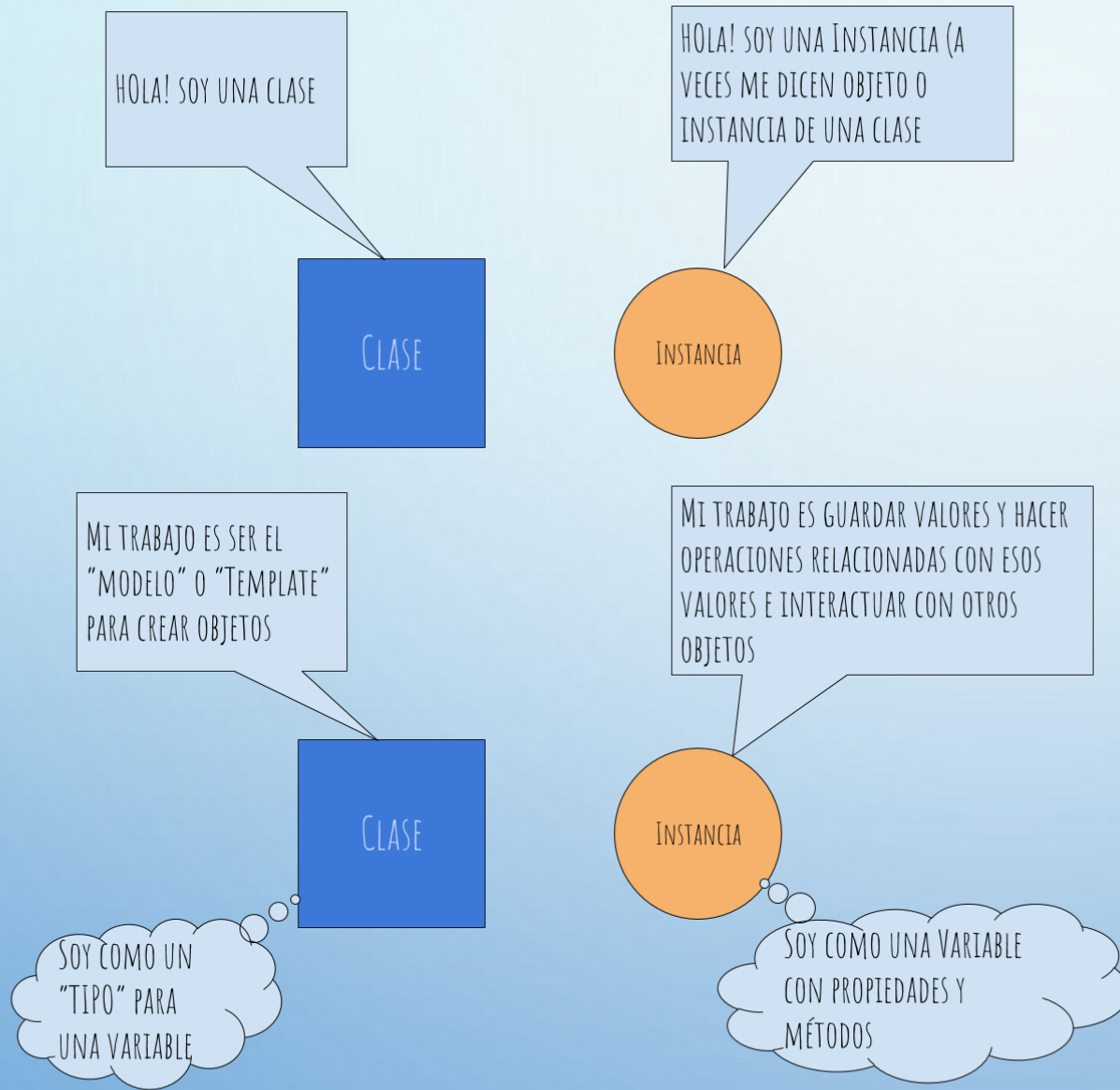
//array
let list1: number[] = [1, 2, 3];
let list2: Array<number> = [4, 5, 6];
```

Esto ofrece
diversas ventajas
en tiempo de
desarrollo: el
compilador detecta
cualquier problema
que pueda tener su
código, antes de
ejecutarse.

Objetos basados en clases



Fuente: <https://gustavodohara.com/>



Objetos basados en clases

- JavaScript tradicional utiliza funciones y herencia basada en prototipos para construir componentes reutilizables
- A partir de ECMAScript 2015 (ES6), podemos construir aplicaciones con filosofía orientada a objetos.
- En TypeScript, la **orientación a objetos** tiene más funciones integradas, **soportando clases abstractas, modificadores de acceso** e interfaces entre otras características.

Objetos basados en clases

Características

- Herencia
- Modificadores Public, private, and protected (public por defecto)
- Modificador Readonly.
- Descriptores de acceso (Accessors)
- Propiedades static
- Clases abstractas
- Interfaces

Ejemplo Objetos basados en clases

En este ejemplo vemos algunas de las características de las clases de Typescript, como ser herencia y modificadores de acceso. La sintaxis es similar a las clases de C # o Java.

```
class Person {
  protected name: string;
  protected constructor(theName: string) { this.name = theName;}
}

// Employee can extend Person
class Employee extends Person {
  private department: string;

  constructor(name: string, department: string) {
    super(name);
    this.department = department;
  }

  public getElevatorPitch() {
    return `Hello, my name is ${this.name} and I work in ${this.department}.`;
  }
}

let howard = new Employee("Howard", "Sales");
let john = new Person("John"); // Error: The 'Person' constructor is protected
```

En TypeScript, podemos utilizar patrones orientados a objetos. Una clase es capaz de extender las clases existentes para crear otras nuevas mediante herencia.

Crearemos un proyecto de Typescript

- Crear una carpeta e inicializar un proyecto en node.js: `npm init`
- Instalar el compilador: `npm install -D typescript`
- Crear el proyecto typescript: `tsc --init`
- Agregar "ES2015","DOM" en tsconfig.json sección "lib": []
- Generar un archivo.ts con código de prueba
- Compilar con `.\node_modules\.bin\tsc`
- Ejecutar `node archivo.js`