



Índice

| | |
|---|----|
| Introducción | 3 |
| Módulos | 5 |
| Archivos | 7 |
| Servidor | 9 |
| Node Package Manager (npm) | 11 |
| JSON-Server | 13 |
| Express | 14 |
| - Response | 15 |
| - Request | 16 |
| - Next | 18 |
| API | 19 |
| - Estructura | 20 |
| - JWT (Json Web Tokens) | 26 |
| Aplicaciones de escritorio (electron) | |

Introducción

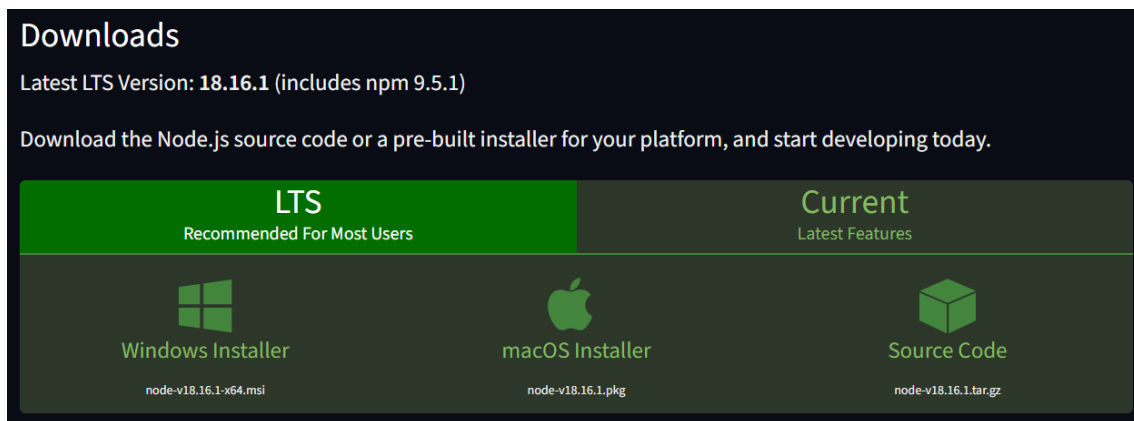
NodeJS es un entorno de ejecución de JavaScript, lo cual permite utilizar JavaScript para la creación de aplicaciones y backend de una página web.

Posee una arquitectura orientada a eventos (tiene un bucle que se va ejecutando cada vez que le llega un evento en un solo thread)

Utiliza motor V8 (chrome)

Instalación

Se puede instalar **NodeJS** desde la página oficial <https://nodejs.org/en/download>



La versión LTS es la más estable, mientras que Current es la última sin soporte actualizado.

Una vez descargado e instalado, comprobar que la instalación fue exitosa abriendo la consola del sistema y verificar la versión instalada.

```
Command Prompt
Microsoft Windows [Version 10.0.17758.1]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\Admin>Node --version
v14.17.3

C:\Users\Admin>npm --version
6.14.13

C:\Users\Admin>
```

— Verificación de la instalación de Node.js en Windows.

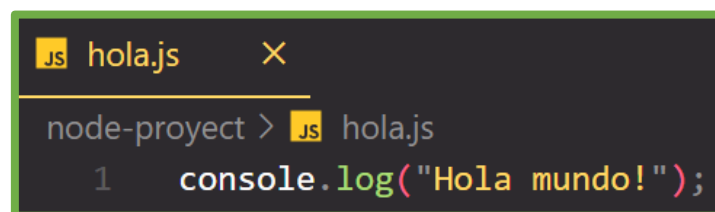
Nuevo proyecto

Para iniciar un nuevo proyecto en NodeJS, es necesario crear el directorio donde se ubicará el proyecto.

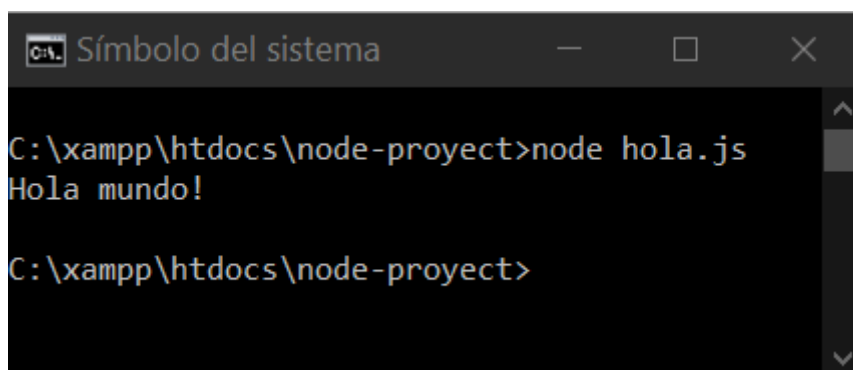
Luego, crear el primer archivo JS del proyecto (en general, **index.js**) en el cual se puede utilizar código JavaScript.

Ejecución del programa

Se puede ejecutar el código generado mediante el comando **node** en consola.



```
JS hola.js X
node-proyect > JS hola.js
1 console.log("Hola mundo!");
```



```
Símbolo del sistema
C:\xampp\htdocs\node-proyect>node hola.js
Hola mundo!
C:\xampp\htdocs\node-proyect>
```

Módulos

División del código en partes más pequeñas, en archivos.

La palabra reservada **require** permite “importar” módulos.

La palabra reservada **exports** permite “exportar” funciones de un módulo.

```
JS index.js X
const math = require('./math.js');
let resultado = math.sumatoria(1, 2, 3);
console.log(resultado);
```

```
C:\xampp\htdocs\node-proyect>node index.js
6
```

```
JS math.js X
function sumar (...operandos) {
  let sumaTotal = 0;
  for (let i = 0; i < operandos.length; i++) {
    sumaTotal += operandos[i];
  }
  return sumaTotal;
}
exports.sumatoria = sumar;
```

↓
clave valor

Al importar un módulo, lo que se está haciendo es importar un objeto con las funciones que el módulo posee:

```
JS index.js X
const math = require('./math.js');
console.log(math);
```

```
C:\xampp\htdocs\node-proyect>node index.js
{ sumatoria: [Function: sumar] }
```

Esto significa que se puede exportar directamente el objeto con todas las funciones, aunque también se pueden exportar variables y funciones individuales.

```
JS math.js X
const sumar = (a, b) => a + b;
const restar = (a, b) => a - b;
const multiplicar = (a, b) => a * b;
const dividir = (a, b) => b !== 0 ? a / b : console.error("Error");

const Math = {
  sumar: sumar,
  restar: restar,
  multiplicar: multiplicar,
  dividir: dividir
};

module.exports = Math;
```

```
C:\xampp\htdocs\node-proyect>node index.js
{
  sumar: [Function: sumar],
  restar: [Function: restar],
  multiplicar: [Function: multiplicar],
  dividir: [Function: dividir]
}
```

Si se desea exportar una sola función, se hace de la siguiente manera.

```
JS module.js X
function saludar (nombre) {
  console.log(`Hola ${nombre}`);
}

module.exports = saludar;
```

```
JS index.js X
const saludar = require('./module.js');

console.log(saludar);

saludar("Guillermo");
```

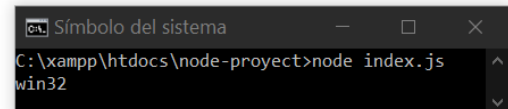
Módulos preconstruidos

NodeJS tiene módulos por defecto, los cuales permiten diversas funcionalidades.

Estos módulos se pueden encontrar en la documentación de NodeJS, así como todas las funciones que tiene cada módulo.

Nota: se requiere escribir el prefijo **node:** antes del nombre del módulo.

```
const os = require('node:os');
console.log(os.platform());
```



```
Símbolo del sistema
C:\xampp\htdocs\node-proyect>node index.js
win32
```

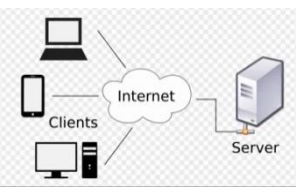
```
const fs = require('node:fs');
```



Archivos

El módulo **FileSystem** permite trabajar con archivos del dispositivo.

Min 49 fazt



Servidor

Los dispositivos se comunican con un servidor por medio de peticiones con un determinado protocolo.

Protocolo HTTP

Utiliza dos parámetros.

- request petición del cliente al servidor.
- response respuesta del servidor al cliente.

HTTP Status Codes

Information [100 - 199]

100 - Continue
101 - Switching Protocols
102 - Processing
103 - Early hints

Success [200 - 299]

200 - Ok
201 - Created
202 - Accepted
204 - No Content
206 - Partial Content

Redirect [300 - 399]

300 - Multiple choices
301 - Moved Permanently
304 - Not Modified
307 - Temporary redirect
308 - Permanent redirect

Client Error [400 - 499]

400 - Bad request
401 - Unauthorized
403 - Forbidden
404 - Not found
409 - Conflict

Server Error [500 - 599]

500 - Internal server error
501 - Not implemented
502 - Bad gateway
503 - Service unavailable
504 - Gateway timeout

```
const http = require('http');

// Manejar el servidor tras crearse
function handleServer (request, response) {
    response.writeHead(200, { 'Content-type': 'text/html' });
    response.write("<h1>Hola mundo desde NodeJS</h1>");
    response.end(); // reinicia bucle para recibir otras peticiones
}

// Crear servidor (demora un tiempo)
const server = http.createServer(handleServer);

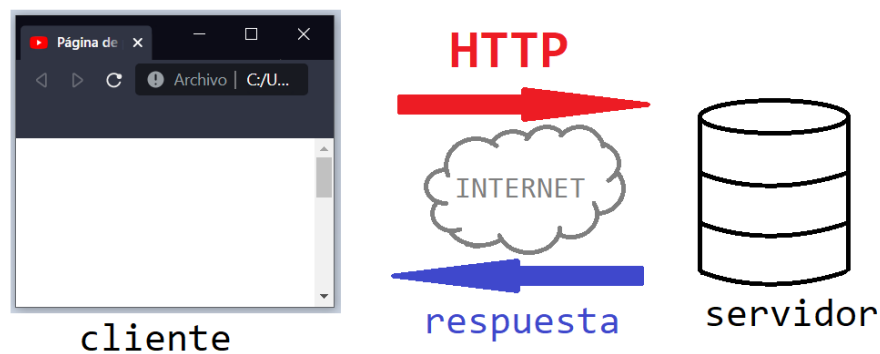
// Publicar servidor en el puerto 3000
server.listen(3000, () => console.log("Server on port 3000"));
```

Peticiones HTTP

El protocolo HTTP es una petición que realiza el usuario / cliente hacia un servidor, y dicho servidor le devuelve una data (información).

Esto es unidireccional, es decir, sólo se puede obtener información y no guardarla dentro de un servidor (no mediante HTTP).

- Cliente: todo a lo que el usuario puede acceder desde su interfaz (navegador).
- Servidor: donde se procesa toda la información.

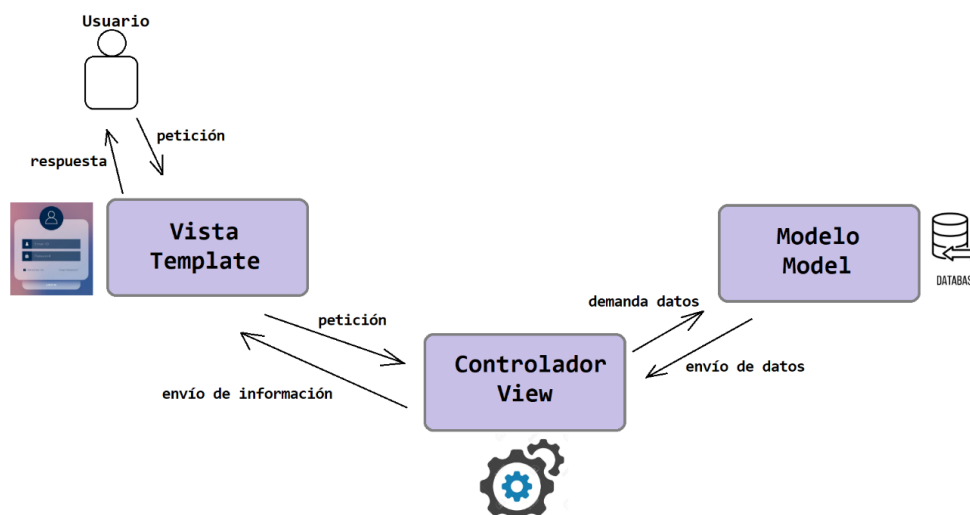


Cada vez que el servidor envía una respuesta al cliente, se actualiza la página web activa para poder visualizarse.

Modelo Vista Controlador

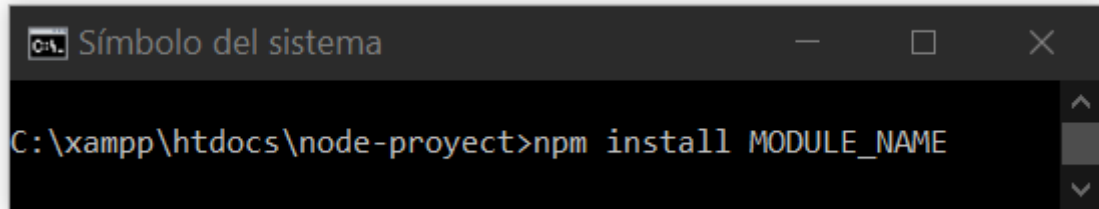
Patrón que consiste en dividir cualquier aplicación en tres grandes módulos.

- **Modelo:** encargado de gestionar los datos, en general con una Base de Datos.
- **Vista:** encargado de mostrar la información al usuario, la interfaz gráfica.
- **Controlador:** encargado de gestionar las comunicaciones entre la vista y el modelo.



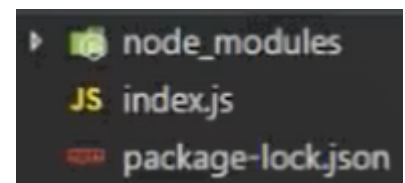
Node Package Manager

Permite administrar paquetes y módulos para nuestro proyecto, entre ellos, módulos creados por terceros y almacenados en www.npmjs.com



Cuando se instala un módulo externo con el comando **npm** en consola para nuestro proyecto, se añadirán algunos archivos.

- ✓ node_modules módulo instalado
- ✓ package-lock.json información del módulo instalado



Creación de un proyecto con NPM

Permite almacenar toda la información del proyecto, junto con todos los módulos que se hayan utilizado para poder retomarlo en otro dispositivo de ser necesario.

npm init

Tras esto, se pedirá una serie de datos para guardar el proyecto con la información ingresada.

Finalizando el proceso, se creará un **package.json** de configuración de meta-información.

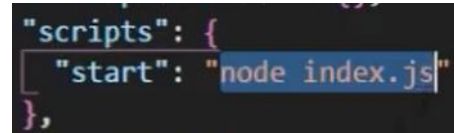
Instalación de un proyecto previamente creado

Encontrándonos en consola dentro del directorio donde se encuentra el **package.json**, se puede ejecutar el comando **npm install** para que se instalen todos los módulos necesarios para iniciar el proyecto.

Personalización de comandos

Dentro del **package.json** se encuentran los comandos personalizados de npm.

Esto quiere decir que, al ejecutar un comando en la consola, se ejecutará lo que está establecido en esta sección.

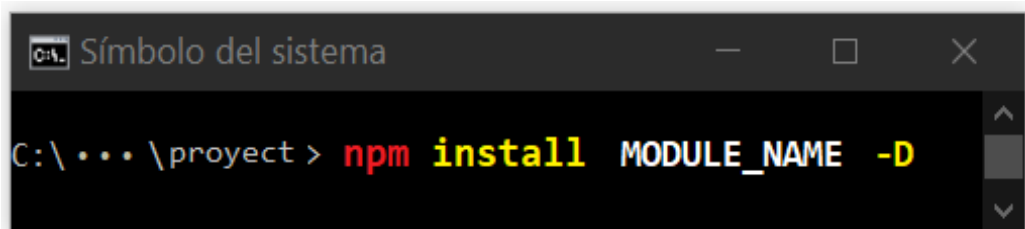


```
"scripts": {  
  "start": "node index.js"  
},
```

En este ejemplo, al utilizar el comando **npm run start**, se ejecutará el comando **node index.js**, iniciando el código que se encuentra en index.js

Módulos exclusivos de modo desarrollo

Se pueden instalar módulos que sólo estén en el modo desarrollo de nuestro proyecto, es decir, que no se utilizarán finalmente en el proyecto terminado.

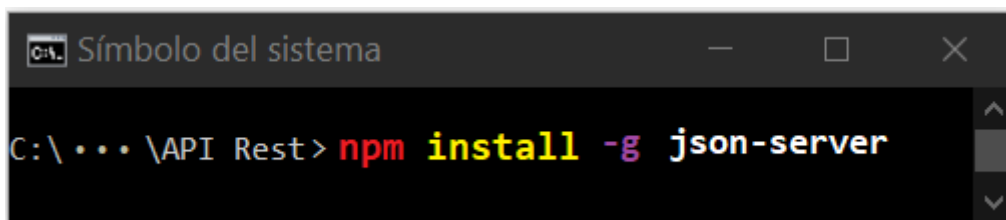


Símbolo del sistema

```
C:\...\proyect > npm install MODULE_NAME -D
```

JSON-Server

Permite la creación de una API sencilla a partir de un archivo JSON.



```
Símbolo del sistema
C:\... \API Rest> npm install -g json-server
```

Una vez instalado, se crea un archivo JSON que será la información almacenada.

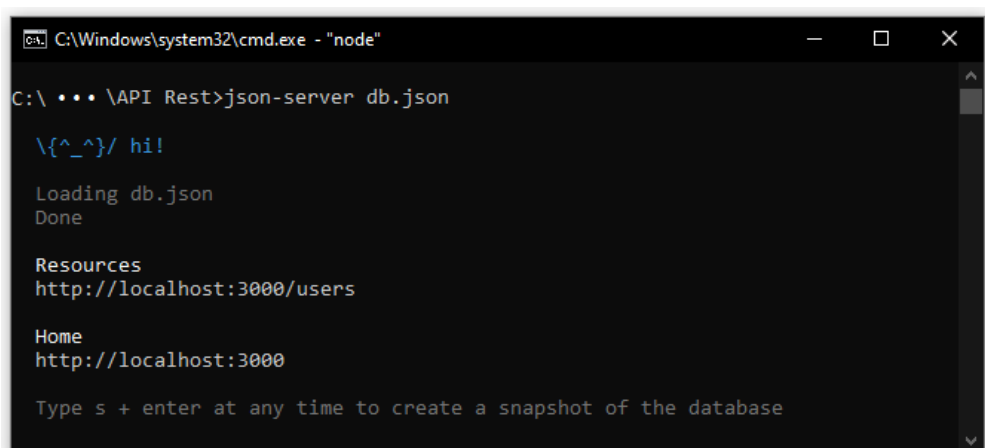


```
{
  "users": [
    {
      "id": 0,
      "name": "Guille"
    },
    {
      "id": 1,
      "name": "Nico"
    }
  ]
}
```

Creado nuestra “base de datos” en JSON, se ejecuta el siguiente comando

json-server db.json

└─ Nombre del JSON



```
C:\Windows\system32\cmd.exe - "node"
C:\... \API Rest> json-server db.json

\{^_^}/ hi!

Loading db.json
Done

Resources
http://localhost:3000/users

Home
http://localhost:3000

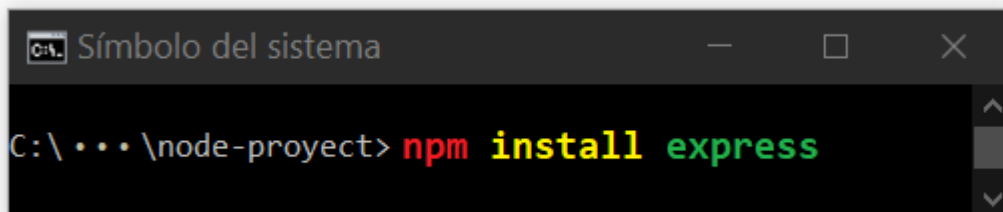
Type s + enter at any time to create a snapshot of the database
```

Así, si accedes a <http://localhost:3000/users/1> devolverá la información del id 1

Express

Es un framework de NodeJS que permite la ejecución de servidores y backend.

Se instala desde NPM a través de la consola.



```
C:\... \node-proyecto> npm install express
```

Un ejemplo sencillo de uso es el mismo ejemplo que se usó para la creación de un servidor con NodeJS, pero utilizando Express:

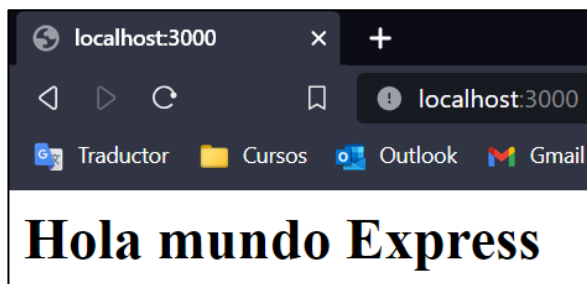
```
// Importar framework Express
const express = require('express');

// Crear servidor con Express
const server = express();

// Definir cómo responderá el servidor
server.get('/', (request, response) => {
  response.send('<h1>Hola mundo Express</h1>');
  response.end();
});

// Definir puerto del servidor
server.listen(3000, () => {
  console.log("Server on port 3000");
});
```

node index.js



Sequelize

Permite el uso de bases de datos relacionales (SQL).

```
npm install sequelize pg pg-hstore
```

Response

El response (abreviado **res**) es lo que el servidor responde o devuelve ante una petición del cliente / frontend.

```
import axios from "axios";
export const getData = async () => {
  const { data } = await axios.get("/"); // throwable
  return data;
}
```

```
const express = require('express') // framework
const server = express()

// settings
server.set('port', PORT) // variable port
server.set('json spaces', 2) // 2 espacios en json

// final response
const result = (request, response) => {
  response.status(200).json({ success: true }); // ok
  response.end()
}

// default route
server.get('/', result)

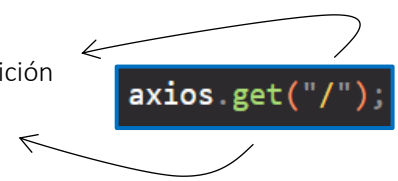
// Start server
server.listen(server.get('port'), () => {
  console.log(`Server on port ${server.get('port')}`);
});
```

Request

El request (abreviado **req**) contiene información de la petición solicitada por el frontend.

Algunos de ellos son:


- ✓ url obtiene la URL a la cual se le realizó la petición
- ✓ method obtiene el nombre del método requerido
- ✓ httpVersion
- ✓ complete
- ✓ statusCode
- ✓ statusMessage
- ✓ withCredentials será TRUE sólo si se ha admitido compartir cookies



```
axios.get("/");
```

Además, dentro del request se encontrará información adicional de la petición

- ✓ params dato que es enviado junto al endpoint.


 frontend

```
export const getUserData = async (id) => {
  const { data } = await axios.get(`/ ${id}`) // throwable
  return data
}
```

```
server.get('/:id', (request, response) => {
  const { id } = request.params;
  response.status(200).json({ id: id }); // ok
  response.end()
})
```

 backend

- ✓ query son los parámetros de la URL, los cuales se escriben luego de un signo de pregunta al final en formato ?key=value

 frontend

```
export const getName = async (name) => {
  const { data } = await axios.get(`/ ?name=${id}`) // throwable
  return data
}
```

```
server.get('/', (request, response) => {
  const { name } = request.query;
  response.status(200).json({ name: name }); // ok
  response.end()
})
```

 backend

- ✓ **body** se envían junto a la función **axios** cuando se utiliza **post** o **put**.

frontend

```
export const postData = async (json) => {
  const { data } = await axios.post("/", json); // throwable
  return data;
}
```

backend

```
server.post('/', (request, response) => {
  const { name, age, married } = request.body;
  response.status(200).json({ name, age, married }); // ok
  response.end()
})
```

```
{
  "name": "Guille",
  "age": 28,
  "married": false
}
```

- ✓ **header** datos que pueden enviarse junto a la función **axios** al utilizar **get**.

frontend

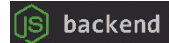
```
export const getDataWithCredentials = async () => { // throwable
  const { data } = await axios.get("/",
    {
      headers: {
        'User': true
      }
    }
  );
  return data;
}
```

backend

```
server.get('/', (request, response) => {
  const user = request.header('User');
  response.status(200).json({ userName: user }); // ok
  response.end()
})
```

Next

Es un parámetro extra del callback que permite hacer validaciones antes de dar una respuesta final desde el servidor.



```
const verifyToken = (request, response, next) => {
  const { token } = request.body;

  if (!token) return response.status(401).json({ result: "No token" });
  else if (token === 'ClaveSecreta') {
    request.userType = "ADMIN"; // añade una prop al request
    next();
  }
  else return response.status(401).json({ result: 'Invalid token' });
}

const verifyPermission = (requiredPermission) => {
  return (request, response, next) => {
    const { userType } = request;
    if (!userType) return response.status(401).json({ result: 'Unauthenticated.' });
    else if (userType === requiredPermission) next();
    else return response.status(403).json({ result: 'No permission.' });
  };
}

server.post('/',
  verifyToken,
  verifyPermission('ADMIN'),
  (request, response) => {
    // sólo llega hasta acá si en los 'verify' llegó a next
    response.status(200).json({ success: true })
  }
)
```

Si se quiere que alguno de los procesos (en este caso llamados 'verify') se ejecuten en todas las peticiones, se las puede configurar directamente en el método **use** de **server**.

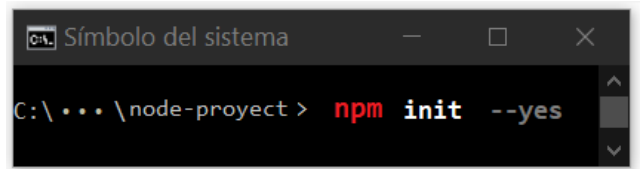
```
server.use(verifyToken)
```

API

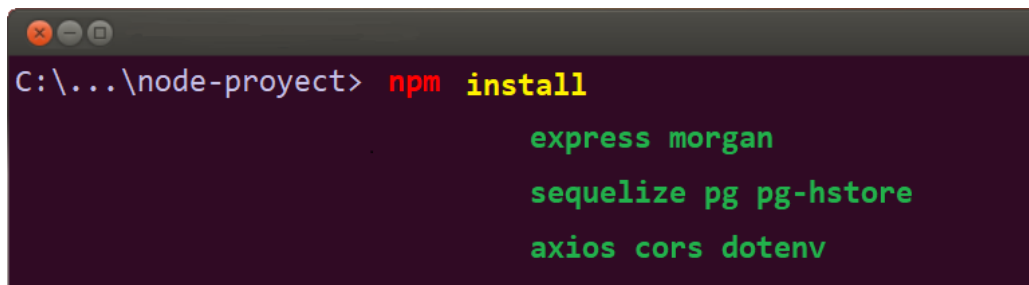
Servidor “**backend**” que conecta el **frontend** (vista del usuario) con la base de datos.

Se puede crear con Express y Sequelize (para las bases de datos relacionales).

Lo primero a hacer, es inicializar el proyecto:
y luego importar los módulos necesarios.



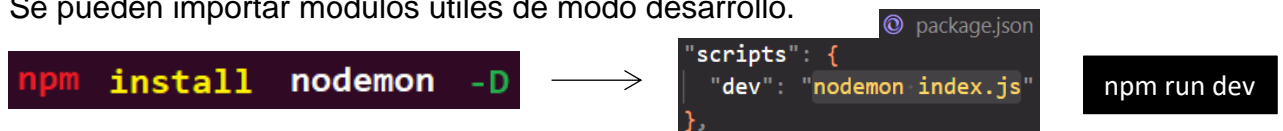
```
Símbolo del sistema
C:\...\node-project > npm init --yes
```



```
C:\...\node-project> npm install
    express morgan
    sequelize pg pg-hstore
    axios cors dotenv
```

- **morgan** permite visualizar las peticiones entrantes por consola
- **axios** permite realizar peticiones a la base de datos
- **cors** permite el manejo de las políticas CORS
- **dotenv** permite manejar claves de acceso de archivos **.env**

Se pueden importar módulos útiles de modo desarrollo.



```
npm install nodemon -D
```

→

```
package.json
{
  "scripts": {
    "dev": "nodemon index.js"
  }
}
```

npm run dev

- **nodemon** permite ejecutar el servidor de manera tal que, al realizar cambios en el código, el servidor se reinicie para que los cambios queden efectivos en el momento (y no tener que reiniciarlo manualmente)

Estructura de la API

Se expondrá un ejemplo en el cual se trabajará con una tabla que almacene mensajes de usuarios con la siguiente información: **name**, **email**, **message**.

| id | name | email | message |
|----|--------|------------------|-----------------|
| 1 | Guille | guille@gmail.com | Primer mensaje |
| 2 | Josi | josi@gmail.com | Segundo mensaje |
| 3 | Marce | marce@gmail.com | Tercer mensaje |

index.js

```
const server = require("../src/server");

// Start server
server.listen(server.get('port'), () => {
  console.log(`Server on port ${server.get('port')}`);
});
```

server.js

```
const express = require('express') // framework
const router = require('./routes') // routes
const morgan = require('morgan')
const cors = require('cors')

const { PORT } = require('./config')

const server = express()

// middlewares
server.use(morgan('dev'))
server.use(express.urlencoded({extended:false}))
server.use(express.json())
server.use(cors())
server.use(router)

// settings
server.set('port', PORT) // variable port
server.set('json spaces', 2) // 2 espacios en json

module.exports = server;
```

```
src
├── controllers \ messages
│   ├── deleteMessageController.js
│   ├── getMessageController.js
│   ├── getMessagesController.js
│   ├── postMessageController.js
│   ├── putMessageController.js
│   └── selectMessagesByNameController.js
├── handlers
│   └── messageHandlers.js
├── models
│   └── Message.js
├── routes
│   ├── index.js
│   └── messageRoutes.js
├── config.js
├── db.js
├── server.js
├── .env
├── index.js
├── package-lock.json
└── package.json
```

config.js

```
require('dotenv').config(); // carga archivo .env

const PORT = process.env.PORT || 3000;
const DB_HOST = process.env.DB_HOST || "localhost";
const DB_USER = process.env.DB_USER || "root";
const DB_PASSWORD = process.env.DB_PASSWORD || "";
const DB_DATABASE = process.env.DB_DATABASE || "lodepica";
const DB_NAME = process.env.DB_NAME || "test";
const DB_PORT = process.env.DB_PORT || 3000;
const DB_PARAMS = process.env.DB_PARAMS || undefined;

module.exports = {
  PORT,
  DB_HOST,
  DB_NAME,
  DB_USER,
  DB_PASSWORD,
  DB_DATABASE,
  DB_PORT,
  DB_PARAMS
}
```

.env

```
DB_USER = "fl0user"
DB_PASSWORD = "K7xNQErZV3qd"
DB_HOST = "link-del-host"
DB_PORT = "5432"
DB_NAME = "test"
DB_PARAMS = "sslmode=require"
PORT = 3000
```

db.js

```
const { Sequelize } = require("sequelize");
const { DB_USER, DB_PASSWORD, DB_HOST, DB_NAME, DB_PORT, DB_PARAMS } = require('./config')

const fs = require("fs"); // Lectura de archivos
const path = require("path"); // manipulación de path

const sequelize = new Sequelize( // Define la base de datos y su tipo
  `postgres://${DB_USER}:${DB_PASSWORD}@${DB_HOST}:${DB_PORT}/${DB_NAME}${DB_PARAMS}?'+DB_PARAMS+"",
  { logging: false, native: false }
);

// Load models
const basename = path.basename(__filename);
const modelDefiners = [];
fs.readdirSync(path.join(__dirname, "/models"))
  .filter((file) => file.indexOf(".") !== 0 && file !== basename && file.slice(-3) === ".js")
  .forEach((file) => { modelDefiners.push(require(path.join(__dirname, "/models", file))); });
modelDefiners.forEach((model) => model(sequelize));

let entries = Object.entries(sequelize.models);
let capsEntries = entries.map((entry) => [
  entry[0][0].toUpperCase() + entry[0].slice(1),
  entry[1],
]);
sequelize.models = Object.fromEntries(entries);
console.log(entries);

// Modelos
/*const { Driver, Team } = sequelize.models;
Driver.belongsToMany(Team, { through: "driver_team" });
Team.belongsToMany(Driver, { through: "driver_team" });*/

// Aca vendrian las relaciones
// Product.hasMany(Reviews);

module.exports = {
  ...sequelize.models, // para poder importar los modelos así: const { Product, User } = require('./db.js')
  conn: sequelize, // para importar la conexión { conn } = require('./db.js');
};
```

➤ Models

Define las tablas de la base de datos, junto con las columnas que contiene cada tabla y los tipos de dato que se pueden ingresar en cada una.

► models / db.js: Message.js

```
const { DataTypes } = require("sequelize");
module.exports = (sequelize) => {
  sequelize.define(
    "Message", // Nombre del modelo
    // Cuando se ejecute una query a la base de datos,
    // sequelize plurizará el nombre del modelo.
    // Es decir, la query la hará a la tabla "Messages"
    {
      id: {
        type: DataTypes.UUID,
        allowNull: false,
        primaryKey: true,
        //defaultValue: DataTypes.UUIDV4,
        autoIncrement: true, // el id se autoincrementa
      },
      name: {
        type: DataTypes.STRING,
        allowNull: true,
      },
      email: {
        type: DataTypes.STRING,
        allowNull: true,
        unique: true, // no se puede repetir
      },
      message: {
        type: DataTypes.STRING,
        allowNull: false, // puede ser null
      },
    },
    // Extra options
    {
      timestamps: false, // Fecha en que se crea la celda
      tableName: "messages" // Nombre literal de la tabla
    }
  );
};
```

➤ Routers

Maneja las rutas internas del servidor, desde las cuales se interactúa de diferente forma con la API.

El archivo index.js dentro de un directorio (como routes) permite que, al importarse un directorio con **require**, se importe el contenido del index de ese directorio.

► routes / (); index.js

```
const { Router } = require("express");
const router = Router();

// Importar nuestras subrutas
const messagesRoutes = require("./messageRoutes");

// Declaración de nuestras rutas + subrutas
router.use("/messages", messagesRoutes);

module.exports = router;
```

► routes / (); messageRoutes.js

```
const { Router } = require("express");

// Importar handlers
const {
  getMessagesHandler,
  getMessageHandler,
  postMessageHandler,
  deleteMessageHandler,
  putMessageHandler,
} = require("../handlers/messageHandlers");

// Subrutas "/messages" según tipo de petición
const messageRoutes = Router();
messageRoutes.get("/", getMessagesHandler); // obtener todos
messageRoutes.get("/:id", getMessageHandler); // obtener por id
messageRoutes.post("/", postMessageHandler); // crear nuevo
messageRoutes.delete("/:id", deleteMessageHandler); // borrar
messageRoutes.put("/:id", putMessageHandler); // editar por id

module.exports = messageRoutes;
```

```
// Importar controllers (funciones que actúan sobre la base de datos)
const { createMessage } = require("../controllers/messages/postMessageController");
const { deleteMessage } = require("../controllers/messages/deleteMessageController");
const { getAllMessages } = require("../controllers/messages/getMessagesController");
const { getMessageById } = require("../controllers/messages/getMessageController");
const { getMessagesByName } = require("../controllers/messages/selectMessagesByNameController");
const { editMessage } = require("../controllers/messages/putMessageController");

// Crear handlers (funciones ejecutadas al ingresar en una ruta)
// Obtener según id
const getMessageHandler = async (req, res) => {
  try {
    const { id } = req.params;
    if (id) {
      const getId = await getMessageById(id);
      if (getId) res.status(200).json(getId); // ok
      else res.status(404).json({}); // not found
    }
  } catch (error) {
    res.status(500).json({ detail: error, error: error.message }); // internal server error
  }
};

// Obtener todos
const getMessagesHandler = async (req, res) => {
  try {
    const { name } = req.query;
    if (name) { // Si hay un name en la query, sólo se obtienen esos
      const messages = await getMessagesByName(name);
      res.status(200).json(messages); // ok
    } else {
      const messages = await getAllMessages();
      res.status(200).json(messages); // ok
    }
  } catch (error) {
    res.status(500).json({ detail: error, error: error.message }); // internal server error
  }
};

// Agregar un nuevo mensaje
const postMessageHandler = async (req, res) => {
  const { name, email, message } = req.body;
  try {
    const created = await createMessage(
      name,
      email,
      message
    );
    res.status(201).json(created); // created
  } catch (error) {
    res.status(400).json({ detail: error, error: error.message }); // bad request
  }
};

// Borrar según id
const deleteMessageHandler = async (req, res) => {
  try {
    const { id } = req.params;
    if (id) {
      const removed = await deleteMessage(id);
      if (removed) res.status(200).json(removed); // ok
      else res.status(204).json({id:false}); // no content
    }
  } catch (error) {
    res.status(500).json({ detail: error, error: error.message }); // internal server error
  }
};

// Editar según id
const putMessageHandler = async (req, res) => {
  try {
    const { id } = req.params;
    if (id) {
      const rowsEdited = await editMessage(id, req.body);
      if (rowsEdited) res.status(200).json(rowsEdited); // ok
      else res.status(204).json({id:false}); // no content
    }
  } catch (error) {
    res.status(500).json({ detail: error, error: error.message }); // internal server error
  }
};

// Exportar handlers
module.exports = {
  getMessageHandler,
  getMessagesHandler,
  postMessageHandler,
  putMessageHandler,
  deleteMessageHandler,
};
```

➤ Handlers

Son los que tienen las funciones permitidas dentro de una ruta.

Es decir, son los que “manejan” lo que sucede dentro de la ruta “messages”.

► handlers / **()**: messageRoutes.js

➤ Controllers

Funciones que interactúan directamente con la base de datos.

(); getMessagesController.js

```
const { Message } = require("../db");

const getAllMessages = async () => {
  const response = await Message.findAll()
  return response
}

module.exports = { getAllMessages };
```

(); getMessageController.js

```
const { Message } = require("../db");

const getMessageById = async (id) => {
  const message = await Message.findOne({
    where: { id: id },
  });
  if (message) { // encontrado
    return message.toJSON();
  }
};

module.exports = { getMessageById };
```

(); deleteMessageController.js

```
const { Message } = require("../db");

const deleteMessage = async (id) => {
  const messageId = await Message.destroy({
    where: { id: id },
  });
  return { id: messageId };
}

module.exports = { deleteMessage };
```

(); putMessageController.js

```
const { Message } = require("../db");

const editMessage = async (id, put) => {
  const messagesEdited = await Message.update(put, {
    where: { id: id },
  });
  return messagesEdited; // array with ids edited
}

module.exports = { editMessage };
```

(); postMessageController.js

```
const { Message } = require("../db");

const createMessage = async (
  name,
  email,
  message
) => {

  const newMessage = await Message.create({
    // id se autoincrementa
    name,
    email,
    message
  });
  return newMessage;
}

module.exports = { createMessage };
```

(); selectMessagesByNameController.js

```
const { Message } = require("../db");

const getMessagesByName = async (name) => {
  const response = await Message.findAll({
    where: { name: name },
  })
  return response
}

module.exports = { getMessagesByName };
```


Request usados

Dentro del request (abreviado **req**) realizado al servidor, puede haber distintos datos según cómo se realice y utilice **axios**.

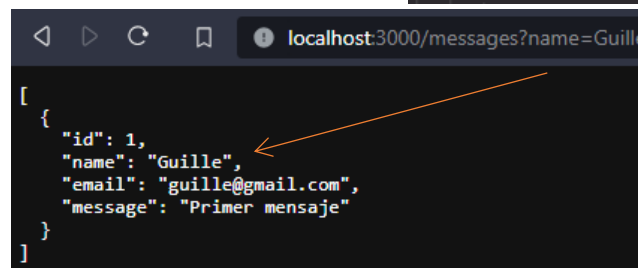


| id | name | email | message |
|----|--------|------------------|-----------------|
| 1 | Guille | guille@gmail.com | Primer mensaje |
| 2 | Josi | josi@gmail.com | Segundo mensaje |
| 3 | Marce | marce@gmail.com | Tercer mensaje |

- query

Son aquellos parámetros de la url (se escriben luego de un signo de pregunta)

```
const getMessagesHandler = async
try {
  const { name } = req.query;
  if (name) { // Si hay un nam
```



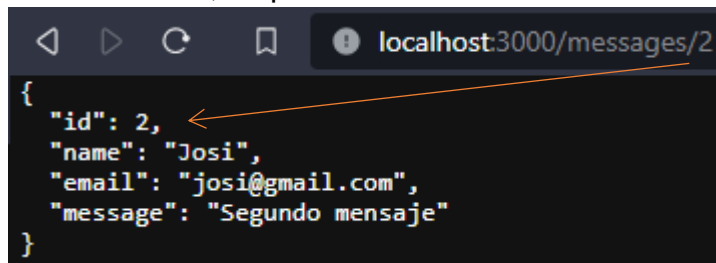
```
[
  {
    "id": 1,
    "name": "Guille",
    "email": "guille@gmail.com",
    "message": "Primer mensaje"
  }
]
```

- params

Aquellos que se envían junto al endpoint que fueron definidos como una url dinámica:

```
messageRoutes.get("/:id", getMessageHandler); // obtener por id
```

De esta forma, se puede utilizar:



```
{
  "id": 2,
  "name": "Josi",
  "email": "josi@gmail.com",
  "message": "Segundo mensaje"
}
```

```
// Obtener según id
const getMessageHandler = async
try {
  const { id } = req.params;
  if (id) {
```

- body

Aquellos que se envían junto a la función **axios**. Estos datos se envían en formato JSON.

```
// Agregar un nuevo mensaje
const postMessageHandler = async (req, res)
const { name, email, message } = req.body;
try {
```

```
axios.post(endpoint, post);
```

Nota: el id no es necesario debido a que el modelo fue configurado para autoincrementarlo

```
{
  "name": "Carla",
  "email": "carla@gmail.com",
  "message": "Cuarto mensaje"
}
```

Json Web Tokens

Permite gestionar el acceso a una base de datos desde el backend (servidor Express) en función de si un usuario está autenticado o no en el frontend (navegador del cliente). Generalmente se utilizan sistemas de autenticación y autorización.

Al autenticarse, el frontend podría recibir un token que debe incluirse en las solicitudes posteriores al backend.

```
const response = await axios.get('http://localhost:5000/datos-privados', {  
  headers: {  
    'Authorization': `Bearer ${token}` // Incluye el token en la cabecera  
  }  
});
```

La palabra "Bearer" se utiliza comúnmente para indicar que se está enviando un token JWT.



Tengo una página web de una pizzería. En el frontend, hecho en React, tengo la lista de todas las pizzas disponibles para pedir. El backend, hecho con Express y Sequelize, es el encargado de administrar la base de datos donde se guardan todas las pizzas que hay disponibles y posee un sistema de autenticación. ¿Cómo hay que hacer para que el dueño de la pizzería pueda modificar la base de datos? Es decir, ¿Debo agregar una ruta extra en el frontend que sea visible sólo para el usuario autorizado o debo hacer otra cosa?



Para permitir que el dueño de la pizzería modifique la base de datos desde el frontend, hay varias estrategias que podrías considerar. Aquí te presento algunas:



Opción 1: Crear una ruta protegida en el frontend

Puedes agregar una ruta adicional en el frontend que solo sea accesible para usuarios autenticados, como el dueño de la pizzería. Esta ruta puede proporcionar un formulario o una interfaz para agregar, editar o eliminar pizzas. Debes asegurarte de que esta ruta esté protegida por autenticación.

- El usuario debe iniciar sesión como dueño de la pizzería.
- Una vez autenticado, al acceder a esta ruta, se le mostrará la interfaz para administrar las pizzas.
- Al enviar los cambios desde el formulario en el frontend, estos datos se enviarían al backend a través de una API REST protegida con autenticación.

Opción 2: Crear un panel de administración separado

Desarrolla un panel de administración aparte del sitio web principal. Este panel estará protegido por autenticación y solo accesible para el dueño de la pizzería. Aquí, el dueño puede gestionar todas las pizzas, realizar cambios en la base de datos y administrar el contenido de la página.

- Al iniciar sesión en el panel de administración, el dueño obtendrá acceso a todas las funciones de gestión de pizzas.
- Este panel se comunica con el backend a través de API para realizar operaciones CRUD (crear, leer, actualizar, eliminar) en la base de datos.

Consideraciones de seguridad:

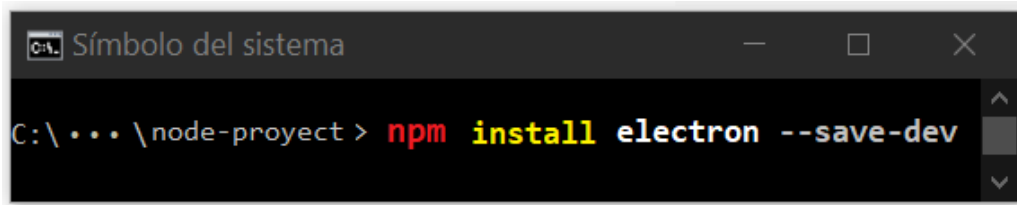
Independientemente del método que elijas, es fundamental implementar medidas de seguridad:

- **Autenticación y autorización:** Asegúrate de que solo el dueño tenga acceso a estas funciones de administración. Utiliza un sistema de autenticación seguro y gestiona los roles de usuario para garantizar que solo el usuario autorizado pueda realizar cambios.
- **Validación de datos:** Realiza la validación tanto en el frontend como en el backend para prevenir la introducción de datos maliciosos o incorrectos en la base de datos.
- **Seguridad de la API:** Asegúrate de que las rutas y endpoints de la API que manipulan la base de datos estén protegidos y sean seguros, utilizando tokens de autenticación o estrategias de autorización adecuadas.

Al implementar estas funcionalidades, podrás permitir al dueño de la pizzería administrar la base de datos desde el frontend de manera segura y eficiente.

Aplicaciones de escritorio

Electron es un framework que permite crear aplicaciones de escritorio ejecutables utilizando tecnologías web como HTML, CSS y JavaScript, y puedes empaquetar un proyecto web junto con un servidor local sin necesidad de configurar manualmente un servidor web.



```
C:\... \node-proyect > npm install electron --save-dev
```

Será necesario crear un archivo **main.js** que será el punto de entrada a la aplicación y que conecte con la carpeta del proyecto (con un **index.html** de entrada):

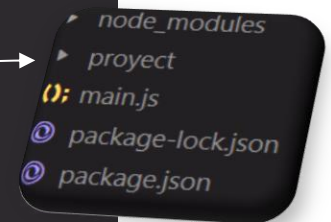
```
const { app, BrowserWindow } = require('electron'); // npm i electron
const express = require('express'); // npm i express
const server = express();
const path = require('path');

const PORT = 3000; // Puerto donde se ejecutará el servidor

// Configura el servidor web local
server.use(express.static(path.join(__dirname, 'proyect')));

// Crea una ventana de navegador Electron
app.on('ready', () => {
  const mainWindow = new BrowserWindow({ width: 800, height: 600 });
  mainWindow.loadURL(`http://localhost:${PORT}`);
});

// Inicia el servidor
server.listen(3000, () => {
  console.log(`Local server port on http://localhost:${PORT}`);
});
```

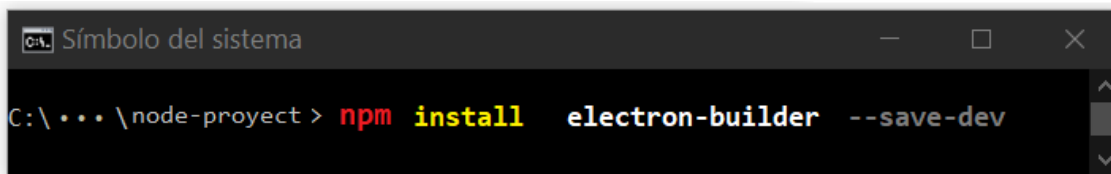


Luego, en **package.json** se configura el comando para poder inicializar la app con el comando **npm run start**.

```
"scripts": {
  "start": "electron main.js"
}
```

Ejecutable

Para generar un ejecutable de Electrón, es necesario instalar el builder.



```
C:\... \node-project > npm install electron-builder --save-dev
```

Será necesario crear **electron-builder.json** con la configuración de la aplicación.

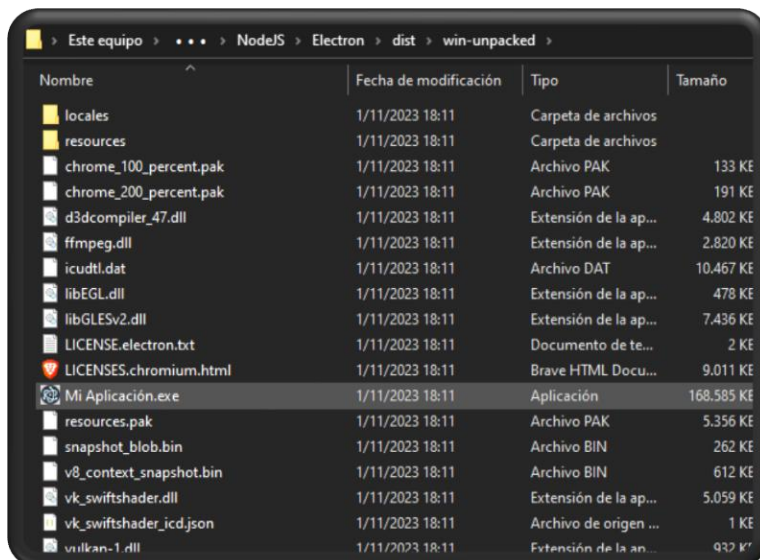
```
{
  "appId": "com.example.myapp",
  "productName": "Mi Aplicación",
  "directories": {
    "output": "dist"
  },
  "files": [
    "main.js",
    "project/**/*"
  ],
  "asar": true
}
```



Configurar el comando para construir el proyecto y asegurarse que el archivo **package.json** esté bien estructurado.

```
"scripts": {
  "start": "electron main.js",
  "build": "electron-builder"
}
```

Ejecutar el comando **npm run build** para construir el proyecto (dentro de **dist**).



```
package.json
{
  "name": "electron",
  "version": "1.0.0",
  "description": "una descripción",
  "main": "main.js",
  "scripts": {
    "start": "electron main.js",
    "build": "electron-builder"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.18.2"
  },
  "devDependencies": {
    "electron": "^27.0.2",
    "electron-builder": "^24.6.4"
  },
  "build": {
    "win": {
      "icon": "ruta/al/icono.ico"
    }
  }
}
```

Nota: la prop "build" es sólo para personalizar el ícono, puede quitarse

