

Conexión entre Backend y Frontend en Vue.js

Maria Fernanda Robayo Laguna

Laura Sofía Cangrejo Perafan

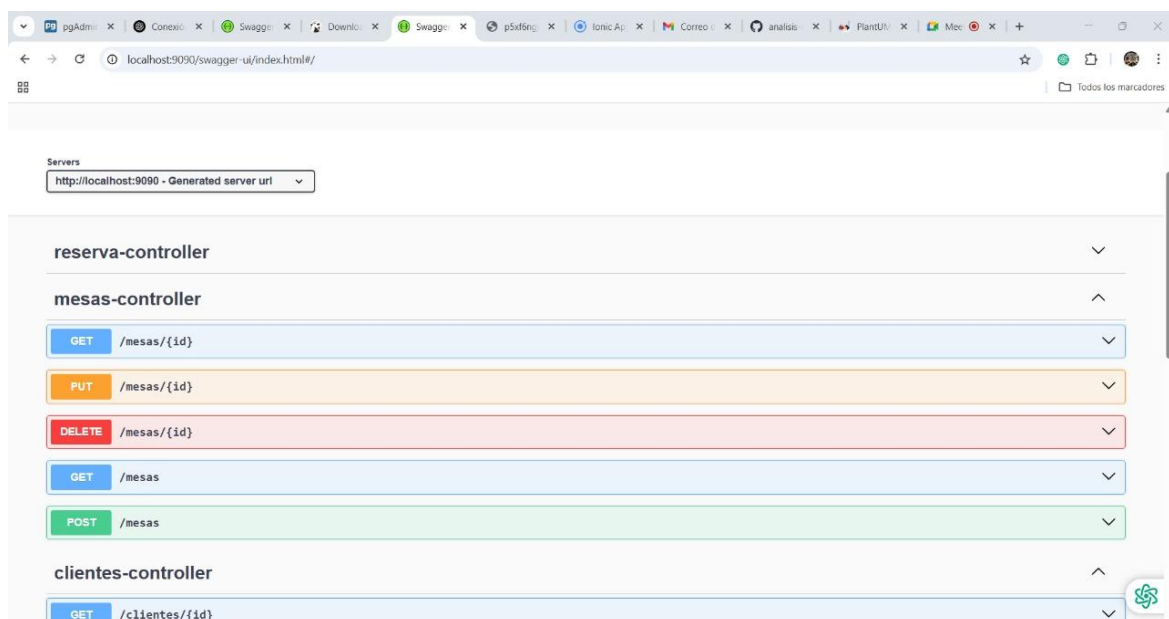
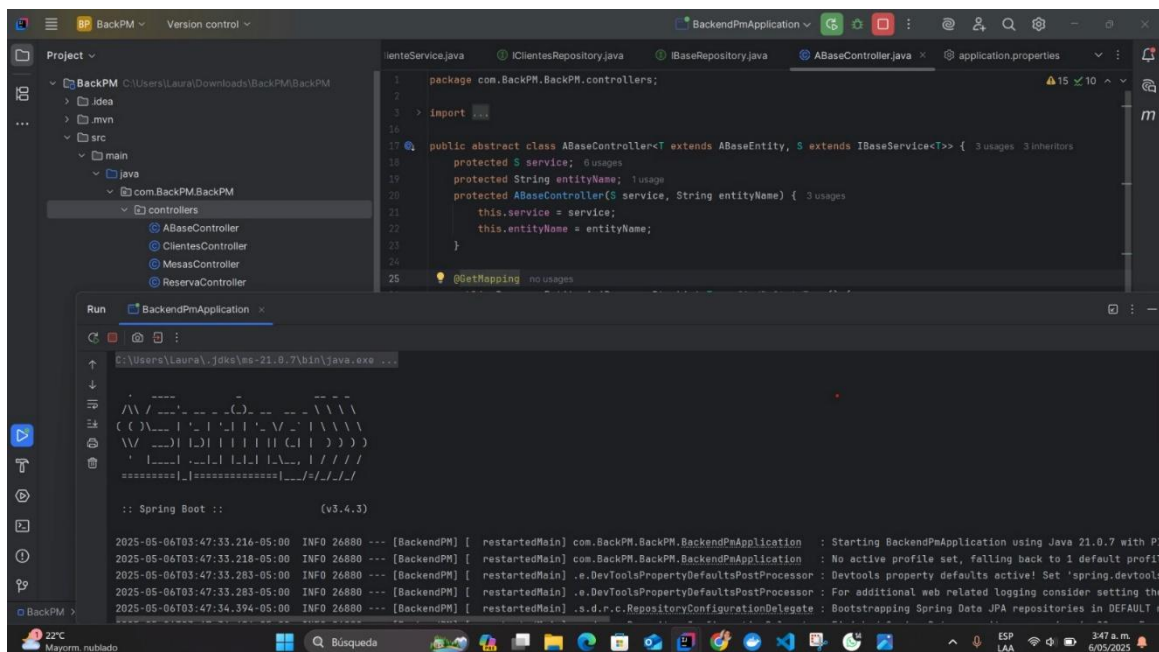
Docente: Jesús Ariel González Bonilla

Corporación universitaria de Huila-Corhuila

06/ mayo /2025

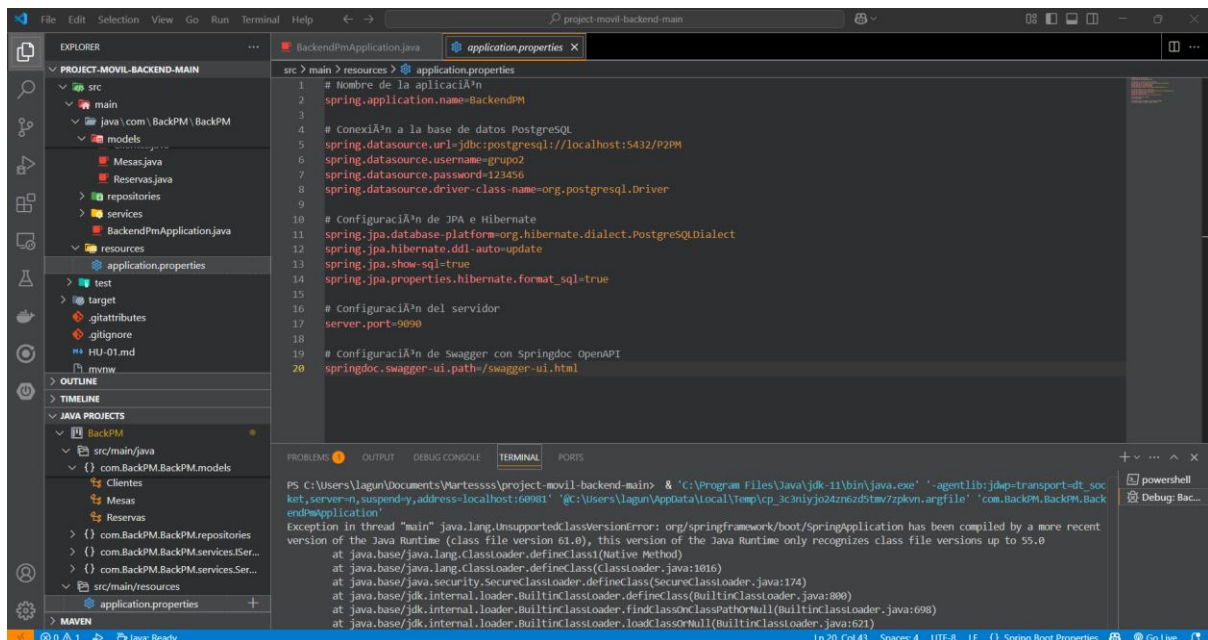
Paso 1: Verifico que el backend esté corriendo y que Swagger funcione bien

Primero me aseguro de que el backend esté levantado sin errores. Para eso lo ejecuto (en mi caso corre en el puerto 9090) y luego entro a Swagger para confirmar que todos los endpoints estén visibles y accesibles. Por ejemplo, desde <http://localhost:9090/swagger-ui/index.html> puedo ver las rutas de **mesas**, **clientes** y **reservas**, y ahí mismo puedo probarlas para ver si responden bien. Si todo eso funciona, ya puedo pasar al frontend con confianza porque sé que el backend está listo para recibir peticiones.



Paso 2: Verifico el puerto y la conexión con la base de datos

Después de confirmar que el backend levanta bien, reviso que esté usando el puerto correcto y que esté conectado a la base de datos. En mi archivo `application.properties`, tengo definido que el servidor corre en el puerto 9090 y que la base de datos es PostgreSQL, alojada localmente en el puerto 5432. También tengo configurado el nombre de la base (P2PM), el usuario (grupo2) y la contraseña (123456). Si todo eso está bien, significa que el backend puede comunicarse con la base y ejecutar consultas sin problemas.



The screenshot shows an IDE with the `application.properties` file open. The file contains the following configuration:

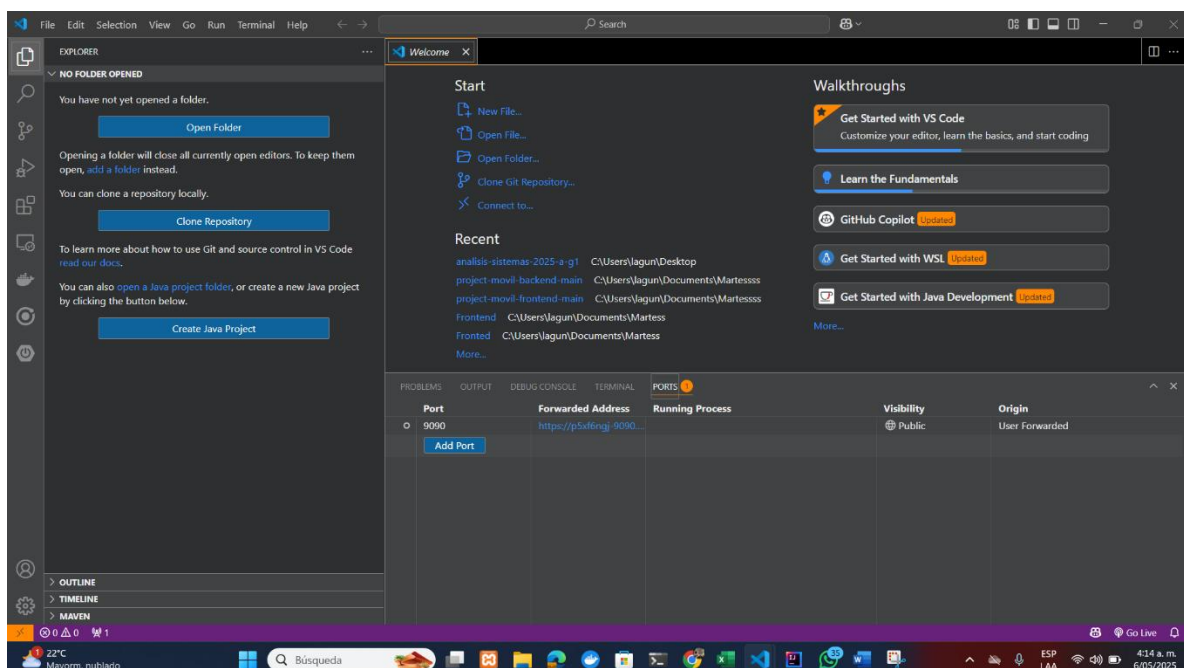
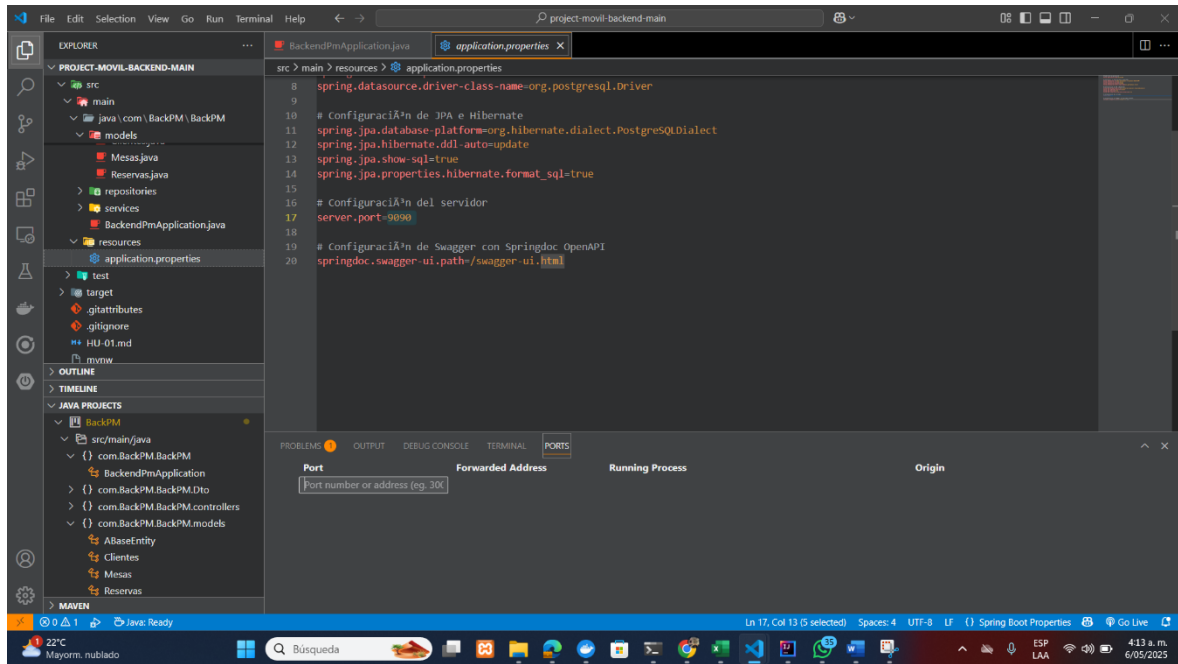
```
1 # Nombre de la aplicación
2 spring.application.name=BackendPM
3
4 # Conexión a la base de datos PostgreSQL
5 spring.datasource.url=jdbc:postgresql://localhost:5432/P2PM
6 spring.datasource.username=grupo2
7 spring.datasource.password=123456
8 spring.datasource.driver-class-name=org.postgresql.Driver
9
10 # Configuración de JPA e Hibernate
11 spring.jpa.database-platform=org.hibernate.dialect.PostgreSQLDialect
12 spring.jpa.hibernate.ddl-auto=update
13 spring.jpa.show-sql=true
14 spring.jpa.properties.hibernate.format_sql=true
15
16 # Configuración del servidor
17 server.port=9090
18
19 # Configuración de Swagger con Springdoc OpenAPI
20 springdoc.swagger-ui.path=/swagger-ui.html
```

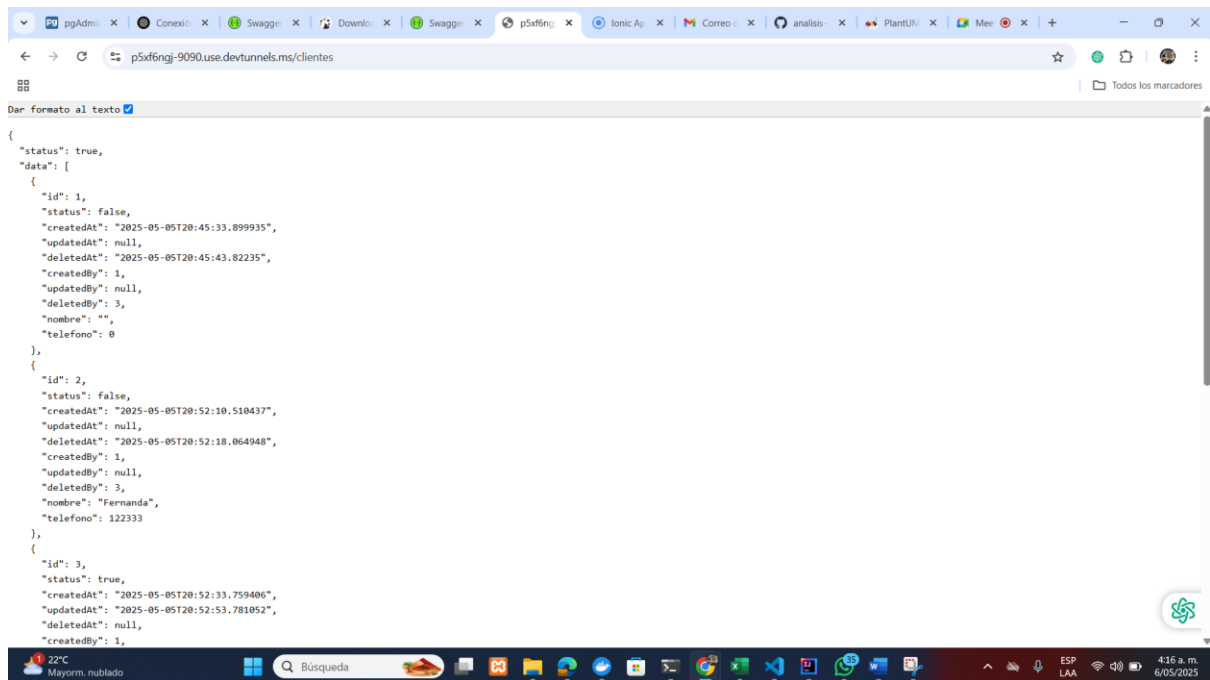
The terminal window shows the following error message:

```
PS C:\Users\lagun\Documents\Wartesss\project-movil-backend-main> & 'C:\Program Files\Java\jdk-11\bin\java.exe' '-agentlib:jdwp=transport=dt_socket,server=n,suspend=y,address=localhost:60981' '-Dc:\Users\lagun\AppData\Local\Temp\cp_3c3niyjo24zme2d3tm7zpkvm.argfile' '-Dcom.BackPM.BackPM.BackendPMApplication'
Exception in thread "main" java.lang.UnsupportedClassVersionError: org.springframework.boot.SpringApplication has been compiled by a more recent version of the Java Runtime (class file version 61.0), this version of the Java Runtime only recognizes class file versions up to 55.0
    at java.base/java.lang.ClassLoader.defineClass1(Native Method)
    at java.base/java.lang.ClassLoader.defineClass(ClassLoader.java:1016)
    at java.base/java.security.SecureClassLoader.defineClass(SecureClassLoader.java:174)
    at java.base/jdk.internal.loader.BuiltinClassLoader.defineClass(BuiltinClassLoader.java:800)
    at java.base/jdk.internal.loader.BuiltinClassLoader.findClassOnClassPathOrNull(BuiltinClassLoader.java:698)
    at java.base/jdk.internal.loader.BuiltinClassLoader.loadClassOrNull(BuiltinClassLoader.java:621)
```

### Paso 3: Hacemos un puente con el puerto 9090

Aquí lo que hacemos es exponer el puerto 9090 desde Visual Studio Code para poder acceder a nuestro backend desde el navegador. En la pestaña de PORTS, agregamos el puerto 9090 y se crea un enlace público (forwarded address) que nos permite acceder directamente a nuestra API.





Paso 4: Conectamos el frontend al backend usando la URL pública

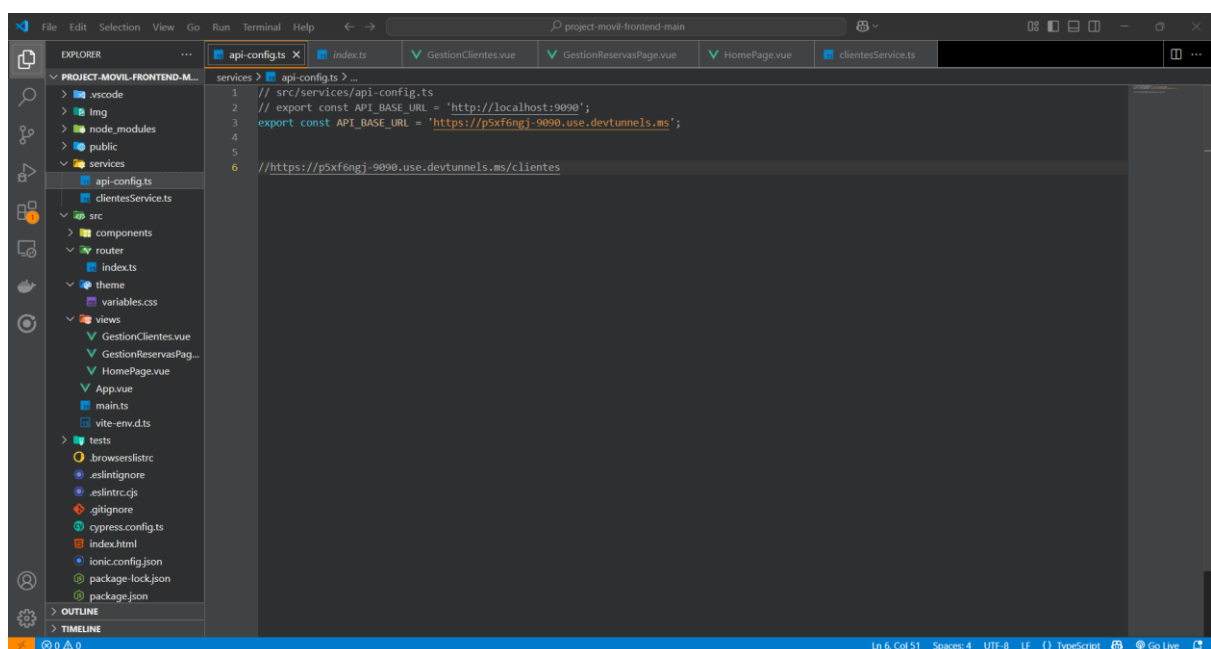
Después de crear el túnel en el Paso 3, ya tenemos un enlace público que nos permite acceder a nuestra API desde cualquier lugar.

Ahora, vamos al proyecto del frontend y hacemos lo siguiente

1. Abrimos el archivo:

`src/services/api-config.ts`

2. Allí pegamos la URL que generó el túnel, reemplazando la dirección local:



### Paso 5: Configuración del archivo cliente.service.ts

Después de definir la URL del backend en api-config.ts, ahora creamos el archivo cliente.service.ts, que es el responsable de gestionar todas las operaciones relacionadas con clientes desde el frontend.

¿Qué contiene este archivo?

Este archivo importa axios, una librería que usamos para hacer peticiones HTTP al backend, y define varias funciones que permiten:

**GET** → Para consultar datos.

**POST** → Para guardar nuevos datos.

**PUT** → Para actualizar datos.

**DELETE** → Para eliminar datos.

**getAllClientes()**: Consulta la lista de clientes desde el backend.

```
// Obtener todos los clientes
export const getAllClientes = async (): Promise<ClienteData[]> => {
  try {
    const { data } = await axios.get(CLIENTE_API_URL)
    // Filtramos los clientes donde deleteAt es null
    const clientesActivos = data.data.filter((cliente: any) => cliente.deletedAt === null)
    return clientesActivos
  } catch (error) {
    console.error('Error al obtener clientes:', error)
    throw error
  }
}
```

**findClienteById(id):** Busca un cliente por su ID.

```
// Buscar cliente por ID
export const findClienteById = async (id: number): Promise<ClienteData> => {
  try {
    const { data } = await axios.get(`${CLIENTE_API_URL}/${id}`)
    return data
  } catch (error) {
    console.error(`Error al obtener cliente ID ${id}:`, error)
    throw error
  }
}
```

**saveCliente(data):** Envía los datos de un nuevo cliente al servidor.

```
// Crear nuevo cliente
export const saveCliente = async (clienteData: ClienteData): Promise<any> => {
  try {
    const { data } = await axios.post(CLIENTE_API_URL, clienteData)
    return data
  } catch (error) {
    console.error('Error al guardar cliente:', error)
    throw error
  }
}
```

**updateCliente(data, id):** Actualiza un cliente existente.

```
// Actualizar cliente existente
export const updateCliente = async (clienteData: ClienteData, id: number): Promise<any> => {
  try {
    const { data } = await axios.put(`${CLIENTE_API_URL}/${id}`, clienteData)
    return data
  } catch (error) {
    console.error(`Error al actualizar cliente ID ${id}:`, error)
    throw error
  }
}
```

**deleteCliente(id):** Elimina un cliente por su ID.

```
// Eliminar cliente por ID
export const deleteCliente = async (id: number): Promise<any> => {
  try {
    const { data } = await axios.delete(`${CLIENTE_API_URL}/${id}`)
    return data
  } catch (error) {
    console.error(`Error al eliminar cliente ID ${id}:`, error)
    throw error
  }
}
```

Paso 6: Crear y conectar la vista GestionCliente.vue

En este paso desarrollamos la interfaz de usuario para gestionar los clientes desde el frontend con Vue + Ionic. Esta vista permite crear, editar, eliminar y visualizar clientes que vienen del backend.



## 1. Importaciones

```
import { ref, onMounted } from 'vue'
import {
  IonPage,
  IonHeader,
  IonToolbar,
  IonTitle,
  IonContent,
  IonCard,
  IonCardHeader,
  IonCardTitle,
  IonCardContent,
  IonItem,
  IonLabel,
  IonInput,
  IonToggle,
  IonButton,
  IonList,
  toastController,
} from '@ionic/vue'
```

Se cargan componentes de **Ionic**, funciones para el CRUD de clientes y tipos necesarios.

Como componentes visuales de Ionic (como botones, formularios, tarjetas),

## 2. Variables reactivas

```
const cliente = ref<ClienteData>({ nombre: '', telefono: 0, status: true })
const clientes = ref<(ClienteData & { id: number })[]>([])
const editando = ref(false)
const idEditando = ref<number | null>(null)
```

Almacenan el cliente actual, la lista completa de clientes y si estamos **editando o creando** uno nuevo.

### 3. Toast de notificación

```
const mostrarToast = async (mensaje: string, color: 'success' | 'danger') => {  
  const toast = await toastController.create({  
    message: mensaje,  
    duration: 2000,  
    color,  
    position: 'top',  
  })  
  await toast.present()  
}
```

Muestra un mensaje flotante (toast) al usuario, útil para confirmar acciones o errores.

### 4. Cargar clientes

```
const cargarClientes = async () => {  
  try {  
    clientes.value = await getAllClientes()  
  } catch {  
    await mostrarToast('Error al cargar clientes', 'danger')  
  }  
}
```

Llama al backend para **obtener todos los clientes** y los guarda en la lista.

### 5. Guardar o actualizar cliente

```
const guardarCliente = async () => {  
  try {  
    if (editando.value && idEditando.value !== null) {  
      await updateCliente(cliente.value, idEditando.value)  
      await mostrarToast('Cliente actualizado', 'success')  
    } else {  
      await saveCliente(cliente.value)  
      await mostrarToast('Cliente guardado', 'success')  
    }  
    clearData()  
    cargarClientes()  
  } catch {  
    await mostrarToast('Error al guardar cliente', 'danger')  
  }  
}
```

Si estamos editando, actualiza el cliente. Si no, **crea uno nuevo**. Luego recarga la lista.

## 6. Editar cliente existente

```
const editarCliente = (cli: ClienteData & { id: number }) => {  
  cliente.value = { nombre: cli.nombre, telefono: cli.telefono, status: cli.status }  
  editando.value = true  
  idEditando.value = cli.id  
}
```

Rellena el formulario con los datos del cliente seleccionado para **editarlo**.

## 7. Eliminar cliente

```
const eliminarCliente = async (id: number) => {  
  try {  
    await deleteCliente(id)  
    await mostrarToast('Cliente eliminado', 'success')  
    cargarClientes()  
  } catch {  
    await mostrarToast('Error al eliminar cliente', 'danger')  
  }  
}
```

Llama al backend para **eliminar un cliente** por su ID.

## 8. Limpiar formulario

```
// Función para limpiar los datos y el formulario  
const clearData = () => {  
  cliente.value = { nombre: '', telefono: 0, status: true }  
  editando.value = false  
  idEditando.value = null  
}
```

Limpia los campos del formulario y **reinicia los estados** de edición.

## 9. Cargar clientes al iniciar

```
53   onMounted(() => {  
54     cargarClientes()  
55   })  
56 </script>  
57  
58 <style scoped>  
59   ion-card {  
60     margin-top: 20px;  
61   }  
62   ion-item {  
63     --inner-padding-end: 10px;  
64   }  
65 </style>  
66
```

Apenas se carga la vista, se **llaman los datos** desde el backend automáticamente.

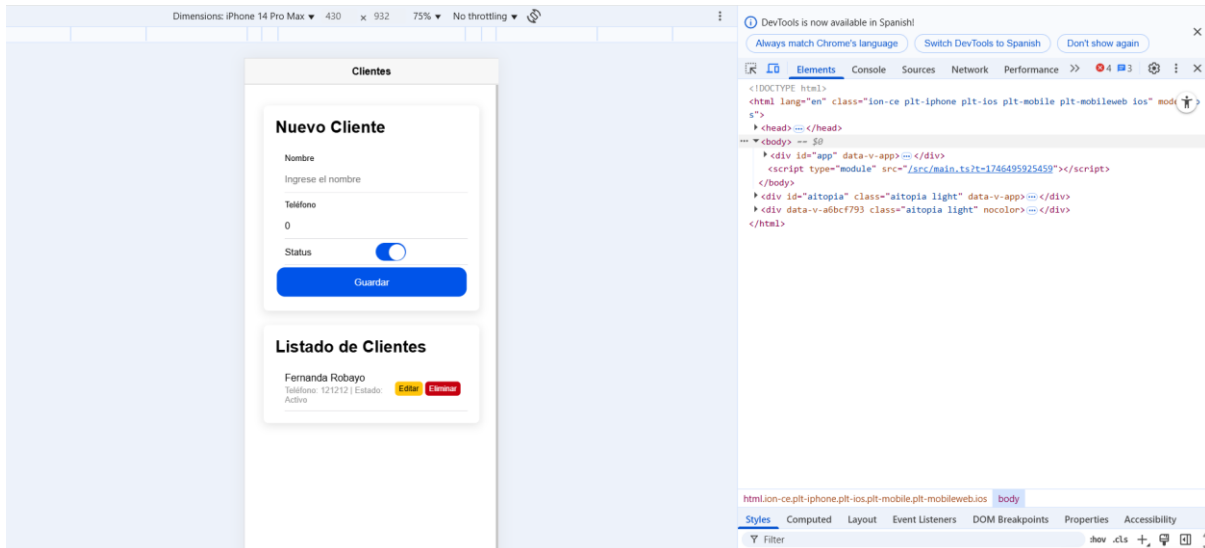
Paso 7: Inicializamos el proyecto frontend con ionic serve

```
C:\Users\lagun\Documents\Martessss\project-movil-frontend-main>ionic serve  
> vite.cmd --host=localhost --port=8100  
[vite]   → Local:   http://localhost:8100/  
[vite]   → press h + enter to show help  
  
[INFO] Development server running!  
  
Local: http://localhost:8100  
  
Use Ctrl+C to quit this process  
  
[INFO] Browser window opened to http://localhost:8100!  
  
C:\Users\lagun\Documents\Martessss\project-movil-frontend-main>ionic serve  
> vite.cmd --host=localhost --port=8100  
[vite]   → Local:   http://localhost:8100/  
[vite]   → press h + enter to show help  
  
[INFO] Development server running!  
  
Local: http://localhost:8100  
  
Use Ctrl+C to quit this process  
  
[INFO] Browser window opened to http://localhost:8100!  
  
[vite] 8:43:30 p. m. [vite] hmr update /src/views/GestionClientes.vue  
[vite] 8:43:30 p. m. [vite] hmr update /src/views/GestionClientes.vue  
[vite] 8:43:30 p. m. [vite] hmr update /src/views/GestionClientes.vue  
[vite] 8:43:31 p. m. [vite] hmr update /src/views/GestionClientes.vue  
[vite] 8:43:31 p. m. [vite] hmr update /src/views/GestionClientes.vue  
[vite] 8:45:24 p. m. [vite] hmr update /src/views/GestionClientes.vue
```

Paso 8: Vemos la vista e interactuamos con ella

Ahora que hemos levantado la aplicación con el comando ionic serve, vamos a interactuar con la vista que hemos creado y comprobar que todo funciona correctamente.

### 1. Agregamos nuevo cliente



### 2. Editamos clientes y actualizamos

