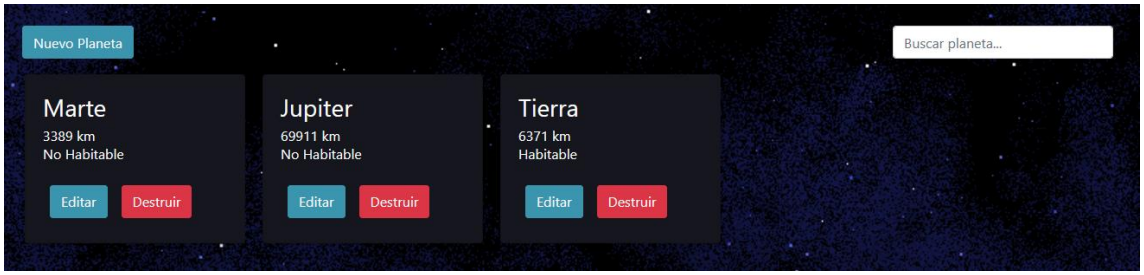
Proyecto Planeta

Este proyecto está conformado por una Api Rest desarrollado por su parte el backend en Spring haciendo uso de los cuatro tipos básicos de solicitudes HTTP implementando los métodos de controlador GET, POST, PUT y DELETE, y por otro lado el frontend usando Angular CLI 11 y Node.js consumiendo los datos de una base de datos en SQL.



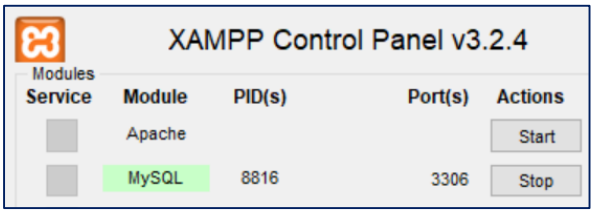
Herramientas usadas:

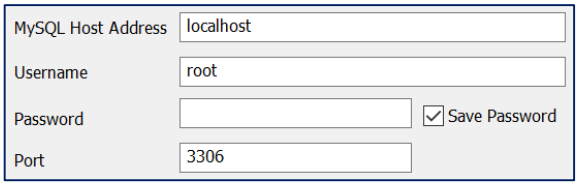
- XAMPP → para habilitar el puerto que usara el host
- SQLyog → donde tendremos nuestra base de datos
- Postman → para la verificación de los métodos del Backend
- Spring Tool Suite → donde está la lógica del Backend
- Visual Studio Code → donde está la lógica del Frontend

DATA BASE

CREACIÓN DE LA BASE DE DATOS EN SQLYOG

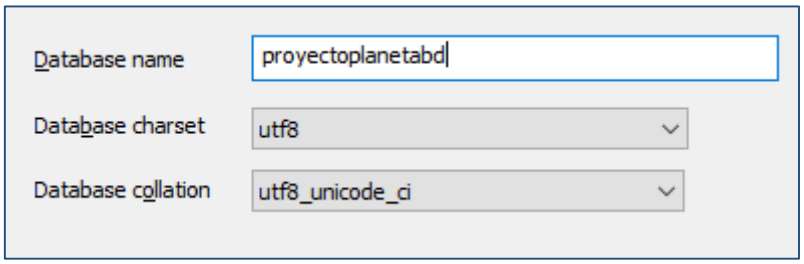
Para la conexión de la api rest con la base de datos se usara Xampp, que es un paquete de software libre, que consiste principalmente en el sistema de gestión de bases de datos MySQL, el servidor web Apache y los intérpretes para lenguajes de script PHP y Perl.





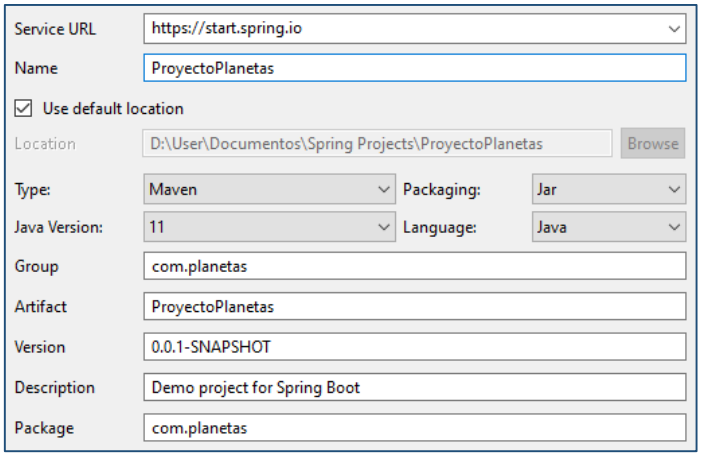
La configuración del application.properties y la del sql debe ser la misma para que funcione correctamente y pueda conectarse

Ruta por teclado Ctrl + D para crear la base de datos:



BACKEND

Construido a partir de un Spring Starter Project.



Name: corresponde al nombre del proyecto
Type: corresponde al selector de librerías, en este caso Maven
Group: com.planetas (teniendo a com que corresponde al nombre de la empresa, dominio)
Artifact corresponde al mismo nombre del proyecto
Package es donde estarán todos los archivos del proyecto que puede llevar el mismo nombre que el group.

Dependencias

- MySQL: tipo de base de datos a usar
- JPA: es una abstracción de Hibernate y permite usarlo
- JDBC: conector de la base de datos
- Rest Repositories: para facilitar la comunicación de recursos de los repositorios
- Rest Repositories HAL Browse
- Spring Web Services: facilita el contacto con servicios SOAP
- Spring REST Docs: ayuda a la documentación de servicios RESTful

Estas dependencias se encuentran en el archivo pom.xml, el cual contiene lo necesario para generar el fichero ejecutable.

Clase principal se encuentra en el paquete com.planetas y es **ApiPlanetasApplication** la cual se crea de manera automática dentro del package base del proyecto, por lo tanto, cualquier otra clase que se cree debe estar dentro de este package, en un sub package.

El aspecto que tiene por defecto es:

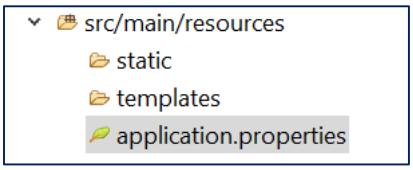
```java
@SpringBootApplication
public class ApiPlanetasApplication {

    public static void main(String[] args) {
        SpringApplication.run(ApiPlanetasApplication.class, args);
    }

}
```

La anotación @SpringBootApplication está conformada por tres anotaciones:

- **@SpringBootConfiguration** → que es la configuración automática que se puede sobrescribir en el application.properties
- **@EnableAutoConfiguration** → para habilitar dicha configuración
- **@ComponentScan** → que permite buscar y registrar todas las clases y componentes en el contenedor anotadas con @RestController, @Controller, @Component, @Repository y @Service



Configuración del application.properties, que es el archivo principal de configuración:

```
spring.datasource.url=jdbc:mysql://localhost:3306/proyectoplanetabd?useUnicode=true&useJDBCCompliantTimezoneShift=true&useLegacyDatetimeCode=false&serverTimezone=UTC
spring.datasource.username=root
spring.datasource.password=

spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto=update
spring.jpa.database-platform=org.hibernate.dialect.MySQL57Dialect

server.port=8080
```

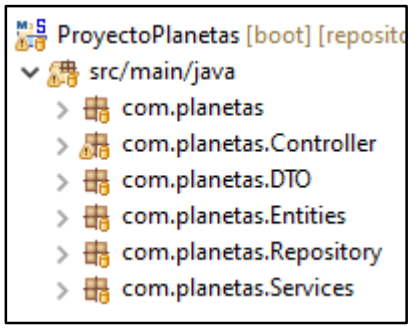
spring.datasource.url → dirección o posición de la base de datos y el gestor a usar
Para solucionar un error de horario por estar en Argentina (unicode). Esto le cambia el timeZone de Argentina por el UTC (el global).
(?useUnicode=true&useJDBCCompliantTimezoneShift=true&useLegacyDatetimeCode=false&serverTimezone=UTC)
dialecto, tipo de base de datos, dirección, nombre de la base de datos

- **spring.datasource.username** → usuario de la base de datos
- **spring.datasource.password** → contraseña de la base de datos
- **spring.jpa.show-sql** → configuración de hibernate, para hacerle seguimiento a los procesos mostrándolos. Es un boolean, se activa o desactiva. Se muestra en consola las query que se estén usando
- **spring.jpa.hibernate.ddl-auto** → configura la forma que se crearan las tablas de la base de datos. Con update, las crea levantado el proyecto y luego las va actualziando

- **spring.jpa.database-platform** → tipo de comunicación que tendrá con la base de datos, determina dialecto de la base de datos. La plataforma jpa que se está usando: MySQL5InnoDBDialect, está obsoleta (deprecated), esto se puede ver yendo a las dependencias de maven >> hibernate-core >> org.hibernate.dialect >> MySQL5InnoDBDialect. Debido a esto lo mejor es usar otra version, ejemplo: MySQL57Dialect, la cual tiene soporte para InnoDBStorageEngine.
- **server-port** → configuración del puerto, no es obligatorio. Se usa en el caso de conflicto de puertos.

Implementación de Spring Boot MVC

PAQUETES



Usando este patrón de diseño, se crean sub package para organizar los archivos. Teniendo al Controller como intermediario entre la interfaz y la lógica de negocio. Al Entity que son las clases que definen los objetos donde se recuperan y guardan los datos de la base. El Service donde está el algoritmo de la lógica de negocio. El Repository que cumple como interfaz entre el usuario final y la aplicación

Entity

Se anotan los atributos privados, constructor completo y vacío, setters y getters. Luego se decora para que Spring sepa que se trata de una clase Entity:
@Entity → decorador o anotación que indica a Spring que es una clase entidad de JPA
@Table(name="Personas") → indica el nombre que tendrá la tabla en la base de datos
@Id → indica que el id corresponde a la clave primaria de la clase (primary key)
@GeneratedValue(strategy = GenerationType.IDENTITY) → indica como se genera esta primary key en la base de datos. Es una estrategia de generacion. Identity es para que se generen de manera incremental
@Column(name="Nombre_del_Atributo") → da el nombre de la columna del atributo, por defecto si no se coloca queda con el nombre del atributo. Se le puede agregar unique = true, para que no se repita el nombre.

```
@Entity
@Table(name="Planetas")
public class PlanetaEntity {

	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	private int id;

	@Column(name="NombrePlaneta")
	private String name;

	@Column(name="TamañoPlaneta")
```

```
	private Long size;

	@Column(name="VidaEnPlaneta")
	private boolean habitable;

	//Constructors
	public PlanetaEntity() { }
	public PlanetaEntity(int id, String name, Long size, boolean
habitable) {
		this.id = id;
		this.name = name;
		this.size = size;
		this.habitable = habitable;
	}

	//Getters & Setters
	public int getId() {
		return id;
	}
	public void setId(int id) {
		this.id = id;
	}

	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}

	public Long getSize() {
		return size;
	}
	public void setSize(Long size) {
		this.size = size;
	}

	public boolean getHabitable() {
		return habitable;
	}
	public void setHabitable(boolean habitable) {
		this.habitable = habitable;
	}


}
```

DTO

Data Transfer Object. Encapsulamiento de la entidad
Luego a la clase se le implementa la interfaz Serializable, que permite la serialización de objetos. Esto es convertir cualquier objeto en una secuencia de bytes para poder posteriormente ser leídos así restaurar el objeto original. Al importarlo requiere que se aplique el atributo necesario para que se implemente adecuadamente la serialización.

```
public class PlanetaDTO implements Serializable {

	private static final long serialVersionUID = 1L;
```

```
	private int id;
	private String name;
	private Long size;
	private boolean habitable;

	//Constructors
	public PlanetaDTO() {}
	public PlanetaDTO(int id, String name, Long size, boolean habitable) {
		this.id = id;
		this.name = name;
		this.size = size;
		this.habitable = habitable;
	}

	//Getters & Setters
	public int getId() {
		return id;
	}
	public void setId(int id) {
		this.id = id;
	}

	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}

	public Long getSize() {
		return size;
	}
	public void setSize(Long size) {
		this.size = size;
	}

	public boolean getHabitable() {
		return habitable;
	}
	public void setHabitable(boolean habitable) {
		this.habitable = habitable;
	}


}
```

REPOSITORY

Clase de acceso a datos. Su función es acceder, hacer consultas y operaciones en la base de datos. Interactúa entre la base de datos y el service a través del uso del entity. Es una clase DAO (Data Access Object). Spring Data JPA se hereda a través del Repository, de modo que nos habilita todos los métodos CRUD que necesitamos para hacer las peticiones. También se puede implementar métodos propios customizados a través de anotaciones **@Query**

Genérico: **<clave, valor>**

<T, ID> correspondiendo T para clave: clase PersonaEntity que llevara los datos que se guardan o adquieren de la base de datos. Y ID, correspondiendo al tipo de valor de la id de dicha clase. (Por alguna razon no puedo ver la clase JpaRepository, por lo tanto, en vez de esa clase se podría usar el CrudRepository...)

Jpa administra la persistencia de la base de datos. Ahorra la escritura de las query manualmente. Es una especificación creada para acceder y hacer el mapeo entre los objetos y la base de datos. Define la especificación, las interfaces y las anotaciones que permiten hacer eso.

```
public interface PlanetaRepository extends JpaRepository<PlanetaEntity,
Integer> {

}
```

SERVICE

En el fondo puede contener las clases DAO y dentro de sus métodos podrían colaborar o interactuar diferentes DAO, realizando consultas a diferentes tablas, todo bajo una misma transacción. Estos métodos se envuelven en una misma transacción por lo que pueden trabajar juntos distintos DAOs. La función del Service es evitar ensuciar el Controller con la clase DAO de forma directa. Por lo tanto, el service maneja los datos traídos desde la base de datos.
Es importante usar el bean **@Service** para que Spring sepa que se está haciendo con esta clase. Se puede utilizar una interfaz para universalizar los métodos. Se lo puede hacer de tipo genérico asignando un objeto genérico tipo, ejemplo, T en el caso de trabajar con diversos dtos.

```
public interface ObjectServices <T> {
	public List<T> findAll() throws Exception;
	public T findById(int id) throws Exception;
	public T save(T t) throws Exception;
	public T update(T t, int i) throws Exception;
	public boolean delete(int id) throws Exception;
}
```

Es decir que el service tendrá que implementar esta interfaz y en vez de usar tipo T, usara el dto correspondiente, ya que depende de lo que se recibe y devuelve al controlador.

Son por lo tanto tres capas para interactuar con la base de datos: el repositorio interactúa con la base de datos, el servicio maneja los datos que se traen de la base de datos y el controlador que recibe y envía datos al exterior.

```
@Service
public class PlanetaService implements ObjectServices <PlanetaDTO> {

	private PlanetaRepository repository;

	//Constructor
	public PlanetaService(PlanetaRepository repository) {
		this.repository = repository;
	}
```

```
	//Methods

	//----------FindAll----------
	@Transactional
	public List<PlanetaDTO> findAll() throws Exception {
		List<PlanetaEntity> planetas = repository.findAll();
		List<PlanetaDTO> listaDTO = new ArrayList<PlanetaDTO> ();

		try {
			for (PlanetaEntity planeta : planetas) {

				PlanetaDTO dto = new PlanetaDTO();

				dto.setId(planeta.getId());
				dto.setName(planeta.getName());
				dto.setSize(planeta.getSize());
				dto.setHabitable(planeta.getHabitable());

				listaDTO.add(dto);
			}
			return listaDTO;

		} catch (Exception e) { throw new Exception(); }
	}

	//----------FindById----------
	@Transactional
	public PlanetaDTO findById(int id) throws Exception {

		Optional<PlanetaEntity> planetaOptional =
repository.findById(id);

		try {
			PlanetaDTO dto = new PlanetaDTO();
			PlanetaEntity planeta = planetaOptional.get();

			dto.setId(id);
			dto.setName(planeta.getName());
			dto.setSize(planeta.getSize());
			dto.setHabitable(planeta.getHabitable());

			return dto;

		} catch (Exception e) { throw new Exception(); }
	}

	//----------Save----------
	@Transactional
	public PlanetaDTO save(PlanetaDTO dto) throws Exception {

		PlanetaEntity planeta = new PlanetaEntity();

		planeta.setId(dto.getId());
		planeta.setName(dto.getName());
		planeta.setSize(dto.getSize());
		planeta.setHabitable(dto.getHabitable());

		try {
```

```
                planeta = (PlanetaEntity) repository.save(planeta);
                dto.setId(planeta.getId());
                return dto;

            } catch (Exception e) { throw new Exception(); }
      }

      //----------Update----------
      @Transactional
      public PlanetaDTO update(PlanetaDTO dto, int id) throws Exception {

            Optional<PlanetaEntity> planetaOptional =
repository.findById(id);

            try {
                  PlanetaEntity planeta = planetaOptional.get();

                  planeta.setId(id);
                  planeta.setName(dto.getName());
                  planeta.setSize(dto.getSize());
                  planeta.setHabitable(dto.getHabitable());

                  repository.save(planeta);
                  dto.setId(planeta.getId());

                  return dto;
            } catch (Exception e) { throw new Exception();}
      }

      //----------Delete----------
        @Transactional
        public boolean delete(int id) throws Exception{
              try {
                    if(repository.existsById(id)) {
                          repository.deleteById(id);
                          return true;
                    } else {
                          throw new Exception();
                    }
              } catch (Exception e) { throw new Exception();} }

}
```

CONTROLLER

El controlador se encarga de comunicar el service con el exterior (cliente). Devuelve un ResponseEntity con el estado http correspondiente según la respuesta del método service al que llame. En este caso se usarán tres anotaciones: **@RestController**, **@RequestMapping** con el path que el puntero de destino de la base de datos y **@CrossOrigin**. El Controller entonces recibe y envía datos al exterior

```
@RestController
@RequestMapping("api/v1/planetas")
@CrossOrigin(origins = "*" , methods = {RequestMethod.GET,
RequestMethod.POST, RequestMethod.PUT, RequestMethod.DELETE})
public class PlanetaController implements ObjectController<PlanetaDTO> {

      PlanetaService service;
```

```
	//Constructor
	public PlanetaController(PlanetaService service) {
		super();
		this.service = service;
	}

	//Methods

	//----------GetAll----------
	@GetMapping("/")
	@Transactional
	public ResponseEntity getAll() {
		try {
			return
ResponseEntity.status(HttpStatus.OK).body(service.findAll());
		} catch (Exception e) {
			return
ResponseEntity.status(HttpStatus.NOT_FOUND).body("Error al buscar
registros");
		}
	}

	//----------GetOne----------
	@GetMapping("/{id}")
	@Transactional
	public ResponseEntity getOne(@PathVariable int id) {
		try {
			return
ResponseEntity.status(HttpStatus.OK).body(service.findById(id));
		} catch (Exception e) {
			return
ResponseEntity.status(HttpStatus.NOT_FOUND).body("Registro no encontrado");
		}
	}

	//----------Post----------
	@PostMapping("/")
	@Transactional
	public ResponseEntity post(@RequestBody PlanetaDTO dto) {
		try {
			return
ResponseEntity.status(HttpStatus.OK).body(service.save(dto));
		} catch (Exception e) {
			return
ResponseEntity.status(HttpStatus.BAD_REQUEST).body("Error al crear");
		}
	}

	//----------Put----------
	@PutMapping("/{id}")
	@Transactional
	public ResponseEntity put(@RequestBody PlanetaDTO dto, @PathVariable
int id) {
		try {
			service.update(dto, id);
			return ResponseEntity.status(HttpStatus.OK).body("Registro
actualizado");
		} catch (Exception e) {
```

```
                  return
ResponseEntity.status(HttpStatus.BAD_REQUEST).body("Error al actualziar
registro");
            }
      }

      //----------Delete----------
        @DeleteMapping("/{id}")
        @Transactional public ResponseEntity delete(@PathVariable int id) {
              try {
                    service.delete(id);
                    return
ResponseEntity.status(HttpStatus.OK).body("Registro eliminado");
              } catch (Exception e) {
                    return
ResponseEntity.status(HttpStatus.BAD_REQUEST).body("Error al eliminar
registro");
              }
        }

}
```

@RestController → para indicar que es un controlador de tipo Rest
@RequestMapping → para mapear el RestController, con esta anotacion se genera el url (endpoint)
@CrossOrigin → "origins = *" hace referencia a la url que se especificó antes. Esto permite que el browser maneje de forma segura las solicitudes http de origen cruzado de un cliente cuyo origen sería, en este caso, el localhost.

También se le puede implementar una interfaz del tipo genérico

Se crea un objeto service a partir de la clase service para recibir los estados http.
El ResponseEntity maneja las respuestas http entre el cliente y el servidor. Maneja tanto la cabecera, cuerpo y códigos de estado de las respuestas permitiendo total libertad para configurar lo que se quiere enviar desde los endpoints.

```
public interface ObjectController <T> {
      public ResponseEntity <T> getAll();
      public ResponseEntity <T> getOne(int id);
      public ResponseEntity <T> post(T t);
      public ResponseEntity <T> put(T t, int id);
      public ResponseEntity <T> delete(int id);
}
```

PRUEBA DEL BACKEND CON POSTMAN

Para empezar… correr la aplicación. Verificar que se este conectada con la base de datos y que una vez haya iniciado la aplicación, se hayan creado las tablas.

En este caso, el url estadar seria: localhost:8080/api/v1/planetas/

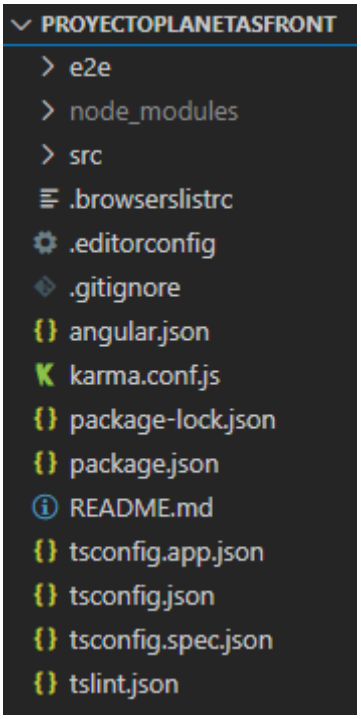
En donde se define como localhost en puerto 8080 (ya indicado desde el application.properties), api como nombre de la aplicación, v1 como la versión y planetas como el tipo de dato en plural al usa en el objeto PlanetaController.

FRONTEND

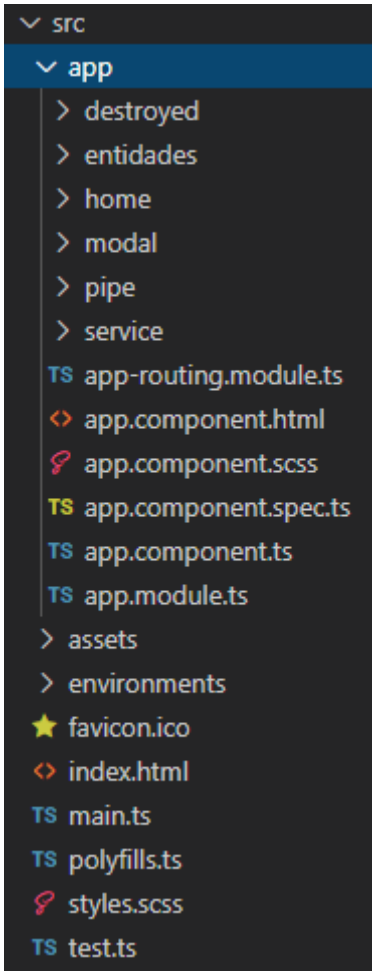
Para crear el proyecto nuevo, comando CLI: ng new proyectname a través de la consola, la cual se accede con Ctrl+ñ
Angular routing, indicamos que no para hacerlo manualmente
El formato de estilo seleccionado es SCSS

Una vez creado el proyecto, se lo levanta el programa usando el comando: ng serve --open
o sino su abreviatura: ng serve -o

En el caso de que sugiera la instalación de pluggings, le damos a okey para que instale todos los necesarios. Por ejemplo el linter, necesario para detectar errores sintácticos en tiempo de desarrollo



e2e → directorio que contiene lo necesario para realizar testing.
node_modules → librerias y dependencias, manejado de manera automática. Todo se registra dentro del package.json
package.json → contienen las dependencias. Es como el pom de Spring
src → carpeta que contiene el codigo fuente
.editorconfig → contienen la configuración del editor
.gitignore → archivo de github del repositorio, donde se puede subir y compartir el proyecto, permite ignorar archivos o carpetas que no se quieran compartir (ejemplo: /dist directorio que contiene el codigo compilado, lo cual se lo ignora con la /) Para esto hay que activarlo yendo a la configuración del IDE: ctrl + ; En package se busca el componente tree view, en settings, por defecto aparece desactivado Hide VCS Ignored Files. Al activarlo desaparece de forma automática todo lo indicado como "ignorado" en este directorio. Sino, en vez de ir a package, se va a Core donde esta Ignored Names, allí se anota el nombre del directorio que se quiera ocultar.
angular.json → archivo principal de configuración de proyecto



app → carpeta con todo lo necesario del componente principal
app.component.css → hoja de estilo exclusivos para el app.component
app.component.html → plantilla o vista que responde al app.component.ts, contiene el html
app.component.spec.ts → para pruebas unitarias
app.component.ts → es una clase que representa una parte de la aplicación web. Similar al controller de Spring. Para mostrar el contenido se usa el selector (en el lugar donde se use la etiqueta), que esta asociado también a una plantilla (vista, template)
app.modules.ts → repositorio donde se registran los componentes, clases de servicio, módulos, configuraciones de la base de datos
assets → donde se guardan los contenidos estáticos de la aplicación
environment.ts → ambientes de desarrollo y producción
favicon.ico → icono que se muestra en el navegador
index.html → es la puerta de entrada a la aplicación. En el body se incluye la etiqueta html app-root que contiene el nombre del selector del app.component.ts (en este caso: app-root).
main.ts → clase principal que arranca y levanta la aplicación (AppModule)
polyfills.ts → archivo de configuración para aumentar la compatibilidad de la aplicación
styles.css → hojas de estilos de uso global, para todos los componentes
test.ts → para pruebas unitarias

Las demás carpetas son nuestros componentes creados para el funcionamiento de la api

Instalación de Bootstrap en Angular
Librerias CSS y JS en nuestro index.html

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Planetas</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">

  <script src="https://code.jquery.com/jquery-
3.4.1.slim.min.js" integrity="sha384-
J6qa4849blE2+poT4WnyKhv5vZF5SrPo0iEjwBvKU7imGFAV0wwj1yYfoRSJoZ+n" crossor
igin="anonymous"></script>
```

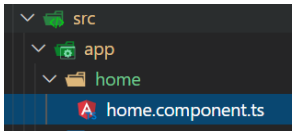
```
  <script src="https://cdn.jsdelivr.net/npm/popper.js@1.16.0/dist/umd/popper.min.js" integrity="sha384-Q6E9RHvbIyZFJoft+2mJbHaEWldlvI9IOYy5n3zV9zzTtmI3UksdQRVvoxMfooAo" crossorigin="anonymous"></script>
  <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.4.0/css/bootstrap.min.css" integrity="sha384-SI27wrMjH3ZZ89r4o+fGIJtnzkAnFs3E4qz9DIYioCQ5l9Rd/7UAa8DHcaL8jkWt" crossorigin="anonymous">

</head>
<body>
  <app-root></app-root>

  <script src="https://stackpath.bootstrapcdn.com/bootstrap/4.4.0/js/bootstrap.min.js" integrity="sha384-3qaqj0lc6sV/qpzrc1N5DC6i1VRn/HyX4qdPaiEFbn54VjQBEU341pvjz7Dv3n6P" crossorigin="anonymous"></script>

</body>
</html>
```

Creación de componentes manualmente



Los diferentes componentes sirven para mantener organizado el código. En este caso, se crea una carpeta donde colocar los nuevos componentes. Una vez creado el nuevo componente se debe registrar en el app.modules.ts para que angular conozca de su existencia. Por ejemplo, home.component.ts

```
home.component.html   home.component.ts ×   index.html Angular P
ular Projects > ApiPlaneta > src > app > home > home.component.ts > ...
import { Component } from '@angular/core';

@Component({
    selector: 'app-home',
    template: `<h1> template del home.component.ts </h1>`
})

export class HomeComponent {

}
```

Y para que el contenido del nuevo componente aparezca en web, se utiliza la etiqueta del selector escrito dentro del codigo del home.componen.ts

```
import { HomeComponent } from './home/home.component';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    HomeComponent
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Dentro del @NgModule se encuentra:
imports → que es para los módulos
providers → que es para los services
declarations → que es para los componentes
exports → que son los elementos que son compartidos con otros módulos

Para la vista del nuevo componente se crea un html con la misma nomenclatura donde se guardara el codigo que se verá en la pagina web, es decir: home.component.html . Para que el mismo aparezca, la dirección de dicha plantilla debe estar en el templateUrl del componente al que está asociado, es decir agregarlo dentro del @Component del home.component.ts . En el caso de que sea innecesario crear una vista html para un componente, se puede agregar crudamente en un template.

```
@Component({
    selector: 'app-home',
    templateUrl: './home.component.html',
    //template: `<h1> template del home.component.ts </h1>`
})
```

@Component → decorador o anotación con configuración meta-data, para configurar
selector → define el nombre por el cual se identifica el componente, corresponde a una etiqueta html que se incluye en el index
templateUrl → usa como plantilla el archivo .html (url). Es la vista a la que esta asociada
styleUrls → mismo ^ pero con .css que son las hojas de estilo, se puede tener una o mas (separados con ",")

Creación de componentes a través de consola
También se los puede llamar clases. Se accede al lugar donde se quiera crear la nueva carpeta ingresando cd + el nombre de la carpeta hasta llegar a la dirección. Para crear una carpeta nueva se ingresa mkdir + el nombre de la carpeta. Luego para la creación de la clase se ingresa ng g c + nombre del componente
Esto generara todas las clases relacionadas, es decir: la vista, la clase de estilo, la de pruebas unitarias y la de código.
Sino la creación de una clase se hace con ng generate class <nombre de la clase>
Para que la vista de home se muestre en el app.component.html se agrega el selector en la vista del app component, de manera:

```
<app-home></app-home>
```

app.module.ts

```
@NgModule({
  declarations: [
    AppComponent,
    HomeComponent,
    ModalComponent,
    FilterPipe,
    DestroyedComponent
  ],
  imports: [
    BrowserModule,
    HttpClientModule,
    FormsModule,
    ReactiveFormsModule,
    AppRoutingModule
  ],
```

```
  providers: [PlanetaService, ModalComponent],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

app-routing.module.ts

```
const routes: Routes = [
  {path: '', component: HomeComponent},
  {path: '*', pathMatch: 'full', redirectTo: ''}
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

app.component.ts

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss']
})
export class AppComponent {
  title = 'ProyectoPlanetasFront';
}
```

app.component.html

```
<div class="fondo p-5">
  <router-outlet></router-outlet>
</div>
```

app.component.scss

```
.fondo{
    //height:40em; width:auto;
    background-size:cover;
    background-image:url('https://images-wixmp-
ed30a86b8c4ca887773594c2.wixmp.com/f/77801cc4-7cda-4520-b205-
449f4531e4e1/d557zhe-d91204ff-c32f-4e9b-b04c-
bb15e49168af.png?token=eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJ1c
m46YXBwOiIsImlzcyI6InVybjphcHA6Iiwib2JqIjpbW3sicGF0aCI6IlwvZlwvNzc4MDFjYz
QtN2NkYS00NTIwLWIyMDUtNDQ5ZjQ1MzFlNGUxXC9kNTU3emhlLWQ5MTIwNGZmLWMzMmYtNGU
5Yi1iMDRjLWJiMTVlNDkxNjhhZi5wbmcifV1dLCJhdWQiOlsidXJuOnNlcnZpY2U6ZmlsZS5k
b3dubG9hZCJdfQ.g6nnhOGSAlDEv-kPYU-IURS0whSV-1VVulXSVIDMrbU');
    background-position:50% 50%;
    width: 100%;
    height: 100%;
```

```
}
```

planeta.ts

```
export class Planeta{
    id!:number;
    name!:String;
    size!:number;
    habitable!:boolean;

}
```

planeta.service.ts

```
@Injectable({
    providedIn: 'root'
})

export class PlanetaService {

    planeta: Planeta = new Planeta(); //validaciones

    url="http://localhost:8080/api/v1/planetas/";

    constructor(private http:HttpClient) {}

    getAll():Observable<Planeta[]> {
        return this.http.get<Planeta[]>(this.url);
    }

    getOne(id:number):Observable<Planeta> {
        return this.http.get<Planeta>(this.url+id);
    }

    post(planeta:Planeta):Observable<Planeta> {
        console.log(planeta);
        return this.http.post<Planeta>(this.url, planeta);
    }

    put(id:number, planeta:Planeta):Observable<Planeta> {
        console.log(planeta);
        return this.http.put<Planeta>(this.url+id, planeta);
    }

    delete(id:number):Observable<Planeta> {
        return this.http.delete<Planeta>(this.url+id);
    }
}
```

home.component.ts

```ts
@Component({
  selector: 'app-home',
  templateUrl: './home.component.html',
  styleUrls: ['./home.component.scss']
})
export class HomeComponent implements OnInit {

  //variable para pipe
  filterPost = '';

  //variables para modal
  myModal=false;
  idModal!:number;

  //variables para destroyed
  myDestroy=false;

  //objeto planetas
  planetas: Planeta[] = [];

  constructor (private service: PlanetaService) {}

  ngOnInit(): void {
    this.getAll();
  }

  //-------- Metodo ~ Trae todos los datos del Backend
  getAll() {
    this.service.getAll().subscribe(
      data => {
        this.planetas = data;
      }
    );
  }

  //-------- Metodo ~ Elimina carta correspondiente
  delete(id:number) {
      this.myDestroy=true;
      this.service.delete(id).subscribe(
        data => {
          console.log(data);
        }
      );
  }

  //-------- Metodo ~ Despliega el modal
  edit(id:number){
```

```
    this.myModal=true;
    if(id != 0){
      console.log(id);
      this.idModal = id;
    } else {
      this.idModal = 0;
    }
  }

  d(id:number){
    this.myDestroy=true;
    this.idModal = id;
  }

  //-------- Metodo ~ Cierra el modal
  closeModal(e:any){
    this.myModal = e; //= false --> ya que llega desde el modal
    this.myDestroy = e;
  }



}
```

home.component.html

```
<!-- Buscador de Planetas -->
<div class="buscador">
    <form class="form-group">
        <input class="form-control margen-top-
5" type="text" placeholder="Buscar planeta..." style="width: 255px;"
            name="filterPost" [(ngModel)]="filterPost">
    </form>
</div>


<!-- Boton Nuevo Planeta -->
<button class="btn btn-info mb-
2" (click)="edit(0)">Nuevo Planeta</button>

<div class="ml-2">
    <!-- Tabla de Cartas -->
    <div class="row">
        <!-- creamos una fila-->
        <div class="col-xs-
4" *ngFor="let p of planetas | filter:filterPost">
            <!--creamos columna-->
            <div class="card">
                <div class="card-body">
                    <h3 class="card-title"> {{p.name}} </h3>
```

```
                    <h6 class="card-subtitle"> {{p.size}} km </h6>
                    <p class="card-
text" *ngIf="p.habitable == true"> Habitable </p>
                    <p class="card-
text" *ngIf="p.habitable == false"> No Habitable </p>
                    <button class="btn btn-info m-
2" (click)="edit(p.id)">Editar</button>
                    <button class="btn btn-danger m-
2" (click)="delete(p.id)">Destruir</button>
                </div>
            </div>
        </div>
    </div>
</div>

<!-- App Modal -->
<div *ngIf="myModal===true">
    <app-
modal [visible]="myModal" [getId]="idModal" (close)="closeModal($event)">
 </app-modal>
</div>

<!-- App Destroyed -->
<div *ngIf="myDestroy===true">
    <app-
destroyed [vDes]="myDestroy" (close)="closeModal($event)"> </app-
destroyed>
</div>
```

home.component.scss

```
.btn-info {
    color: #ffffff;
    background-color: #3c95af;
    border-color: #3c95af
}

.close {
    color: #ffffff;
}

.cruz{
    position:absolute; //creo que era para que se pudiera mover dentro de
l modal sin limitaciones de otros elementos
    top:0;
    right:5px;
    cursor:pointer;
    font-size: 1.3rem;
    &:hover{ //colorcito al que cambia la x cuando se pasa el mouse por e
ncima
```

```
        color: orange;
    }
}

.buscador {
    position:absolute;
    top:48px;
    right:50px;
}


.card {
    box-shadow: 0 4px 8px 0 rgba(0, 0, 0, 0.2);
    display: inline-block;
    width: 255px;
    margin: 10px;
    color: #ffffff;
    background-color: #16171e;

    //text-align: center; //centra el contenido de la carta
}
```

filter.pipe.ts

```
@Pipe({
    name: 'filter'
})

export class FilterPipe implements PipeTransform {
    transform(planetas: Planeta[], texto: string): Planeta[] {

        if (texto.length === 0) {
            return planetas;
        }
        texto = texto.toLocaleLowerCase();
        return planetas.filter(
            planeta => {
                return planeta.name.toLocaleLowerCase().includes(texto);
            }
        );
    }
}
```

modal.component.ts

```
@Component({
  selector: 'app-modal',
  templateUrl: './modal.component.html',
```

```
  styleUrls: ['./modal.component.scss']
})
export class ModalComponent implements OnInit {

  //funcionalidades del modal
  @Input() visible!: boolean;
  @Output() close: EventEmitter<boolean> = new EventEmitter();

  //variable id auxiliar
  @Input() getId!: number;

  //objeto planeta
  planeta: Planeta = new Planeta();

  //para formularios
  formularioCreado!:FormGroup;

  constructor(
    private service: PlanetaService,
    private formBuilder: FormBuilder
  ) { }

  ngOnInit(): void {

    this.crearFormulario(); //mm no se che

    if(this.getId != 0){
      this.getOne(this.getId);
    }

  }

  //-------- Metodo ~ Trae el registro elegido
  getOne(id: number) {
    this.service.getOne(id).subscribe(data => {
      this.planeta = data;
      console.log(this.planeta);
    });
  }

  //-------- Metodo ~ Cierra el modal
  cancelar(){
    this.close.emit(false);
    location.reload();
  }

  //-------- Metodo ~ Guarda el contenido del modal
  save(id:number){
    this.service.post(this.planeta).subscribe(
```

```
      data => {
        //this.validator();
        console.log(data);
        if(id===0){
          alert("Planeta guardado con exito!");
          this.formularioCreado.reset();
        } else {
          alert("Planeta actualizado con exito!");
        }
      }
    );
  }

  //-------- Metodo ~ Validaciones
  crearFormulario(){
    this.formularioCreado = this.formBuilder.group(
      {
        name: ["", Validators.compose(
          [Validators.required, Validators.pattern('[0-9a-zA-Z ]*')]
        )],
        size: ["", Validators.compose(
          [Validators.required, Validators.pattern('[0-9]*')]
        )]
      }

    );
  }



}
```

modal.component.html

```
<div class="modal" *ngIf="visible" tabindex="-1" role="dialog">
    <div class="contenedor">
        <div>
            <div>
                <div class="modalHeader">
                    <h4 *ngIf="getId != 0">Editar: {{planeta.name}}</h4>
                    <h4 *ngIf="getId === 0">Nuevo Planeta</h4>
                    <span (click)="cancelar()">x</span>
                </div>

                <div class="modalContent">

                    <div [formGroup]="formularioCreado">
                        <div class="input-group mb-3">
                            <span style="width: 90px;" class="input-
group-text" id="basic-addon1">Nombre</span>
```

```
                            <input type="text" class="form-
control" formControlName="name" [(ngModel)]="planeta.name">
                            <div class="alert alert-danger" role="alert"
                                *ngIf="formularioCreado.controls['name'].
invalid && formularioCreado.controls['name'].dirty">
                                Nombre no valido
                            </div>
                        </div>

                        <div class="input-group mb-3">
                            <span style="width: 90px;" class="input-
group-text" id="basic-addon1">Tamaño</span>
                            <input type="text" class="form-
control" formControlName="size" [(ngModel)]="planeta.size">
                            <div class="alert alert-danger" role="alert"
                                *ngIf="formularioCreado.controls['size'].
invalid && formularioCreado.controls['size'].dirty">
                                Tamaño no valido
                            </div>
                        </div>
                    </div>

                    <div class="input-group mb-3">
                        <span style="width: 90px;" class="input-group-
text mr-3" id="basic-addon1">Habitable</span>
                        <p class="font-weight-bold">Si</p>
                        <input class="mr-2 ml-
2" type="radio" name="habitable" [(ngModel)]="planeta.habitable" [value]=
"true">
                        <p class="font-weight-bold">No</p>
                        <input class="mr-2 ml-
2" type="radio" name="habitable" [(ngModel)]="planeta.habitable" [value]=
"false">
                    </div>

                    <div class="modal-footer">
                        <button class="btn btn-
info" (click)="save(getId)"
                            [disabled]="!formularioCreado.valid">Guardar<
/button>
                        <button class="btn btn-
info" (click)="cancelar()">Cerrar</button>
                    </div>
                </div>

            </div>
        </div>
    </div>
</div>
```

modal.component.scss

```scss
:host {
    .modal {
        display: table;
        width: 100%;
        height: 100%;
        background-color: rgba(0, 0, 0, .4); //transparencia negra
    }
    .contenedor {
        display: table-cell;
        vertical-align: middle; //lo centra a nivel vertical
        & > div {
            background-
image:url( 'https://bestanimations.com/uploads/gifs/1184781408saturn-
planet-animation-11.gif' );
            background-size: 500px;
            width: 300px;
            height: 300px;
            color: #ffffff;
            background-color: #16171e;
            margin: 0 auto; //lo centra a nivel horizontal
            padding: 1rem; //marquito para interior del modal
            border-radius: 5px; //vertices redondeadas
            box-shadow: 0 0 20px rgba(0, 0, 0, .4);
        }
    }
    .modalHeader {
        position:relative;
        h2{
            margin:0;
        }
        span{
            position:absolute; //creo que era para que se pudiera mover d
entro del modal sin limitaciones de otros elementos
            top:0;
            right:0;
            cursor:pointer;
            font-size: 1.3rem;
            &:hover{ //colorcito al que cambia la x cuando se pasa el mou
se por encima
                color: orange;
            }
        }
    }
    .modalContent {
        padding-top: 1rem; //margen de arriba
    }
    .button {
```

```
        margin: 3px;
    }
    .radioBox {
        position:absolute;
    }

    .alert {
        color: #ffffff;
        border-color: #dd4343;
        background-color: #dd4343;
        position:absolute;
        margin-top: 50px;
        z-index:5;
    }
}
```

destroyed.component.ts

```
@Component({
  selector: 'app-destroyed',
  templateUrl: './destroyed.component.html',
  styleUrls: ['./destroyed.component.scss']
})
export class DestroyedComponent implements OnInit {

  //funcionalidades del modal
  @Input() vDes!: boolean;
  @Output() close: EventEmitter<boolean> = new EventEmitter();

  constructor() { }

  ngOnInit(): void { }

  //-------- Metodo ~ Cierra el modal
  cancelar(){
    this.close.emit(false);
    location.reload();
  }
}
```

destroyed.component.html

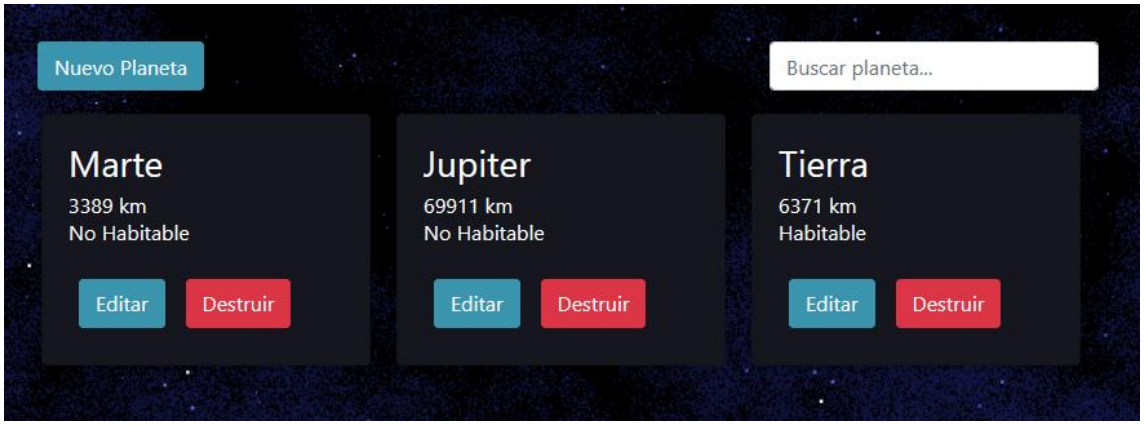
```
<div class="modal" *ngIf="vDes" tabindex="-1" role="dialog">
    <div class="contenedor">
        <div>
            <div>
                <div class="modalHeader">
                    <h4> Planeta destruido </h4>
                    <button class="btn btn-
info" (click)="cancelar()">Aceptar</button>
```

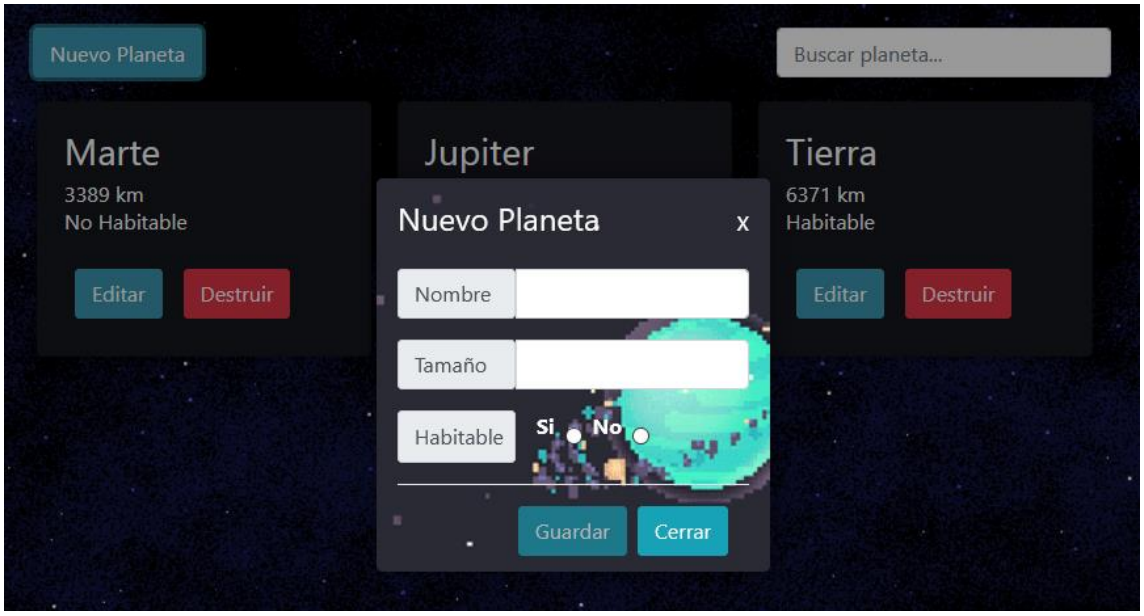
```
                </div>
            </div>
        </div>
    </div>
</div>
```

destroyed.component.scss

```
:host {
    .modal {
        display: table;
        width: 100%;
        height: 100%;
        background-color: rgba(0, 0, 0, .4); //transparencia negra
    }
    .contenedor {
        display: table-cell;
        vertical-align: middle; //lo centra a nivel vertical
        & > div {
            background-
image:url('https://lh3.googleusercontent.com/proxy/nCrlm1Yx4s2A09VNS8Kvit
NJPyvdH1i8nUVDzoAowzy-PQhemc-PeALWl-
kzrulwEt1J4ByGOEb2ulpzMrVmyuYJbmGliKWSmCqIeAS0M3mhpBioQpovfTQ8GGodpJUDKxX
S');
            background-position: 50% 60%;
            background-size: 500px;
            width: 200px;
            height: 130px;
            text-align: center;
            color: #ffffff;
            background-color: #16171e;
            margin: 0 auto; //lo centra a nivel horizontal
            padding: 1rem; //marquito para interior del modal
            border-radius: 5px; //vertices redondeadas
            box-shadow: 0 0 20px rgba(0, 0, 0, .4);
        }
    }
    .modalHeader {
        position:relative;
        h2{
            margin:0;
        }
    }
    .button {
        margin: 3px;
    }
}
```

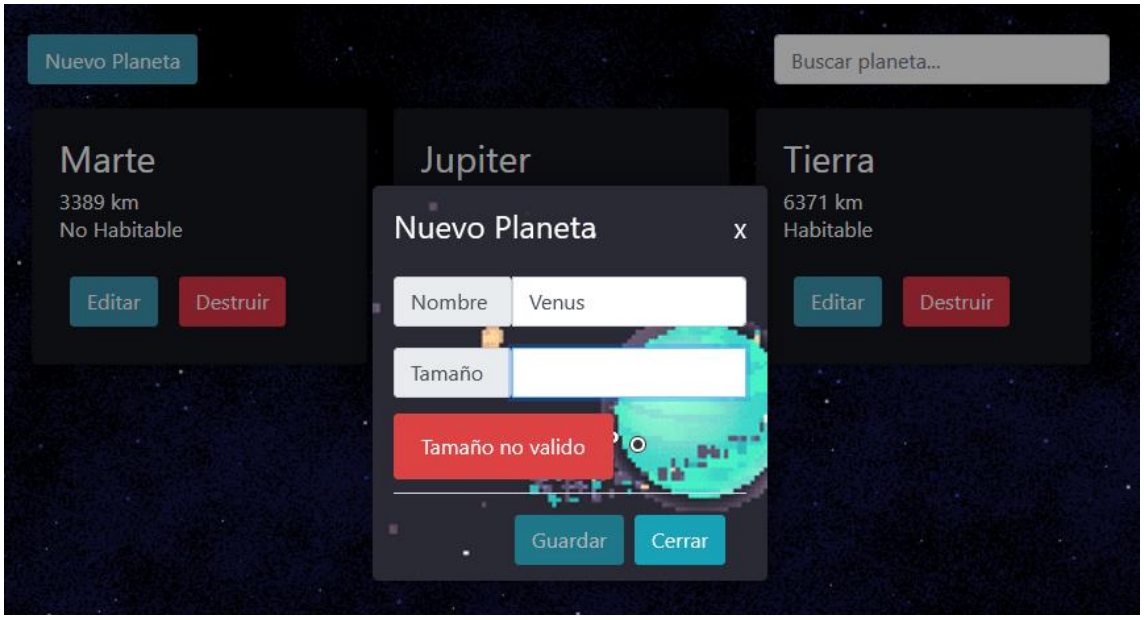
INTERFACES

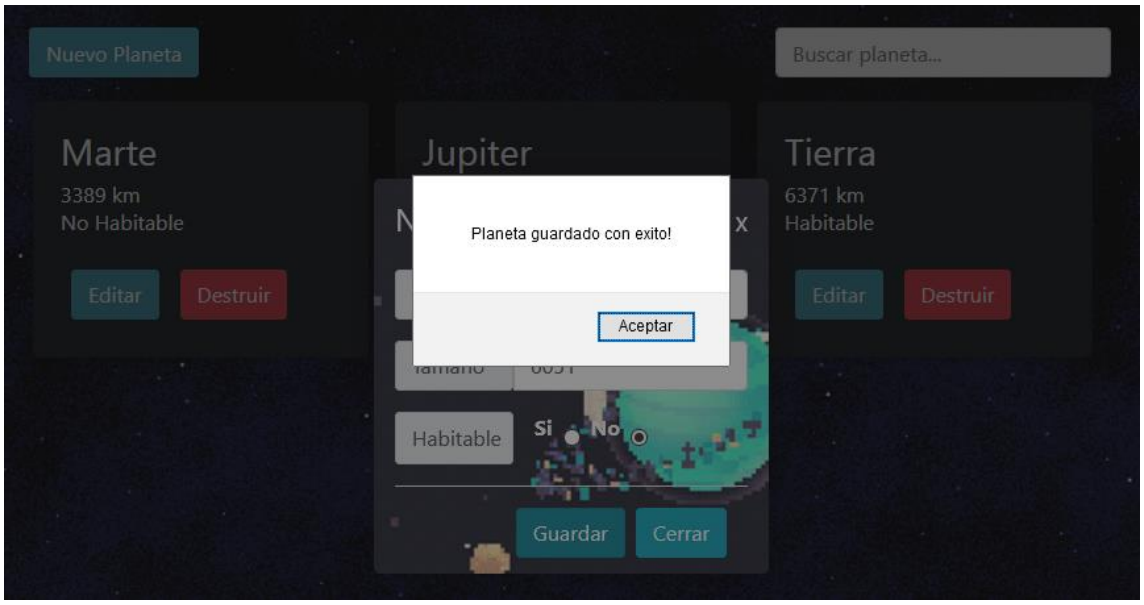

Nuevo Planeta
Buscar planeta...
Marte
3389 km
No Habitable
Editar
Destruir
Jupiter
69911 km
No Habitable
Editar
Destruir
Tierra
6371 km
Habitable
Editar
Destruir

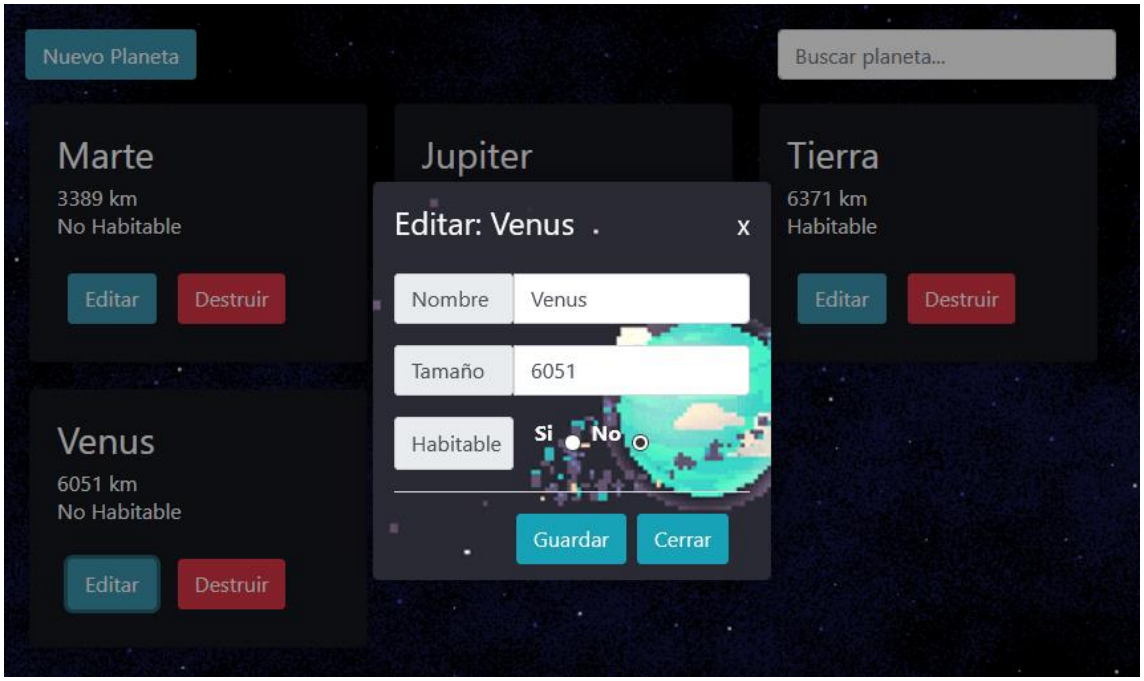

Nuevo Planeta
Buscar planeta...
Marte
3389 km
No Habitable
Editar
Destruir
Jupiter
Tierra
6371 km
Habitable
Editar
Destruir
Nuevo Planeta
x
Nombre
Tamaño
Habitable
Si
No
Guardar
Cerrar

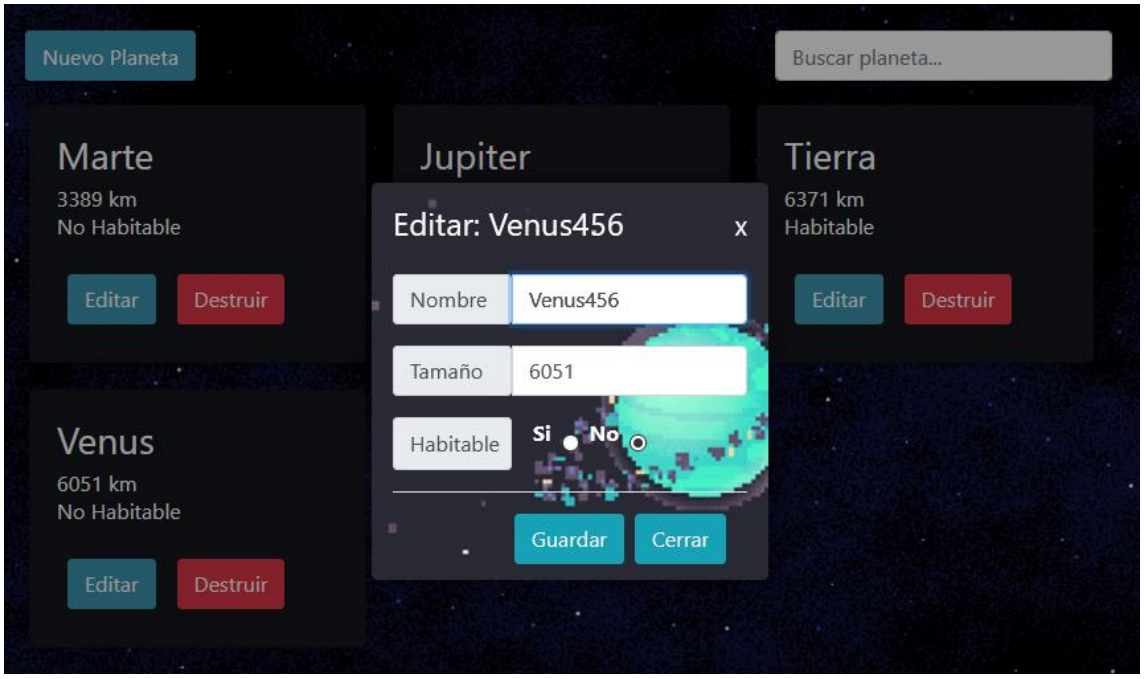

Nuevo Planeta
Buscar planeta...
Marte
3389 km
No Habitable
Editar
Destruir
Jupiter
Nuevo Planeta
x
Nombre
Venus
Tamaño
6051
Habitable
Si
No
Guardar
Cerrar
Tierra
6371 km
Habitable
Editar
Destruir

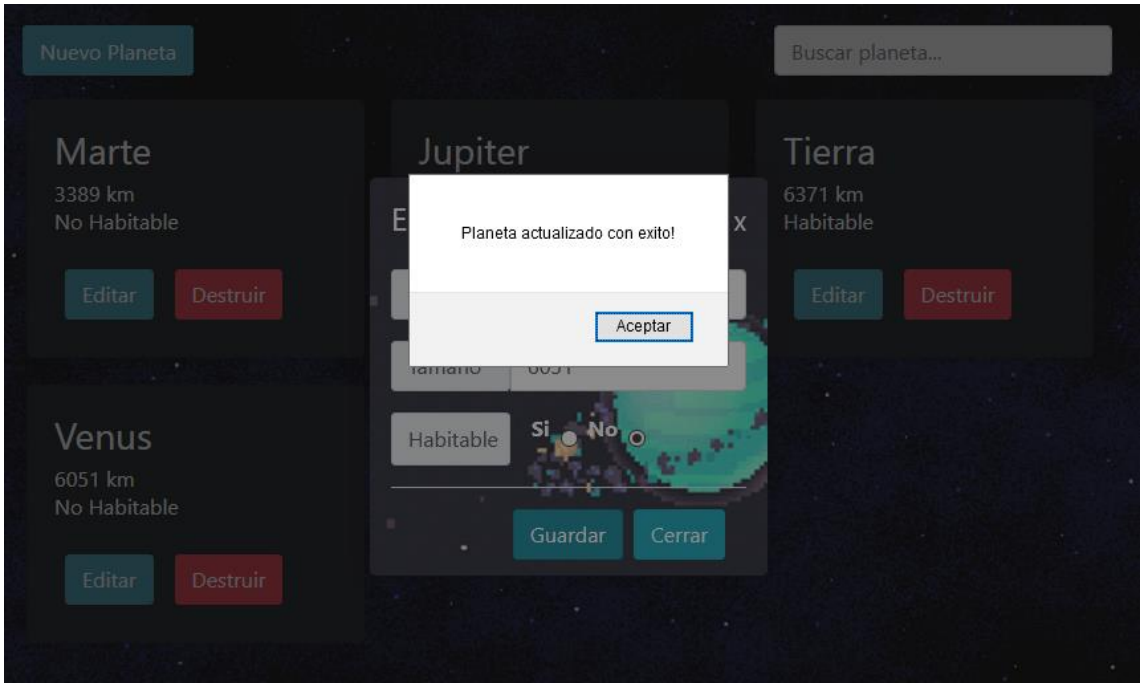

Nuevo Planeta
Buscar planeta...
Marte
3389 km
No Habitable
Editar
Destruir
Jupiter
Nuevo Planeta
x
Nombre
_^dsa
Nombre no valido
Habitable
Si
No
Guardar
Cerrar
Tierra
6371 km
Habitable
Editar
Destruir


Nuevo Planeta
Buscar planeta...
Marte
3389 km
No Habitable
Editar
Destruir
Jupiter
Nuevo Planeta
x
Nombre
Venus
Tamaño
Tamaño no valido
Guardar
Cerrar
Tierra
6371 km
Habitable
Editar
Destruir

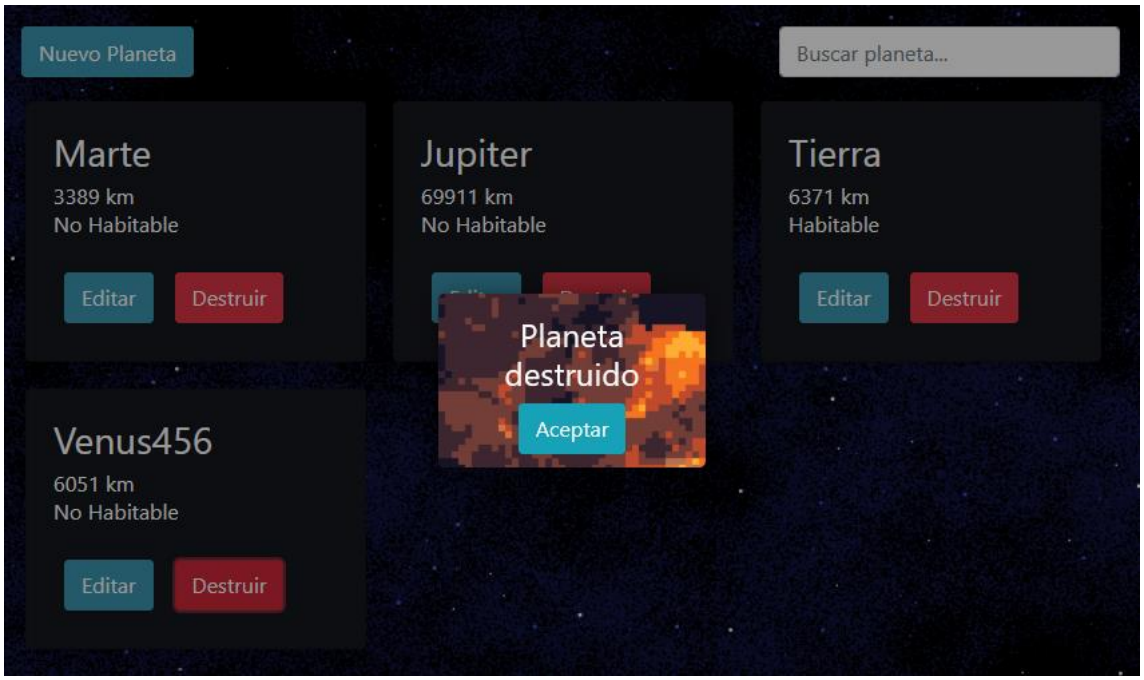

Nuevo Planeta
Buscar planeta...
Marte
3389 km
No Habitable
Editar
Destruir
Jupiter
Planeta guardado con exito!
Aceptar
Habitable
Si
No
Guardar
Cerrar
Tierra
6371 km
Habitable
Editar
Destruir


Nuevo Planeta
Buscar planeta...
Marte
3389 km
No Habitable
Editar
Destruir
Jupiter
Tierra
6371 km
Habitable
Editar
Destruir
Editar: Venus
x
Nombre
Venus
Tamaño
6051
Habitable
Si
No
Guardar
Cerrar
Venus
6051 km
No Habitable
Editar
Destruir


Nuevo Planeta
Buscar planeta...
Marte
3389 km
No Habitable
Editar
Destruir
Jupiter
Tierra
6371 km
Habitable
Editar
Destruir
Editar: Venus456
x
Nombre
Venus456
Tamaño
6051
Habitable
Si
No
Guardar
Cerrar
Venus
6051 km
No Habitable
Editar
Destruir


Nuevo Planeta
Buscar planeta...
Marte
3389 km
No Habitable
Editar
Destruir
Jupiter
Tierra
6371 km
Habitable
Editar
Destruir
Planeta actualizado con exito!
Aceptar
Habitable
Si
No
Guardar
Cerrar
Venus
6051 km
No Habitable
Editar
Destruir


Nuevo Planeta
a
Marte
3389 km
No Habitable
Editar
Destruir
Tierra
6371 km
Habitable
Editar
Destruir


Nuevo Planeta
Buscar planeta...
Marte
3389 km
No Habitable
Editar
Destruir
Jupiter
69911 km
No Habitable
Tierra
6371 km
Habitable
Editar
Destruir
Planeta destruido
Aceptar
Venus456
6051 km
No Habitable
Editar
Destruir

GIT

git init

```
Sabrina@DESKTOP-OPP3921 MINGW64 /d/angularprojects/ProyectoPlanetasFront (master
)
$ git remote add origin https://github.com/SZDohmen/ProyectoPlaneta-Frontend.git
```

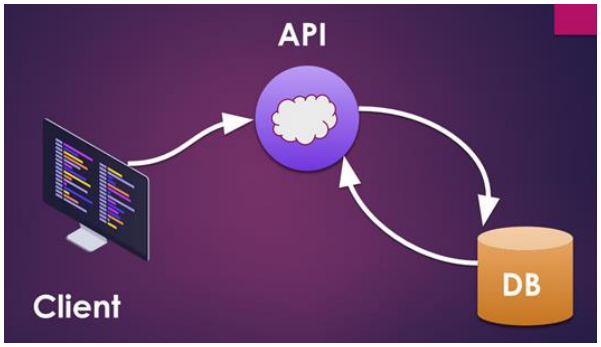
```
Sabrina@DESKTOP-OPP3921 MINGW64 /d/angularprojects/ProyectoPlanetasFront (master
)
$ git push -u origin master
```

```
Sabrina@DESKTOP-OPP3921 MINGW64 /d/angularprojects/ProyectoPlanetasFront (master
)
$ git add .
```

```
Sabrina@DESKTOP-OPP3921 MINGW64 /d/angularprojects/ProyectoPlanetasFront (master
)
$ git commit -m "all files"
```

```
Sabrina@DESKTOP-OPP3921 MINGW64 /d/angularprojects/ProyectoPlanetasFront (master
)
$ git push origin master
```

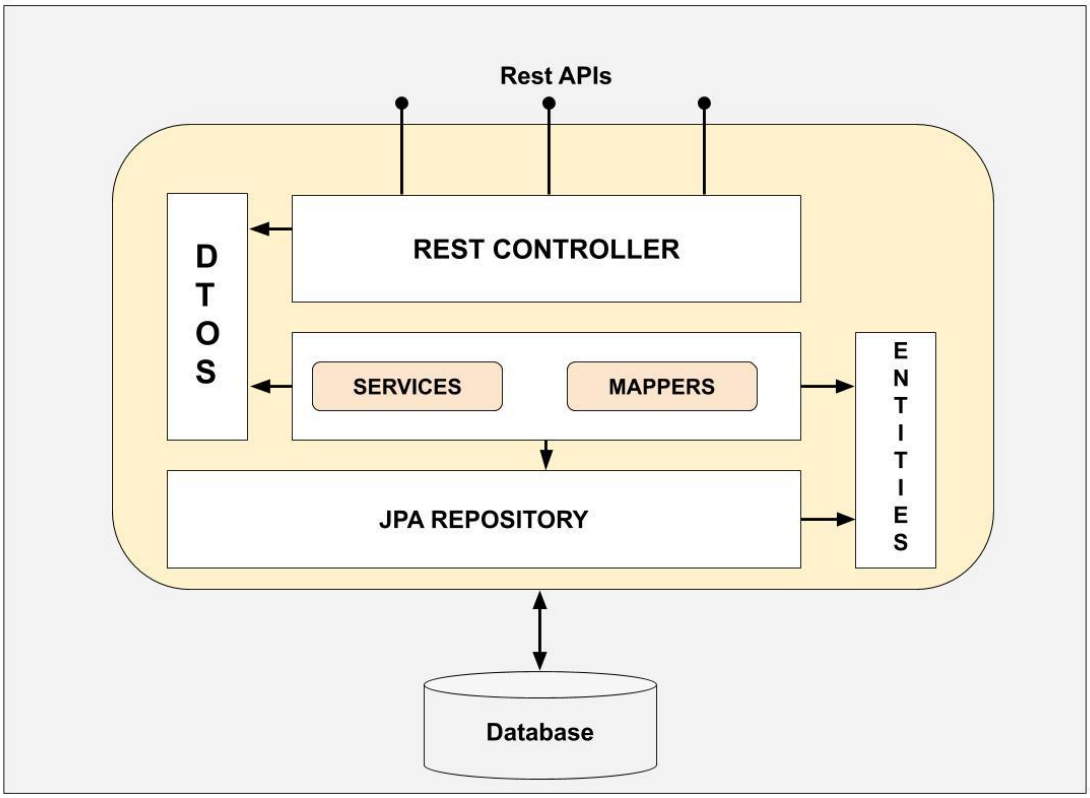
CONCEPTOS



Se entiende como API (*Application Programming Interface*) a una Interfaz de Programación de Aplicaciones. Es un sistema (interfaz) que permite que otras aplicaciones se comuniquen entre sí.

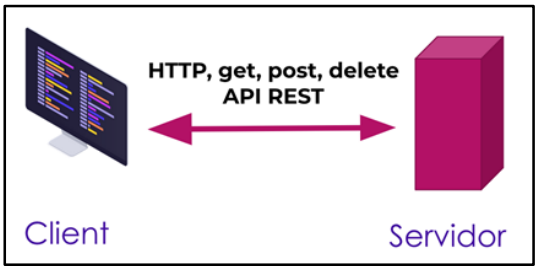
Un servicio web es una colección de estándares y protocolos que aplicaciones y sistemas usan para intercambiar datos a través de internet. Se utilizan para crear aplicaciones que necesitan ofrecer interoperabilidad, es decir, comunicarse con otras aplicaciones sin depender del sistema operativo subyacente y los lenguaje de programación.

La estructura REST (*Representational State Transfer*) se la define como una Transferencia de Estado de Representación. Es una arquitectura de Sw que proporciona características y protocolos subsayentes que rigen el comportamiento de los clientes y los servidores.



Un servicio Rest no tiene estado (*stateless*), es decir que el cliente debe pasarle el estado en cada llamada. Esto hace que cada vez que se quiera hacer una petición (paso de datos), el servicio Rest no recordará las anteriores. Está orientado a operar sobre recursos y no sobre servicios.

Como ventaja esta la posibilidad de crear cliente/servidor. Se centra en el rendimiento a gran escala para sistemas distribuidos hipermedia. Independiente de las tecnologías y los lenguajes. Requiere menos recursos del servidor, es más flexible y de performance superior.



RESTfull hace referencia a un sitio web que implementa la arquitectura REST, es decir, cliente/servidor basado en protocolo http. No mantiene estado y los recursos son accedidos por medio de url.

Utiliza explícitamente los métodos http:

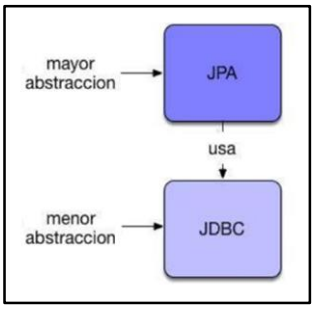
- ***GET***: para obtener un recurso
- ***POST***: para crear un recurso en el servidor
- ***PUT***: para cambiar el estado de un recurso o actualizarlo
- ***DELETE***: para eliminar un recurso

```
{
    "planetas": [
        { "name": "Tierra", "size": 6371, "habitable":"true"},
        { "name": "Marte", "size": 3389, "habitable":"true"},
        { "name": "Jupiter", "size": 69911, "habitable":"true"}
    ]
}
```

Un JSON (*JavaScript Object Notation*) es un formato de texto ligero para el intercambio de datos, el cual se usará en este proyecto.

Persistencia es la propiedad que permite que un objeto persista en el tiempo, es decir, que el objeto siga existiendo después de finalizado el programa que le dio origen.

Para persistir objetos se utiliza una base de datos del tipo relacional. Por un lado, se tiene los objetos y que por otro se guardan los datos en forma relacional (en una tabla), no como objetos. Esto se logra mediante el mapeo de los objetos en dichas tablas y viceversa. Dicho mecanismo es un ORM, un Mapeo de Objeto Relacional (*Object-Relational Mapping*). Esto permite la transición de lenguaje orientado a objeto a una base de datos relacional. Para persistir los objetos se usa: JPA, Hibernate, Spring Data JPA.



Java Persistence API, brinda interfaces que luego son implementadas por distintos proveedores. Ofrece un conjunto de anotaciones (del paquete java.persistence) para anotar los objetos e indicar como se realizará el mapeo en la base de dato. El lenguaje de consultas, llamado JPQL (*Java Persistence Query Language*) para consultar los objetos persistentes. Proporciona especificaciones para persistir, leer y administrar datos desde su objeto java hacia las relaciones en la BD.

Un framework de mapeo de objeto racional para java es Hibernate, que facilita el mapeo de atributos entre una base de datos relacional y el modelo de objetos de una aplicación. Funciona mediante el marcado de clases por medio de anotaciones, que sirven para el mapeo entre entidades POJO en java y tablas en SQL. Hibernate es un ORM que implementa JPA.

En resumen: Hibernate es un ORM que implementa JPA para indicar como mapear cualquier objeto a una base de datos y viceversa, es decir que se ocupa de la persistencia o recuperación de los objetos.

Spring Data es parte del Framework Spring, que consiste en un gran número de submódulos cuyo objetivo es facilitar el acceso de datos en aplicaciones basadas en Spring.

Mientras que Spring Data JPA tiene como objetivo mejorar la implementación de las capas de acceso a datos reduciendo el esfuerzo de persistir o acceder a los datos de una BD. Es otra capa encima de JPA, que requiere un proveedor de JPA como Hibernate. Las dependencias para ello se escriben en el archivo pom.xml

Una de las herramientas más útiles a la hora de utilizar librerías de terceros (como JPA e Hibernate) es Maven. Es una herramienta capaz de gestionar un proyecto de software Java completo. Desde la etapa en la que se comprueba que el código es correcto, hasta que se despliega la aplicación, pasando por la ejecución de pruebas y generación de informes y documentación. Es decir que nos va a acompañar durante todo el ciclo de vida de nuestra aplicación.

Por otro lado, Maven es un sistema de gestión de dependencias (o librerías) muy potente que nos permite identificar que librerías utiliza el software que estamos desarrollando en un fichero de configuración POM. Esto nos permite utilizar todas las dependencias y librerías sin necesidad de descargarlas manualmente gracias a su repositorio remoto (Maven Central o Maven Repository).