VALIDACIÓN Y VERIFICACIÓN DE SOFTWARE (INF 732)
GUÍA DE LABORATORIO 1
Hola mundo

En el ámbito del desarrollo de software, la creación de aplicaciones robustas, escalables y mantenibles es un objetivo fundamental. Para lograrlo, es esencial contar con herramientas y frameworks que faciliten la estructuración del código y la implementación de buenas prácticas. NestJS emerge como una opción poderosa en el ecosistema de Node.js, ofreciendo un enfoque modular y basado en TypeScript que permite construir aplicaciones del lado del servidor de manera eficiente. Esta guía de laboratorio tiene como objetivo introducir a los estudiantes en el uso de NestJS, guiándolos en la creación de una aplicación básica desde cero. A través de la implementación de un sencillo "Hola Mundo", se explorarán conceptos clave como la creación de módulos, controladores, servicios y la inyección de dependencias, además de la realización de pruebas unitarias para garantizar el correcto funcionamiento del código. Este laboratorio no solo busca familiarizar a los estudiantes con el framework, sino también sentar las bases para el desarrollo de aplicaciones más complejas en el futuro.

Objetivo

El objetivo principal de la guía es proporcionar una introducción práctica al desarrollo de aplicaciones utilizando **NestJS**, un framework de Node.js para construir aplicaciones del lado del servidor de manera eficiente y escalable. A través de esta guía, se busca:

1. **Familiarizarse con NestJS**: Aprender los conceptos básicos de NestJS, como la creación de módulos, controladores y servicios, así como la estructura de un proyecto típico en este framework.
2. **Configurar un entorno de desarrollo**: Asegurarse de que los estudiantes tengan las herramientas necesarias instaladas (como Node.js, npm/yarn y Nest CLI) para comenzar a desarrollar aplicaciones con NestJS.
3. **Crear una aplicación básica**: Guiar a los estudiantes en la creación de una aplicación simple ("Hola Mundo") que les permita entender cómo se estructuran los componentes en NestJS, cómo se manejan las solicitudes HTTP y cómo se organiza el código en módulos.

4. **Entender la inyección de dependencias**: Mostrar cómo NestJS utiliza la inyección de dependencias para conectar diferentes componentes de la aplicación, como controladores y servicios.
5. **Realizar pruebas unitarias**: Introducir a los estudiantes en la creación de pruebas unitarias para verificar el comportamiento de los componentes de la aplicación, utilizando herramientas como Jest.
6. **Ejecutar y probar la aplicación**: Enseñar cómo ejecutar la aplicación y cómo probarla en un entorno local, asegurándose de que funcione correctamente.

Prerrequisitos

Antes de comenzar, asegúrate de tener instalado:

- **Node.js** (versión 14.x o superior)

```
node -v
```

- **npm** (Node Package Manager) o **yarn**

```
npm -v
```

- **Nest CLI** (opcional, pero recomendado)

```
nest -v
```

Puedes instalar el CLI de NestJS globalmente con el siguiente comando:

```
npm install -g @nestjs/cli
```

1. Crear un Nuevo Proyecto

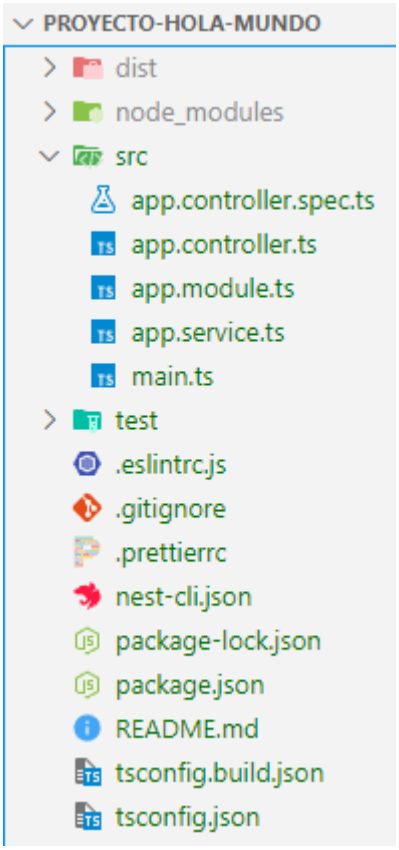
Con el CLI de NestJS, puedes crear un nuevo proyecto con el siguiente comando:

```
nest new proyecto-hola-mundo
```

Esto iniciará un asistente que te preguntará qué administrador de paquetes deseas usar (npm o yarn). Selecciona el que prefieras y espera a que se instalen las dependencias.

2. Estructura

Una vez creado el proyecto, deberías ver una estructura similar a esta:



app.service.ts

```
import { Injectable } from '@nestjs/common';

@Injectable()
export class AppService {
  getHello(): string {
    return 'Hello World!';
  }
}
```

Importación del Decorador Injectable

```
import { Injectable } from '@nestjs/common';
```

- **@nestjs/common**: Es uno de los módulos principales de NestJS que contiene muchos de los decoradores y clases más comunes, como Injectable, Controller, Module, entre otros.

- **Injectable**: Es un decorador que se usa para marcar una clase como un proveedor en NestJS. Los proveedores son clases que pueden ser inyectadas en otras clases mediante la inyección de dependencias.

Definición de la Clase AppService

```
@Injectable()
export class AppService {
```

- @Injectable(): Este decorador indica que la clase AppService es un proveedor. Esto permite que NestJS pueda inyectar esta clase en otros lugares donde sea necesaria.
- export class AppService { ... }: Aquí se define la clase AppService como un servicio que estará disponible para inyectar en otros componentes de la aplicación (por ejemplo, controladores).

Método getHello

```
getHello(): string {
  return 'Hello World!';
}
```

- getHello(): string: Este es un método dentro de la clase AppService. Este método no toma parámetros y devuelve un string.
- return 'Hello World!';: El método simplemente devuelve la cadena 'Hello World!'.

Resumen

- **Propósito del Servicio**: AppService es un servicio básico que podría usarse en un controlador para manejar la lógica de negocio de una aplicación. En este caso, su única funcionalidad es devolver la cadena 'Hello World!' cuando se llama al método getHello.
- **Uso en un Controlador**: Generalmente, un servicio como este se inyecta en un controlador para separar la lógica de negocio de la lógica de manejo de solicitudes. Un controlador podría llamar al método getHello de AppService para obtener el mensaje y luego enviarlo como respuesta a una solicitud HTTP.

app.controller.ts

```
import { Controller, Get } from '@nestjs/common';
import { AppService } from './app.service';

@Controller()
export class AppController {
  constructor(private readonly appService: AppService) {}

  @Get()
  getHello(): string {
    return this.appService.getHello();
  }
}
```

Este código define un controlador en una aplicación NestJS. Los controladores en NestJS son responsables de manejar las solicitudes entrantes y devolver respuestas al cliente. Aquí te explico cada parte del código:

Importaciones

```
import { Controller, Get } from '@nestjs/common';
import { AppService } from './app.service';
```

- **Controller y Get**: Estos son decoradores proporcionados por NestJS y se importan del módulo @nestjs/common.
    - **@Controller()**: Se usa para definir una clase como un controlador, lo que le permite manejar rutas HTTP específicas.
    - **@Get()**: Es un decorador que define un manejador de rutas para las solicitudes GET.
- **AppService**: Es un servicio que se importa desde el archivo app.service.ts. Este servicio es inyectado en el controlador para que pueda ser utilizado.

Definición del Controlador

```
@Controller()
export class AppController {
```

- **@Controller()**: Este decorador indica que AppController es un controlador. Si no se especifica una ruta en el decorador (como en este caso), el controlador manejará las rutas relativas a la raíz (/).
- **export class AppController { ... }**: Define la clase AppController como el controlador de la aplicación.

Constructor

```
constructor(private readonly appService: AppService) {}
```

- **Inyección de Dependencias**: El constructor de AppController tiene un parámetro llamado appService de tipo AppService.
    - **private readonly appService: AppService**: Esto significa que appService es una propiedad privada y de solo lectura que es inyectada en la clase cuando se crea una instancia de AppController. Esta es la manera en que NestJS realiza la inyección de dependencias.
    - **AppService**: Es un servicio que contiene la lógica de negocio de la aplicación. En este caso, el servicio tiene un método getHello() que devuelve un saludo.

Método getHello

```
@Get()
 getHello(): string {
  return this.appService.getHello();
 }
```

- **@Get()**: Este decorador asocia el método getHello con las solicitudes HTTP GET. Debido a que no se especifica una ruta dentro del decorador @Get(), este método manejará las solicitudes GET a la ruta raíz (/).
- **getHello(): string**: Este método no toma ningún parámetro y devuelve un string.
- **return this.appService.getHello();**: Este método llama al método getHello del servicio AppService, que devuelve la cadena 'Hello World!'. El resultado de esta llamada es lo que se envía de vuelta como respuesta a la solicitud GET.

Resumen del Flujo

1. **Solicitud GET**: Cuando un cliente hace una solicitud GET a la raíz (/) de la aplicación, esta solicitud es manejada por el método getHello de AppController.
2. **Llamada al Servicio**: El método getHello del controlador llama al método getHello del servicio AppService, que devuelve la cadena 'Hello World!'.
3. **Respuesta al Cliente**: El controlador devuelve esta cadena como respuesta al cliente.

app.module.ts

```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';

@Module({
  imports: [],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

Este código define un módulo en NestJS, que es la forma principal de organizar y estructurar una aplicación NestJS. Los módulos permiten agrupar controladores, servicios y otros proveedores de manera lógica. Aquí te explico cada parte del código:

Importaciones

```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
```

- **Module**: Es un decorador que se usa para definir un módulo en NestJS. Se importa desde @nestjs/common, que es uno de los módulos principales de NestJS.

- **AppController** y **AppService**: Estos son componentes que se importan de otros archivos (app.controller.ts y app.service.ts, respectivamente). AppController es un controlador que maneja las solicitudes HTTP, y AppService es un servicio que contiene la lógica de negocio de la aplicación.

Definición del Módulo

```
@Module({
  imports: [],
  controllers: [AppController],
  providers: [AppService],
})
```

- **@Module({...})**: Este decorador marca la clase AppModule como un módulo de NestJS. Dentro del decorador @Module, se configuran varias propiedades que definen qué componentes están disponibles dentro de este módulo.

Propiedades del Decorador @Module

1. **imports: []**:
    - Esta propiedad se usa para importar otros módulos que este módulo necesita para funcionar. En este caso, el array está vacío porque AppModule no depende de otros módulos.
2. **controllers: [AppController]**:
    - Aquí se definen los controladores que pertenecen a este módulo. En este caso, AppController es el único controlador registrado en AppModule. Los controladores manejan las solicitudes HTTP entrantes y devuelven respuestas.
3. **providers: [AppService]**:
    - Esta propiedad define los proveedores que están disponibles dentro del módulo. Un proveedor puede ser un servicio, un valor, una clase, etc. En este caso, AppService es un proveedor que se puede inyectar en otros componentes (como el AppController). Los servicios encapsulan la lógica de negocio y son inyectables en controladores y otros servicios.

Exportación del Módulo

```
export class AppModule {}
```

- **export class AppModule {}**: Define la clase AppModule como el módulo principal de la aplicación. En NestJS, cada aplicación tiene al menos un módulo raíz, que en este caso es AppModule. Este módulo se usa para organizar y configurar los diferentes componentes de la aplicación.

main.ts

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  await app.listen(3000);
}
bootstrap();
```

Importaciones

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
```

- **NestFactory**: Es una clase proporcionada por el módulo @nestjs/core. Esta clase es responsable de crear instancias de la aplicación NestJS. Se utiliza para inicializar la aplicación y configurar el entorno en el que se ejecutará.
- **AppModule**: Es el módulo principal de la aplicación que define la estructura y los componentes principales (controladores, servicios, etc.). Este módulo se importa desde el archivo app.module.ts.

Función bootstrap

```
async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  await app.listen(3000);
}
```

- **async function bootstrap()**:
  - bootstrap es una función asíncrona que se utiliza para inicializar la aplicación NestJS. La función es asíncrona porque realiza operaciones que involucran promesas, como la creación de la aplicación y la puesta en marcha del servidor.
- **const app = await NestFactory.create(AppModule);**:
  - Este comando crea una instancia de la aplicación NestJS.
  - **NestFactory.create(AppModule)**: Este método toma el módulo raíz (AppModule) como argumento y crea la aplicación a partir de él.
  - **await**: Se usa porque NestFactory.create() devuelve una promesa, y await asegura que la aplicación se cree antes de continuar con el siguiente paso.
- **await app.listen(3000);**:
  - Este comando inicia el servidor HTTP en el puerto 3000.
  - **app.listen(3000)**: Hace que la aplicación escuche solicitudes HTTP en el puerto 3000. Una vez que el servidor está en funcionamiento, la aplicación estará disponible para manejar solicitudes en http://localhost:3000.
  - **await**: Se usa nuevamente porque app.listen() es una operación asíncrona que también devuelve una promesa. Esto asegura que la aplicación esté completamente iniciada antes de que la función bootstrap termine su ejecución.

Llamada a bootstrap

```
bootstrap();
```

- **bootstrap();**: Este comando invoca la función bootstrap para iniciar el proceso de inicialización de la aplicación. Es el punto de partida de la aplicación NestJS y sin esta llamada, el servidor no se iniciaría.

Resumen del Flujo

1. **Creación de la Aplicación**: Se crea una instancia de la aplicación NestJS basada en el módulo raíz AppModule.
2. **Inicio del Servidor**: La aplicación comienza a escuchar solicitudes HTTP en el puerto 3000.

3. **Disponibilidad de la Aplicación**: Una vez que el servidor está en marcha, la aplicación está lista para manejar las solicitudes que lleguen a http://localhost:3000.

Para ejecutar la aplicación, utiliza el siguiente comando:

```
npm run start
```

Esto iniciará un servidor en http://localhost:3000. Si visitas esta URL en tu navegador, deberías ver el mensaje "¡Hola Mundo!".

app.controller.spec.ts

```
import { Test, TestingModule } from '@nestjs/testing';
import { AppController } from './app.controller';
import { AppService } from './app.service';

describe('AppController', () => {
  let appController: AppController;

  beforeEach(async () => {
    const app: TestingModule = await Test.createTestingModule({
      controllers: [AppController],
      providers: [AppService],
    }).compile();

    appController = app.get<AppController>(AppController);
  });

  describe('root', () => {
    it('should return "Hello World!"', () => {
      expect(appController.getHello()).toBe('Hello World!');
    });
  });
});
```

Este código es un ejemplo de un test unitario en una aplicación NestJS. Utiliza el marco de pruebas integrado de NestJS para verificar que el

comportamiento del controlador sea el esperado. Aquí te explico cada parte del código:

Importaciones

```
import { Test, TestingModule } from '@nestjs/testing';
import { AppController } from './app.controller';
import { AppService } from './app.service';
```

- **Test** y **TestingModule**: Estos son utilidades proporcionadas por @nestjs/testing para configurar y ejecutar pruebas unitarias en NestJS.
    - **Test**: Es una clase que proporciona métodos estáticos para crear instancias de TestingModule.
    - **TestingModule**: Es una clase que representa un módulo de prueba, permitiendo configurar el entorno para la prueba.
- **AppController** y **AppService**: Son las clases que se están probando, importadas desde sus respectivos archivos.

Descripción del Conjunto de Pruebas

```
describe('AppController', () => {
  let appController: AppController;
```

- **describe('AppController', () => { ... })**:
    - describe es una función global proporcionada por Jest (el marco de pruebas utilizado por defecto en NestJS) que define un conjunto de pruebas. En este caso, el conjunto de pruebas se centra en el AppController.
- **let appController: AppController;**:
    - appController es una variable que se declara aquí y se inicializa antes de cada prueba para que pueda ser utilizada dentro del conjunto de pruebas.

Configuración Antes de Cada Prueba

```
beforeEach(async () => {
    const app: TestingModule = await Test.createTestingModule({
      controllers: [AppController],
```

```
    providers: [AppService],
  }).compile();

  appController = app.get<AppController>(AppController);
});
```

- **beforeEach(async () => { ... })**:
  - beforeEach es una función que se ejecuta antes de cada prueba individual en el conjunto. Aquí se utiliza para configurar el entorno de prueba.
- **const app: TestingModule = await Test.createTestingModule({...}).compile();**
  - Este código crea un TestingModule, que es un módulo de prueba configurado con los controladores y proveedores necesarios.
  - **Test.createTestingModule({...})**: Se utiliza para configurar el módulo de prueba, registrando el AppController y AppService.
  - **compile()**: Compila el módulo de prueba, resolviendo cualquier inyección de dependencias.
- **appController = app.get<AppController>(AppController);**:
  - Una vez que el módulo de prueba está compilado, se obtiene una instancia del AppController desde el módulo de prueba utilizando app.get<AppController>(AppController). Esta instancia se guarda en la variable appController para ser utilizada en las pruebas.

Prueba Individual

```
describe('root', () => {
  it('should return "Hello World!"', () => {
    expect(appController.getHello()).toBe('Hello World!');
  });
});
```

- **describe('root', () => { ... })**:
  - Este describe anidado agrupa pruebas relacionadas con la ruta raíz (/) que maneja el AppController.

- **it('should return "Hello World!"', () => { ... })**:
  - it es otra función proporcionada por Jest que define una prueba individual. Esta prueba verifica que el método getHello() del AppController devuelva 'Hello World!'.
- **expect(appController.getHello()).toBe('Hello World!');**:
  - expect es una función de Jest que se utiliza para hacer afirmaciones sobre el resultado de una operación. En este caso, se espera que la llamada a appController.getHello() devuelva exactamente 'Hello World!'.
  - **toBe('Hello World!')**: Es una aserción que verifica que el valor retornado sea exactamente igual a 'Hello World!'.

Resumen

- **Configuración de Pruebas**: El código configura un entorno de prueba para el AppController y su dependencia AppService.
- **Prueba del Método getHello()**: La prueba verifica que el método getHello() en el AppController devuelva 'Hello World!', como se espera.
- **Uso de Jest**: Jest es el marco de pruebas que se utiliza, proporcionando funciones como describe, it, y expect para estructurar y realizar las pruebas.

Para ejecutar los test, utiliza el siguiente comando:

```
npm run test
```