NestJS and TypeScript Fundamentals

Course Code: CSC 4182 Course Title: Advanced Programming In Web Technologies

Dept. of Computer Science
Faculty of Science and Technology

Lecture No:	**2**	**Week No:**	**01**	**Semester:**	
Lecturer:	*Sazzad Hossain; sazzad@aiub.edu*				

Lecture Outline

- ✓ NestJS
- ✓ Node.js Vs Express.js Vs NestJS
- ✓ TypeScript(TS)?
- ✓ Declare Variables in JavaScript/TypeScript
- ✓ Data Types in TypeScript
- ✓ TypeScript Decorators
- ✓ TypeScript Generics
- ✓ TypeScript Async/Await and Promises

NestJS

- Nest (NestJS) is a framework for building efficient, scalable **Node.js** server-side applications.
- It uses **progressive** JavaScript, is built with and fully supports **TypeScript**
- Combines elements of OOP (Object Oriented Programming), FP (Functional Programming), and FRP (Functional Reactive Programming).
- makes use of robust HTTP Server frameworks like ExpressJS

Node.js Vs Express.js Vs NestJS

	Node.js	**Express.js**	**NestJS**
Abstraction Level	Low-level runtime environment for JavaScript/TypeScript	Minimalistic web application framework	Full-featured framework
Structure and Organization	No strict guidelines for code structure	Minimal structure guidelines	Opinionated structure and organization
Code Reusability	Limited code reusability without additional patterns or libraries	Limited code reusability without additional patterns or libraries	Enhanced code reusability using modules and dependency injection
Decorators and Metadata	Not natively supported	Not natively supported	Utilizes decorators and metadata for defining routes, controllers, and modules

Node.js Vs Express.js Vs NestJS

	Node.js	**Express.js**	**NestJS**
Dependency Injection	Not natively supported	Not natively supported	Built-in support for dependency injection, promoting loose coupling and modularity
Testing	No built-in testing utilities	Basic testing utilities	Robust testing utilities and support for unit testing, integration testing, and e2e testing
Developer Productivity	Requires manual configuration and setup	Requires manual configuration and setup	CLI tool for project generation, code scaffolding, and automating repetitive tasks
Ecosystem and Community	Extensive ecosystem and large community support	Extensive ecosystem and large community support	Growing ecosystem and supportive community

Node.js Vs Express.js Vs NestJS

Building RESTful API using Node.js, Express.js, and NestJS.

Node.js Example:

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

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello, World!');
});

server.listen(3000, 'localhost', () => {
  console.log('Server running on port 3000');
});
```

Node.js Vs Express.js Vs NestJS

Building RESTful API using Node.js, Express.js, and NestJS.

Express.js Example:

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

const app = express();

app.get('/', (req, res) => {
  res.send('Hello, World!');
});

app.listen(3000, () => {
  console.log('Server running on port 3000');
});
```

Node.js Vs Express.js Vs NestJS

Building RESTful API using Node.js, Express.js, and NestJS.

NestJS Example:

```
import { Controller, Get } from '@nestjs/common';

@Controller()
export class AppController {
  @Get()
  getHello(): string {
    return 'Hello, World!';
  }
}
```

TypeScript(TS)?

TypeScript is a **strongly typed** programming language that builds on JavaScript

- TypeScript adds additional syntax to JavaScript to support a tighter integration with editor. **Catch errors** early in editor.
- TypeScript code converts to JavaScript, which **runs anywhere JavaScript runs**; In a browser or Node.js/Server.
- TypeScript understands JavaScript and uses **type inference to give you great tooling** without additional code.

History of TypeScript

- Developed by Microsoft, was first announced by Anders Hejlsberg in October 2012.
- It was introduced as a **superset** of JavaScript that adds optional **static typing, class-based object-oriented programming,** and improved tooling support to JavaScript development.
- Gained significant popularity in the JavaScript community due to its ability to **catch errors at compile-time**, provide **better tooling** support, and enable developers to build more **maintainable** and **scalable** applications.

Application of TypeScript

- **Web Development:** TypeScript is commonly used for building web applications, both on the client-side and server-side. Popular web frameworks like **Angular, React,** and **Vue.js** have built-in support for TypeScript.
- **Node.js Development:** Node.js frameworks and libraries, such as NestJS and TypeORM, are built using TypeScript.
- **Mobile App Development:** Frameworks like React Native and NativeScript to build cross-platform mobile apps.
- **Testing and Automation:** TypeScript is often used for writing tests and automation scripts.

Declare Variables in JavaScript/TypeScript

Variables can be declared using the **let**, **const**, or **var** keywords.

The **let** keyword is used to declare variables that **can be reassigned** with a new value.
is used to declare variables that are **block-scoped**. Block scope refers to the **portion of code within curly braces {}** (e.g., inside a function, loop, or conditional statement).

Example:

```
let age: number = 25;
let message: string = 'Hello, TypeScript!';
let isActive: boolean = true;
```

Declare Variables in JavaScript/TypeScript (Contd.)

The **const** keyword is used to declare variables that have a **constant** value and **cannot be reassigned**.

Example:

```
const PI: number = 3.14159;
const fullName: string = 'John Doe';
const isActive: boolean = true;
```

The **var** keyword is the legacy way of declaring variables in JavaScript and TypeScript. Variables declared with var have **function** or **global scope** and are hoisted.

Example:

```
var age: number = 25;
var message: string = 'Hello, TypeScript!';
var isActive: boolean = true;
```

Data Types in TypeScript

- **Boolean**: Represents the logical values true or false.

```
let isCompleted: boolean = true;
```

- **Number**: Represents numeric values, both integer and floating-point.

```
let age: number = 25;
let price: number = 9.99;
```

- **String**: Represents textual data enclosed in single or double quotes.

```
let message: string = 'Hello, World!';
```

- **Any**: Represents a dynamic or flexible type that allows values of any type.

```
let data: any = 'Hello, World!';
data = 42;
data = true;
```

Data Types in TypeScript

- **Array:** Represents a collection of values of a particular type.

```
let numbers: number[] = [1, 2, 3, 4, 5];
let fruits: Array<string> = ['apple', 'banana', 'orange'];
```

- **Void:** Represents the absence of a value. Typically used as the return type for functions that do not return a value.

```
function sayHello(): void {
  console.log('Hello!');}
```

- **Object**: Represents any non-primitive type, such as arrays, functions, or objects.

```
let user: object = {
  name: 'John Doe',
  age: 25,
};
```

TypeScript Decorators

- TypeScript **decorators** provide a way to add metadata and modify the behavior of classes, properties, methods, or parameters at design time.
- Decorators are expressions prefixed with the **@** symbol and are applied to the target they are decorating.
- To enhance or extend the functionality of existing code without modifying its original implementation.
- NestJS heavily utilizes decorators to define routes, controllers, services, and other application components.

TypeScript Decorators


```
import { Controller, Get } from
'@nestjs/common';

@Controller('users')// Control Decorator
export class UsersController {
  @Get()// Get Decorator
  getUsers(): string {
    return 'Get all users';
  }
}
```

TypeScript Generics

Generics allow you to create **reusable components** that can work with different types.

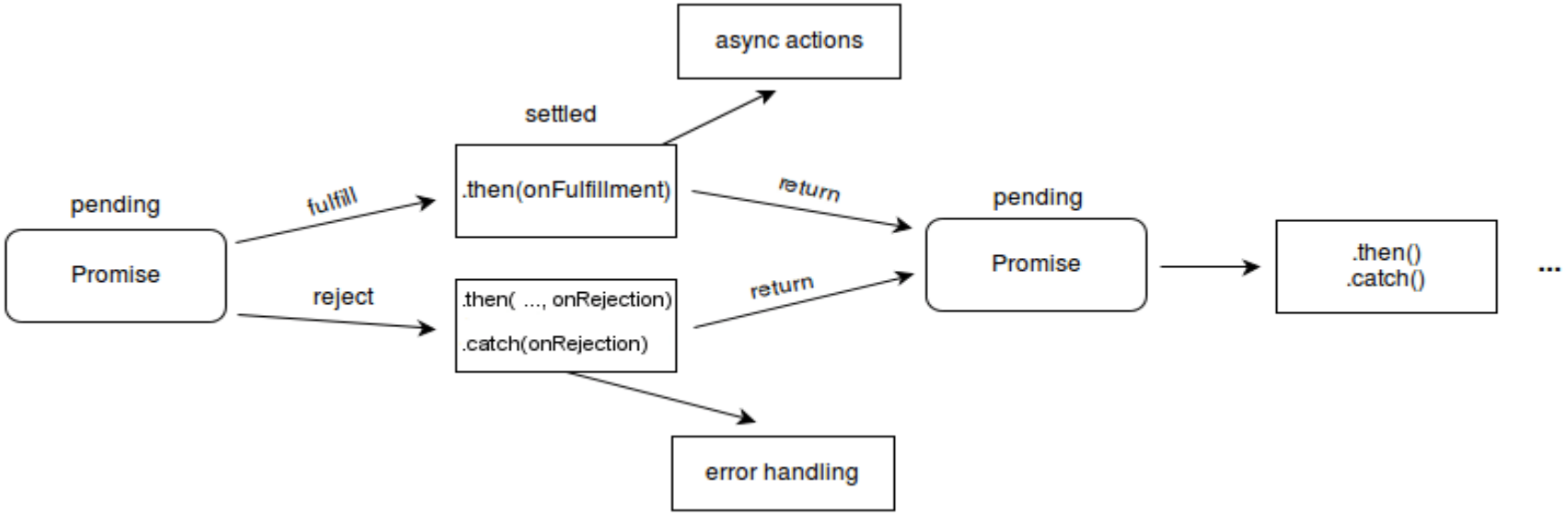
```
class Box<T> {
  private item: T;

  addItem(item: T): void {
    this.item = item;
  }
  getItem(): T {
    return this.item;
  }
}
const stringBox = new Box<string>();
stringBox.addItem('Hello, World!');
console.log(stringBox.getItem()); // Output: Hello, World!
```

TypeScript Async/Await and Promises

Promises are a core feature of JavaScript that provide an abstraction for handling **asynchronous** operations.

A Promise is an **object representing** the eventual completion or failure of an asynchronous operation, and it can be in one of three states: **pending**, **fulfilled**, or **rejected**.

It has methods like **.then()** and **.catch()**



TypeScript Async/Await and Promises

Async/Await is a syntactic feature introduced in **ECMAScript** 2017 that allows you to write asynchronous code in a more synchronous and readable manner.

It is built on top of **Promises** and provides a way to write **asynchronous** code that looks and behaves like **synchronous** code.

The **async** keyword is used to define an asynchronous function, and the **await** keyword is used to **pause** the execution of the function until a Promise is **resolved**.

TypeScript Async/Await and Promises


```typescript
async getUser(id: string): Promise<User> {
  try {
    // Retrieve the user asynchronously
    const user = await this.userService.getUserById(id);
    // Retrieve the posts asynchronously
    const posts = await this.postService.getPostsByUserId(user.id);
    // Assign the retrieved posts to the user object
    user.posts = posts;
    return user;
  } catch (error) {
    // If any error occurs during the asynchronous operations,
throw a NotFoundException
    throw new NotFoundException('User not found');
  }
}
```

By using **await**, the execution of the method is **paused** until the **Promises** returned by the service methods are resolved.

References

1. **W3Schools Online Web Tutorials, URL: http://www.w3schools.com**
2. **Node.js, URL: https://nodejs.org/en/**
3. **Next.js, URL: https://nextjs.org/**
4. **TypeScript URL: https://www.typescriptlang.org/**
5. **MDN Web Docs URL: https://developer.mozilla.org/**


AMERICAN INTERNATIONAL UNIVERSITY
PRAESIDIUM
DISCIPLINA
CIVITATIS
1994
BANGLADESH

Thank You!