Q: What is nunit testing?

NUnit is an open-source unit testing framework primarily used in the .NET ecosystem, including languages like C#.

What NUnit is:

**Open-source**: NUnit is freely available and can be used without any licensing fees.

**Unit testing framework**: It's specifically designed for unit testing, which means testing individual functions, methods, or classes in isolation.

**For .NET**: It's primarily used in .NET development, supporting languages like C#.

Based on JUnit: NUnit was initially developed as a port of JUnit, the popular Java testing framework.

**Why NUnit is used:**

**Test-driven development (TDD):**

NUnit supports TDD, a development approach where tests are written before the code they test, promoting clean and reliable code.

**Automated testing:**

It allows developers to automate the testing process, ensuring that code behaves as expected and catching errors early.

**Rich features:**

NUnit offers various features like attributes for defining tests, assertions for comparing expected and actual results, and support for parameterized tests.

**Q: Abstract VS Interface**

1. Using abstract we cannot achieve multiple inheritance but using an Interface we can achieve multiple inheritance.
2. We can not declare a member field in an Interface.
3. We can not use any access modifier i.e. public, private, protected, internal etc., because within an interface by default everything is public.
4. An Interface member cannot be defined using the keyword static, virtual, abstract or sealed.

**Abstract:**

// Abstract class

abstract class Animal

{

// Abstract method (does not have a body)

public abstract void animalSound();

// Regular method

public void sleep()

{

Console.WriteLine("Zzz");

}

}

// Derived class (inherit from Animal)

class Pig : Animal

{

public override void animalSound()

{

// The body of animalSound() is provided here

Console.WriteLine("The pig says: wee wee");

}

}

class Program

{

static void Main(string[] args)

{

Pig myPig = new Pig(); // Create a Pig object

myPig.animalSound(); // Call the abstract method

myPig.sleep(); // Call the regular method

}

}

**Interface**

// Interface

interface IAnimal

{

void animalSound(); // interface method (does not have a body)

}

// Pig "implements" the IAnimal interface

class Pig : IAnimal

{

public void animalSound()

{

// The body of animalSound() is provided here

Console.WriteLine("The pig says: wee wee");

}

}

class Program

{

static void Main(string[] args)

{

Pig myPig = new Pig(); // Create a Pig object

myPig.animalSound();

}

}

**Q3 What is Range Validation C#**  
Range validation in C# refers to checking whether a given value falls within a specified minimum and maximum range.

using System.ComponentModel.DataAnnotations;

public class Product

{

[Range(1, 1000, ErrorMessage = "Price must be between 1 and 1000.")]

public decimal Price { get; set; }

[Range(18, 65, ErrorMessage = "Age must be between 18 and 65.")]

public int Age { get; set; }

}

**Q-what is input validation test cases?**

**Input validation test cases** are specific scenarios created to test how an application handles various types of input from users. The goal is to ensure that the system behaves correctly, securely, and predictably—accepting valid input and rejecting invalid or malicious input.

**🧪 What Input Validation Test Cases Typically Include:**

1. **Valid Inputs (Positive Test Cases)**
   * Inputs that fall within acceptable range/format.
   * Example: Age = 25, Email = test@example.com, Password = P@ssw0rd!
2. **Invalid Inputs (Negative Test Cases)**
   * Inputs that break rules or constraints.
   * Example:
     + Age below 18 or above 65
     + Empty fields
     + Strings instead of numbers
3. **Boundary Values**
   * Edge cases on the limits of valid input.
   * Example: Age = 18, Age = 65
4. **Empty / Null Inputs**
   * Tests for missing or blank input values.
   * Example: Name = "", Email = null
5. **Special Characters or SQL Injection**
   * To test security against malicious input.
   * Example: '; DROP TABLE Users;--
6. **Wrong Data Types**
   * Providing a string where a number is expected.
   * Example: Age = "twenty"
7. **Exceeding Maximum Length**
   * Testing inputs that go beyond allowed lengths.
   * Example: Username = "averylongusernamethatexceedsmaxlength"

Q-What is Trigger?

**1. Triggers in SQL / Databases**

A **trigger** is a special kind of stored procedure that automatically runs ("is triggered") when certain events happen in the database, like:

* **INSERT**
* **UPDATE**
* **DELETE**

**🔹 Example (SQL Trigger):**

sql

CopyEdit

CREATE TRIGGER trg\_AfterInsert

ON Employees

AFTER INSERT

AS

BEGIN

INSERT INTO AuditLog (Description)

VALUES ('A new employee was added.')

END

So, every time a new record is inserted into the Employees table, this trigger fires and logs the event.

**⚡ 2. Triggers in C# (Events / Delegates)**

In **C#**, a **trigger** often refers to an event being fired.

For example, in a button click:

csharp

CopyEdit

button.Click += (sender, e) => {

Console.WriteLine("Button clicked!");

};

**Q-what is the main components in angular application**

In an Angular application, everything revolves around **components**, **services**, and **modules**. Here’s a breakdown of the **main components** (or building blocks) of an Angular app:

**🔑 1. Components**

* The **heart** of any Angular app.
* Each component controls a part of the UI.
* Made up of:
  + HTML (template)
  + CSS or SCSS (styling)
  + TypeScript (logic)

**Example:**

ts

CopyEdit

@Component({

selector: 'app-hello',

templateUrl: './hello.component.html',

styleUrls: ['./hello.component.css']

})

export class HelloComponent {

message = 'Hello from Angular!';

}

**🧩 2. Modules**

* Angular apps are modular. The **root module** is typically AppModule.
* Modules help organize related components, directives, pipes, and services.

**Example:**

ts

CopyEdit

@NgModule({

declarations: [AppComponent, HelloComponent],

imports: [BrowserModule],

bootstrap: [AppComponent]

})

export class AppModule { }

**🔄 3. Templates**

* Define **how the UI looks**.
* HTML + Angular syntax (\*ngIf, \*ngFor, {{dataBinding}}, etc.)

**🖇️ 4. Data Binding**

* Connects the component’s logic to the template.
* Types:
  + **Interpolation:** {{ title }}
  + **Property binding:** [src]="imageUrl"
  + **Event binding:** (click)="handleClick()"
  + **Two-way binding:** [(ngModel)]="username"

**📦 5. Services and Dependency Injection (DI)**

* For sharing logic/data (like APIs or shared data) between components.
* Use @Injectable() to make a service.

**🌐 6. Routing (Optional but Common)**

* Angular's RouterModule lets you create single-page applications (SPAs) with multiple views.

ts

CopyEdit

const routes: Routes = [

{ path: '', component: HomeComponent },

{ path: 'about', component: AboutComponent }

];

Q-What is root component in angular?

**🌱 What is the Root Component?**

* It's the component that Angular **loads first** when the app starts.
* Usually named: AppComponent
* Declared in: app.component.ts
* Bootstrapped in: AppModule (app.module.ts)
* Rendered inside: index.html via a selector like <app-root>

**📁 Typical Structure**

lua

CopyEdit

src/

├── app/

│ ├── app.component.ts <-- Root component logic

│ ├── app.component.html <-- Root component template

│ ├── app.component.css <-- Optional styling

│ └── app.module.ts <-- Root module

├── index.html <-- Hosts <app-root>

Q-define angular architecture?

Angular is a **component-based framework** for building scalable web apps. Its architecture are:

| **Building Block** | **Description** |
| --- | --- |
| **Modules** | Group related components, directives, pipes, and services. Every Angular app has a **root module** (AppModule). |
| **Components** | Define the UI using templates, styles, and logic. Each component has a class, template, and metadata. |
| **Templates** | HTML + Angular syntax to define the layout of a component. |
| **Metadata** | Decorators (@Component, @NgModule, etc.) that tell Angular how to process a class. |
| **Data Binding** | Mechanism for binding data between component and template (one-way, two-way, etc.). |
| **Directives** | Extend or modify HTML behavior (\*ngIf, \*ngFor, custom directives). |
| **Services** | Business logic, data access, and shared code, usually injected using **Dependency Injection (DI)**. |
| **Routing** | Navigation and page management in a single-page application (SPA). |

**Q- MVC VS Angular?**

| **Concept** | **Description** |
| --- | --- |
| **MVC** | A software design **pattern** that separates an application into three main components: **Model**, **View**, and **Controller**. It's commonly used in backend frameworks like ASP.NET MVC, Spring MVC, etc. |
| **Angular** | A **front-end web framework** built by Google for building **single-page applications (SPAs)** using **components**, **modules**, **services**, and **TypeScript**. |

**Q-What is Dependency Injection in C#?**

**Dependency Injection (DI)** is a **design pattern** that allows a class to **receive its dependencies from the outside**, rather than creating them internally.

builder.Services.AddScoped<IMessageService, EmailService>();

AddScoped: New instance per HTTP request

AddSingleton: One instance for the entire app

AddTransient: New instance every time it's requested

**What is Data set and Data Reader?**

**DataSet (in C# / ADO.NET)**

**A DataSet is an in-memory, disconnected collection of data tables** that can store and manage data from a database. It can hold multiple tables, support relationships between them, and allows data to be modified without needing a constant connection to the database.

**📗 DataReader (in C# / ADO.NET)**

**A DataReader is a fast, forward-only, read-only stream of data** from a database. It requires an open connection and is used to read data quickly and efficiently, one row at a time.

**DataReader**

1. The ADO.NET DataReader is used to retrieve read-only (cannot update data back to a datasource) and forward-only (cannot read backward/random) data from a database.
2. Using of a DataReader increases application performance and reduces system overheads. This is due to one row at a time is stored in memory.
3. You create a DataReader by calling Command.ExecuteReader after creating an instance of the Command object.
4. This is a connected architecture: The data is available as long as the connection with database exists.

**Q-Entity Framework (EF) – Definition**

**Entity Framework (EF)** is an **Object-Relational Mapping (ORM)** framework for .NET that allows developers to **interact with a database using C# objects**, instead of writing raw SQL queries.

**In Simple Terms:**

* EF helps you **connect C# code to a database**.
* You can **create, read, update, and delete (CRUD)** data using C# classes.
* EF handles the SQL generation and database communication for you.

**Q-What is a Web API?**

A **Web API (Application Programming Interface)** is a **set of HTTP-based endpoints** that allow different software applications to **communicate with each other** over the **web**.

In .NET (C#), **Web API** is used to **build RESTful services** that can be accessed using HTTP verbs like **GET**, **POST**, **PUT**, and **DELETE**.

**Q-What is an Azure Function?**

**Azure Function** is a **serverless compute service** provided by Microsoft Azure that lets you run **small pieces of code (called functions)** in the cloud **without worrying about servers**.

**🧠 In Simple Terms:**

* You write **just the code** you need.
* Azure takes care of **running it, scaling it, and managing the infrastructure**.
* It's like a **"function as a service" (FaaS)**.

Q-how to write test case for checking name with validations

Great question! Writing a **test case for checking name validations** ensures that your application only accepts proper input. Below is a **sample test case** format for validating a **"Name"** field.

**✅ Example Test Cases for Name Field Validation**

| **Test Case ID** | **Description** | **Input** | **Expected Result** | **Status** |
| --- | --- | --- | --- | --- |
| TC001 | Name is valid | John Doe | Accepted | ✅ |
| TC002 | Name is empty | `` (empty) | Error: "Name is required" | ✅ |
| TC003 | Name contains numbers | John123 | Error: "Name should only contain letters" | ✅ |
| TC004 | Name contains special characters | @John! | Error: "Invalid characters in name" | ✅ |
| TC005 | Name is too short | J | Error: "Name must be at least 2 characters" | ✅ |
| TC006 | Name is too long | 256+ characters | Error: "Name must be less than 255 chars" | ✅ |
| TC007 | Name with leading/trailing spaces | John | Trimmed and Accepted | ✅ |
| TC008 | Name with mixed case letters | jOhN doE | Accepted | ✅ |

**💡 If you're writing unit tests (e.g. in C# with xUnit or NUnit):**

**Example C# Unit Test**

csharp

CopyEdit

[Theory]

[InlineData("John Doe", true)]

[InlineData("", false)]

[InlineData("John123", false)]

[InlineData("@John!", false)]

public void TestNameValidation(string name, bool expected)

{

bool result = ValidateName(name);

Assert.Equal(expected, result);

}

public bool ValidateName(string name)

{

if (string.IsNullOrWhiteSpace(name)) return false;

if (name.Length < 2 || name.Length > 255) return false;

if (!Regex.IsMatch(name, @"^[a-zA-Z\s]+$")) return false;

return true;

}

**What is Mock Testing?**

**Mock testing** is a **testing technique** where you **create fake (mock) objects** to **simulate the behavior of real objects** in unit tests—especially when the real objects are complex, slow, or not yet available.

**🎯 Why Use Mocks?**

Because:

* You don't want to hit a real **database**, **API**, or **email service** in every test.
* You want to test your code **independently** of its dependencies.
* You want **faster** and more **reliable** tests.

**What are joins in sql**

Different Types of SQL JOINs

Here are the different types of the JOINs in SQL:

* (INNER) JOIN: Returns records that have matching values in both tables
  + eg-

SELECT ProductID, ProductName, CategoryName  
FROM Products  
INNER JOIN Categories ON Products.CategoryID = Categories.CategoryID;

* LEFT (OUTER) JOIN: Returns all records from the left table, and the matched records from the right table
  + Eg-

SELECT Customers.CustomerName, Orders.OrderID  
FROM Customers  
LEFT JOIN Orders ON Customers.CustomerID = Orders.CustomerID  
ORDER BY Customers.CustomerName;

* RIGHT (OUTER) JOIN: Returns all records from the right table, and the matched records from the left table
  + Eg

SELECT Orders.OrderID, Employees.LastName, Employees.FirstName  
FROM Orders  
RIGHT JOIN Employees ON Orders.EmployeeID = Employees.EmployeeID  
ORDER BY Orders.OrderID;

* FULL (OUTER) JOIN: Returns all records when there is a match in either left or right table

Eg:

SELECT Customers.CustomerName, Orders.OrderID  
FROM Customers  
FULL OUTER JOIN Orders ON Customers.CustomerID=Orders.CustomerID  
ORDER BY Customers.CustomerName;

**Q-What is the difference between nUnit and xUnit?**

Both NUnit and xUnit are popular testing frameworks for .NET applications, but they differ in features and approaches to writing and organizing tests. Here's a comparison to help you decide which suits your needs:

| **Feature** | **NUnit** | **xUnit** |
| --- | --- | --- |
| **Philosophy** | Traditional, well-established framework. | Modern, emphasizes extensibility and flexibility. |
| **Assertions** | Provides a wide range of assertions with explicit syntax. | Similar assertion capabilities, often written with simpler syntax. |
| **Attributes** | Relies on attributes like [Test], [SetUp], [TearDown], and [TestFixture]. | Uses [Fact] and [Theory], along with custom data attributes for data-driven tests. |
| **Parallel Test Execution** | Supported, requires configuration. | Built-in support for parallel execution by default. |
| **Dependency Injection** | Less integration with DI frameworks. | Designed with DI in mind, making it a better fit for modern apps. |
| **Popularity** | Widely adopted in legacy projects and some current ones. | Increasingly favored for new projects due to modern features. |
| **Compatibility** | Compatible with older .NET Framework versions. | Focuses on .NET Core and newer versions. |

**Q- What is the difference between abstract and encapsulation**

Abstract and encapsulation are both fundamental concepts in object-oriented programming, but they serve different purposes. Here's a breakdown of their differences:

| **Aspect** | **Abstract** | **Encapsulation** |
| --- | --- | --- |
| **Definition** | The process of hiding implementation details and showcasing only the essential features of an object. | Bundling data (attributes) and methods (functions) into a single unit and restricting access to some components. |
| **Focus** | Deals with *what* an object does (behavior and interface). | Deals with *how* data is accessed and protected. |
| **Key Mechanism** | Achieved using abstract classes and interfaces. | Achieved using access modifiers like private, protected, or public. |
| **Purpose** | To define a blueprint or contract for derived classes without providing implementation details. | To ensure data security and prevent unintended interference. |
| **Examples** | Abstract methods (defined without implementation) in an abstract class. | Hiding sensitive data with private fields and exposing them through getters/setters. |

To summarize: **abstraction** is about providing a high-level view and leaving out complex details, while **encapsulation** is about safeguarding and controlling access to those details. They often work together—for instance, an abstract class defines a blueprint, while encapsulation protects its implementation.

**Q-what is entityframework and ADO.net**

**Entity Framework (EF)** and **ADO.NET** — both are used in .NET for **data access**, but they work quite differently:

**🔹 ADO.NET (ActiveX Data Objects .NET)**

**What it is:**  
ADO.NET is a **low-level** data access technology that allows you to directly interact with databases using **SQL commands**. It provides full control over your SQL queries, connections, and transactions.

**Key Features:**

* Works with **DataReader**, **DataSet**, **DataTable**, etc.
* You write **raw SQL queries** or stored procedures manually.
* Faster and more efficient for **performance-critical** applications.
* Gives you **fine-grained control** over how data is retrieved, manipulated, and updated.

**Example (C#):**

csharp

CopyEdit

SqlConnection con = new SqlConnection("your\_connection\_string");

SqlCommand cmd = new SqlCommand("SELECT \* FROM Users", con);

con.Open();

SqlDataReader reader = cmd.ExecuteReader();

while(reader.Read())

{

Console.WriteLine(reader["Username"]);

}

con.Close();

**🔹 Entity Framework (EF)**

**What it is:**  
Entity Framework is an **Object-Relational Mapper (ORM)**. It abstracts database access using C# classes instead of raw SQL. It allows developers to work with data using **.NET objects**.

**Key Features:**

* Supports **LINQ to Entities** for querying.
* Handles **migrations**, relationships, and **schema generation**.
* Automatically maps classes to tables (Code First / Database First).
* Slower than ADO.NET for large-scale operations, but **much faster for development**.

**Example (C# with EF Core):**

csharp

CopyEdit

using (var context = new AppDbContext())

{

var users = context.Users.ToList();

foreach (var user in users)

{

Console.WriteLine(user.Username);

}

}

**🔄 Main Differences:**

| **Feature** | **ADO.NET** | **Entity Framework** |
| --- | --- | --- |
| Abstraction Level | Low-level (manual SQL) | High-level (ORM) |
| Speed/Performance | Faster | Slightly slower |
| Ease of Development | More code & boilerplate | Easier & cleaner code |
| Learning Curve | Steeper for beginners | Easier for beginners |
| Query Language | SQL | LINQ |
| Schema Management | Manual | Automatic with migrations |

Want a real-world analogy too? Think of:

* **ADO.NET** = You manually write letters and post them yourself.
* **Entity Framework** = You use an email client that formats, sends, and tracks messages for you.

Let me know if you'd like to see a comparison with code side-by-side or dive into EF Core migrations or DbContext.

4o

**You said:**

difference?

**ChatGPT said:**

Sure! Here's a **quick and clear difference** between **ADO.NET** and **Entity Framework (EF)**:

**🔍 ADO.NET vs Entity Framework (EF)**

| **Feature** | **ADO.NET** | **Entity Framework (EF)** |
| --- | --- | --- |
| **Type** | Low-level Data Access | High-level ORM (Object-Relational Mapper) |
| **How It Works** | Manual SQL queries | Works with C# classes and LINQ |
| **Code Amount** | More code (boilerplate) | Less code (automatic mapping) |
| **Performance** | Faster (especially for large datasets) | Slightly slower due to abstraction |
| **Control Over SQL** | Full control | Limited (can use raw SQL if needed) |
| **Query Language** | SQL | LINQ (Language Integrated Query) |
| **Database Interaction** | Manual connection, command, reader | Handled by DbContext and models |
| **Setup & Learning Curve** | More setup, steeper learning curve | Easier for beginners |
| **Schema Management** | Manual | Handled via Migrations (Code First) |
| **Use Case** | Performance-critical apps | Rapid development and maintenance-friendly apps |

**Q-How asp.net mvc will work?**

Workflow in ASP.NET MVC:

1. A user makes a request by entering a URL or interacting with the application (e.g., clicking a button).
2. The **Routing System** maps the request to a specific **Controller** and **Action Method**.
3. The **Controller** processes the request, interacts with the **Model** if needed (e.g., querying data), and prepares the data for the **View**.
4. The **View** generates the HTML response, which is sent back to the user's browser.

Example:

Suppose you have a blog application:

1. The user visits http://example.com/Post/Details/1 (request).
2. The routing system maps this to the PostController's Details action method.
3. Inside the action method, the Controller fetches blog post data (e.g., post ID 1) from the **Model**.
4. The Controller passes this data to the **View**, which renders the blog post details as HTML.

**Q-what is stored procedure?**

**🔹 What is a Stored Procedure?**

A **Stored Procedure** is a **predefined set of SQL statements** stored in the **database**. It can

**🔧 Why use Stored Procedures?**

✅ **Reusable** – Write once, use many times  
✅ **Faster** – Stored on the server and precompiled  
✅ **Secure** – You can control access (e.g., allow calling it without giving full table access)  
✅ **Maintainable** – Business logic is centralized in the database  
✅ **Transactional** – You can handle multiple queries atomically

**🧾 Example of a Stored Procedure (SQL Server)**

Example

CREATE PROCEDURE SelectAllCustomers @City nvarchar(30)  
AS  
SELECT \* FROM Customers WHERE City = @City  
GO;

Execute the stored procedure above as follows:

Example

EXEC SelectAllCustomers @City = 'London';

**Q-What is a REST API?**

**REST** stands for **Representational State Transfer**. A **REST API** is a set of web services (URLs) that allow **communication between client and server** over the **HTTP protocol**, usually for data operations like Create, Read, Update, and Delete (**CRUD**).