

KOENIG
step forward



API Testing



Module 1: Java Basics



Module 1: Java Basics

- Setting up Java
- Data Types and Operators
- Control Statements
- Arrays and Strings
- OOPS- Class , Object, Method and Constructors
- “this” Keyword
- “static” Keyword
- “super” Keyword
- “final” Keyword
- Inheritance and Interfaces
- Exception Handling
- Java Array Lists and HashMap



Introduction to Java

- Java is a high-level, object-oriented, class-based programming language developed by Sun Microsystems (now owned by Oracle) in 1995.
- It is widely used for building:
 - Desktop applications
 - Web applications
 - Mobile applications (especially Android)
 - Enterprise software
 - Embedded systems



Introduction to Java

- Key Features of Java

Feature	Description
Simple	Easy to learn with a clean syntax similar to C or C++ but without complexity like pointers.
Object-Oriented	Everything in Java revolves around objects and classes.
Platform-Independent	"Write Once, Run Anywhere" — the most significant feature of Java.
Robust	Strong memory management, exception handling, and type checking.
Secure	Provides a secure execution environment through bytecode verification and the Security Manager.
Multithreaded	Built-in support for concurrent programming (multiple threads).



Introduction to Java

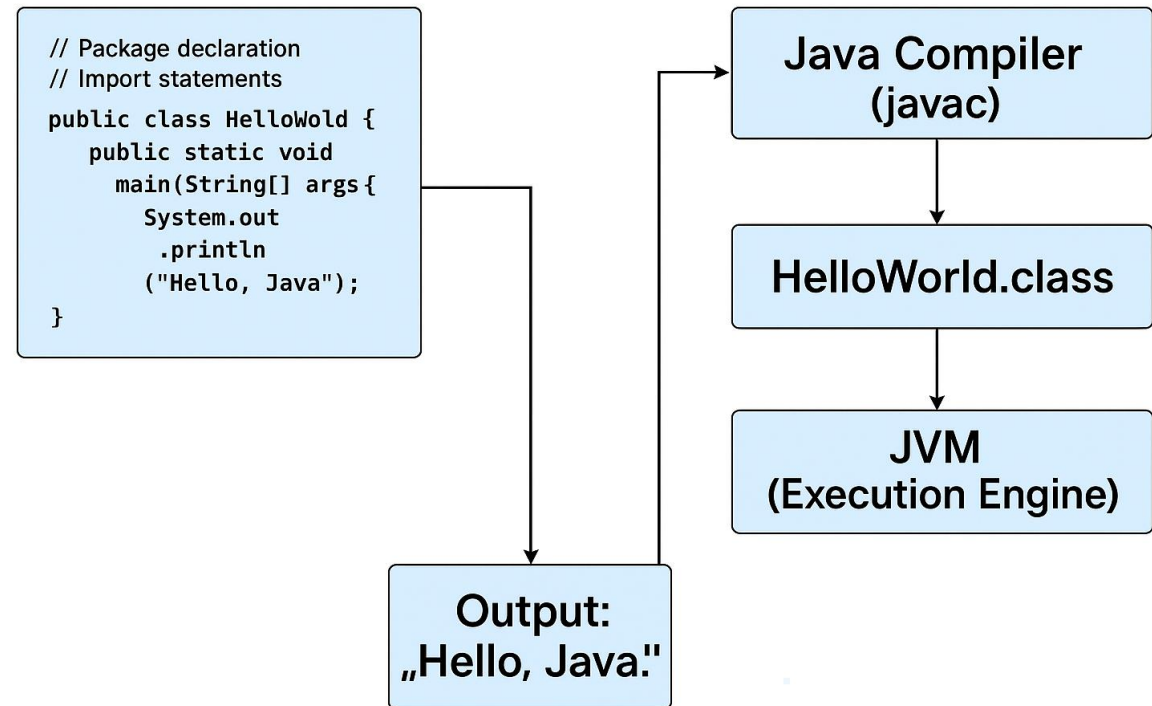
- Structure of a Java Class

```
// Package declaration (optional)
package com.example;

// Import statements (optional)
import java.util.Scanner;

// Class declaration
public class HelloWorld {

    // Main method – entry point
    public static void main(String[] args) {
        // Print statement
        System.out.println("Hello, Java!");
    }
}
```





Setting up Java

- IntelliJ IDEA Community Edition is a free and open-source integrated development environment (IDE) developed by JetBrains.
- It is designed primarily for Java development, but also supports other JVM-based languages like Kotlin, Groovy, and Scala.
- Download and install the IntelliJ IDEA Community Edition
 - Go to the Official Website: <https://www.jetbrains.com/idea/download>
 - Choose Community Edition.
 - Click Download for your operating system (Windows/Linux/macOS).
 - Run the downloaded installer and follow the installation wizard.



Setting up Java

- Launch IntelliJ IDEA and Create New Project
 1. Open IntelliJ IDEA.
 2. Click “New Project”.
 3. In the New Project Wizard:
 - Select Java (make sure JDK is installed and selected)
 - Choose a Project SDK (you can download a JDK from [here](#) too)
 - Click Next, then Finish
- If you don't have a JDK:
 - IntelliJ lets you download JDK automatically.
 - Recommended: JDK 21 (LTS) or higher




Setting up Java

- Create Java Class
 1. In the Project Explorer, right-click src → New → Java Class.
 2. Enter class name, e.g., Main.
 3. Write your Java code inside:

```
public class Main {  
    public static void main(String[] args) {  
        System.out.println("Welcome to Java!");  
    }  
}
```



Setting up Java

- Run the Java Program
 1. Right-click inside the code editor → Run 'Main.main()'
 2. Or click the green  Run button at the top right.
 3. The output will appear in the console at the bottom:

Welcome to Java!



Java Data Types

- A data type in Java defines the kind of data a variable can store, how much memory it uses, and the operations allowed on it.
- Data types ensure type safety during compilation.
- Data types prevent invalid operations (e.g., adding a number to a string incorrectly).
- Data types help the compiler allocate memory efficiently.
- In Java, data types are categorized into two main types
 - Primitive Data Types (Built-in)
 - Reference Data Types



Java Data Types

- Primitive Data Types (Built-in)
 - These are the most basic data types provided by Java. They are not objects and store actual values.

Type	Size	Description	Example
byte	1 byte	Whole number from -128 to 127	byte b = 100;
short	2 bytes	Whole number from -32K to 32K	short s = 30000;
int	4 bytes	Common integer type	int x = 100000;
long	8 bytes	Large integer values	long l = 123456L;
float	4 bytes	Decimal, single precision	float f = 3.14f;
double	8 bytes	Decimal, double precision	double d = 3.14159;
char	2 bytes	Single character (Unicode)	char c = 'A';
boolean	1 bit	True/False	boolean b = true;



Java Data Types

- Reference Data Types
 - These types refer to objects, not actual values. Memory stores references (addresses) of the object.

Type	Description	Example
String	Represents a sequence of characters	<code>String s = "Hello";</code>
Arrays	Group of similar data elements	<code>int[] arr = {1,2,3};</code>
Classes	Blueprint for objects	<code>class Person {}</code> <code>Person p = new Person();</code>
Interfaces	Reference to a set of methods (contract)	<code>interface MyInterface {}</code> <code>class MyClass implements MyInterface {}</code> <code>MyClass obj = new MyClass();</code>



Java Variables

- Variable Declaration
 - Declaring a variable means reserving memory space and specifying the data type.
- Syntax:
 dataType variableName;
- Example:
 int age; // declaration
 String name; // declaration



Java Variables

- Variable Initialization
 - Initialization means assigning a value to the variable for the first time.
- Syntax:
 - `variableName = value;`
 - Or declare and initialize in one line:
`dataType variableName = value;`
- Example:
 - `int age = 25; // declare + initialize`
 - `String name = "John"; // declare + initialize`



Java Operators

- Arithmetic Operators
 - Used to perform basic mathematical operations.

Operator	Description	Example	Result
+	Addition	5 + 2	7
-	Subtraction	5 - 2	3
*	Multiplication	5 * 2	10
/	Division	5 / 2	2 (int)
%	Modulus (Remainder)	5 % 2	1



Java Operators

- Relational (Comparison) Operators
 - Used to compare two values.

Operator	Description	Example	Result
==	Equal to	a == b	true / false
!=	Not equal to	a != b	true / false
>	Greater than	a > b	true / false
<	Less than	a < b	true / false
>=	Greater or equal	a >= b	true / false
<=	Less or equal	a <= b	true / false



Java Operators

- Logical Operators
 - Used to combine multiple boolean expressions.

Operator	Description	Example	Result
&&	Logical AND	<code>a > 3 && b < 10</code>	true if both true
 	Logical OR	<code>a > 3 b < 10</code>	true if any one true
!	Logical NOT	<code>!(a > b)</code>	Opposite of condition



Java Operators

- Unary Operators
 - Unary operators operate on a single operand.

Operator	Description	Example	Result
+	Unary plus	+a	a
-	Unary minus	-a	Negative a
++	Increment (prefix/postfix)	++a or a++	Increases a
--	Decrement (prefix/postfix)	--a or a--	Decreases a
!	Logical NOT	!true	false
~	Bitwise complement	~5	-6



Java Operators

- Ternary Operator
 - The ternary operator is a shortcut for if-else and takes three operands.
- Syntax:
condition ? value_if_true : value_if_false;
- Example:
int a = 10, b = 20;
int max = (a > b) ? a : b;
System.out.println("Max = " + max); // Max = 20



Control Statements

- if Statement
 - Used to execute a block of code only if a condition is true.
- Syntax:

```
if (condition) {  
    // code to execute if condition is true  
}
```



Control Statements

- if-else Statement
 - Used to execute one block if condition is true, otherwise another block.
- Syntax:

```
if (condition) {  
    // true block  
} else {  
    // false block  
}
```



Control Statements

- nested if-else Statement
 - Used when we have to make multiple decisions (conditions inside conditions).

- Syntax:

```
if (condition1) {  
    if (condition2) {  
        // inner true block  
    } else {  
        // inner false block  
    }  
} else {  
    // outer false block  
}
```




Control Statements

- switch-case Statement
 - Used to test a variable against multiple constant values.

- Syntax:

```
switch (variable) {  
    case value1:  
        // code block  
        break;  
    case value2:  
        // code block  
        break;  
    default:    // default code block  
}
```



Control Statements

- In Java, loops are used to execute a block of code repeatedly until a certain condition is met.

- for Loop Syntax:

```
for (initialization; condition; update) {  
    // loop body  
}
```

- while Loop

- Executes as long as the condition is true.

- Syntax:

```
while (condition) {  
    // loop body  
}
```



Control Statements

- do-while Loop
 - Executes at least once, even if the condition is false.
- Syntax:

```
do {  
    // loop body  
} while (condition);
```



Control Statements

- In Java, break, and continue are control flow keywords used to change the normal flow of program execution, especially inside loops and methods.
- break Keyword
 - Used to exit a loop or switch statement immediately.
- Syntax:
 break;



Control Statements

- continue Keyword
 - Used to skip the current iteration and continue with the next iteration of the loop.
 - Syntax:
`continue;`



Java Arrays

- An array is a list of elements of the same type arranged in a single row.
- Use arrays for linear data (e.g., marks, ages).
- Syntax:

// Declaration

```
dataType[] arrayName;
```

// Declaration + Initialization

```
dataType[] arrayName = new dataType[size];
```

// Declaration + Initialization + Values

```
dataType[] arrayName = {val1, val2, val3};
```



Java Arrays

- Array Example:

```
int[] numbers = {10, 20, 30, 40};  
for (int i = 0; i < numbers.length; i++) {  
    System.out.println(numbers[i]);  
}
```

- Output:

```
10  
20  
30  
40
```



String in Java

- In Java, String is one of the most commonly used classes, representing a sequence of characters.
- Strings in Java are immutable, meaning once created, they cannot be changed.
- Declaring and Initializing Strings
 - `String str1 = "Hello";` `// String literal`



Common String methods

- `length()`
 - Returns the number of characters in the string.
`String str = "Java";`
`System.out.println(str.length());` // Output: 4
- `equals(String anotherString)`
 - Compares the contents of two strings (case-sensitive).
 - `String a = "Java";`
 - `String b = "Java";`
 - `System.out.println(a.equals(b));` // true



Common String methods

- `equalsIgnoreCase(String str)`
 - Compares content, ignoring case.
`"Java".equalsIgnoreCase("java"); // true`
- `substring(int beginIndex, int endIndex)`
 - Extracts part of the string from `beginIndex` (inclusive) to `endIndex` (exclusive).
`String str = "Programming";`
`System.out.println(str.substring(0, 6)); // Output: Progra`
`System.out.println(str.substring(6)); // Output: mming`



Common String methods

- `toLowerCase()`
 - Converts all characters to lowercase.
`String s = "HeLLo";`
`System.out.println(s.toLowerCase()); // hello`
- `toUpperCase()`
 - Converts all characters to uppercase.
`String s = "HeLLo";`
`System.out.println(s.toUpperCase()); // HELLO`



Common String methods

- trim()
 - Removes leading and trailing whitespaces.
`String s = " Hello Java ";`
`System.out.println(s.trim()); // "Hello Java"`



OOPS- Class , Object, Method and Constructors

- Class

- A class is a blueprint for creating objects.
- It defines attributes (variables) and methods (functions).

```
public class Car {  
    // Attributes (fields)  
    String brand;  
    int speed;  
    // Method  
    void startEngine() {  
        System.out.println(brand + " engine started.");  
    }  
}
```



OOPS- Class , Object, Method and Constructors

- Object

- An object is an instance of a class. It represents a real-world entity with state and behavior.

```
public class Main {  
    public static void main(String[] args) {  
        Car car1 = new Car();    // Object creation  
        car1.brand = "Honda";    // Accessing attribute  
        car1.startEngine();      // Calling method  
    }  
}
```



OOPS- Class , Object, Method and Constructors

- Attributes

- Attributes (also called fields or instance variables) define the state of an object.

- Example:

```
String brand;  
int speed;
```

- Methods

- Methods define the behavior of an object. They are like functions defined inside a class.

- Example:

```
void startEngine() {  
    System.out.println("Engine started");  
}
```



OOPS- Class , Object, Method and Constructors

- In Java, classes define attributes (fields) and methods (functions).
- When you create an object from a class, that object becomes a reference to access these members (variables and methods) using the dot (.) operator.
- It allows encapsulation (grouping data and methods together).
- You interact with real-world entities as objects (e.g., Student, Car).
- Helps in organizing and reusing code.



OOPS- Class , Object, Method and Constructors

```
public class Car {  
    String color;  
    void drive() {  
        System.out.println("The car is driving.");  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Car myCar = new Car();    // Object creation  
        myCar.color = "Red";      // Accessing attribute  
        myCar.drive();            // Calling method  
    }  
}
```



OOPS- Class , Object, Method and Constructors

- In Java, a constructor is a special method used to initialize objects.
- It has the same name as the class and no return type (not even void).
- Default Constructor
 - A default constructor is a constructor with no parameters.
 - If you don't define any constructor in a class, Java provides one automatically.

- Syntax:

```
public class Car {  
    Car() { // Default constructor  
        System.out.println("Car object created!");  
    }  
}
```



OOPS- Class , Object, Method and Constructors

- Parameterized Constructor
 - A parameterized constructor accepts arguments to initialize the object with specific values.

- Syntax:

```
public class Car {  
    String color;  
    Car(String c) { // Parameterized constructor  
        color = c;  
    }  
}
```



Use of 'this' keyword

- The 'this' keyword in Java is a reference variable that refers to the current object of a class.
- Distinguish Between Instance and Local Variables
 - When local variables (e.g., constructor parameters) shadow instance variables, this helps refer to the current instance variable.

```
public class Student {  
    String name;  
    Student(String name) {  
        this.name = name; // 'this.name' refers to instance variable  
    }  
}
```



Use of 'this' keyword

- Invoke Current Class Constructor

- You can call another constructor in the same class using this().

```
public class Student {  
    String name;  
    int age;  
    Student() {  
        this("Unknown", 0); // Calls the parameterized constructor  
    }  
    Student(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
}
```



Use of 'this' keyword

- Access Current Class Methods and Variables
 - Even when there's no ambiguity, this can explicitly access members of the class.

```
public class Example {  
    void print() {  
        System.out.println("Print method");  
    }  
  
    void callPrint() {  
        this.print(); // Same as just print(), but explicit  
    }  
}
```



Use of 'static' keyword

- In Java, static is a keyword used for memory management.
- It can be applied to variables, methods, blocks, and nested classes.
- Static members belong to the class rather than to any specific object.
- Static Variables (Class Variables)
 - Shared among all objects of the class.
 - Initialized only once, at the time of class loading.



Use of 'static' keyword

- Static Variables (Class Variables) - Example

```
class Student {  
    static String school = "Greenwood High"; // static variable  
    String name;  
  
    Student(String name) {  
        this.name = name;  
    }  
}
```

- Usage:

```
System.out.println(Student.school); // Accessed without creating an object
```




Use of 'static' keyword

- Static Methods

- Can be called without creating an object.
- Cannot access non-static (instance) members directly.
- Often used for utility or helper methods.

```
class MathUtils {  
    static int square(int n) {  
        return n * n;  
    }  
}
```

- Usage:

```
int result = MathUtils.square(5); // No object needed
```



Use of 'static' keyword

- Static Block

- Runs once when the class is loaded.
- Used for static initialization (e.g., setting up static variables).

```
class Config {  
    static String appName;  
  
    static {  
        appName = "MyApp";  
        System.out.println("Static block executed");  
    }  
}
```

- Usage:

```
System.out.println(Config.appName); // Static block runs only once
```



Inheritance in Java

- Inheritance allows one class to inherit the properties and behaviors (fields and methods) of another class.
- It helps with code reusability and creating hierarchical relationships.
- Represented using extends keyword.
- It defines a parent-child or superclass-subclass relationship.
- It's also called class inheritance.



Inheritance in Java

- Inheritance – Example

- Dog IS-A Animal → Because Dog inherits from Animal.

```
class Animal {  
    void eat() {  
        System.out.println("This animal eats food.");  
    }  
}  
  
class Dog extends Animal {  
    void bark() {  
        System.out.println("The dog barks.");  
    }  
}
```



Use of 'super' keyword

- The 'super' keyword in Java is used to refer to the immediate parent class. It's mainly used in inheritance scenarios.
- Access Parent Class Variables
 - If a child class has a variable with the same name as the parent, super helps to distinguish.
- Access Parent Class Methods
 - If a child overrides a method from the parent, super can call the parent's version.
- Call Parent Constructor
 - super() is used inside a child constructor to call the parent constructor.
 - Must be the first statement in the constructor.



Use of 'final' keyword

- The final keyword in Java is used to restrict modification.
- final Variable
 - A final variable's value cannot be changed once initialized.
- Example:

```
final int MAX_VALUE = 100;  
MAX_VALUE = 200; // Error: Cannot assign a value to final variable
```
- Use case:
 - Constants, fixed configuration, unchangeable IDs.



Use of 'final' keyword

- final Method

- A final method cannot be overridden by subclasses.

- Example:

```
class Animal {  
    final void sound() {  
        System.out.println("Animal  
makes sound");  
    }  
}
```

```
class Dog extends Animal {  
    void sound() { // Error  
        System.out.println("Dog barks");  
    }  
}
```

- Use case:

- To prevent altering core behavior in subclasses.



Use of 'final' keyword

- final Class
 - A final class cannot be extended (inherited from).
- Example:

```
final class MathUtils {  
    // Utility methods  
}  
  
// Attempting to extend  
class AdvancedMath extends MathUtils { // Error  
}
```
- Use case:
 - Security, immutability, and utility classes like `java.lang.String`, `java.lang.Math`.



Use of 'final' keyword

- Final with Reference Types
 - A final object reference cannot point to another object, but its internal state can change.
- Example:

```
final Student john = new Student("John", 25);  
john.setAge(26); // Allowed  
john = new Student("Peter", 24); // Not allowed
```



Interface in Java

- An interface in Java is a blueprint of a class.
- It defines method signatures (abstract methods) without implementation.
- Classes implement interfaces to define the actual behavior.

Benefit	Explanation
Abstraction	Focuses on what the class should do, not how.
Multiple inheritance	A class can implement multiple interfaces, unlike extending classes.
Flexibility and decoupling	Code becomes more modular and testable.
Design pattern integration	Essential in Strategy, Factory, Observer, etc.



Interface in Java

- Interfaces – Example

```
interface Vehicle {  
    void start(); // public & abstract by default  
    void stop();  
}  
class Car implements Vehicle {  
    public void start() {  
        System.out.println("Car starts");  
    }  
    public void stop() {  
        System.out.println("Car stops");  
    }  
}
```



Interface in Java

- In Java, interfaces allow abstraction and are the only way to achieve multiple inheritance of type.
- An interface is a reference type, similar to a class, that can contain:
 - Abstract methods (implicitly public and abstract)
 - Default and static methods (from Java 8)
 - Constants (public static final)

```
interface Animal {  
    void makeSound(); // abstract method  
}
```



Interface in Java

- Multiple Inheritance via Interface
 - Java supports multiple inheritance of behavior using interfaces

```
interface A {  
    void methodA();  
}
```

```
interface B {  
    void methodB();  
}
```

```
// Multiple inheritance using interfaces  
class C implements A, B {  
    public void methodA() {  
        System.out.println("From A");  
    }  
    public void methodB() {  
        System.out.println("From B");  
    }  
}
```



Exception Handling

- An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions.
- Java provides a powerful exception-handling mechanism to gracefully manage errors like:
 - Division by zero
 - Accessing invalid array index
 - File not found
 - Null references, etc.



Exception Handling

- Java exceptions are mainly divided into two types:
- Checked Exceptions (Compile-Time Exceptions)
 - These are checked by the compiler at compile time.
 - The program won't compile unless these are either handled with try-catch or declared using throws.
- Examples:
 - IOException
 - FileNotFoundException
 - SQLException



Exception Handling

- Unchecked Exceptions (Runtime Exceptions)
 - These are not checked by the compiler.
 - Occur due to programming mistakes, like dividing by zero or null access.
 - You can handle them, but it's not mandatory.
- Examples:
 - ArithmeticException
 - NullPointerException
 - ArrayIndexOutOfBoundsException



Exception Handling

- The try-catch Block
 - In Java, the try-catch block is used to handle exceptions gracefully without crashing the program.

- Syntax:

```
try {  
    // Code that might throw an exception  
} catch (ExceptionType e) {  
    // Code to handle the exception  
}
```



Exception Handling

- Multiple Catch Blocks
 - You can handle different types of exceptions using multiple catch blocks.
 - Each block catches a specific exception.

- Syntax:

```
try {  
    // risky code  
} catch (ArithmeticException e) {  
    // handle arithmetic error  
} catch (ArrayIndexOutOfBoundsException e) {  
    // handle array index error  
} catch (Exception e) {  
    // handle all other exceptions  
}
```



Exception Handling

- finally Block
 - The finally block is always executed after the try block (with or without a catch), regardless of whether an exception is thrown or not.

- Syntax:

```
try {  
    // risky code  
} catch (Exception e) {  
    // handle exception  
} finally {  
    // cleanup code that always runs  
}
```



Exception Handling

- throw Keyword

- The throw keyword is used to explicitly throw an exception (usually custom or runtime exception).
- Syntax:

```
throw new ExceptionType("message");
```

- throws Keyword

- The throws keyword is used in method declaration to indicate that the method might throw a checked exception, and the caller must handle it.
- Syntax:

```
public void readFile() throws IOException {  
    // code that might throw IOException  
}
```



Exception Handling

- A custom exception is a user-defined class that extends Java's Exception classes to indicate specific error conditions.

- Extend Exception

```
public class InvalidAgeException extends Exception {  
    public InvalidAgeException(String message) {  
        super(message);  
    }  
}
```



Java Array Lists and HashMap

- The Java Collections Framework is a unified architecture for storing, retrieving, and manipulating groups of objects (data collections) efficiently.
- Key Components of the Collections Framework
 - Interfaces
 - Abstract data structures (e.g., List, Set, Map)
 - Implementations
 - Concrete classes (e.g., ArrayList, HashSet, LinkedList, HashMap)
 - Algorithms
 - Utility methods for searching, sorting, etc. (e.g., Collections.sort())



Java Array Lists and HashMap

- List
 - A List is an ordered collection that allows duplicate elements.
- Key Features:
 - Maintains insertion order
 - Allows duplicate values
 - Access via index (0-based)
- Common Implementations:
 - ArrayList
 - LinkedList



Java Array Lists and HashMap

- ArrayList – Dynamic Array

```
import java.util.ArrayList;
public class ArrayListDemo {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("Java");
        list.add("Python");
        list.add("Java"); // Allows duplicates
        System.out.println("ArrayList: " + list); // [Java, Python, Java]
    }
}
```

- Use case: Random access, frequent reads
- Backed by array → Fast for get(index)



Java Array Lists and HashMap

- Map
 - A Map stores key-value pairs.
- Key Features:
 - Unique keys
 - Values can be duplicated
 - Not a subtype of Collection
- Common Implementations:
 - HashMap – no order
 - LinkedHashMap – insertion order



Java Array Lists and HashMap

- HashMap – Key-value pairs

```
import java.util.HashMap;
public class HashMapDemo {
    public static void main(String[] args) {
        HashMap<Integer, String> map = new HashMap<>();
        map.put(101, "Alice");
        map.put(102, "Bob");
        map.put(101, "Charlie"); // Replaces value for key 101
        System.out.println("HashMap: " + map); // {101=Charlie, 102=Bob}
        System.out.println("Value for key 102: " + map.get(102)); // Bob
    }
}
```

- Use case: Lookup by key
- Fast access, keys must be unique



Hands-on Lab: Library Book Management (Using ArrayList)

- Create a system to manage library books, where each book has: bookId, title, author, and price.
- Use ArrayList to store and display the book details



Module 2: Overview of API Testing



Module 2: Overview of API Testing

- Client Server Architecture
- What is API and API Testing?
- Difference between API Testing and Unit Testing
- Overview of Web Services
- HTTP Structure
- HTTP Methods
- Status Codes



Client Server Architecture

- Client–Server Architecture is a distributed computing model where tasks are divided between clients (requesters of services) and servers (providers of services).
 - Client → Sends a request (e.g., browser, mobile app).
 - Server → Processes the request and sends back a response (e.g., web server, database server).
- It's the backbone of most networked applications, including the web, databases, email, and cloud systems.



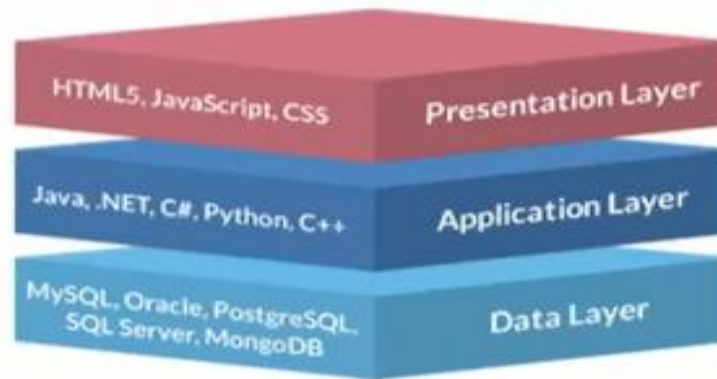
Client Server Architecture

- Types of Client–Server Architecture
- 1-Tier (Monolithic)
 - Client, server, and database in one system
 - Example: Standalone desktop apps
- 2-Tier
 - Client \leftrightarrow Server (DB or Application server)
 - Example: Client app directly communicating with DB
- 3-Tier (Most Common)
 - Presentation Layer (Client/UI)
 - Application Layer (Server/Business Logic)
 - Data Layer (Database Server)
 - Example: Web apps with frontend, backend, and database

Client Server Architecture



Client/Server Architecture





Client Server Architecture

- Presentation Layer
 - The presentation tier is the user interface and communication layer of the application, where the end user interacts with the application.
 - Its main purpose is to display information to and collect information from the user.
- Business Layer
 - In this middle layer, information that is collected in the presentation layer is processed - sometimes against other information in the data layer- using business logic, a specific set of business rules
 - The application tier can also add, delete, or modify data in the data layer.
- Data Layer
 - The data layer, sometimes called database layer, data access layer or back-end, is where the information that is processed by the application is stored and managed.



What is API and API Testing?

- API (Application Programming Interface) is a set of rules and protocols that allows one software application to communicate with another.
- It defines how requests and responses should be made.
- Think of it like a waiter in a restaurant:
 - You (client) place an order.
 - The waiter (API) takes it to the kitchen (server).
 - The kitchen prepares food and gives it back via the waiter.
- Examples:
 - REST API: GET /users → Returns a list of users in JSON.
 - Payment API: Allows apps to connect with PayPal or Stripe.
 - Maps API: Embeds Google Maps into apps.



What is API and API Testing?

- API Testing is a type of software testing that verifies APIs work correctly, reliably, securely, and perform well.
- Unlike UI testing, it directly tests the API's endpoints, requests, and responses.
- Key Focus of API Testing
 - Functionality → Does the API return correct results?
 - Reliability → Does it respond consistently?
 - Security → Is it protected against unauthorized access?
 - Performance → How fast does it respond under load?
 - Error Handling → Are invalid inputs handled properly?



Difference between API Testing and Unit Testing

Feature	API Testing	Unit Testing
Definition	Tests the interfaces between different software components to verify that requests and responses work correctly.	Tests individual units or components of the code (like functions or methods) in isolation.
Focus	Functionality, reliability, performance, and security of APIs.	Logic correctness of a single function/class/module.
Scope	Broader: Can involve multiple modules, databases, or servers.	Narrow: Focused on a single unit or method.
Dependencies	Often depends on external systems like databases, servers, or other APIs.	Independent; mocks or stubs are used to isolate the unit.
Performed By	QA engineers, testers, sometimes developers.	Developers (usually during development phase).
Tools	Postman, Rest Assured, SoapUI.	JUnit, NUnit, Jest, Mocha, PyTest.
Goal	Ensure API behaves as expected under various scenarios, including invalid inputs and load.	Ensure a single unit works correctly according to the design.
Testing Level	Medium to high level (integration/functional).	Low level (code-level).



Overview of Web Services

- A Web Service is a software system that allows applications to communicate over a network (usually the Internet) using standard protocols.
- It enables interoperability between different systems (different languages, platforms, or devices).
- Essentially, a Web Service exposes functions or data that other applications can use.



Overview of Web Services

- Key Characteristics of Web Services:
- **Platform-independent**
 - Works across different operating systems and programming languages.
- **Standards-based**
 - Uses protocols like HTTP, HTTPS, SOAP, REST, and data formats like XML, JSON.
- **Reusable**
 - Can be used by multiple applications.
- **Network accessible**
 - Applications communicate over LAN or Internet.



Overview of Web Services

- Types of Web Services
- **SOAP Web Services**
 - Protocol-based (Simple Object Access Protocol)
 - Uses XML for message format
 - Has strict standards (WSDL for describing services)
 - Supports built-in error handling and security
- **RESTful Web Services**
 - Architectural style (Representational State Transfer)
 - Can use JSON, XML, or plain text
 - Uses standard HTTP methods: GET, POST, PUT, DELETE
 - Lightweight, faster, and widely used for web/mobile apps



HTTP Structure

- HTTP (Hypertext Transfer Protocol) is the foundation of data communication on the web.
- It defines how clients (browsers, apps) request data from servers and how servers respond.
- Protocol used for web pages, APIs, and resources.
- Stateless: Each request is independent.



HTTP Structure

- HTTP Request Structure
 - An HTTP request is sent by the client to the server.
- Basic Structure:
 - <Method> <URL> <HTTP Version>**
 - Headers**
 - [Optional Body]**
- Request Line
 - Method: Type of operation (GET, POST, PUT, DELETE, PATCH)
 - URL: Resource address (/users/1)
 - HTTP Version: e.g., HTTP/1.1
- Example:
 - GET /users/1 HTTP/1.1**



HTTP Structure

- Headers
 - Provide metadata about the request.
- Examples:
 - Content-Type: application/json
 - Authorization: Bearer abc123
- Body (Optional)
 - Contains data sent to the server (used in POST, PUT, PATCH).
- Example (JSON):

```
{  
  "username": "john",  
  "password": "1234"  
}
```



HTTP Structure

- HTTP Response Structure
 - An HTTP response is sent by the server to the client.
- Basic Structure:
 - <HTTP Version> <Status Code> <Status Message>**
 - Headers**
 - [Optional Body]**
- Status Line
 - HTTP/1.1 200 OK**
 - Status codes indicate result:
 - 2xx → Success (200 OK)
 - 3xx → Redirection (301 Moved Permanently)
 - 4xx → Client error (404 Not Found)
 - 5xx → Server error (500 Internal Server Error)



HTTP Structure

- Headers
 - Provide metadata about the response.
- Examples:
 - Content-Type: application/json
 - Content-Length: 123
- Body (Optional)
 - Contains requested data or error message.
- Example (JSON):

```
{  
  "id": 1,  
  "name": "Ashok Kumar",  
  "email": "ashok@example.com"  
}
```



HTTP Methods

- HTTP Methods (also called verbs) define the action a client wants to perform on a resource on the server.
- Common HTTP Methods

Method	Purpose	Request Body	Example Usage
GET	Retrieve data from server	No	GET /users/1 → Fetch user with ID 1
POST	Create a new resource	Yes	POST /users with JSON body { "name": "John" }
PUT	Update an existing resource (or create if not exists)	Yes	PUT /users/1 with JSON body { "name": "John Smith" }
PATCH	Partially update an existing resource	Yes	PATCH /users/1 with JSON { "email": "john@example.com" }
DELETE	Remove a resource	Usually no	DELETE /users/1 → Delete user with ID 1



Status Codes

- HTTP Status Codes are three-digit numbers returned by the server to indicate the result of a client's request.
- They help clients understand whether a request succeeded, failed, or requires further action.

Category	Range	Meaning
1xx	100–199	Informational → Request received, continuing process
2xx	200–299	Success → Request was successfully processed
3xx	300–399	Redirection → Client needs to take additional action
4xx	400–499	Client Error → Issue with request (wrong input, unauthorized)
5xx	500–599	Server Error → Server failed to process a valid request



Status Codes

Status Code	Meaning	Example / Use
200 OK	Request succeeded	GET /users/1 → Returns user data
201 Created	Resource successfully created	POST /users → New user created
204 No Content	Request succeeded but no body	DELETE /users/1 → User deleted
301 Moved Permanently	Resource permanently moved	Redirect URL changed
302 Found	Resource temporarily moved	Temporary redirect
400 Bad Request	Invalid request syntax	Missing required fields
401 Unauthorized	Authentication required or failed	Access protected resource without token
403 Forbidden	Request valid but server refuses	User has no permission
404 Not Found	Resource not found	Request non-existing user
405 Method Not Allowed	HTTP method not allowed	POST on /users/1 where only GET allowed
500 Internal Server Error	Server encountered an error	Server crashed or exception occurred
503 Service Unavailable	Server temporarily unavailable	Server under maintenance



Module 3: Testing API's using Postman Tool



Module 3: Testing API's using Postman Tool

- Overview of Postman Tool
- Setup and Installation
- Basic HTTP requests in Postman
- Create Dummy API's
- Basic of JSON and JSON Path
- Validating Responses in Postman and JavaScripting
- Postman Scripts and Variables
- Chaining Requests



Module 3: Testing API's using Postman Tool

- Data Driven Testing Using JSON and CSV Files
- File Upload and Download API
- Authorization Types
- cURL
- Validating XML Response
- Converting XML Resposne to JSON
- Documentation and Publishing



Overview of Postman Tool

- Postman is a popular API testing and development tool used by developers and testers to send requests, receive responses, and automate API testing.
- Supports REST, SOAP, GraphQL, gRPC APIs
- Available as desktop app, web app, and Chrome extension
- Used for manual testing, automated testing, and collaboration



Overview of Postman Tool

- Advantages of Postman
 - Easy to use for beginners and advanced users
 - Supports both manual and automated testing
 - Provides visualization of responses (JSON, XML, HTML)
 - Enables team collaboration and version control
 - Reduces dependency on UI for testing APIs
- Typical Use Cases
 - Manual API testing – Quickly send requests and check responses.
 - Automated API testing – Validate responses using test scripts.
 - Monitoring & Debugging – Check API uptime and diagnose issues.



Setup and Installation

- Download Postman
 - Visit the official website: <https://www.postman.com/downloads/>
 - Select your Operating System (Windows, macOS, Linux)
 - Click Download
- Installation for Windows:
 - Run the downloaded .exe file
 - Click Next → Install → Finish
 - Launch Postman
- Sign In / Create Account
 - Postman can be used without an account, but signing in allows:
 - Syncing collections across devices
 - Collaboration with team
 - Options: Email, Google, GitHub



Basic HTTP requests in Postman

- Open Postman
 - Launch Postman
 - Click “New → Request”
 - Name your request and optionally add it to a collection
- Select HTTP Method
 - Postman supports all HTTP methods:
 - GET – Retrieve data
 - POST – Create a new resource
 - PUT – Update an existing resource
 - PATCH – Partially update a resource
 - DELETE – Delete a resource
 - Use the dropdown next to the URL field to select the method.



Basic HTTP requests in Postman

- Enter Request URL
 - Type the endpoint URL in the request URL field
 - Example test URL: <https://jsonplaceholder.typicode.com/posts>
- Add Body (For POST, PUT, PATCH)
 - Click Body tab → Select raw → Choose JSON
 - Example POST body:

```
{  
  "title": "My First Post",  
  "body": "This is a test post",  
  "userId": 1  
}
```



Basic HTTP requests in Postman

- Send Request
 - Click Send
 - Postman shows:
 - Status Code → e.g., 200 OK, 201 Created
 - Response Body → Data returned by API
 - Response Time → How fast the server responded
 - Headers → Metadata
- Save Request (Optional)
 - Click Save → Add to Collection for future use



Create Dummy API's

- Dummy APIs (or Mock APIs) are fake APIs that simulate real API behavior.
- They are used for:
 - Practicing API testing
 - Developing frontend apps before backend is ready
 - Testing automation scripts



Create Dummy API's

- Using JSONPlaceholder (Free Online Service)
 - URL: <https://jsonplaceholder.typicode.com/>
 - Provides pre-defined endpoints like /posts, /users, /comments
- Example:
 - GET all posts → <https://jsonplaceholder.typicode.com/posts>
 - GET single post → <https://jsonplaceholder.typicode.com/posts/1>
 - POST new post → <https://jsonplaceholder.typicode.com/posts>



Create Dummy API's

- Using JSON Server (Local Dummy API)
 - Install JSON Server globally: `$ npm install -g json-server@0.17.4`
- Create a file db.json:

```
{  
  "users": [ { "id": 1, "name": "John", "email": "john@example.com" } ]  
}
```
- Run server:
`$ json-server --watch db.json --port 3000`
- Test endpoints:
 - GET all users → `http://localhost:3000/users`
 - GET single user → `http://localhost:3000/users/1`
 - POST new user → `http://localhost:3000/users`



Basic of JSON and JSON Path

- JSON (JavaScript Object Notation) is a lightweight data interchange format used to store and exchange data between client and server.
- Human-readable and easy to parse
- Language-independent (works with Java, Python, JS, etc.)
- JSON Structure:
 - Object → { "key": "value" }
 - Array → [item1, item2]



Basic of JSON and JSON Path

- Example:

```
{  
  "id": 1,  
  "name": "John Smith",  
  "email": "john@example.com",  
  "roles": ["Admin", "User"],  
  "address": {  
    "city": "NewYork",  
    "country": "United States"  
  }  
}
```



Basic of JSON and JSON Path

- JSONPath is a query language for JSON, similar to XPath for XML.
 - It allows you to extract specific data from JSON responses.
 - Very useful in API testing and automation.
- JSONPath Syntax

Symbol	Meaning	Example
\$	Root object	\$ → Whole JSON
.	Child operator	\$.name → "John Smith"
[]	Array index	\$.roles[0] → "Admin"
*	Wildcard (all elements)	\$.roles[*] → ["Admin", "User"]
..	Recursive search	\$..city → "NewYork"
[?()]	Filter expression	\$.roles[?(@ == 'Admin')] → "Admin"



Validating Responses in Postman and JavaScripting

- Response validation ensures that an API returns the expected data, status codes, and structure.
 - Postman uses JavaScript in the “Tests” tab to write validation scripts.
 - Helps automate API testing instead of manually checking responses.
- Step 1: Send a Request
 - Open Postman
 - Create a GET or POST request
 - Click Send to get the response
 - Example GET request: GET <https://jsonplaceholder.typicode.com/users/1>



Validating Responses in Postman and JavaScripting

- Step 2: Open Tests Tab
 - Click Tests tab in Postman request
 - This is where you write JavaScript code to validate response
- Step 3: Validate Status Code

```
pm.test("Status code is 200", function () {  
  pm.response.to.have.status(200);  
});
```

 - Checks if the API returns 200 OK



Validating Responses in Postman and JavaScripting

- Step 4: Validate Response Body

- Check if JSON key exists

```
pm.test("Response has name field", function () {  
    var jsonData = pm.response.json();  
    pm.expect(jsonData).to.have.property("name");  
});
```

- Check key value

```
pm.test("Name is John Smith", function () {  
    var jsonData = pm.response.json();  
    pm.expect(jsonData.name).to.eql("John Smith");  
});
```



Validating Responses in Postman and JavaScripting

- Step 5: Validate JSON Array
 - For endpoints returning an array:

```
pm.test("Response has at least one user", function () {  
  var jsonData = pm.response.json();  
  pm.expect(jsonData.length).to.be.above(0);  
});
```



Validating Responses in Postman and JavaScripting

- Step 6: Validate Headers

```
pm.test("Content-Type is application/json", function () {  
  pm.response.to.have.header("Content-Type", "application/json; charset=utf-8");  
});
```

- Step 7: Validate Response Time

```
pm.test("Response time is less than 500ms", function () {  
  pm.expect(pm.response.responseTime).to.be.below(500);  
});
```

- Step 8: Run Tests

- Click Send → Postman runs tests automatically
- Check Test Results tab → Shows which tests passed or failed



Postman Scripts and Variables

- Postman scripts are JavaScript code snippets that you can run before or after a request to perform checks, setup, or dynamic operations.
- Types of Scripts
 1. Pre-request Script
 - Runs before sending the request
 - Used for:
 - Generating dynamic values (timestamp, UUID, token)
 - Setting environment or collection variables
 - Encoding/decoding data
 - Example:

```
// Generate a random number and set as variable
let randomNum = Math.floor(Math.random() * 1000);
pm.environment.set("randomId", randomNum);
```



Postman Scripts and Variables

- Types of Scripts

- 2. Test Script

- Runs after receiving the response
 - Used for:
 - Validating response (status code, body, headers)
 - Storing response data in variables
 - Chaining requests

- Example:

```
pm.test("Status code is 200", function () {  
    pm.response.to.have.status(200);  
});  
// Store userId from response  
let jsonData = pm.response.json();  
pm.environment.set("userId", jsonData.id);
```



Postman Scripts and Variables

- Postman Variables
 - Variables store dynamic values that can be reused across requests, environments, or collections.
- Using Variables
 - Syntax: `{{variableName}}`
 - Example:
 - URL: <https://api.example.com/users/{{userId}}>



Postman Scripts and Variables

- Setting Variables via Script

```
// Set environment variable
```

```
pm.environment.set("variableName", "abc123");
```

```
// Get variable value
```

```
let value = pm.environment.get("variableName");
```

```
// Unset variable
```

```
pm.environment.unset("variableName");
```



Chaining Requests

- Chaining Requests means using the output of one API request as input for another in Postman.
 - Useful for workflows like get all users → get user details.
 - Achieved using variables and pre-request/test scripts.



Chaining Requests

- Steps to Chain Requests in Postman
- Step 1: Send First Request
- Example: Get All Users
 - GET <https://jsonplaceholder.typicode.com/users>
 - Response:

```
[  
  {  
    "id":1,  
  }  
]
```



Chaining Requests

- Step 2: Store Response Data in Variables

- Go to Tests tab of first request
- Store userId:

```
var jsonData = pm.response.json();  
pm.environment.set("userId", jsonData[5].id);
```

- Step 3: Use Variables in Second Request

- Example: Get User Details API

GET <https://jsonplaceholder.typicode.com/users/{{userId}}>

- No need to manually copy values

Data Driven Testing Using JSON and CSV Files



- Data Driven Testing (DDT) is Running the same test case multiple times with different sets of input data.
- It Helps validate API behavior under multiple inputs without rewriting tests.
- In Postman, using Collection Runner with external data files (.json or .csv).



Hands-On Lab: API Testing using Postman

- You are asked to test a dummy REST API for a Student Management System using json-server as a backend and Postman for API testing.
- The API should support basic CRUD operations on students.
- Each student record should contain:
 - id (number, auto-increment)
 - name (string)
 - email (string)
 - course (string)
- You will:
 - Set up a dummy API using json-server
 - Manually test endpoints in Postman
 - Write test scripts in Postman to validate responses
 - Use DDT to create 5 students



File Upload and Download API

- File Upload API
 - An API that allows sending files (images, PDFs, docs, etc.) from client → server.
 - Usually uses HTTP POST with multipart/form-data content type.
- File Download API
 - An API that allows retrieving files from the server → client.
 - Usually uses HTTP GET with a Content-Disposition header to force download.



Authorization Types

- No Auth
 - No authentication/authorization required.
 - Anyone can access the API endpoint.
 - Example: Public APIs (e.g., fetching weather info).
- API Key
 - A unique key (string) provided to the client.
 - Sent via Header or Query Parameter.
 - Example:
 - GET /data?apiKey=12345
 - or
 - Header: x-api-key: 12345
 - Use case: Simple authentication for public APIs with rate limits.



Authorization Types

- Basic Auth
 - Uses username:password (Base64 encoded) in the request header.
 - Example Header:
Authorization: Basic YWRtaW46cGFzc3dvcmQ=
 - Easy to implement but less secure if not using HTTPS.
 - Use case: Internal services, quick authentication.
- Bearer Token (Token-based Auth)
 - Client sends a token (like JWT) in the header.
 - Example:
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...
 - Token usually has an expiry time.
 - Use case: REST APIs, OAuth2 implementations.



cURL

- cURL = Client URL
- A command-line tool used to send HTTP requests and interact with APIs.
- Supports many protocols (HTTP, HTTPS, FTP, etc.).
- Installed by default on most Linux/macOS systems. On Windows, it comes with newer versions or can be installed separately.

```
curl https://jsonplaceholder.typicode.com/posts/1
```




Validating XML Response

- Many APIs (especially older SOAP or enterprise APIs) still return XML responses instead of JSON.
- Postman supports both, and you can validate XML using built-in XML parser inside Tests tab.
- XPath-like Validation
 - You can directly traverse the converted JSON structure (since XML is parsed into objects).



Converting XML Resposne to JSON

- You can use `xml2Json()` function to convert XML → JSON object.

```
// Convert XML response into JSON
```

```
let responseJson = xml2Json(pm.response.text());
```

```
// Validate root element
```

```
pm.test("Root element is note", function () {
```

```
    pm.expect(responseJson).to.have.property("note");
```

```
});
```



Documentation and Publishing

- Documentation is a detailed description of your APIs—how they work, endpoints, parameters, request/response examples.
- Postman allows you to auto-generate documentation from your collections and share them with teams or public.
- Postman generates a URL for your docs:
`https://documenter.getpostman.com/view/123456/collection/abcde`
- Anyone with the link can read your API documentation without Postman account.



Module 4: Web Services Testing with SOAP UI and Ready API Tools

Module 4: Web Services Testing with SOAP UI and Ready API Tools



- Introduction to SOAP UI
- Installation of SOAP UI
- SOAP UI Features
- Assertions in SOAP UI
- SOAP UI Properties
- Property Transfer
- Data-Driven Testing
- Data Driven Testing using Multiple Data Sources in SOAP UI Pro (Ready API)



Introduction to SOAP UI

- SOAP UI is a popular API testing tool used for testing web services, both SOAP and REST APIs.
- Developed by SmartBear.
- Allows functional, security, and load testing of APIs.
- Provides both free (open-source) and Pro (paid) versions.
- Ideal for manual and automated API testing, especially in enterprise environments.



Introduction to SOAP UI

- Key Features of SOAP UI

Feature	Description
SOAP Support	Test SOAP web services with WSDL.
REST Support	Test REST APIs with JSON, XML, or other payloads.
Assertions	Validate responses using built-in assertions (contains, XPath, JSONPath, schema, etc.).
Data-Driven Testing	Run tests with multiple data sets from Excel, CSV, or databases.
Mock Services	Simulate APIs to test client applications before real API is ready.
Load Testing	Run performance/load tests on APIs.



Installation of SOAP UI

- Go to the official website: <https://www.soapui.org/downloads/soapui.html>
- Choose the version:
 - SOAP UI Open Source (Free)
 - SOAP UI Pro / ReadyAPI (Paid, with trial)
- Download the installer based on your OS.
- Run the downloaded .exe installer.
- Follow the installation wizard
 - Click Install → Wait for installation to finish.
- Launch SOAP UI from Start Menu.



SOAP UI Features

- SOAP UI is a powerful API testing tool with features for both SOAP and REST services.
- SOAP Web Service Testing
 - Fully supports SOAP APIs.
 - Automatically reads WSDL files.
 - Generates requests and sample responses.
 - Allows SOAP header, body, and security testing.



SOAP UI Features

- REST API Testing
 - Supports REST APIs with JSON, XML, or plain text.
 - Allows testing of GET, POST, PUT, DELETE requests.
 - Supports query/path parameters and custom headers.
 - Easy integration with data-driven testing.
- Assertions
 - SOAP UI provides built-in assertions to validate responses:
 - Contains / Not Contains
 - XPath Validation
 - JSONPath Validation
 - Response Assertions
 - Schema Validation (XML/JSON)



SOAP UI Features

- Data-Driven Testing
 - Supports running tests with multiple input datasets.
 - Can read data from:
 - Excel files
 - Databases
 - Helps in automated regression testing.
- User-Friendly Interface
 - Drag-and-drop GUI for creating projects, test suites, and test cases.
 - Organize APIs into folders, suites, and steps.
 - Visualize requests, responses, and logs easily.



Assertations in SOAP UI

- Assertions are used to validate API responses automatically.
- They check whether the response meets certain conditions, such as content, or status code.
- Assertions can be applied to SOAP and REST requests.
- How to Add Assertions in SOAP UI
 - Open a request in SOAP UI.
 - Click Assertions tab at the bottom of the request window.
 - Click Add Assertion → Choose Type (e.g., Contains, JSONPath Match).
 - Configure the assertion parameters:
 - String to check, XPath/JSONPath, expected value, time, etc.
 - Click OK → Run request → Assertion will pass or fail automatically.



SOAP UI Properties

- Properties in SOAP UI are variables that store values which can be reused across requests, test cases, test suites, or projects.
- They make tests dynamic, maintainable, and data-driven.
- Properties can store URLs, IDs, tokens, or any test data.
- Add Properties
 - Right-click on Project → Show Project View → Properties Tab
 - Click + (Add Property) → Give name & value
 - Example: baseURL = <http://localhost:3000>
- Use property expansion to reference property values:
`${#Project#baseURL}`



Property Transfer

- Property Transfer is a feature in SOAP UI used to pass values dynamically from one test step to another.
- Useful for chaining requests, data-driven testing, and dynamic workflows.
- Values can be transferred between:
 - Responses → Properties
 - Properties → Request Fields
- Use Cases
 - Extract ID from a GET request and use it in a PUT or DELETE request.



Data-Driven Testing

- Data-Driven Testing (DDT) is a testing approach where the same test case is executed multiple times with different input data.
- SOAP UI allows REST/SOAP requests to read test data from:
 - Excel / CSV files
 - Database
 - Properties (XML/JSON)
- Reusability and automated testing for multiple scenarios.



Data Driven Testing using Multiple Data Sources in SOAP UI Pro (Ready API)

- Allows executing a single test case with multiple data sets from different sources.
- Example:
 - DataSource1: CSV with product details
 - DataSource2: Excel with discount info
- You can combine them dynamically and feed a single request or multiple requests.



Hands-On Lab: API Testing using Soap UI

- You are tasked with **testing a local JSON-based dummy API** created using json-server. The goal is to:
 - Retrieve a list of all users.
 - Extract the **ID of a specific user** (e.g., 3rd user) using **Property Transfer**.
 - Retrieve the user details using the extracted user ID.
 - Validate the **response using Assertions** (status code, JSON element values).
 - Update the user's email.
 - Delete the user using id.
- **API Endpoints (json-server):**
 - Get all users: `http://localhost:3000/users`
 - Get single user: `http://localhost:3000/users/{userId}`
 - Update user: PUT `http://localhost:3000/users/{userId}`
 - Delete user: DELETE `http://localhost:3000/users/{userId}`



Module 5: API Testing using Rest Assured



Module 5: API Testing using Rest Assured

- Environment Setup and Http Methods
- Create Request Payload(Request Body)
- Types of Parameters, Headers , Cookies and Logging
- Parsing JSON Responses
- Parsing XML Response
- File Upload and Download API
- JSON & XML Schema Validations
- Serialization and DE-Serialization
- Authentication Types
- API Chaining



What is Rest Assured

- Rest Assured is a Java library for testing REST APIs.
- It simplifies making HTTP requests, validating responses, and chaining calls.
- Works with JUnit / TestNG for automation.

- Add Rest Assured Dependency

```
<dependency>  
  <groupId>io.rest-assured</groupId>  
  <artifactId>rest-assured</artifactId>  
  <version>5.5.6</version>  
</dependency>
```



Environment Setup and Http Methods

- Environment Setup
 - Install JDK, IDE, Maven/Gradle
 - Add rest-assured, slf4j-simple, jackson-databind and testng dependencies
- HTTP Methods
 - GET → Fetch resource
 - POST → Create new resource
 - PUT → Replace resource
 - PATCH → Partial update
 - DELETE → Remove resource



Create Request PayLoad(Request Body)

- In Rest Assured, you can build request payloads using Java HashMap

```
Map<String, Object> payload = new HashMap<>();
```

```
    payload.put("name", "Sam Smith");
```

```
    payload.put("email", "sam@example.com");
```



Types of Parameters, Headers , Cookies and Logging

- Types of Parameters in Rest Assured
 - Query Parameters (?key=value)
 - Path Parameters (/users/{id})
 - Form Parameters (for x-www-form-urlencoded)
- Headers in Rest Assured
 - Used to pass metadata like content type, authorization, etc.
- Cookies in Rest Assured
 - Send Cookies: `.cookie("sessionId", "12345")`
 - Get Cookies from Response: `String value = response.getCookie("sessionId");`



Types of Parameters, Headers , Cookies and Logging

- Logging in Rest Assured
 - Logging helps debug requests & responses.
- Log All
 - `given().log().all().get("/users");`
- Log Only Request
 - `given().log().uri().log().headers().get("/users");`
- Log Response
 - `given().get("/users").then().log().all();`



Parsing JSON Responses

- Basic JSON Parsing

- You can extract values from JSON responses using `.path()` → Returns the value of a JSON path
- Example:

```
RestAssured.baseURI = "https://jsonplaceholder.typicode.com";  
Response response = given().get("/users/1");  
// Get as String  
String resString = response.asString();  
System.out.println("Response: " + resString);
```



Parsing XML Response

- XML parsing is slightly different from JSON but Rest Assured makes it easy using XmlPath or XPath expressions.
- Example:

```
RestAssured.baseURI = "https://www.w3schools.com/xml";  
Response response = given().get("/note.xml");  
System.out.println("Response:\n" + response.asString());  
XmlPath xmlPath = new XmlPath(response.asString());  
String to = xmlPath.getString("note.to");
```



File Upload and Download API

- File Upload and Download using Rest Assured
- Upload
 - Use `multiPart("fieldName", File)`
 - API returns JSON with uploaded file info
- Download
 - Send GET to the file URL
 - Save response as `InputStream` → write to local file



JSON & XML Schema Validations

- JSON Schema Validation
 - JSON Schema defines the structure, data types, and constraints of a JSON object.
 - It ensures that the API response conforms to the expected format, not just the values.
- XML Schema Validation
 - XML Schema (XSD) defines the structure, elements, attributes, and data types of XML documents.
 - It ensures that the API XML response follows the expected structure and type rules.



Serialization and DE-Serialization

- Serialization is the process of converting a Java object into a format that can be sent over the network or stored, typically JSON or XML.
 - Purpose: Send Java objects in HTTP request body or save them to a file.
 - Common formats: JSON, XML, Binary
- Deserialization is the process of converting JSON or XML response into a Java object.
 - Purpose: Read API response and work with it in Java.
 - Common formats: JSON → POJO, XML → POJO



Authentication Types

- Authentication in rest assured

Auth Type	How to Pass in Rest Assured	Example API
Basic Auth	<code>.auth().preemptive().basic(username, password)</code>	<code>postman-echo.com/basic-auth</code>
Bearer Token	<code>.header("Authorization", "Bearer " + token)</code>	<code>dummyjson.com/auth/login</code>



API Chaining

- API Chaining is the process of using data from the response of one API request as input for another API request.
- In real-world API chaining scenario:
 - Step 1: Get a list → extract dynamic data
 - Step 2: Use extracted data to call dependent API → validate response



Extra: Common API Mistakes to be Tested



Common API Mistakes to be Tested

- Input Validation Mistakes
 - **Missing validation:** API accepts invalid/malformed inputs (e.g., negative age, wrong email format).
 - **Overly strict validation:** Rejects valid values (e.g., international phone numbers).
 - **Improper error handling:** Returns 500 Internal Server Error instead of 400 Bad Request.
- Authentication & Authorization Flaws
 - No authentication on sensitive endpoints.
 - Token not expired properly (still usable after expiry).
 - Basic Auth credentials hardcoded in code.



Common API Mistakes to be Tested

- Improper Use of HTTP Status Codes
 - Returning 200 OK even for errors.
 - Using 500 for client errors.
 - Missing 201 Created for successful POST requests.
- Inconsistent Data Structures
 - Missing required fields in response.
 - Data type mismatches (id as string in one API, integer in another).
 - Field names inconsistent (userId vs id).



Common API Mistakes to be Tested

- Performance & Scalability Issues
 - API too slow due to unoptimized queries.
 - No pagination on large dataset endpoints → returning thousands of records.
 - Memory leaks with large file uploads.
- File Upload/Download Issues
 - Upload endpoint accepts all file types (potential security issue).
 - Large file upload not restricted.
 - Corrupt file download due to wrong encoding.

Hands-On Lab: API Testing using Rest Assured



- You are tasked with testing a dummy **product management API** (hosted via json-server) to verify basic CRUD operations using Rest Assured.
- Test basic CRUD functionality - GET / POST / PUT / DELETE
- Assertions - Verify status codes, response body
- Extract the ID of a specific product (e.g., 4th product) and Retrieve the product details using the extracted product ID.



Happy Learning :)