Basics of Web Development and Internet

Internet: Internet is a global networks of computers connected.

Web: World Wide Web is a way of accessing information over the medium of the internet.

What is Client and Server?

Client:

- ➔ A device or application that requests services or resources from a server.
- ➔ A Clients is typically a web browser that users interact with to access web pages.
- ➔ A Client can also be other types of software like an email client or a mobile app.

Characteristics of a client:

- ➔ User interface
- ➔ Requests services
- ➔ Receives Data

What is a Server?

- ➔ A device or application that provides services or resources to clients.
- ➔ A server is designed to handle requests from multiple clients

➔ A server hosts websites and respond to requests.

Characteristics of a server:

➔ Always on.
➔ Handles multiple requests
➔ Sends data

How do the Interact?

➔ Users will interact with devices such as mobile phone or computer and using these devices it will send a request to the server.
➔ Server will receive the requests it will process it and sends them a response back to the client.
➔ This response could be a requested webpage, the result of a query or simply some confirmation of the receipt.

Examples:

➔ Web Browsing
➔ Email

What are APIs?

- ➔ API stands for application programming interface. It is a set of rules or protocols that allows one software application to interact with another.
- ➔ Basically it defines the format or a way in which two software applications can communicate with each other.

Restaurant:

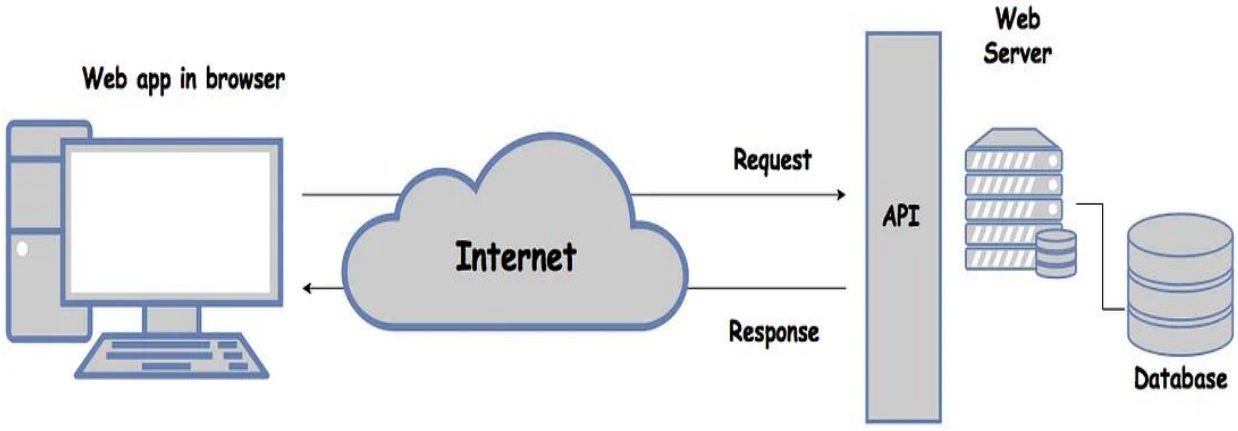
Customer is application

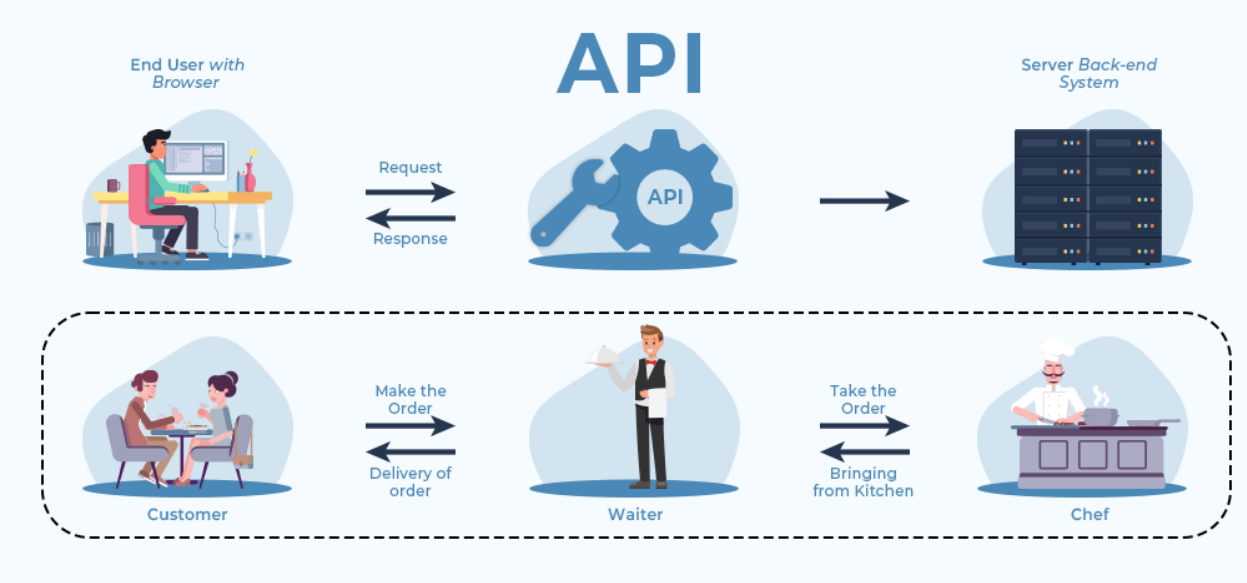
Kitchen is another system service

Menu is API specification

Waiter is API

Food is Response





APIs can be

- ➔ Private
- ➔ Partner
- ➔ Public

The Need of APIs

- ➔ Reduces Manual Effort
- ➔ Automates Everything

Types of API Request

1. Get Request
2. Post Request
3. Put Request
4. Delete Request

1. Get Request

➔ Retrieve of Get resources from server

➔ Used only to read data

➔ 200 ok (successful)

➔ 404 not found.

2. Post Request

➔ Create resources from server

➔ 201 created

➔ 404 not found

➔ 400 bad request

3. Put Request

➔ Update existing resources on Server

➔ 200 ok

➔ 404 not found

➔ 400 bad request

4. Delete Request

➔ It is used to delete resources from server

What is REST API?

- REST API, stands for Representation State Transfer Application Programming Interface. It follows the design principle of REST architectural style.
- Rest is Stateless, meaning each request from a client to the server must contain all the information that server needs to understand the request.

Principles of REST API

- ➔ **Client-Server Architecture**
- ➔ **Stateless**
- ➔ **Can be cached**
- ➔ **Opaque (not transparent) in terms of layers.**
- ➔ **Uniform interface**

Web Services built following the REST architectural style are known as RESTful Web services.

Common Methods:

- ➔ GET
- ➔ POST
- ➔ PUT
- ➔ DELETE

Benefits:

- ➔ Simplicity
- ➔ Scalability
- ➔ Flexibility
- ➔ Portability
- ➔ Visibility

HTTP vs HTTPS

- **HTTP:** HyperText transfer protocol.
- **HTTPs:** HyperText transfer protocol secure.

➔ Both of these are protocol that are used for transmitting the information over the internet.

➔ They operate based on a client-server model, where a client (web browser) sends a request to the server hosting a website.

➔ Both protocols use similar method to perform actions on the web server as well as status code.

➔ HTTP and HTTPs are both stateless protocols, meaning they do not inherently remember anything about the previous web session.

➔ Both HTTP and HTTPs can transfer data in various formats including HTML, XML, JSON and plain text.

Status Codes in APIs

Http Status Codes are three-digit code that indicate the output of an API request.

Classification of Status Codes

- ➔ 1xx (Informational)
- ➔ 2xx (Successful)
- ➔ 3xx (Redirection)
- ➔ 4xx (Client Error)
- ➔ 5xx (Server Error)

Commonly Used Status Codes

- ➔ 200 ok
- ➔ 201 created
- ➔ 204 no content
- ➔ 301 moved permanently
- ➔ 400 bad request
- ➔ 401 unauthorized
- ➔ 403 forbidden
- ➔ 404 not found
- ➔ 500 internal server error

Resource

- ➔ A resource is any piece of information that can be named or identified on the web.
- ➔ Can represent any type of object, data or service that can be accessed by clients.
- ➔ A resource is not just limited to documents files; it can be anything from a text file, an image, a collection of other resource, a non-virtual object like a person, and even abstract concepts like a service.
- ➔ In a social media application, resources could include a user profile, a photo, a list of friends or even a specific post or comment.

URI (Uniform Resource Identifier)

- ➔ A URI is a string of characters used to identify a resource on the internet either by location, name or both.
- ➔ It provides a mechanism for accessing the representation of a resource over the network, typically through specific protocols such as http or https.
- ➔ URIs are broad category that includes both URLs (Uniform Resource Locators) and URNs (Uniform Resource Names)

Sub – Resources

- ➔ A Sub-Resource is a resource that is hierarchically under another resource.
- ➔ It's a part of a larger resource and can be accessed by extending the URI of the parent resource.
- ➔ Sub-Resources are often used in RESTful API's to maintain a logical hierarchy of data and to facilitate easy access to related resources.

Example:

- ➔ In a blogging platform, you might have a users resource identified by a URI(/users). A specific user could access a resource accessible at /users/{userId}.
- ➔ If each user can have blog posts, a post would be a sub-resource of that user, identified by something like /users/{userId}/posts/{postId}.

Importance in Web Development

- ➔ Organization
- ➔ Accessibility
- ➔ Scalability

Why Web Frameworks?

➔ Websites ha a lot in common.

➔ Security, Database, URLs, Authentication, Testing and Debugging.

- **Thinks of Building House:**

➔ You would need Blueprint and Tools.

➔ That's how web development works.

➔ Developer's had to build from scratch.

- **What if?**

➔ You could have prefabricated components?

➔ Could you assemble faster?

➔ Would you reduce errors?

For solving these issues web frameworks comes into picture.

What is Web Framework?

➔ Web Framework is nothing but collection of tools and modules that is needed to do standard tasks across every web application.

Popular Web Framework?

➔ Spring Boot (Java)
➔ Django (Python)
➔ Flask (Python)
➔ Express (JavaScript)
➔ Ruby on Rails (Ruby)

Introduction to Spring Boot

➔ Initially developed by Rod Johnson in 2002.
➔ First version released in March 2004.
➔ Since then, major developments and versions released.

- Spring simplifies enterprise application development.

Key Principles:

➔ **Simplicity:** Simplify complex technology.
➔ **Modularity:** Encourage modular application architecture through loose coupling.
➔ **Testability:** Promote good programming practice. Such as programming with interfaces, dependency injection.

Key Components of Spring

➔ Core spring framework (Dependency injection, aspect oriented programming, transaction management and so on.)

- ➔ Spring Web
- ➔ Spring Data
- ➔ Spring Security
- ➔ Spring Cloud

Use Cases

- ➔ Enterprise Application
- ➔ Microservices Architecture.
- ➔ Web Applications.

Tight Coupling and Loose Coupling

Coupling refers to how closely connected different components or systems are.

Tight Coupling:

- ➔ Tight coupling describes a scenario where software components are highly dependent on each other.

Loose Coupling:

➔ Loose coupling describes a scenario where software components are less dependent on each other.

Importance in Software Design

➔ Flexibility and Maintainability.
➔ Scalability
➔ Testing

Achieving Loose Coupling

➔ Interfaces and Abstraction
➔ Dependency Injection
➔ Event Driven Architecture

Hands on:

Create classes in this package: com.tight.coupling

➔ UserDatabase.java
➔ UserManager.java
➔ TightCouplingExample.java

```
public class UserDatabase {
    public String getUserDetails() {
        return "User Details from Database.";
    }
}

public class UserManager {
    private UserDatabase userDatabase = new UserDatabase();
    public String getUserInfo() {
        return userDatabase.getUserDetails();
    }
}


public class TightCouplingExample {
    public static void main(String[] args) {
        UserManager userManager = new UserManager();
        System.out.println(userManager.getUserInfo());
    }
}
```

Loose Coupling:

Hands On: Create classes in package: com.loose.coupling

- ➔ LooseCouplingExample
- ➔ NewDatabase
- ➔ UserDatabase
- ➔ WebServiceDatabase
- ➔ UserManager

Create an interface named:

- ➔ UserDataProvider

```
public interface UserDataProvider {
   String getUserDetails();
}

public class NewDatabase implements UserDataProvider {
   @Override
   public String getUserDetails() {
      return "getting details from new database.";
   }
}
```

```
public class UserDatabase implements UserDataProvider{
    @Override
    public String getUserDetails() {
        return "User Details from Database";
    }
}


public class WebServiceDatabase implements
UserDataProvider{
    @Override
    public String getUserDetails() {
        return "Fetching Data From Web Service";
    }
}


public class UserManager {
    private UserDataProvider userDataProvider;

    public UserManager(UserDataProvider userDataProvider) {
        this.userDataProvider = userDataProvider;
    }
    public String getUserInfo() {
        return userDataProvider.getUserDetails();
    }
}
```

```
public class LooseCouplingExample {
   public static void main(String[] args) {
      UserDataProvider userDatabase = new UserDatabase();
      UserManager userManagerWithDatabase = new
UserManager(userDatabase);

System.out.println(userManagerWithDatabase.getUserInfo());

      UserDataProvider WebServiceDataProvider = new
WebServiceDatabase();
      UserManager webServiceUserManager = new
UserManager(WebServiceDataProvider);
      System.out.println(webServiceUserManager.getUserInfo());

      UserDataProvider newUserDatabase = new
NewDatabase();
      UserManager newUserManager = new
UserManager(newUserDatabase);
      System.out.println(newUserManager.getUserInfo());
   }
}
```

Core Concepts of Spring

Loose Coupling: Loose coupling is a design principle that aims to reduce the dependencies between components within a system.

Inversion of Control: Inversion of control is a design principle where the control of object creation and lifecycle management is transferred from the application code to an external container or framework.

Dependency Injection: Dependency injection is a design pattern commonly used in object-oriented programming where the dependencies of a class are provided externally rather than being created within the class itself.

Beans: Objects that are managed by framework are known as Beans.

Spring Container and Configuration

Types of Spring Containers:

- ➔ Application Context
- ➔ Bean Factory

Following configuration contains bean definition:

Search for, maven repository

In maven repository, search,

- ➔ Spring core
- ➔ Spring context

Add these two dependencies in pom.xml file

Like this:

```
<dependencies>
   <!--
https://mvnrepository.com/artifact/org.springframework/spring-core -->
   <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-core</artifactId>
      <version>6.2.2</version>
   </dependency>

   <!--
https://mvnrepository.com/artifact/org.springframework/spring-context -->
   <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>6.2.2</version>
   </dependency>
</dependencies>
```

What is pom.xml?

➔ Project Object Model, it is an XML file that contains information about the project and configuration details.

Creating First Bean:

Java ->

Create class My bean in package car.example.bean

Resource ->

Create a new xml file

ApplicationBeanContext.xml

Search for xml schema based configuration spring.

Add this snippet to ApplicationBeanContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="
      http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-
beans.xsd">

   <!-- bean definitions here -->

</beans>
```

```
package car.example.bean;

public class MyBean {
   private String message;
   public void setMessage(String message) {
      this.message = message;
   }
   public void showMessage() {
      System.out.println("Message: " + message);
   }

   @Override
   public String toString() {
      return "MyBean [message=" + message + "]";
   }
}
```

In applicationBeanContext.xml

Add this,

```
<!-- bean definitions here -->
<bean id="myBean" class="car.example.bean.MyBean">
    <property value="I am a first Bean"
name="message"></property>
</bean>
```

Create a class named App in car.example.bean:

```
package car.example.bean;

import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationC
ontext;

public class App {
   public static void main(String[] args) {
      ApplicationContext context = new
ClassPathXmlApplicationContext("applicationBeanContext.xml")
;
      MyBean myBean = (MyBean) context.getBean("myBean");
      System.out.println(myBean);
   }
}
```

Bean:

➔ Objects that are managed by frameworks are known as bean.

Bean definition:

➔ A bean definition includes configuration metadata that the container needs to know to create and manage the bean.

Bean Configuration:

➔ Bean definition can be provide in various ways, including XML configuration files, annotations, and java-based configuration.

➔ Beans are configured using XML files, where each bean is defined within <bean> tags with attributes specifying class, properties and dependencies.

➔ Beans can be configured using annotations like @Component, @Service, @Repository

Lifecycle of Beans:

➔ Instantiation

➔ Population of properties

➔ Initialization

➔ Ready for use

➔ Destruction

Dependency Injection

➔ Dependency Injection (DI) is a design pattern used in software development to achieve loose coupling between classes by removing the direct dependency instantiation from the dependent class itself.

Types:

➔ Constructor Injection
➔ Setter Injection

Constructor Injection

➔ Dependencies are provided to the dependent class through its constructor.
➔ Dependencies are passed as arguments to the constructor when the dependent class is instantiated.
➔ Constructor injection ensures that the dependencies are available when the object is created.

Create following classes in car.example.constructor.injection

➔ App
➔ Car
➔ Specification

```
package car.example.constructor.injection;

public class Specification {
    private String make;
    private String model;

    public String getMake() {
        return make;
    }
    public void setMake(String make) {
        this.make = make;
    }
    public String getModel() {
        return model;
    }
    public void setModel(String model) {
        this.model = model;
    }
    @Override
    public String toString() {
        return "Specification{" +
                "make='" + make + '\'' +
                ", model='" + model + '\'' +
                '}';
    }
}
```

```
package car.example.constructor.injection;

public class Car {
   private Specification specification;
   public void displayDetails() {
      System.out.println("Car specification: " +
specification.toString());
   }
   public Car (Specification specification) {
      this.specification = specification;
   }
}
```

then go to resources -> create a new xml file named applicationConstructorInjection.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="
      http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-
beans.xsd">
```

```
    <!-- bean definitions here -->
    <bean id="carSpecification"
    class="car.example.constructor.injection.Specification">
       <property name="make" value="Toyota"/>
       <property name="model" value="Corolla"/>
    </bean>
    <bean id="myCar"
    class="car.example.constructor.injection.Car">
       <constructor-arg ref="carSpecification"/>
    </bean>
</beans>
```

```
package car.example.constructor.injection;

import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationC
ontext;

public class App {
    public static void main(String[] args) {
        ApplicationContext context = new
ClassPathXmlApplicationContext("applicationConstructorInjecti
on.xml");
        Car myCar = (Car) context.getBean("myCar");
        myCar.displayDetails();
    }
}
```

Homework: setter injection

Introduction to Auto-wiring and its types

➔ Automatically resolves and inject dependencies between beans without requiring explicit definitions of wiring in XML or even in java configurations.

Types:

➔ Auto-wiring by Name
➔ Auto-wiring by Type
➔ Auto-wiring by constructor

Annotations

➔ Annotations in Java provide a way to add metadata to your code.

Commonly Used Spring Annotations

@Component

@Autowired

@Qualifier

@Value

@Repository

@Service

@Controller

@RequestMapping

@SpringBootApplication

Components and ComponentScan

➔ Component refers to a Java class that is managed by the Spring IoC container.

➔ Component is a special kind of bean that is designed to auto detect.

Defining Components in Spring

➔ Using XML.

➔ Using Annotations

Example:

- Using XML

```
<bean id=”myComponent”
class=”com.example.MyComponent”/>
```

- Using Annotations

```
Import org.springframework.stereotype.Component
@Component // Marks the class as a Spring component
Public class MyComponent {
// class Implementation }
```

Component Scanning is a feature helps to automatically detect and register beans from predefined package paths.

Using XML

<!—Enable component scanning →

<context:component-scan base-package=”car.example.componentScan”/>

Hands On:

Go to Resources directory and create a new file:

- → ComponentScanDemo.xml
- →

Google: component scan xml

Paste the template in ComponentScanDemo.xml file

Write : <context: component-scan base-package=”com.componentScan”/>

create two class in com.componentscan

- → Employee
- → App

package com.componentscan;

import org.springframework.beans.factory.annotation.Value;

```
import org.springframework.stereotype.Component;

@Component("employee")
public class Employee {
   private int employeeId;
   @Value("John")
   private String firstName;
   @Value("${java.home")
   private String lastName;
   @Value("#{4 * 4}")
   private double salary;

   public int getEmployeeId() {
      return employeeId;
   }

   public void setEmployeeId(int employeeId) {
      this.employeeId = employeeId;
   }

   public String getFirstName() {
      return firstName;
   }

   public void setFirstName(String firstName) {
      this.firstName = firstName;
   }
```

```
    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public double getSalary() {
        return salary;
    }

    public void setSalary(double salary) {
        this.salary = salary;
    }

    @Override
    public String toString() {
        return "Employee{" +
                "employeeId=" + employeeId +
                ", firstName='" + firstName + '\'' +
                ", lastName='" + lastName + '\'' +
                ", salary=" + salary +
                '}';
    }
}
```

```
package com.componentscan;

import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicatio
nContext;

public class App {
   public static void main(String[] args) {
      ApplicationContext context = new
ClassPathXmlApplicationContext("componentScanDemo.xml"
);

      Employee employee = (Employee)
context.getBean("employee");
      System.out.println(employee.toString());
   }
}
```