

UNIT IV

Web Services: Why Web services, SOA - Service Oriented Architecture, , Types of Web Services, SOAP based Web Services, RESTful Web Services, How to create RESTful Services

Web Services

1: Why Web Services:

Introduction to Web Services and Service-Oriented Architecture (SOA)

Web services and Service-Oriented Architecture (SOA) are fundamental concepts in modern software development, enabling interoperability, reusability, and flexibility in building distributed systems. Let's explore the key aspects of web services and SOA based on the provided search results.

2. What are Web Services:

Web services are software systems designed to support interoperable machine-to-machine interaction over a network. They provide a standardized way of integrating web-based applications using open standards such as XML, SOAP, WSDL, and UDDI. Web services can be accessed over standard internet protocols, making them independent of platforms and programming languages.

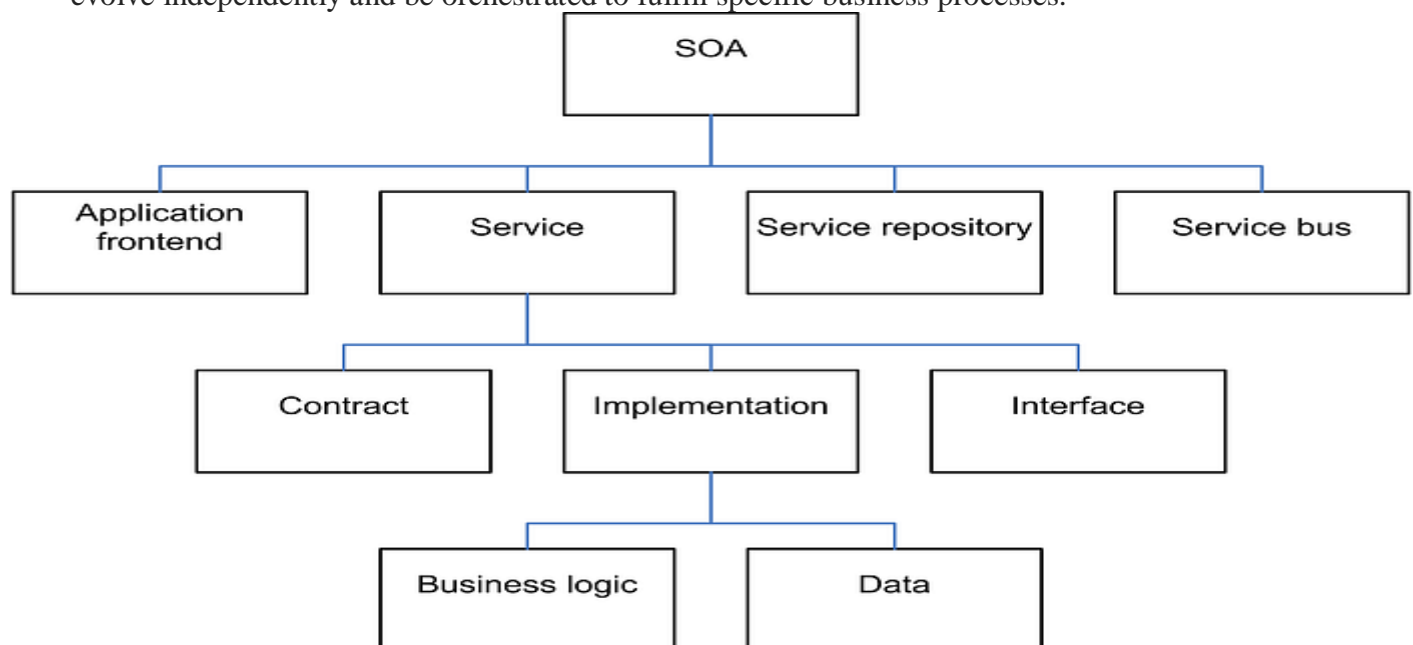
1. **Interoperability:** Web services facilitate seamless communication between different systems, regardless of their underlying technologies, by using standard internet protocols and data formats.
2. **Standardization:** They adhere to standardized protocols such as SOAP (Simple Object Access Protocol) and REST (Representational State Transfer), ensuring consistent communication and data exchange.

3. **Service Description:** Web services are described using WSDL (Web Services Description Language), which provides a machine-readable description of the service's capabilities and communication protocols.

3. Service-Oriented Architecture (SOA):

SOA is a software development approach that emphasizes the creation of reusable and interoperable software services. It enables the integration of disparate systems and applications by encapsulating business functions as services with well-defined interfaces. SOA promotes loose coupling, flexibility, and reusability of software components.

1. **Reusability and Interoperability:** SOA promotes the reuse of software components and facilitates interoperability between different systems and applications.
2. **Service Granularity:** Services in SOA are designed to be granular, representing specific business capabilities or functions, and can be composed to create complex applications.
3. **Decoupled Architecture:** SOA encourages loose coupling between services, allowing them to evolve independently and be orchestrated to fulfill specific business processes.



Benefits of SOA and Web Services:

- **Interoperability:** Web services and SOA enable seamless integration and communication between disparate systems and applications.
- **Reusability:** Services designed in an SOA can be reused across multiple applications, reducing redundancy and promoting efficiency.
- **Flexibility:** SOA allows for the flexible composition and orchestration of services to meet evolving business requirements.

→SOA is based on some key principles which are mentioned below

Standardized Service Contract — Services adhere to a service description. A service must have some sort of description which describes what the service is about. This makes it easier for client applications to understand what the service does.

Loose Coupling — Less dependency on each other. This is one of the main characteristics of web services which just states that there should be as less dependency as possible between the web services and the client invoking the web service. So if the service functionality changes at any point in time, it should not break the client application or stop it from working.

Service Abstraction — Services hide the logic they encapsulate from the outside world. The service should not expose how it executes its functionality; it should just tell the client application on what it does and not on how it does it.

Service Reusability — Logic is divided into services with the intent of maximizing reuse. In any development company re-usability is a big topic because obviously one wouldn't want to spend time

and effort building the same code again and again across multiple applications which require them. Hence, once the code for a web service is written it should have the ability work with various application types.

Service Autonomy — Services should have control over the logic they encapsulate. The service knows everything on what functionality it offers and hence should also have complete control over the code it contains.

Service Statelessness — Ideally, services should be stateless. This means that services should not withhold information from one state to the other. This would need to be done from either the client application. An example can be an order placed on a shopping site. Now you can have a web service which gives you the price of a particular item. But if the items are added to a shopping cart and the web page navigates to the page where you do the payment, the responsibility of the price of the item to be transferred to the payment page should not be done by the web service. Instead, it needs to be done by the web application.

Service Discoverability — Services can be discovered (usually in a service registry). We have already seen this in the concept of the UDDI, which performs a registry which can hold information about the web service.

Service Compos ability — Services break big problems into little problems. One should never embed all functionality of an application into one single service but instead, break the service down into modules each with a separate business functionality.

Service Interoperability — Services should use standards that allow diverse subscribers to use the service. In web services, standards as XML and communication over HTTP is used to ensure it conforms to this principle.

How work SOA:

To understand SOA, consider the architecture commonly found on college campuses: the data-oriented system. In a data-oriented system, campus departments share information by sending it in batch-file transfers from one department to another. For example, the library might need to check individuals who wish to borrow materials to find out if they are full-time students. To do so, the library would query the student records department, which, in the case of UW–Madison, would send all 45,000 student records in a batch. The library would then load the batch into its system's database, simply to locate the data it needs for a small subset of those students.

Research firm Adventure notes that university departments can easily misinterpret data sent by batch transfers because departments use different business rules or different versions of data to suit their unique needs. Recreational facilities, for example, might define a current student as one who is enrolled and has paid all college fees and is therefore authorized to use campus facilities. The library, on the other hand, might define a current student — one eligible to check out books — as one who is enrolled but has not necessarily paid all fees. Such differences can create obvious data-accuracy problems and minimize the usefulness of information.

The fact that the batch-file method includes all possibly needed data increases the security risk and places a large burden on already strained computing resources. Batch-file transfers inherent can be cumbersome, time-consuming processes prone to errors. As departments transfer batch files across campus, different “versions of truth” begin to circulate. In contrast, SOA is based on a service-level agreement between a service owner (source) and the service consumer. The transaction between the two is tracked at all points.

Pros and Cons of SOA:

Pros :

1. Service Reusability

In SOA, an application is built by assembling small, self-contained, and loosely coupled pieces of functionality. Therefore, the services can be reused in multiple applications independent of their interactions with other services.

2. Easy Maintainability

Since a service is an independent entity, it can be easily updated or maintained without having to worry about other services. Large, complex applications can thus be managed easily.

3. Greater Reliability

SOA-based applications are more reliable since small, independent services are easier to test and debug as compared to massive chunks of code.

4. Location Independence

The services are usually published to a directory where consumers can look them up. This approach allows a service to change its location at any time. However, the consumers are always able to locate their requested service through the directory look up.

5. Improved Scalability and Availability

Multiple instances of a single service can run on different servers at the same time. This increases scalability and availability of the service.

6. Improved Software Quality

Since services can be reused, there is no scope for redundant functionality. This helps reduce errors due to inconsistent data, and thereby improves the quality of code.

7. Platform Independence

SOA facilitates the development of a complex product by integrating different products from different vendors independent of the platform and technology.

8. Increased Productivity

Developers can reuse existing legacy applications and build additional functionality without having to develop the entire thing from scratch. This increases the developers' productivity, and at the same time, substantially reduces the cost of developing an application.

Cons:

Ø Increased Overhead

Every time a service interacts with another service, complete validation of every input parameter takes place. This increases the response time and machine load, and thereby reduces the overall performance.

Ø Complex Service Management

The service needs to ensure that messages have been delivered in a timely manner. But as services keep exchanging messages to perform tasks, the number of these messages can go into millions even for a single application. This poses a big challenge to manage such a huge population of services.

Ø High Investment Cost

Implementation of SOA requires a large upfront investment by means of technology, development, and human resource

Example of SOA:

To deliver services outside the firewall to new markets: First Citizens Bank not only provides services to its own customers, but also to about 20 other institutions, including check imaging, check processing, outsourced customer service, and “bank in a box” for getting community-sized bank everything they need to be up and running. Underneath these services is an SOA-enabled mainframe operation.

To provide real-time analysis of business events: Through real-time analysis, OfficeMax is able to order out-of-stock items from the point of sale, employ predictive monitoring of core business processes such as order fulfilment, and conduct real-time analysis of business transactions, to quickly measure and track product affinity, hot sellers, proactive inventory response, price error checks, and cross-channel analysis.

To streamline the business: Whitney National Bank in New Orleans built a winning SOA formula that helped the bank attain measurable results on a number of fronts, including cost savings, integration, and more impactful IT operations. Metrics and progress are tracked month to month — not a “fire-and-forget business case.”

To improve federal government operations: The US Government Accountability Office (GAO) issued guidelines intended to help government agencies achieve enterprise transformation through enterprise architecture. The guidelines and conclusions offer a strong business case for commercial businesses also seeking to achieve greater agility and market strength through shared IT services. As GAO explains it, effective use of an enterprise architecture achieves a wide range of benefits.

To improve state and local government operations: The money isn’t there to advance new initiatives, but state governments may have other tools at their disposal to drive new innovations — through shared IT service. Along these lines, a new study released by the National Association of State Chief Information Officers (NASCIO), TechAmerica and Grant Thornton, says well-managed and focused IT initiatives may help pick up the slack where spending is being cut back.

To improve healthcare delivery: If there’s any sector of the economy that desperately needs good information technology, that’s the healthcare sector — subject to a dizzying array of government mandates, fighting cost overruns at every corner, and trying to keep up with the latest developments in care and protocols.

4.Types of Web Services;

Types of Web Services are

1. SOAP-based Web services
2. RESTful Web services

Web services are essential for enabling communication and data exchange between different software systems over a network. Two primary types of web services are SOAP-based and RESTful services. Each has its own set of characteristics, advantages, and use cases.

5.SOAP-based Web Services:

SOAP (Simple Object Access Protocol) is a protocol for exchanging structured information in the implementation of web services. It relies on XML for message formatting and is based on a client-server model. SOAP web services are known for their robustness and standardization.

Simple Object Access Protocol, as the name says, is a standard protocol intended to exchange structured messages in a decentralized, distributed environment. SOAP is a protocol based on XML to exchange messages across the internet. It has a set of rules which define and describe the messaging format and processing rules for information exchanged between the sender and receiver. SOAP can use two different transfer protocols to exchange the messages: HTTP (Hypertext Transfer Protocol) and SMTP (Simple Mail Transport Protocol), but the most popular protocol used by SOAP services is HTTP. Below we have a SOAP

Request message as well as an example of the SOAP message structure:

```
OST /soap/event HTTP/1.1
```

```
Content-Length: 309
```

```
SOAPAction: "getEvent"
```

```
Content-Type: text/xml; charset=utf-8
```

```
Host: ems-sv258:1774
```

```
Connection: Keep-Alive<?xml version="1.0" ?>
```

```
<soapenv:Envelope
```

```
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
```

```
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
```

```
xmlns:ns1="http://soap.com/act">

<soapenv:Body>

  <ans:getNote xmlns:ans="http://soap.com/act">

    <eventID>1000</eventID>

  </ans:getNote>

</soapenv:Body>

</soapenv:Envelope>
```

Since SOAP standards for message exchange are based on XML and usually use HTTP as a transport protocol, it is possible to communicate between different applications with different operational systems.

One specific characteristic of SOAP is the coupling between the service provider (server) and the service consumer (client). There is a rigid contract between the server and client in SOAP and the consumer must know the exact message structure to send to the server. If anything changes on either side, client or server, it will break the communication.

Another important characteristic of SOAP is its association with Web Service Description Language (WSDL). The WSDL has the description of SOAP service and how it works. This makes the client building process easier once IDEs and frameworks can be utilized to automate the implementation based on WSDL.

SOAP also has built-in error handling. When any message has an error or is missing required information, the error response message contains information about what is wrong and can be used by the client to fix the problem. This is very important to consumers when they are not the owner of the service, otherwise, the client won't be able to understand what is wrong with the request.

How to work SOAP:

SOAP primarily uses the standard HTTP request/response model (see Figure A).

The client wraps a method call in SOAP/XML, which is then posted over HTTP to the server. The XML request is parsed to read the method name and parameters passed and delegated for processing. The XML response is then sent back to the client, containing the return value — or fault data — of the method call. Finally, the client may parse the response XML to make use of the return value.

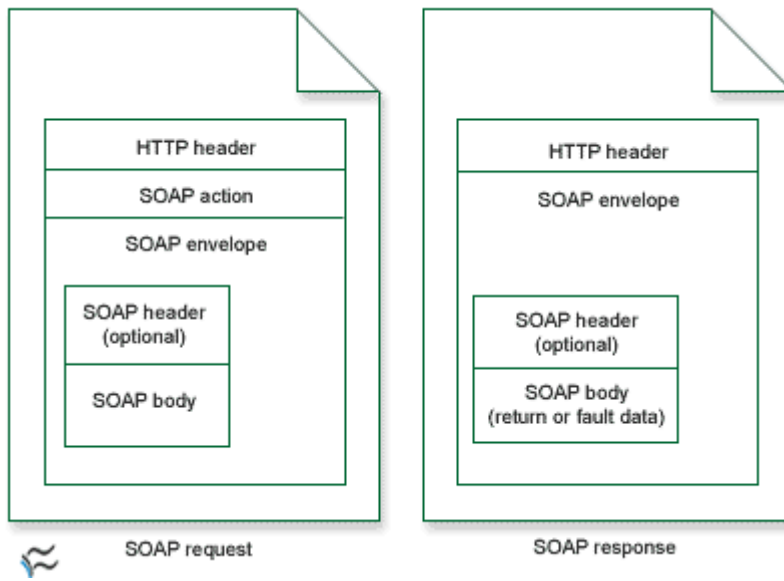


Figure -A

But HTTP is not the only transfer protocol that SOAP supports. In version 1.1 of SOAP, the specification was expanded to cover other transfer protocols such as Simple Mail Transfer Protocol (SMTP), File Transfer Protocol (FTP), or any other protocol that can transfer text. These other protocols could be used to make asynchronous SOAP calls, meaning the client application could invoke the SOAP service but would not have to wait for a response from the server.

Cons and Pros:

Pros:

- 1. WS Security:** SOAP defines its own security known as WS Security.
- 2. Language and Platform independent:** SOAP web services can be written in any programming language and executed in any platform

Cons

- 1. Slow:** SOAP uses XML format that must be parsed to be read. It defines many standards that must be followed while developing the SOAP applications. So it is slow and consumes more bandwidth and resource.
- 2. WSDL dependent:** SOAP uses WSDL and doesn't have any other mechanism to discover the service.

Example of SOAP:

In order to call a SOAP API, you'll most likely need to include a SOAP library with your programming language. Although it's possible to make SOAP API calls without SOAP libraries, it's more efficient to work with an abstraction rather than crafting the messages yourself. The SOAP messages are verbose, mainly due to reliance on XML.

While the following examples use Python for readability, remember that SOAP is agnostic regarding your programming language. To retrieve a user profile from a fictitious SOAP API, you might make the following request using the Zeep library:

```
from zeep import Client

client = Client('http://www.example.com/exampleapi')
result = client.service.GetUser(123) # request user with ID 123

name = result['Username']
```

In this example, we initiate a SOAP client based upon the SOAP endpoint. Then we call the service, invoking the `getuser` option with a user ID parameter. It's a simple example, but disguises even more detail of the SOAP messages behind the scenes.

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
  <soap:Header>
  </soap:Header>
  <soap:Body>
    <m:GetUser>
      <m:UserId>123</m:UserId>
    </m:GetUser>
  </soap:Body>
</soap:Envelope>
```

Let's look at how this SOAP call might be structured:

And the response might look something like this:

```
<?xml version="1.0"?>

<soap:Envelope
xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">

  <soap:Body>
    <m:GetUserResponse>
      <m:Username>Tony Stark</m:Username>
    </m:GetUserResponse>
  </soap:Body>

</soap:Envelope>
```

Even in this simple example, the actual data within the message is surrounded by the SOAP structure. Compared to some more modern API request examples, SOAP may appear overly complex. Keep in mind that most developers making SOAP API calls are using a library, which provides a friendlier interface.

That said, it is possible to make SOAP API calls through a typical HTTP request (most SOAP services use HTTP, though the specification is independent of protocol). Here is the same call above using the Python requests library:

```
import requests
req_headers = {"content-type": "text/xml"}
req_body = "<?xml version='1.0'?">"
req_body += "<soap:Envelope xmlns:soap='http://www.w3.org/2003/05/soap-envelope'"
req_body += "<soap:Header></soap:Header>"
req_body += "<soap:Body>"
req_body += "<m:GetUser>"
req_body += "<m:UserId>123</m:UserId>"
req_body += "</m:GetUser>"
req_body += "</soap:Body>"
req_body += "</soap:Envelope>"
response = requests.post(
    "http://www.example.com/exampleapi",
    data=req_body,
    headers=req_headers
)
```

In this case, response.content would include the raw XML response, which needs to be parsed in order to determine the username and any other data the SOAP API returns.

Advantages:

- **Standardization:** SOAP has a well-defined set of standards, making it suitable for enterprise-level applications that require robust security and transactional support.
- **Extensibility:** SOAP's extensibility allows for the addition of new features and standards without breaking existing implementations.

6.RESTful Web Services:

REST (Representational State Transfer) is an architectural style for designing networked applications. RESTful web services are known for their simplicity, scalability, and flexibility.

Key Characteristics:

1. Architectural Style:

- REST is not a protocol but an architectural style that uses standard HTTP methods (GET, POST, PUT, DELETE) for communication
- Resources in RESTful services are identified by URIs (Uniform Resource Identifiers) and can be manipulated using standard HTTP methods.

2. Message Formats:

- RESTful services can use multiple formats for data exchange, including JSON, XML, HTML, and plain text. JSON is the most commonly used format due to its lightweight nature and ease of use.

3. Service Creation with Spring Boot:

- Spring Boot, in combination with Spring Web MVC (also known as Spring REST), simplifies the development of RESTful web services. It provides tools and annotations to create RESTful endpoints efficiently
- RESTful services are often the first step in developing microservices, making them a crucial part of modern application architectures

REST is a web standard architecture that achieves data communication using a standard interface such as HTTP or other transfer protocols that use standard Uniform Resource Identifier (URI). The design is such that each component in a RESTful web service is a resource that can be accessed using standard HTTP methods (if the chosen protocol is HTTP). Resources which can be thought of as objects in the concept of Object oriented programming (OOP) are identified by URIs and the resources are represented in several ways such as JSON, XML, Text etc. though JSON is currently the more favoured choice.

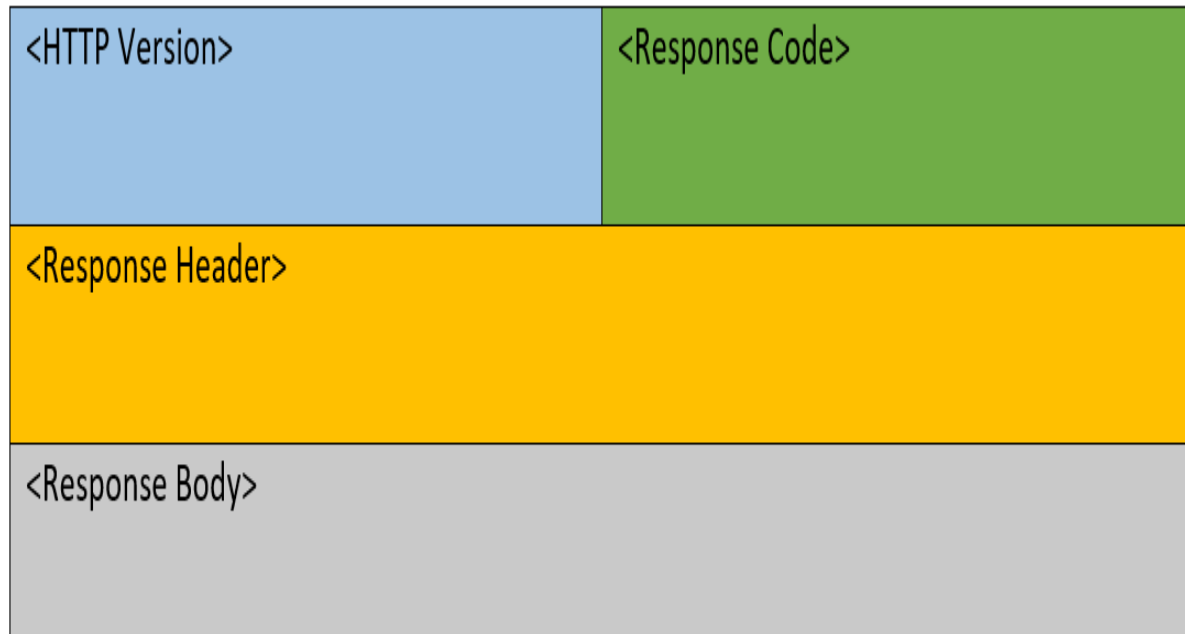
RESTful services have the following properties: Representations, Messages, URIs, Uniform interface, Stateless, Links between resources and caching. A quick look into these properties below using HTTP

1. Representation- resources are represented in different formats as earlier stated and should be a complete representation of the resource.
2. Messages- This is how the client and server interact. Along with the data, messages contain metadata about the message. When accessing a RESTful resource using HTTP, the commonly used methods are GET (reads a resource), PUT (creates a resource), DELETE (removes a resource) and POST (updates an existing resource)

The figures below show the request and response message formats

<VERB>: one of the HTTP methods	<URI> the resource URI to perform the operation	<HTTP Version>
<Request Header> contains information about the message such as format of the message body, format supported by client etc.		
<Request Body> holds the actual message content		

The Response Header and Body hold similar information to the request, only that the information is different as it is the server response



3. URIs – Each resource needs at least one URI to identify a resource(s) and the operation is determined by the HTTP verb/action. Hence a URI can be called with different actions.
4. Stateless- Restful web services are stateless and any session state is held on the client not server. This ensures that every client to server request has the necessary information to understand the request and handles each request independently.
5. Links Between Resources- the representation of a resource can have links to other resources.
6. Caching- the data produced when a request is made the first time is stored and used the next time in order to stop regenerating same information for the same request and improves performance. The HTTP headers help to control caching such as age which tracks how long ago the data was fetched from the server, expires date and time the resource representation expires etc.

The above gives an oversight of the six properties of REST and it is important to remember that:

- REST is not coupled to HTTP and is actually protocol independent. It is simply not a mapping of CRUD to the HTTP methods.
- REST makes it relatively easy to integrate with websites and are exposed using XML (one of many ways) for easy consumption

Advantages:

- **Simplicity:** RESTful services are simple to implement and use, making them ideal for web and mobile applications.

- **Scalability:** The stateless nature of RESTful services allows for easy scaling and load balancing.

Comparison of SOAP and RESTful Web Services

While both SOAP and RESTful web services enable communication between different systems, they have distinct differences:

- **Protocol vs. Architectural Style:** SOAP is a protocol with strict standards, while REST is an architectural style that leverages standard HTTP methods.
- **Message Format:** SOAP uses XML exclusively, whereas REST can use multiple formats, with JSON being the most popular.
- **Complexity:** SOAP is more complex and suitable for enterprise-level applications requiring robust security and transactional support. REST is simpler and more flexible, making it ideal for web and mobile applications.

7. Creating RESTful Services with Spring Boot

RESTful web services are a fundamental part of modern web applications and microservices architectures. They provide a standardized way to enable communication between different systems over HTTP. Spring Boot, combined with Spring Web MVC (also known as Spring REST), simplifies the development of RESTful web services. Let's explore the process of creating RESTful services with Spring Boot in detail.

Introduction to RESTful Services:

REST (Representational State Transfer) is an architectural style that uses standard HTTP methods to interact with resources identified by URIs (Uniform Resource Identifiers). RESTful services are stateless, scalable, and can use multiple data formats, with JSON being the most common.

Key Characteristics of RESTful Services:

1. **Stateless:** Each request from a client to a server must contain all the information needed to understand and process the request.
2. **Resource-Based:** Resources are identified by URIs, and interactions with these resources are performed using standard HTTP methods (GET, POST, PUT, DELETE).
3. **Representation:** Resources can have multiple representations, such as JSON, XML, or HTML.

Setting Up a Spring Boot Project

To create a RESTful service with Spring Boot, you need to set up a Spring Boot project. You can use Spring Initializr (<https://start.spring.io/>) to generate a project with the necessary dependencies.

Dependencies:

- Spring Web: For building web applications, including RESTful services.
- Spring Boot DevTools: For development and testing.
- Spring Data JPA: For database access (optional, if you need to interact with a database).
- H2 Database: An in-memory database for testing (optional).

REST stands for **REpresentational State Transfer**. It is developed by **Roy Thomas Fielding**, who also developed HTTP. The main goal of RESTful web services is to make web services **more effective**. RESTful web services try to define services using the different concepts that are already present in HTTP. REST is an **architectural approach**, not a protocol.

It does not define the standard message exchange format. We can build REST services with both XML and JSON. JSON is more popular format with REST. The **key abstraction** is a resource in REST. A resource can be anything. It can be accessed through a **Uniform Resource Identifier (URI)**. For example:

The resource has representations like XML, HTML, and JSON. The current state capture by representational resource. When we request a resource, we provide the representation of the resource. The important methods of HTTP are:

- **GET:** It reads a resource.
 - **PUT:** It updates an existing resource.
 - **POST:** It creates a new resource.
 - **DELETE:** It deletes the resource.
-

For example, if we want to perform the following actions in the social media application, we get the corresponding results.

POST /users: It creates a user.

GET /users/{id}: It retrieves the detail of a user.

GET /users: It retrieves the detail of all users.

DELETE /users: It deletes all users.

DELETE /users/{id}: It deletes a user.

GET /users/{id}/posts/post_id: It retrieve the detail of a specific post.

POST / users/{id}/ posts: It creates a post of the user.

Further, we will implement these URI in our project.

HTTP also defines the following standard status code:

- **404:** RESOURCE NOT FOUND
 - **200:** SUCCESS
 - **201:** CREATED
 - **401:** UNAUTHORIZED
 - **500:** SERVER ERROR
-

RESTful Service Constraints

- There must be a service producer and service consumer.
- The service is stateless.
- The service result must be cacheable.
- The interface is uniform and exposing resources.
- The service should assume a layered architecture.

Advantages of RESTful web services

- RESTful web services are **platform-independent**.
- It can be written in any programming language and can be executed on any platform.
- It provides different data format like **JSON, text, HTML, and XML**.
- It is fast in comparison to SOAP because there is no strict specification like SOAP.
- These are **reusable**.
- They are **language neutral**.

Building a Reactive RESTful Web Service:

This guide walks you through the process of creating a "Hello, Spring!" RESTful web service with Spring WebFlux (new as of Spring Boot 2.0) and then consumes that service with a WebClient (also new as of Spring Boot 2.0).

This guide shows the functional way of using Spring WebFlux. You can also [use annotations with WebFlux](#).

What You Will Build:

You will build a RESTful web service with Spring Webflux and a WebClient consumer of that service. You will be able to see output in both System.out and at:

```
Copyhttp://localhost:8080/hello
```

What You Will Need:

- About 15 minutes
- A favorite text editor or IDE
- [Java 17](#) or later
- [Gradle 7.5+](#) or [Maven 3.5+](#)
- You can also import the code straight into your IDE:
- [Spring Tool Suite \(STS\)](#)
- [IntelliJ IDEA](#)
- [VSCode](#)

How to complete this guide:

Like most Spring [Getting Started guides](#), you can start from scratch and complete each step or you can bypass basic setup steps that are already familiar to you. Either way, you end up with working code.

To **start from scratch**, move on to [Starting with Spring Initializr](#).

To **skip the basics**, do the following:

- [Download](#) and unzip the source repository for this guide, or clone it using [Git](#): `git clone https://github.com/spring-guides/gs-reactive-rest-service.git`
- `cd` into `gs-reactive-rest-service/initial`
- Jump ahead to [Create a WebFlux Handler](#).

When you finish, you can check your results against the code in `gs-reactive-rest-service/complete`.

Starting with Spring Initializr:

You can use this [pre-initialized project](#) and click Generate to download a ZIP file. This project is configured to fit the examples in this tutorial.

To manually initialize the project:

1. Navigate to <https://start.spring.io>. This service pulls in all the dependencies you need for an application and does most of the setup for you.
2. Choose either Gradle or Maven and the language you want to use. This guide assumes that you chose Java.
3. Click **Dependencies** and select **Spring Reactive Web**.
4. Click **Generate**.
5. Download the resulting ZIP file, which is an archive of a web application that is configured with your choices.

If your IDE has the Spring Initializr integration, you can complete this process from your IDE.

You can also fork the project from Github and open it in your IDE or other editor.

Create a WebFlux Handler:

We're going to start with a `Greeting` POJO that will be serialized as JSON by our RESTful service:

`src/main/java/hello/Greeting.java`

```
Copypackage com.example.reactivewebservice;

public class Greeting {

    private String message;

    public Greeting() {
```

```

    }

    public Greeting(String message) {
        this.message = message;
    }

    public String getMessage() {
        return this.message;
    }

    public void setMessage(String message) {
        this.message = message;
    }

    @Override
    public String toString() {
        return "Greeting{" +
            "message='" + message + '\'' +
            '}';
    }
}

```

In the Spring Reactive approach, we use a handler to handle the request and create a response, as shown in the following example:

[src/main/java/hello/GreetingHandler.java](#)

```

Copypackage com.example.reactivewebservice;

import org.springframework.http.MediaType;
import org.springframework.stereotype.Component;
import org.springframework.web.reactive.function.BodyInserters;
import
org.springframework.web.reactive.function.server.ServerRequest;
import
org.springframework.web.reactive.function.server.ServerResponse;

import reactor.core.publisher.Mono;

@Component
public class GreetingHandler {

    public Mono<ServerResponse> hello(ServerRequest request) {
        return
ServerResponse.ok().contentType(MediaType.APPLICATION_JSON)
            .body(BodyInserters.fromValue(new Greeting("Hello, Spring!")));
    }
}

```

This simple reactive class always returns a JSON body with a “Hello, Spring!” greeting. It could return many other things, including a stream of items from a database, a stream of items that were generated by calculations, and so on. Note the reactive code:

a `Mono` object that holds a `ServerResponse` body.

Create a Router:

In this application, we use a router to handle the only route we expose (`/hello`), as shown in the following example:

`src/main/java/hello/GreetingRouter.java`

```
Copypackage com.example.reactivewebservice;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.MediaType;
import
org.springframework.web.reactive.function.server.RouterFunction;
import
org.springframework.web.reactive.function.server.RouterFunctions;
import
org.springframework.web.reactive.function.server.ServerResponse;

import static
org.springframework.web.reactive.function.server.RequestPredicates.GET;
import static
org.springframework.web.reactive.function.server.RequestPredicates.accept;

@Configuration(proxyBeanMethods = false)
public class GreetingRouter {

    @Bean
    public RouterFunction<ServerResponse> route(GreetingHandler
greetingHandler) {

        return RouterFunctions
            .route(GET("/hello").and(accept(MediaType.APPLICATION_JSON)),
greetingHandler::hello);
    }
}
```

The router listens for traffic on the `/hello` path and returns the value provided by our reactive handler class.

Create a WebClient:

The Spring `RestTemplate` class is, by nature, blocking. Consequently, we do not want to use it in a reactive application. For reactive applications, Spring offers the `WebClient` class, which is non-blocking. We use a `WebClient`-based implementation to consume our RESTful service:

`src/main/java/hello/GreetingClient.java`

```
Copypackage com.example.reactivewebservice;

import reactor.core.publisher.Mono;

import org.springframework.http.MediaType;
import org.springframework.stereotype.Component;
import org.springframework.web.reactive.function.client.WebClient;

@Component
public class GreetingClient {

    private final WebClient client;

    // Spring Boot auto-configures a `WebClient.Builder` instance with
    // nice defaults and customizations.
    // We can use it to create a dedicated `WebClient` for our
    // component.
    public GreetingClient(WebClient.Builder builder) {
        this.client = builder.baseUrl("http://localhost:8080").build();
    }

    public Mono<String> getMessage() {
        return
this.client.get().uri("/hello").accept(MediaType.APPLICATION_JSON)
                .retrieve()
                .bodyToMono(Greeting.class)
                .map(Greeting::getMessage);
    }
}
```

The `WebClient` class uses reactive features, in the form of a `Mono` to hold the content of the message (returned by the `getMessage` method). This is using a function API, rather than an imperative one, to chain reactive operators. It can take time to get used to [Reactive APIs](#), but the `WebClient` has interesting features and can also be used in traditional Spring MVC applications.

You can use `WebClient` to communicate with non-reactive, blocking services, too.

Make the Application Executable:

We're going to use the `main()` method to drive our application and get the Greeting message from our endpoint.

`src/main/java/hello/Application.java`

```
Copypackage com.example.reactivewebservice;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ConfigurableApplicationContext;

@SpringBootApplication
public class ReactiveWebServiceApplication {

    public static void main(String[] args) {
        ConfigurableApplicationContext context =
SpringApplication.run(ReactiveWebServiceApplication.class, args);
        GreetingClient greetingClient =
context.getBean(GreetingClient.class);
        // We need to block for the content here or the JVM might exit
before the message is logged
        System.out.println(">> message = " +
greetingClient.getMessage().block());
    }
}
```

`@SpringBootApplication` is a convenience annotation that adds all of the following:

- `@Configuration`: Tags the class as a source of bean definitions for the application context.
- `@EnableAutoConfiguration`: Tells Spring Boot to start adding beans based on classpath settings, other beans, and various property settings. For example, if `spring-webmvc` is on the classpath, this annotation flags the application as a web application and activates key behaviors, such as setting up a `DispatcherServlet`.
- `@ComponentScan`: Tells Spring to look for other components, configurations, and services in the `hello` package, letting it find the controllers.

The `main()` method uses Spring Boot's `SpringApplication.run()` method to launch an application. Did you notice that there was not a single line of XML? There is no `web.xml` file, either. This web application is 100% pure Java and you did not have to deal with configuring any plumbing or infrastructure.

Build an executable JAR

You can run the application from the command line with Gradle or Maven. You can also build a single executable JAR file that contains all the necessary dependencies, classes, and resources and run that. Building an executable jar makes it easy to ship, version, and

deploy the service as an application throughout the development lifecycle, across different environments, and so forth.

If you use Gradle, you can run the application by using `./gradlew bootRun`.

Alternatively, you can build the JAR file by using `./gradlew build` and then run the JAR file, as follows:

```
java -jar build/libs/gs-reactive-rest-service-0.1.0.jar
```

If you use Maven, you can run the application by using `./mvnw spring-boot:run`.

Alternatively, you can build the JAR file with `./mvnw clean package` and then run the JAR file, as follows:

```
java -jar target/gs-reactive-rest-service-0.1.0.jar
```

The steps described here create a runnable JAR. You can also [build a classic WAR file](#).

Logging output is displayed. The service should be up and running within a few seconds. Once the service has started, you can see a line that reads:

```
>> message = Hello, Spring!
```

That line comes from the reactive content being consumed by the WebClient. Naturally, you can find something more interesting to do with your output than put it in System.out.

Test the Application:

Now that the application is running, you can test it. To start with, you can open a browser and go to <http://localhost:8080/hello> and see, “Hello, Spring!” For this guide, we also created a test class to get you started on testing with the `WebTestClient` class.

```
src/test/java/hello/GreetingRouterTest.java
```

```
Copypackage com.example.reactivewebservice;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.http.MediaType;
import
org.springframework.test.context.junit.jupiter.SpringExtension;
import org.springframework.test.web.reactive.server.WebTestClient;

import static org.assertj.core.api.Assertions.assertThat;

@ExtendWith(SpringExtension.class)
// We create a `@SpringBootTest`, starting an actual server on a
`RANDOM_PORT`
@SpringBootTest(webEnvironment =
SpringBootTest.WebEnvironment.RANDOM_PORT)
public class GreetingRouterTest {
```

```

    // Spring Boot will create a `WebTestClient` for you,
    // already configure and ready to issue requests against
    "localhost:RANDOM_PORT"
    @Autowired
    private WebTestClient webTestClient;

    @Test
    public void testHello() {
        webTestClient
            // Create a GET request to test an endpoint
            .get().uri("/hello")
            .accept(MediaType.APPLICATION_JSON)
            .exchange()
            // and use the dedicated DSL to test assertions against the
            response
            .expectStatus().isOk()
            .expectBody(Greeting.class).value(greeting -> {
                assertThat(greeting.getMessage()).isEqualTo("Hello,
Spring!");
            });
    }
}

```

API & MICROSERVICES

IMPORTANT QUESTIONS - UNIT-4

1. What are Web Services:
2. Why Web services,
3. Explain SOA - Service Oriented Architecture, ,
4. Explain Types of Web Services,
5. Explain SOAP based Web Services,
6. 6, Explain RESTful Web Services,
7. How to create RESTful Services