1. Uniform Interface: leads to performance, scalability, modifiability. Http methods
2. All data and functionality can be accessed using URI’s
3. Stateless protocol (http)
4. Self-descriptive messages, content can be accessed in HTML, XML, plain text, PDF, JPEG, JSON, and others.
5. Stateful interaction via hyperlinks

JAX-RS is a Java programming language API designed to make it easy to develop applications that use the REST architecture. Developers decorate Java programming language class files with JAX-RS annotations to define resources and the actions that can be performed on those resources. JAX-RS annotations are runtime annotations; therefore, runtime reflection will generate the helper classes and artifacts for the resource

Implementations: jax-rs: jersey, restlet, resteasy, spring

Test rest app: postman

 With Amazon S3, buckets and objects can be created, listed, and retrieved using either a REST-style HTTP interface or a SOAP interface.

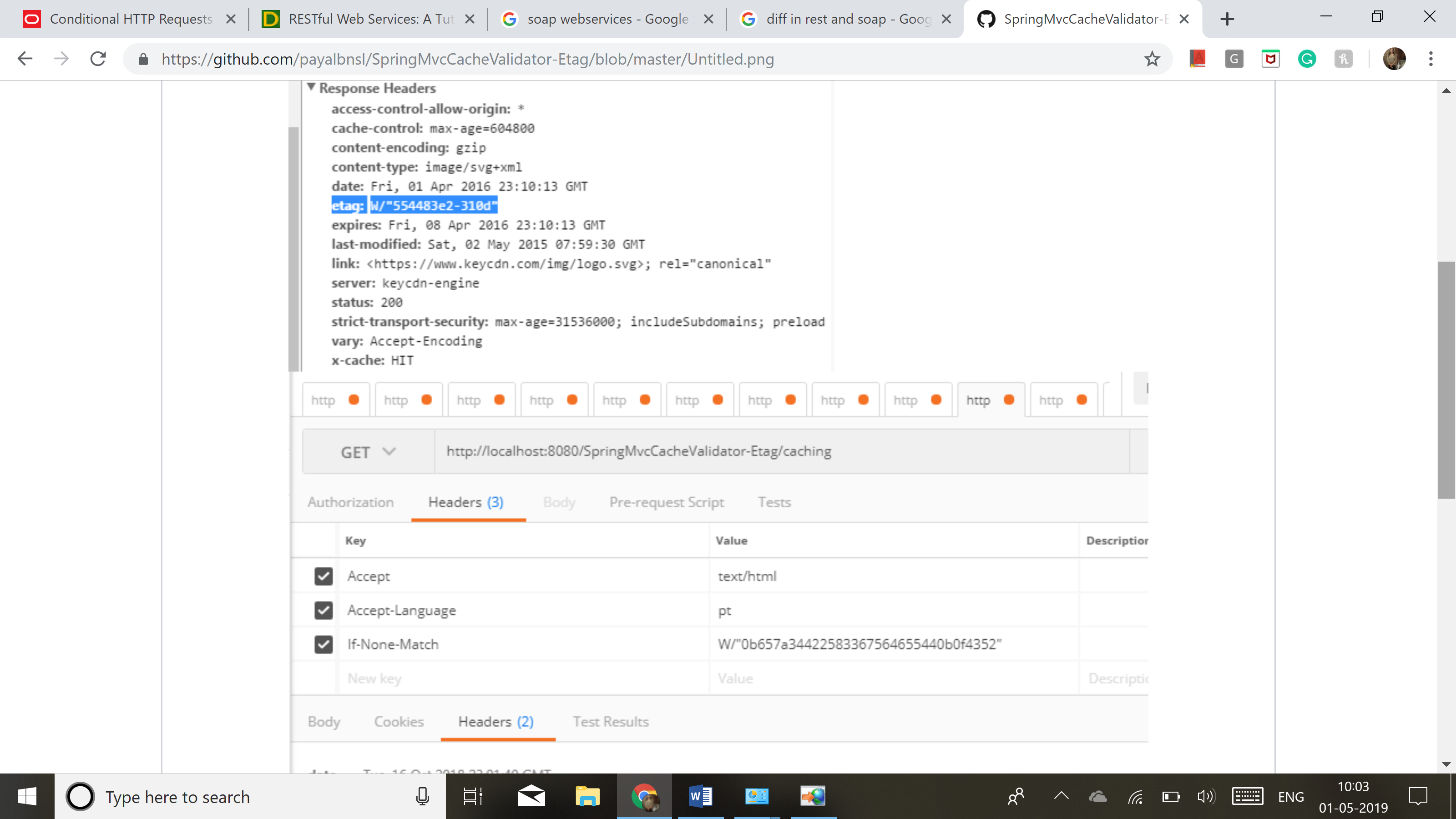
witter public timeline messages and lets you view and update your Twitter status.

Default bean scope for web app?

A GET request can return a Not Modified (304) response if the representation has not changed since the previous request. For example, a web site can return 304 responses for all its static images that have not changed since the previous request.

A PUT request can return a Precondition Failed (412) response if the representation has been modified since the last request. The conditional PUT can help avoid the lost update problem.

Conditional HTTP requests can be used with the Last-Modified and ETag headers. The Last-Modified header can represent dates with granularity of one second.



The @Produces and @Consumes annotations handle static content negotiation in JAX-RS. These annotations specify the content preferences of the server. HTTP headers such as Accept, Content-Type, and Accept-Language define the content negotiation preferences of the client.

Java Architecture for XML Binding (JAXB) is an XML-to-Java binding technology that simplifies the development of web services by enabling transformations between schema and Java objects and between XML instance documents and Java object instances.

 you can use the JAXB binding runtime to marshal and unmarshal your XML documents to and from Java objects and use the resulting Java classes to assemble a web services application.

JAXB is a library that was part of jdk5 to jdk 8. Or u may use Jackson for marshalling/ unmarshalling.

From schema file to java file: xjc abc.xsd

The critical difference between DTDs and XML Schema is that XML Schema utilize an XML-based syntax, whereas DTDs have a unique syntax held over from SGML DTDs. Although DTDs are often criticized because of this need to learn a new syntax, the syntax itself is quite terse. The opposite is true for XML Schema, which are verbose, but also make use of tags and XML so that authors of XML should find the syntax of XML Schema less intimidating.

The goal of DTDs was to retain a level of compatibility with SGML for applications that might want to convert SGML DTDs into XML DTDs. However, in keeping with one of the goals of XML, "terseness in XML markup is of minimal importance," there is no real concern with keeping the syntax brief.

[...]

So what are some of the other differences which might be especially important when we are converting a DTD? Let's take a look.

**Typing**

The most significant difference between DTDs and XML Schema is the capability to create and use datatypes in Schema in conjunction with element and attribute declarations. In fact, it's such an important difference that one half of the XML Schema Recommendation is devoted to datatyping and XML Schema. We cover datatypes in detail in Part III of this book, "XML Schema Datatypes."

[...]

**Occurrence Constraints**

Another area where DTDs and Schema differ significantly is with occurrence constraints. If you recall from our previous examples in Chapter 2, "Schema Structure" (or your own work with DTDs), there are three symbols that you can use to limit the number of occurrences of an element: \*, + and ?.

[...]

**Enumerations**

So, let's say we had a element, and we wanted to be able to define a size attribute for the shirt, which allowed users to choose a size: small, medium, or large. Our DTD would look like this:

<!ELEMENT item (shirt)>

<!ELEMENT shirt (#PCDATA)>

<!ATTLIST shirt

size\_value (small | medium | large)>

[...]

But what if we wanted size to be an element? We can't do that with a DTD. DTDs do not provide for enumerations in an element's text content. However, because of datatypes with Schema, when we declared the enumeration in the preceding example, we actually created a simpleTypecalled size\_values which we can now use with an element:

<xs:element name="size" type="size\_value">

To make a call from one webservice to other in rest we use RestTemplate

To cache the data of one webservice in other, we may use EhCache library provided by Spring

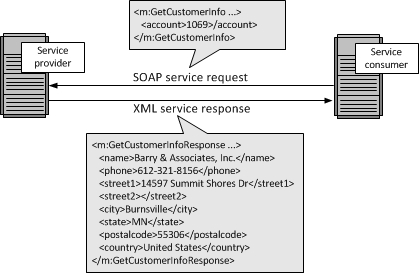
<https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-caching.html>

Soap:

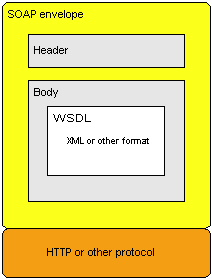
**SOAP** or **Simple Object Access Protocol** is a protocol designed to exchange information in the form of Web Services. It is primarily based on XML documents exchanged over HTTP, but it's possible to transmit messages through other mediums like email and JMS.

SOAP web services are generally based on a a [Web Services Description Language](https://en.wikipedia.org/wiki/Web_Services_Description_Language) or WSDL, which is an XML contract that defines all the data and services offered by a given web service. The client and the server both use this contract as a basis for exchanging information and making [remote procedural calls](https://en.wikipedia.org/wiki/Remote_procedure_call).

The following figure illustrates using SOAP for Web Services. Also see [Web Services Explained](https://www.service-architecture.com/articles/web-services/web_services_explained.html).



SOAP is an alternative to [Representational State Transfer (REST)](https://www.service-architecture.com/articles/web-services/representational_state_transfer_rest.html) and [JavaScript Object Notation (JSON)](https://www.service-architecture.com/articles/web-services/javascript_object_notation_json.html).



The SOAP envelope contains two parts:

1. An optional header providing information on authentication, encoding of data, or how a recipient of a SOAP message should process the message.
2. The body that contains the message. These messages can be defined using the WSDL specification.

The SOAP web services architecture is based on interactions between three components: a service provider, a service requester, and an optional service registry.

**The service provider**

The collection of software that provides a web service.

* The application program
* The middleware
* The platform on which they run

**The service requester**

The collection of software that is responsible for requesting a web service from a service provider.

* The application program
* The middleware
* The platform on which they run

**The service registry**

The service registry is a central location where service providers can publish their service descriptions and where service requesters can find those service descriptions.

The registry is an optional component of the web services architecture because service requesters and providers can communicate without it in many situations. For example, the organization that provides a service can distribute the service description directly to the users of the service in a number of ways, including offering the service as a download from an FTP site.

Using a service registry offers a number of advantages to both the requester and provider; for example, using the IBM® WebSphere® Service Registry and Repository (WSRR) can help the requester to find services more quickly and can help the provider to enforce version control of the services being offered.

CICS® provides direct support for implementing service requester and service provider components. However, you need additional software to deploy a service registry in CICS. If you use the IBM WebSphere Service Registry and Repository (WSRR), CICS provides support for WSRR through the web services assistant. Alternatively, you can deploy a service registry on another platform.

**Interactions between a service provider, a service requester, and, a service registry**

The interactions between the service provider, service requester, and service registry involve the following operations:

**Publish**

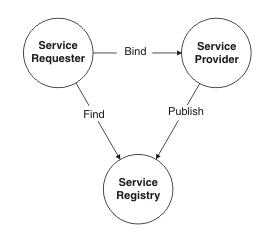
When a service registry is used, a service provider publishes its service description in a service registry for the service requester to find.

**Find**

When a service registry is used, a service requester finds the service description in the registry.

**Bind**

The service requester uses the service description to bind with the service provider and interact with the web service implementation.

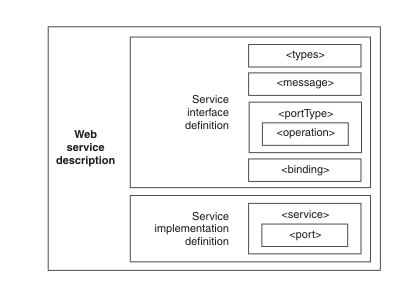
*Figure 1. web services components and interactions*

A web service description is a document by which the *service provider* communicates the specifications for starting the web service to the *service requester*. Web service descriptions are expressed in the XML application known as Web Service Description Language (WSDL).

The service description describes the web service in such a way as to minimize the amount of shared knowledge and customized programming that is needed to ensure communication between the service provider and the service requester. For example, neither the requester nor the provider needs to be aware of the platform on which the other runs, nor of the programming language in which the other is written.

A service description can conform to either the WSDL 1.1 or WSDL 2.0 specification. Each has differences in both the terminology and major elements that can be included in the service description. The following information uses WSDL 1.1 terminology and elements to explain the purpose of the service description.

The structure of WSDL allows a service description to be partitioned into two definitions:

* An abstract *service interface definition* that describes the interfaces of the service and makes it possible to write programs that implement and start the service.
* A concrete *service implementation definition* that describes the location on the network (or *endpoint*) of the web service of the provider and other implementation-specific details. It enables a service requester to connect to the service provider.
* WSDL 1.1 document uses the following major elements in the definition of network services:
* **<types>**
* A container for data type definitions using some type system (such as XML Schema). Defines the data types used within the message. The <types> element is not required when all messages consist of simple data types.
* **<message>**
* Specifies which XML data types are used to define the input and output parameters of an operation.
* **<portType>**
* Defines the set of operations supported by one or more endpoints. Within a <portType> element, each operation is described by an <operation> element.
* **<operation>**
* Specifies which XML messages can appear in the input and output data flows. An operation is comparable with a method signature in a programming language.
* **<binding>**
* Describes the protocol, data format, security, and other attributes for a particular <portType> element.
* **<port>**
* Specifies the network address of an endpoint and associates it with a <binding> element.
* **<service>**
* Defines the web service as a collection of related endpoints. A <service> element contains one or more <port> elements.
* *Figure 1. Structure of a web service description*
* <?xml version="1.0" encoding="UTF-8" standalone="no"?>
* <wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
* xmlns:tns="http://www.cleverbuilder.com/BookService/" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
* xmlns:xsd="http://www.w3.org/2001/XMLSchema" name="BookService"
* targetNamespace="http://www.cleverbuilder.com/BookService/">
* <wsdl:documentation>Definition for a web service called BookService,
* which can be used to add or retrieve books from a collection.
* </wsdl:documentation>
* <!--
* The `types` element defines the data types (XML elements)
* that are used by the web service.
* -->
* <wsdl:types>
* <xsd:schema targetNamespace="http://www.cleverbuilder.com/BookService/">
* <xsd:element name="GetBook">
* <xsd:complexType>
* <xsd:sequence>
* <xsd:element name="ID" type="xsd:string" />
* </xsd:sequence>
* </xsd:complexType>
* </xsd:element>
* <xsd:element name="GetBookResponse">
* <xsd:complexType>
* <xsd:sequence>
* <xsd:element name="ID" type="xsd:string"></xsd:element>
* <xsd:element name="Title" type="xsd:string"></xsd:element>
* <xsd:element name="Author" type="xsd:string"></xsd:element>
* </xsd:sequence>
* </xsd:complexType>
* </xsd:element>
* <xsd:element name="AddBook">
* <xsd:complexType>
* <xsd:sequence>
* <xsd:element name="Title" type="xsd:string"></xsd:element>
* <xsd:element name="Author" type="xsd:string"></xsd:element>
* </xsd:sequence>
* </xsd:complexType>
* </xsd:element>
* <xsd:element name="AddBookResponse">
* <xsd:complexType>
* <xsd:sequence>
* <xsd:element name="ID" type="xsd:string"></xsd:element>
* </xsd:sequence>
* </xsd:complexType>
* </xsd:element>
* <xsd:element name="GetAllBooks">
* <xsd:complexType />
* </xsd:element>
* <xsd:element name="GetAllBooksResponse">
* <xsd:complexType>
* <xsd:sequence>
* <xsd:element name="Book" minOccurs="0" maxOccurs="unbounded">
* <xsd:complexType>
* <xsd:sequence>
* <xsd:element name="ID" type="xsd:string"></xsd:element>
* <xsd:element name="Title" type="xsd:string"></xsd:element>
* <xsd:element name="Author" type="xsd:string"></xsd:element>
* </xsd:sequence>
* </xsd:complexType>
* </xsd:element>
* </xsd:sequence>
* </xsd:complexType>
* </xsd:element>
* </xsd:schema>
* </wsdl:types>

* <!--
* A wsdl `message` element is used to define a message
* exchanged between a web service, consisting of zero
* or more `part`s.
* -->
* <wsdl:message name="GetBookRequest">
* <wsdl:part element="tns:GetBook" name="parameters" />
* </wsdl:message>
* <wsdl:message name="GetBookResponse">
* <wsdl:part element="tns:GetBookResponse" name="parameters" />
* </wsdl:message>
* <wsdl:message name="AddBookRequest">
* <wsdl:part name="parameters" element="tns:AddBook"></wsdl:part>
* </wsdl:message>
* <wsdl:message name="AddBookResponse">
* <wsdl:part name="parameters" element="tns:AddBookResponse"></wsdl:part>
* </wsdl:message>
* <wsdl:message name="GetAllBooksRequest">
* <wsdl:part name="parameters" element="tns:GetAllBooks"></wsdl:part>
* </wsdl:message>
* <wsdl:message name="GetAllBooksResponse">
* <wsdl:part name="parameters" element="tns:GetAllBooksResponse"></wsdl:part>
* </wsdl:message>
* <!--
* A WSDL `portType` is used to combine multiple `message`s
* (e.g. input, output) into a single operation.
* Here we define three synchronous (input/output) operations
* and the `message`s that must be used for each.
* -->
* <wsdl:portType name="BookService">
* <wsdl:operation name="GetBook">
* <wsdl:input message="tns:GetBookRequest" />
* <wsdl:output message="tns:GetBookResponse" />
* </wsdl:operation>
* <wsdl:operation name="AddBook">
* <wsdl:input message="tns:AddBookRequest"></wsdl:input>
* <wsdl:output message="tns:AddBookResponse"></wsdl:output>
* </wsdl:operation>
* <wsdl:operation name="GetAllBooks">
* <wsdl:input message="tns:GetAllBooksRequest"></wsdl:input>
* <wsdl:output message="tns:GetAllBooksResponse"></wsdl:output>
* </wsdl:operation>
* </wsdl:portType>
* <!--
* The `binding` element defines exactly how each
* `operation` will take place over the network.
* In this case, we are using SOAP.
* -->
* <wsdl:binding name="BookServiceSOAP" type="tns:BookService">
* <soap:binding style="document"
* transport="http://schemas.xmlsoap.org/soap/http" />
* <wsdl:operation name="GetBook">
* <soap:operation soapAction="http://www.cleverbuilder.com/BookService/GetBook" />
* <wsdl:input>
* <soap:body use="literal" />
* </wsdl:input>
* <wsdl:output>
* <soap:body use="literal" />
* </wsdl:output>
* </wsdl:operation>
* <wsdl:operation name="AddBook">
* <soap:operation soapAction="http://www.cleverbuilder.com/BookService/AddBook" />
* <wsdl:input>
* <soap:body use="literal" />
* </wsdl:input>
* <wsdl:output>
* <soap:body use="literal" />
* </wsdl:output>
* </wsdl:operation>
* <wsdl:operation name="GetAllBooks">
* <soap:operation
* soapAction="http://www.cleverbuilder.com/BookService/GetAllBooks" />
* <wsdl:input>
* <soap:body use="literal" />
* </wsdl:input>
* <wsdl:output>
* <soap:body use="literal" />
* </wsdl:output>
* </wsdl:operation>
* </wsdl:binding>
* <!--
* The `service` element finally says where the service
* can be accessed from - in other words, its endpoint.
* -->
* <wsdl:service name="BookService">
* <wsdl:port binding="tns:BookServiceSOAP" name="BookServiceSOAP">
* <soap:address location="http://www.example.org/BookService" />
* </wsdl:port>
* </wsdl:service>
* </wsdl:definitions>

## Defining SOAP and REST

SOAP (Simple Object Access Protocol) and REST (Representational State Transfer) are both web service communication protocols. SOAP was long the standard approach to web service interfaces, although it’s been dominated by REST in recent years, with REST now representing more than 70% of public APIs according to [Stormpath](https://stormpath.com/blog/rest-vs-soap" \t "_blank).

Rest documentation: swagger2, soap has wsdl as contract

REST accesses data while SOAP performs operations through a more [standardized set](http://blog.smartbear.com/apis/understanding-soap-and-rest-basics/) of messaging patterns

SOAP was originally created by Microsoft, and it’s been around a lot longer than REST. This gives it the advantage of being an established, legacy protocol. But REST has been around for a good time now as well. Plus, it entered the scene as a way to access web services in a much simpler way than possible with SOAP by using HTTP.

* REST allows a greater variety of data formats, whereas SOAP only allows XML.
* Coupled with JSON (which typically works better with data and offers faster parsing), REST is generally considered easier to work with.
* Thanks to JSON, REST offers better support for browser clients.
* REST provides superior performance, particularly through caching for information that’s not altered and not dynamic.
* It is the protocol used most often for major services such as Yahoo, Ebay, Amazon, and even Google.
* REST is generally faster and uses less bandwidth. It’s also easier to integrate with existing websites with no need to refactor site infrastructure. This enables developers to work faster rather than spend time rewriting a site from scratch. Instead, they can simply add additional functionality.

## Benefits of SOAP Over REST

if you need more robust security, SOAP’s support for WS-Security can come in handy. It offers some additional assurances for data privacy and integrity. It also provides support for identity verification through intermediaries rather than just point-to-point, as provided by SSL (which is supported by both SOAP and REST).

Another advantage of SOAP is that it offers built-in retry logic to compensate for failed communications. REST, on the other hand, doesn’t have a built-in messaging system. If a communication fails, the client has to deal with it by retrying. There’s also no standard set of rules for REST. This means that both parties (the service and the consumer) need to understand both content and context.

Other benefits of SOAP include:

* SOAP’s standard HTTP protocol makes it easier for it to operate across firewalls and proxies [without modifications](http://searchmicroservices.techtarget.com/tip/REST-vs-SOAP-Choosing-the-best-web-service) to the SOAP protocol itself. But, because it uses the complex XML format, it tends to be slower compared to middleware such as ICE and COBRA.
* Additionally, while it’s rarely needed, some use cases require greater transactional reliability than what can be achieved with HTTP (which limits REST in this capacity). If you need ACID-compliant transactions, SOAP is the way to go.
* In some cases, designing SOAP services can actually be less complex compared to REST. For web services that support complex operations, requiring content and context to be maintained, designing a SOAP service requires less coding in the application layer for transactions, security, trust, and other elements.
* SOAP is highly extensible through other protocols and technologies. In addition to WS-Security, SOAP supports WS-Addressing, WS-Coordination, WS-ReliableMessaging, and a host of other web services standards, a full list of which you can find on [W3C](https://www.w3.org/Submission/).

t the end of the day, the best protocol is the one that makes the most sense for the organization, the types of clients that you need to support, and what you need in terms of flexibility. Most new APIs are built using REST and JSON, simply because it typically consumes less bandwidth and is easier to understand both for developers implementing initial APIs as well as other developers who may write other services against it. Because it’s more easily consumed by most of today’s web browsers, REST+JSON has become the defacto technology for the majority of public APIs. However, SOAP remains a valuable protocol in some circumstances. Plus, you don’t have to look far to find die-hard fans advocating for SOAP for certain use cases.