Service-Oriented Architecture

COURSE NOTES

Version 0.1.0

Table of Contents

Course Overview

This course is the fourth in a specialization on Software Design and Architecture, which was brought to you in partnership by Coursera and the University of Alberta. The previous three courses focused on Object-Oriented Design, Design Patterns, and Software Architecture. This course builds on the previous three, and will focus on Service-Oriented Architecture. However, it can also be a stand-alone course.

In Service-Oriented Architecture, we will explore notions such as coupling, separation of concerns, and layering, and use UML diagrams to understand the structure and behaviour of service-oriented systems, web applications and web services.

First, this course will teach you about systems based on services, service-oriented architecture, and a variety of important web standards. These standards can inform your understanding of architecture in web applications, as well as in systems based on web services. Next, we will cover "first generation" web services. The third module examines REST architecture and REST services, which are also based on web standards.

Upon completion of this course, you will be able to:

1) Explain service-oriented architecture
2) Explain systems based on services, service-oriented architecture, and important web standards
3) Use "first generation" web services
4) Explain REST architecture and REST services

Module 1: Web Technologies

Upon completion of this module, you will be able to:

(a) Define service oriented architecture
(b) Explain the service principles
(c) Describe service systems architecture
(d) Understand the history of the web
(e) Describe web technologies, including:
    a. layered systems
    b. XML/HTML
    c. HTTP
    d. Javascript
(f) Explain the basics of distributed systems
(g) Explain RPC
(h) Explain object brokers

Service-Oriented Architecture

In everyday life, we frequently use **services** to help us out. Services are valuable actions that help fulfill a demand. For example, a laundromat, a restaurant, and ride services are example of services.

Software also makes use of services. In the software industry, deploying a service means providing a tool that other software can use. A service is external to the software requesting it, and often remove – either in another service in the company or somewhere on the internet.

Services are often associated with two roles:

1. the service requester, which is the software requesting the service
2. the service provider, which fulfills requests

These roles echo client and server roles in similar domains, so sometimes these terms are used interchangeably.

Service-oriented architecture examines how to build, use, and combine services. Instead of creating large software suites that do everything, service-oriented architecture reaches software goals by building and using

services, and designing an architecture that supports their use.

This course will cover software-oriented architecture in two contexts:

1. on the Internet (also known as "external" to an organization)
2. in large organizations (also known as "internal" to the organization)

Although the concept is the same in both cases, the results will change based on the context.

Web Services

Internet or **web services** are services that are offered on the Internet. It is possible to build feature-full apps on the Internet by using existing web services external to your application to fulfill some of the tasks. For instance, a web application for travelling may take advantage of services that obtain flight prices on the internet, services that obtain hotel prices, or car rental services. Essentially, a number of services only had to be combined to create an application.

Web services may entail trade-offs. The ease of using existing services must be balanced against qualities of the services, which is not under the control of developers. In these cases, non-functional requirements become very important. These might include:

- Response Time
- Supportability
- Availability

of the service, as determined by outside parties.

Large Organizations

In large enterprises or organizations, code that is built in-house can be turned into services. In turn, these services can be used by different parts of the business, or by adding interfaces to existing software. These services can then be used to drive business or organizational goals. For example, a company might offer an interface for software systems in various departments to request information about support costs in their software. Support costs are offered as a service that departments can query.

Internal service-oriented architecture encourages organizations to build general, reusable software services that can be used and combined as needed. This architecture allows businesses to respond to opportunities quickly, and makes services easier to access for new business units.

Developing an extensive service-oriented architecture (SOA) can present trade-offs for large organizations. A full switch to SOA is costly, and it can be difficult to support despite its benefits. In these cases, services are often introduced bit by bit, separating out the most useful, cross-departmental functions first.

Like any other architecture, SOA requires trade-offs and design decisions in implementation, whether it is through web services or in-house organizational services. Additionally, SOA is not easy. Once in place, however, SOAs are powerful, as they provide modularity, extensibility, and code reuse. Applications can be built through combining services. New services can be created by combining existing services, either from the ground up, or through the addition of interfaces to existing code.

Service Principles

In order to create useful and reusable services, and by extension, service-oriented architecture (SOA), there are certain best practices, guidelines, and principles that have been developed that outline the desired properties that services should have. These desired properties for services are outlined in the table below.

Desired Property	Description
Modular and Loosely Coupled	Services should be module and loosely coupled. This allows services to be reusable and combinable – in other words, services can be mixed and matched if they are modular. In object-oriented programming, loose coupling is achieved by exposing only the relevant elements of a class or component to its client. In SOA, requests are made by passing communication to the service in a way that aligns with its interface. The service performs the necessary operations and then passes back a communication containing the result of the service or a confirmation that the request was fulfilled.

Composable	Services should be used in combination, in order to create usable applications or other services. In order to achieve this property, services should be modular. Just like objects can be combined in object-oriented programming to provide the desired behaviour, services should be able to combined to provide a desired end-goal in SOA.
Platform- and Language-Independent	Services should be platform independent and language independent. For example, a service coded in Java can be used by a service requester coded in Ruby. To achieve platform- and language-independence, communication standards and protocols must be followed. For example, services on the Internet are often requested with an XML file or HTTP request.
Self-Describing	A service should describe how to interact with it. In other words, a service should describe its own interfaces. This includes what input the service takes, and what output the service gives. There are formal standards for describing services, including **web service description language (WSDL**). WSDL will be explored later in this course.
Self-Advertising	Services must make known to potential clients that it is available. In-house organizations may create service catalogues, while distributed applications using web services have standards like **Universal Description, Discovery, and Integration (UDDI)** to connect service providers with potential service requesters.

History of Web-based Systems

In order to best understand service-oriented architectures (SOA) based on web services, it is important to understand the origin and evolution of web-based systems.

Let us begin by reviewing some foundational terms. Often, the terms "**Internet**" and **"World Wide Web**" are used interchangeable. However, these terms actually refer to two different, but interrelated things. The Internet was actually invented almost 20 years before the World Wide Web!

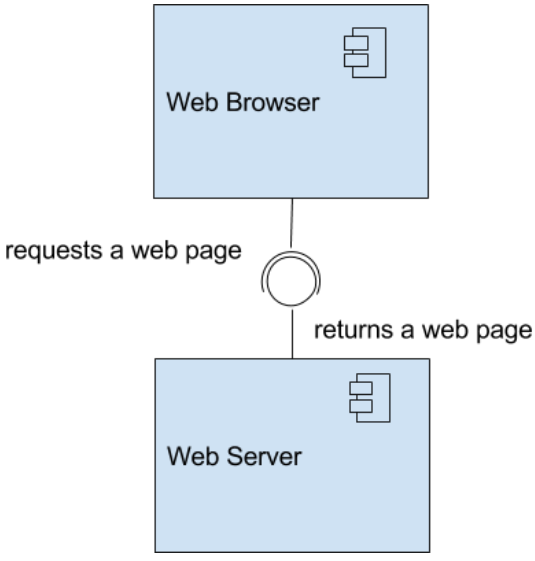
In 1969, a small computer network called ARPANET was created and used by researchers in the United States, in order to send a small amount of data between pairs of computers. This was the first-time data was sent across a computer network. In the following years, small networks were developed at research institutions across the United States. Further, these networks began to connect with each other. Once this network of networks reached a global scale, the Internet was born.

The Internet offered potential to communication information across the global. Tim Berners Lee was inspired by the way the brain links information together through association, and began to investigate how documents could be linked together over networks. This led Lee to propose a web management system which was inspired by hypertext and built on top of the Internet, known as the World Wide Web, in 1990.

The World Wide Web, otherwise known as "the web" led to standards and technologies for computer-based communication over the Internet. Web standards include **Hypertext Markup Language (HTML)**, and **Hypertext Transport Protocol (HTTP)**, which will be explored in later lessons of this course.

The introduction of HTML and web browsers in the early 1990s, which allowed users to view websites, greatly increased the popularity of the web.

Websites are made up of web pages. To view a web page, the web browser makes a request to the web server that hosts the web page, then the web server handles the request by returning the HTML document that corresponds to the requested web page, and then the browser renders this HTML document to the user. The relationship between web browser and web service is a client-server relationship. Both the request and the response are messages conveyed in HTTP, a communication protocol that both the web browser and web server understand.



Static Web Pages

Initially, the web consisted of **static web pages**, which were web pages stored on the server as a separate HTML file.

When a static web page is viewed on a web browser, the HTML rendered on the screen is the same HTML document stored on the web server. The document on the web server has not changed or been customized before being served to the client. In order to change the web page, the corresponding HTML document must be changed. Under this model, even small changes to a web page may require the update of many HTML documents to maintain consistency across the whole website.

Since changes require manual developer intervention, static websites are best used for presenting information that does not change very often, like personal websites or publications.

Dynamic Web Pages

Beginning in 1993, **dynamic web pages** began to emerge in response to the lack of customizability and scalability imposed by static web pages. Dynamic web pages are generated at the time of access. This means that the web page does not exist on a server before it is generated.

When a dynamic web page is viewed, the web server passes on the request to an application to handle. The application can perform a computation, lookup some information in a database, or request information from a web service, which produces dynamic content as output. The application can generate an HTML document for the server, and then the server sends that

back to the web browser, which can then display it for a user.

Making changes to dynamic websites is much easier than for static web pages. Changes need only be applied to one database element or variable in the application to make a change throughout a dynamic website.

Dynamic web pages provide many advantages. They can be customized for the view, they can respond to external events, they generally provide increased functionality compared to their static counterparts, and they are easier for a developer to modify. Currently, dynamic web pages dominate the web, including many types of webpages such as personal blogs and news feeds.

Although it is not always easy to determine if a website is statically or dynamically generated, a good rule of thumb is that more complex websites tend to be dynamic. Static webpages are still excellent tools to use for web pages whose content does not need to be personalized or does not change often.

Web Applications

A growing trend in web-based systems is the use of web-based applications, otherwise known as web applications. **Web applications**, like desktop applications, provide graphical user interfaces that allow users to interact with them, but a web application is run in a web browser and is stored on a remote web server, whereas a desktop application is run and stored locally on a computer.

Web applications are platform independent. This means that they can run on any operating system, provided that a compatible web browser is installed. Web applications eliminate the need for users to download and maintain application software on a computer. However, web applications also require users to have Internet access, because web applications communicate information through HTTP with a web server and/or application server on the backend.

Web applications provide users with a richer, more interactive web experience than simpler dynamic or static web pages. Web applications enable everything from online banking, online learning, online games, calculators, calendars, and more.

Web Services

If web applications or websites integrate with web services, real-time information can be used to create more complex, richer applications. Web services, in turn, can rely on other services. Web services can be used to

satisfy specific needs, such as stock market data, weather reports, or currency conversion.

By treating web services like reusable components, information produced by these web services can be used by many different web applications at the same time. Web applications and web services communicate over the web using open standards like HTTP, XML, and JSON, which are easy for machines to manipulate.

Web services can be accessed through programmatic means, or services provide a user interface that can be embedded in a web page or application. By using web services, request and response of different services is **asynchronous**. This means that the logic is designed to continue running instead of waiting for a response. A page composed of many services can be generated while the individual services are processing requests and sending responses.

Web Technologies

This lesson will examine web technologies used for web-based systems.

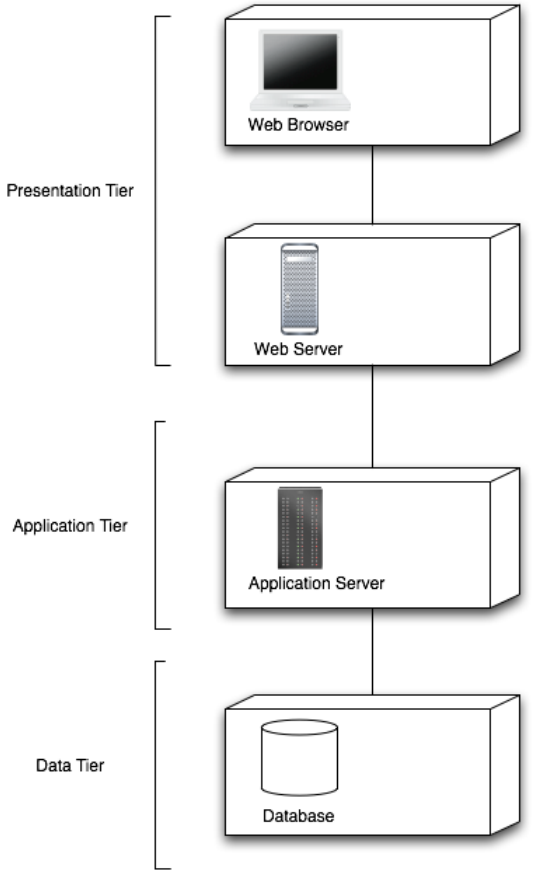
Layered

In the third course of this specialization, the term **layer** was defined as a collection of components that work together toward a common purpose. Layers help identify hierarchies in a system. Knowledge of layers helps software designers restrict how layers interact with each other: components in a layer only interact with components in their own layer or adjacent layers, and this may only be done through interfaces provided by each component. Generally, lower layers provide services to layers above them.

More complex systems often require more layers to help logically separate components. However, the trade-off to adding more layers is that performance suffers, due to an increase in communication required between layers.

Layers are often conceptually organized into **presentation**, **application**, and **data** tiers. In a web-based system, the presentation tier is further divided into two layers: one for the web browser, and one for the web server.

The diagram below illustrates this conceptual organization.



Each layer in the above diagram of a web-based system has a function:

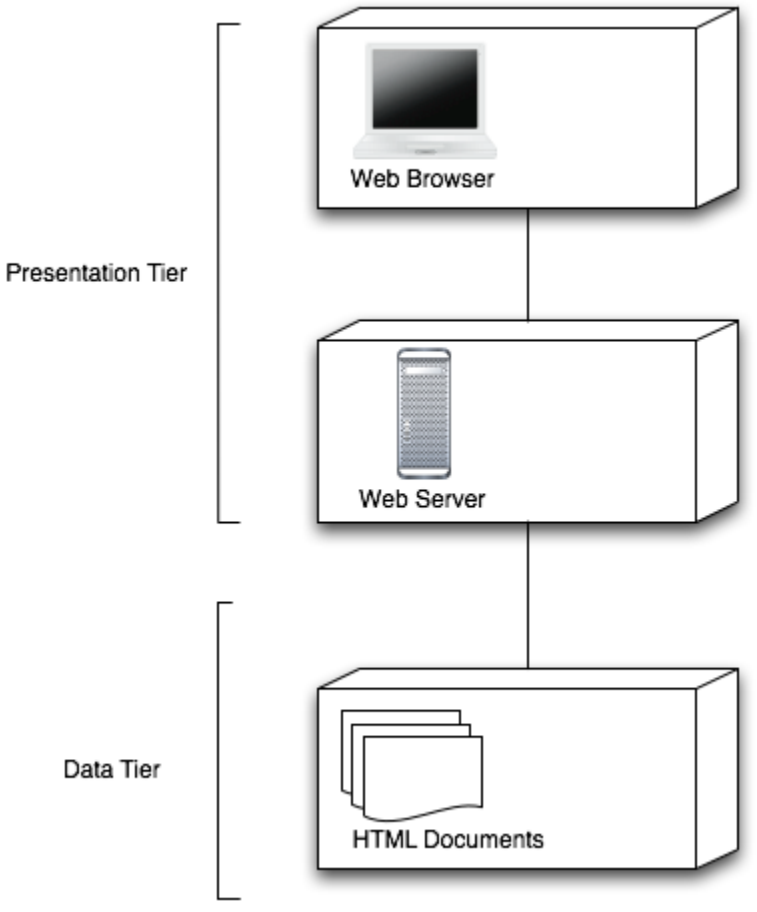
- The **web browser layer** is the topmost layer of the system. It displays information to the user.
- The **web server layer** is directly below the web browser layer. It receives the request from the web browser, obtains the requested content, and returns it to the browser.
- The **application layer** is below the web server layer. It is responsible for ensuring the function or service provided by the system is performed.
- The bottom layer is the **data layer**. This layer is responsible for storing, maintaining, and managing data. Access to data may be read-only, or may allow for both reading and writing. Depending on the system, this access can be in the form of a filesystem or database.

Although these are the typical layers that web systems can be broken up into, there are many ways to separate a web-based system into layers. For

example, not all systems require four layers, and not all systems use these layers the same way.

Layers for Static Web Content

A static web page has a layered architecture of a web browser layer, a web server layer, and a data layer.



The web browser layer typically consists of a web browser, which displays information provided by the web server.

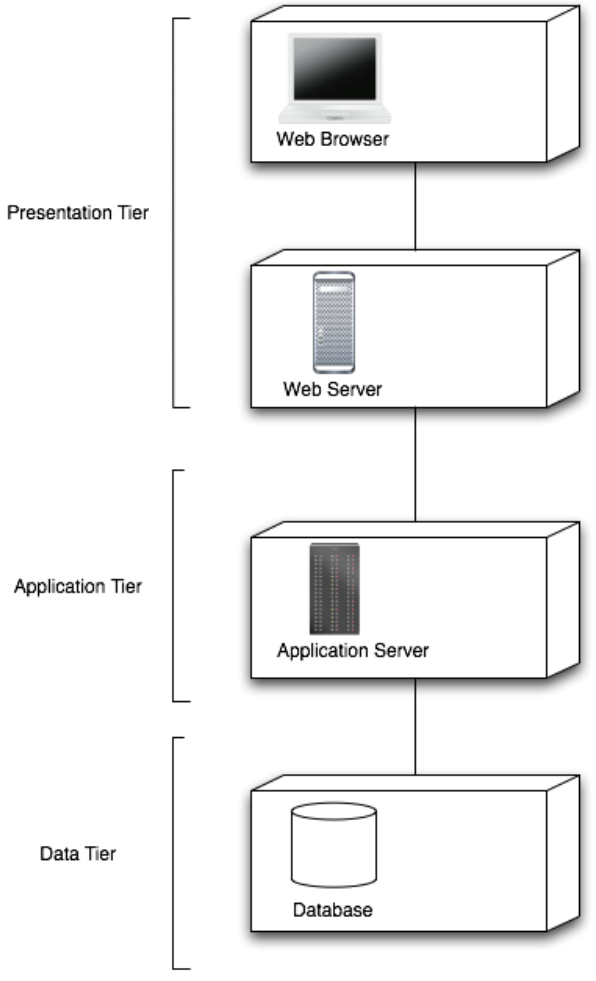
The web server layer receives the request from the web browser, and retrieves the appropriate HTML document stored in the data layer. Once the access to the right HTML document is secured, this layer returns the requested content to the browser.

The data layer consists of HTML documents that are delivered back to the web browser unchanged by the web server. As HTML documents do not change, the filesystem can be read-only.

There is no application layer in a static web content system, as the HTML documents served by the web server are the exact as stored in the file system. No processing has been applied.

Layers for Dynamic Web Content and Web Applications

In a dynamic web content system, a layered architecture requires a web browser layer, web server layer, application layer, and data layer.



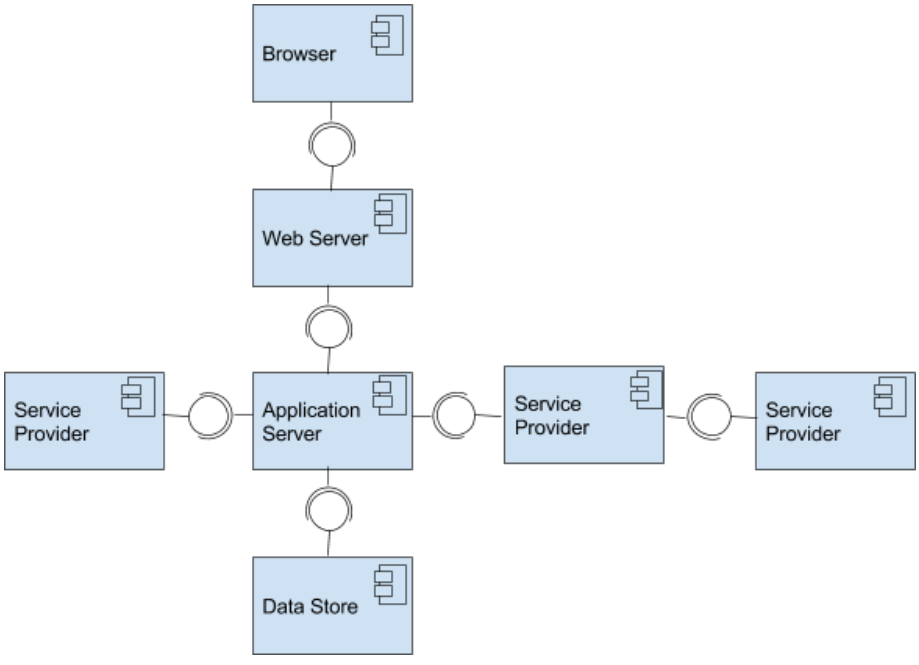
As with static web content, the web browser layer typically consists of a web browser, which displays information provided by the web server. Similarly, the web server layer receives the request from the web browser, and returns the requested content to the browser. Unlike a static web page, however, on a dynamic web page, HTML documents are generated when they are requested by a web browser. The web server passes on the request to an application server in the application layer for processing.

The application layer for dynamic web pages can consist of one or more programs or applications that process the request to generate the resulting content. The application layer may also call upon other web services, and read and write data to a database via the data layer.

This same layering scheme also applies to architectures for complex web applications.

Services View

An alternative view of the architecture can be considered through a UML component diagram. A web-based system can be thought of as a collection of services and service requester/provider pairings.



For example, the database provides data services, and the application server is a server requester to the database. The application server runs programs that may access a variety of web services provided outside the system. Those web services may themselves access other services.

Layered architecture and use of outside web services reinforces the basic design principles of separation of concerns and code reuse. Layers have specific responsibilities, while outside services provide functionality that the system does not have to implement. The scope of using web services is huge, so it raises the challenge of identifying the right ones to use.

XML/HTML/JSON

Web systems user many different types of formats to store and express content. This lesson will explore three common formats:

- **HTML**
- **XML**
- **JSON**

HTML and XML are markup languages, while JSON is a popular, lightweight data-interchange format used in many web applications today.

Markup languages are designed to adorn text in a machine and human readable way, typically to add meaning or structure. Markup languages use tags to mark how certain pieces of text are interpreted. Tags are often standard words that can be defined for some meaning on purpose. In the example below, the text "service-oriented architecture" has been marked-up with tags referring to the text's grammar.

```
<adjective>service-oriented</adjective>
<noun>architecture</noun>
```

There is typically no programming syntax in markup languages. Tags turn a simple text file into something a computer can manipulate or a human can understand.

HTML

Hypertext Markup Language (HTML) is the markup language used to structure text on web pages. Note that HTML provides structure to web pages, but not styling. For example, HTML marks what parts of the page text are the title, headings, paragraphs, etc., so that they can be rendered appropriately by a web browser.

HTML has a predefined set of **tags** that serve different purposes. The code sample below shows the most basic features of an HTML document.

```
<!DOCTYPE html>
<html>
<head>
<title>Page Title</title>
</head>

<body>

<h1>This is a Heading</h1>
<p>This is a paragraph.</p>
```

```
</body>
</html>
```

The **DOCTYPE** tag is the first line in any HTML file, and denotes to the browser that the content is HTML. The rest of the document is contained with the `<html>` tags. In general, HTML documents have two sections: the head and the body. These are contained within the `<head>` and `<body>` tags respectively.

The head usually contains metadata being used by the page. The body contains the main content and information of the web page. The body contains the text, links, images, lists, and other data to present, all tagged appropriate.

Although the structure is tagged, HTML is not meant for styling information. To add aesthetics like fonts and colours, a **cascading style sheet** (CSS) is applied. CSS references the standard HTML tags to apply specific styles to text within those tags. CSS is applied either via a separate CSS file or directly within the HTML document.

The CSS code sample below would style all the text within HTML `<p>` tags, or paragraph tags, with the colour blue.

```
p {
      colour:blue;
}
```

XML

eXtensible Markup Language (XML) is a markup language meant to store and transport data. XML is both machine and human readable. XML is usually used to send structured data within a web-based system.

Below is an example of XML code.

```
<?xml version="1.0" encoding="UTF-8"?>
<note>
      <to>John</to>
      <from>Jane</from>
      <heading>Reminder</heading>
      <body>Don't forget the dogs.</body>
</note>
```

In this example, XML is used to markup a note for someone named John, from someone named Jane, with a heading, "Reminder". The body text reads "Don't forget the dogs."

XML schemas can be defined for the valid tags and their appropriate structure for an XML document.

JSON

JavaScript Object Notation (JSON) is a format used to store and transport data. It is designed to be both machine and human readable. JSON offers many benefits, because JSON can be easily converted to JavaScript objects, and vice-versa. JavaScript is an interpreted programming language commonly supported in modern web browsers.

This ability to easily convert to JavaScript objects and vice-versa makes JSON a popular format when transferring data between web browsers and servers, as well as for passing data around in web applications.

Below is an example of a JSON object.

```
{
      "firstName":"John",
      "lastName":"Doe",
      "Age":"15",
      "favouriteColour":"Red"
}
```

JSON data is written as name/value pairs with JSON objects written inside curly braces.

JSON data can also have arrays of JSON objects. Arrays in JSON are written inside square brackets. The example below lists three JSON objects of "students", with three different people and their properties.

```
{
      "students":[
            {"firstName":"John",
             "lastName":"Smith",
             "Age":"15"
             "favouriteColour":"Red"
            },
            {"firstName":"Donna",
             "lastName":"Doe",
             "Age":"14"
             "favouriteColour":"Green"
            },
            {"firstName":"Thomas",
             "lastName":"Oliver",
             "Age":"12"
             "favouriteColour":"Purple"
```

```
            },
        ]
}
```

HTTP

Hypertext Transfer Protocol (HTTP) is a protocol that dictates how information, including hypertexts, is transferred across the Internet.

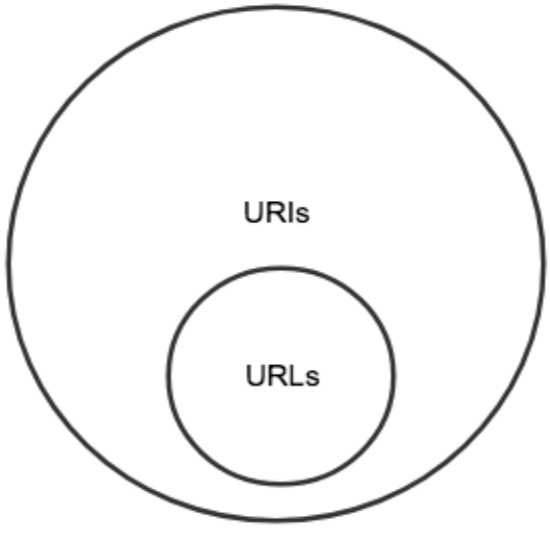
Hypertext is a document embedded with **hyperlinks**, which when clicked, will take you to the intended document or resource. Hyperlinks were originally used to link HTML documents. HTTP was designed to facilitate the use of hypertext, and to support the communication of documents and resources expressed in HTML.

Over time, hyperlinks have been used to link together multimedia resources, such as images, videos, gifs, text, and audio, or documents containing any combination of these. Resources can be static, such as HTML documents, images, or videos, or dynamic, such as programs that produce some output when they are called.

This lesson will focus on HTTP 1.1, although several protocol versions exist.

URIs and URLs

Universal Resource Identifiers (URIs) are addresses used to identify resources. **Universal Resource Locators (URLs)** are a subset of URIs, that are used to locate resources. Both identify the resource, but URLs also tell the protocol how to locate and access the resource. URLs provide the protocol and domain name or IP address of the machine the resource is stored on, and the location of the resource on the machine. All URLs are URIs, but not all URIs are URLs.



URI can be understood through website addresses. Consider the example below.

http://example.com/user/favouriteitems/widebrimsunhat/sunhat.png

This example can be decomposed as below:

Protocol	http
Hostname	example.com
Location of the resource on the host machine	/user/favouriteitems/widebrimsunhat/
Resource	sunhat.png

Instructor's Note: The URI does not need to explicitly provide the IP address. The browser is able to resolve the IP address corresponding to the hostname provided in the URI. The browser will either already know the IP address corresponding to the hostname, or if it doesn't know, it will query a Domain Name System (DNS) server to find out.

TCP

HTTP is built upon a client/server design. To accomplish this, HTTP is built on top of another protocol known as the Transmission Control Protocol (TCP). When a client makes a request to a server, this opens a TCP connection between the client and server allowing for communication. Messages are sent and received through TCP ports. The client/server

relationship exists between a web browser and a web server.

HTTP relies on TCP connections as they allow for reliable, ordered, connection oriented communication. When a browser accesses a URI that starts with "http", a connection between the web browser and a web server is opened on TCP port 80. This port is the default for HTTP messages.

> Instructor's Note: Occasionally, URIs include a port number. This would look as below:
>
> http://**example.com:80**/user/favouriteitems/widebrimsunhat/sunhat.png
>
> This is almost always not used, as production systems typically use the default TCP ports.

HTTP Requests and HTTP Responses

A client **request** consists of a request-line, headers, a blank line, and sometimes a message body.

Element	Description
Request-Line	The request-line includes the request method, request URI and protocol. The URI may end in a query string to specify parameter data for the request, although this is optional. This string is separated from the path of the resource by a question mark.
Headers	Client requests may have a various number of headers, of different kinds. Headers may be mandatory or optional. Mandatory headers allow for the request to be processed and optional headers can be used to give context to the request. Two mandatory headers in any request are the host header, which contains the domain name or IP address of the host, and the accept header, which informs the server what kinds of content the client will accept as a response. Further, if a message body is present, then a content-length header that indicates the size of the body in bytes and a content-type header that indicates the type of the body must be included.

A Blank Line	A blank line follows headers. If no message body is required for the request, then the request ends here.
Message Body	This section contains the message body, if it is required for the request. Message bodies might be HTML documents, JSON, encoded parameters, or other similar content.

Server Response

A server **response** consists of a status-line, headers, a blank-line, and sometimes, a message body.

Element	Description
Status-Line	The request-line includes the protocol version and the HTTP status code. The HTTP status code informs the client of the status of the request. There are many possible HTTP status codes. If the request has been successfully processed, the status code of "200 OK" will likely show.
Headers	As with client requests, server responses may have a various number of headers, of different kinds. Headers may be mandatory or optional. Many optional headers exist and can be used to provide more information and context about the communication. If a message body is present, then a content-length header and a content-type header must be included.
A Blank Line	A blank line follows headers. If no message body is required for the response, then the response ends here.
Message Body	This section contains the message body, if it is required for the response. Message bodies might be HTML documents, JSON, encoded parameters, or other similar content, as with HTTP requests.

Instructor's Note: Numerous possible headers for HTTP requests and responses exist. These optional headers can be used to provide more information and context about the communication.

Encoding

HTTP limits the characters used in URIs, request queries and request bodies to be ASCII. Special or unsafe characters, like space or Unicode, require the encoding of these characters. An example of an unsafe character is a "space".

Unsafe characters are often replaced with a "%" sign, followed by their two-digit hexadecimal digit encoding. For example, a space can be encoded with "%20", or with a "+" sign. The phrase "software design and architecture" can thus be encoded either of the following ways:

- software%20design%20and%20architecture
- software+design+and+architecture

Query strings can also be encoded, using the "=" sign. For example:

- colour=red
- height=very+tall

The ampersand symbol, "&", is used to join all the parameter value pairs. For example:

- colour=red&height=very+tall

GET Method

This lesson has reviewed HTTP requests. Each HTTP request must indicate a request method in its request-line. Request methods are used to indicate to the web server what it should do with the request. The most common request methods are: GET, POST, and PUT.

The **GET method** retrieves the resource given by the URI provided in the request-line. Get methods are used to retrieve web pages, images, or outputs of programs. A query string may be sent along with the request in the URI. Queries always start with a question mark and must be url-encoded. No message body is sent along with a GET request.

Consider an example, where you are reviewing a campsite reservation made on camping.com. At the time of booking, the reservation number 17021 is issued. To review the booking, the url-encoded query string

[?number=17021] must be sent to camping.com/reservation.

The GET request and response would look as below:

Request
GET /reservation?number=17021 HTTP/1.1
Host: camping.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X *x.y*; rv:10.0)
Gecko/20100101 Firefox/10.0
Accept: */*

Response
HTTP/1.1 200 OK
Date: Wed, 17 May 2017 19:15:56 GMT
Content-Length: 10571
Content-Type: text/html; charset=utf-8
Last-Modified: Wed, 17 May 2017 19:12:21 GMT

```
<!DOCTYPE html>
<html lang="en">
…
</html>
```

When the response is received, the web browser will render the HTML document, and the reservation will be reflected on the screen.

POST Method

The **POST method** is another example of a request method. The POST method is used to add or modify a resource according to the message body of the request, on the host specified in the URI of the request. The message body contains the information used to create or update a resource on the website.

POST requests are often used by HTML forms to submit data. If the data in the message body is from an HTML form, then like a GET query, it is url-encoded.

Consider an extrapolation of the previous example. Your family would like to come on the camping trip. You offer to book another campsite for them. As you are creating a new reservation on the website, the POST method will be used for the request. The booking information would be for Mary Teller, whose email is mtetter@gmail.com. Let us assume the reservation dates is 01/07/17, and the campsite is number 35. Note that this information is encoded in the body of the request.

Request

POST /reservation/new/ HTTP/1.1
Host: camping.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X *x.y*; rv:10.0)
Gecko/20100101 Firefox/10.0
Accept: */*
Content-Length: 72
Content-Type: application/x-www-form-urlencoded

name=mary+teller&email=mtetter%40gmail.com&date=01%2F07%2F17&
campsite=35

Response
HTTP/1.1 200 OK
Date: Wed, 17 May 2017 19:15:56 GMT
Content-Length: 1305
Content-Type: text/html; charset=utf-8

```
<!DOCTYPE html>
<html lang="en">
…
 <body>
      <p>Your reservation of campsite 35, for
      01/07/17 was successful! Your reservation
      number is 12231. </p>
…
 </body>
</html>
```

Once the request has been received and processed, the web server sends a response. When the web browser receives the response, it renders the HTML document it received in the message body. This allows the booking confirmation to be viewed on the computer screen.

PUT Method

The **PUT method** is another example of a request method. The PUT method takes the information provided in the body of the request, and creates or updates a resource at the location specified in the URI of the request. The URI specified in the request dictates to the server the identity and location of the enclosed resource.

The PUT method can be used to create or update a resource, like the POST method. The information contained in the request body of a POST method, however, is created or updated under an identity and location determined by the web server, which is not necessarily the identity and location in the supplied URI.

Consider the example used in the previous two sections of this lesson. Imagine that the camping website allows users to add notes to a reservation, for the staff to receive. Imagine you would like to have a campfire during the trip, but you do not have an axe or wood. The reservation can be accessed with the reservation number. To add a note, a PUT request needs to be made to that location. The note will be a JSON object and delivered in the body of the request.

Request
PUT /reservation/17021/ HTTP/1.1
Host: camping.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X *x.y*; rv:10.0)
Gecko/20100101 Firefox/10.0
Accept: */*
Content-Length: 46
Content-Type: application/json

{"note":"We will require firewood and an axe."}

Response
HTTP/1.1 200 OK
Date: Wed, 17 May 2017 19:15:56 GMT
Content-Length: 890
Content-Type: text/html; charset=utf-8

```
<!DOCTYPE html>
<html lang="en">
…
 <body>
      <p>Your note has been added successfully to
      your reservation.</p>
…
 </body>
</html>
```

When the web server processes the PUT request, it sends an HTML document in its message body within its response. When the browser receives the response, it renders the document so that it can be viewed on the computer screen.

HTTP Statelessness

HTTP can also be stateless. This means that the relationship between requests is not preserved.

For example, if a user were browsing an online shopping website, and clicking different items, the HTTP protocol does not keep track of which

items have been previously clicked. Every time a new item is clicked, a new request is sent, but the protocol is unaware if the same client is making the request.

HTTP cookies can be used by websites to track the behaviour of users on the site. When a client makes a request, the site gives an HTTP cookie to store information about the user's browsing session. The cookie is stored by the client, and updated by the server each time the client makes a request to the server. This allows the server to store state information about interactions with a particular client, which can be useful for tracking purposes.

HTTP is fundamental for the web, as it dictates how data is communicated and exchanged. This makes HTTP very important for invoking and accessing web services, and enabling a service-oriented architecture.

Javascript

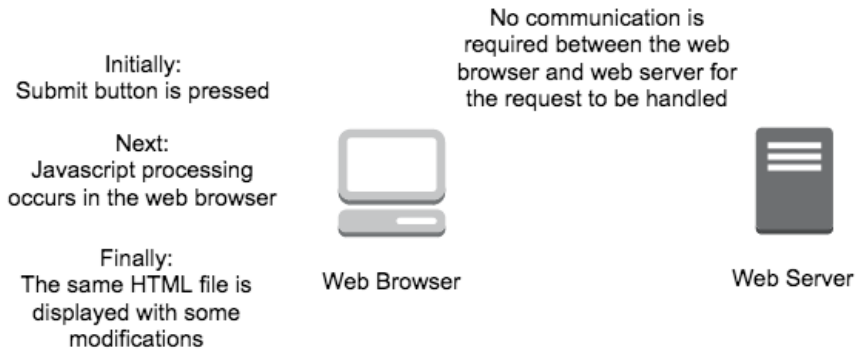
Javascript is a programming language that can be used for a variety of purposes. This lesson will focus on how Javascript can be embedded into HTML documents to make web pages interactive. Javascript is able to modify elements, attributes, styles, and content within the HTML document. Javascript can be embedded between "script" tags in HTML (`<script></script>`).

Javascript is able to provide interactive elements to web pages, because it is an **interpreted language**. This means that Javascript code is interpreted by a web browser at run time. In other words, embedded Javascript is a series of commands that are executed by the web browser when it loads the HTML document.

It is possible for web pages to provide some interaction to users without Javascript, through the use of HTML forms that submit POST or GET requests. The web server responds to the request by providing the web browser with a new HTML document. The result of the interaction will be visible only after the browser has received and loaded the new HTML document.

However, interactions provided to users on a web page embedded with Javascript tend to be more efficient and more usable. With Javascript, a form can be partly checked and processed on the client-side. This means that the browser does not have to wait for the web server to provide a new page, instead, the Javascript on the page can dynamically change the HTML web page that is already loaded in the browser as it runs the client-side in the browser. This offloads some of the processing required to operate this application so that not everything needs to be processed

server-side.



Some interactions with a web page, even with Javascript embedded within, will still require contact with a web server, but it is not always necessary.

Javascript is able to modify elements on a web page by making use of the **HTML Document Object Model** (**DOM)**. When a page is loaded in a web browser, the HTML document becomes a document object. This object can be used by Javascript to obtain and modify the elements and content on the web page. As a result of processing the document object, the content, structure, and style of an HTML document can be modified.

For example, Javascript can be used to modify elements on a web page, such as providing the ability to make image thumbnails grow in size when clicked, and shrunk back down when clicked again, or to hide and reveal text, as for spoilers on text-based web pages.

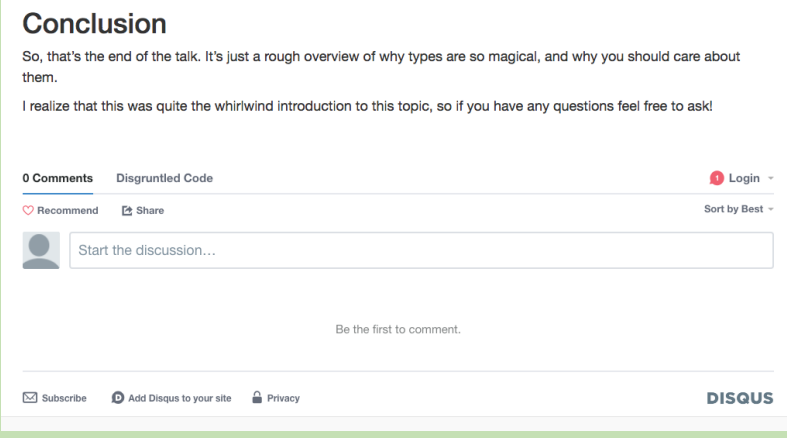
It is not necessary to know Javascript well in order to use it. Pre-made scripts are often available online, from a trusted source. These scripts can be used as is or with small modifications they can be applied to suit a web page's needs.

Javascript embedded into HTML can be a great way to provide users with access to services from a web page.

DID YOU KNOW?

The comment management system Disqus allows users to comment on web pages and administrators to moderate those comments. Disqus is an example of a pre-made Javascript that can be added to a web page through copy and paste into an HTML document. To access Disqus, users are required to sign up for the service.

Links to Disqus can be found in the Course Resources.



Distributed Systems Basics

In recent years, the rapid growth of the Internet and the falling costs of cloud services have given rise to innovative web services, such as Google Docs, PayPal, and Amazon. Private networks now facilitate client-server communications, even for internal company services.

This has changed the way we work – clients and servers often operate in heterogenous environments, where the machine architecture or operating system may be different between a client and server.

Modern client machines are not as powerful as their server counterparts. Client machines are designed to address a different set of concerns than servers in heterogenous environments. For example, client machines may be designed for the experience of a single user, and may be focused on providing exceptional usability. Their operating systems are therefore designed to be intuitive to use and to learn. A server, on the other hand, may need to provide computing power to multiple, concurrent users.

Servers are typically used by system administrators, or other information technology professionals. Their user interfaces may require more expertise to use.

Since machine and operating environments are different between clients and servers, they communicate with the help of **middleware**.

Middleware

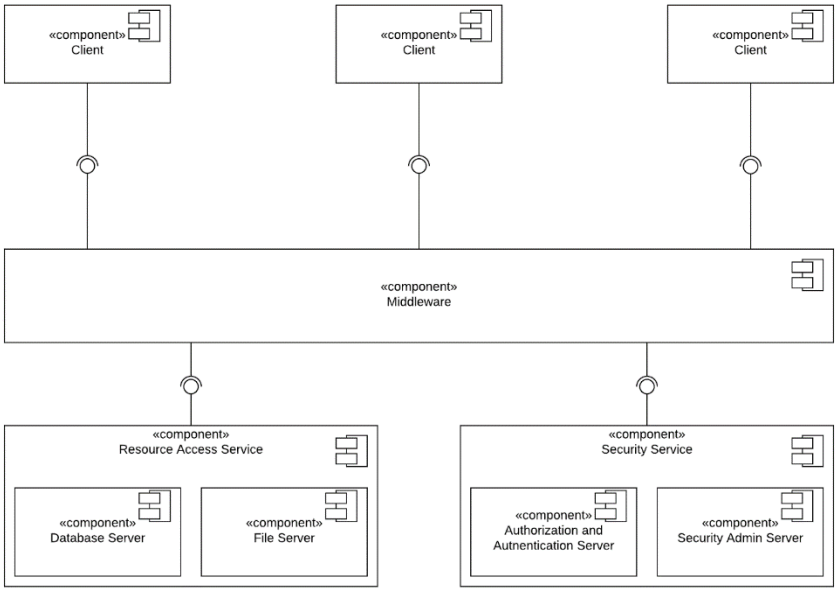
Middleware is a type of architecture used to facilitate communications of services available and requests for these services between two applications that are operating on environmentally different systems.

Network connectivity is an expected mode of operation for modern systems, which rarely work in isolation. New software systems are designed to be able to communicate with other systems over a network. Legacy systems are commonly updated, redesigned, or rewritten in order to be able to utilize network connectivity.

Computer networks have enabled the growth of **distributed computing**. Distributed computing is a system architecture in which computers on a network are able to communicate and coordinate their actions by passing messages through a network.

Distributed computing has enabled developers to design tiered architectures with each layer focusing on specific aspects of the overall systems. Clients and servers can be designed to be specialized, and software and hardware developers are able to create software environments and machine architectures that serve to enhance these specialized characteristics.

Increased specialization runs the risk of impeding communication between client and server. To solve this, middleware provides a common interface to clients and servers in heterogeneous environments. Middleware facilitates communication on a large scale by providing a common interface between entire distributed systems.



Middleware allows developers to access functionalities of a system, without having to implement an entire tier of subsystems in their architecture.

There is a need for middleware to become more sophisticated, or for existing middleware to be extensible, as modern systems become more and more complex. As systems move towards n-tiered architectures, middleware needs to be able to encapsulate more functionalities, from business logic and distribution of client requests to being involved in handling authentication and authorization.

RPC

One example of middleware is **remote procedure call** (**RPC)**. RPC is the basis for middleware systems used for certain web services. RPC allows clients to invoke procedures that are implemented on a server.

In RPC, the client and server are either:

1. on completely separate machines
2. are a different virtual instance on the same machine

In the first case, the physical address space between client and server is clearly different, so the client does not know the physical memory address of the procedure that it wants to call in the server since they do not share the same physical memory.

In the second case, the "virtual" address is shared between client and server. It is up to the operating system to manage each individual virtual instance, and find the correct virtual address for the procedure being invoked.

In either case, the client cannot directly access the procedure being called. RPC facilitates the call between client and server.

History of RPC

Remote procedure calls were developed and introduced by Birell and Nielson in the 1980s. They were created to provide a transparent method of calling procedures that were located on a different machine.

In their initial design, RPCs consisted of three primary components:

1. A client, which is the caller. It is the component making the remote call.
2. A server, which is the callee. It is the component that implements the procedure that is being invoked.
3. An interface definition language (IDL), which is the language through which the client and server communicate.

RPC could also include name and directory services, and binding methods to allow clients to connect to various servers. At this starting point, RPCs were a simple collection of libraries that developers could include in their applications. These libraries contained all the functionalities that were required in order for systems to make remote procedure calls. Eventually, RPCs evolved to become cornerstones for middleware-based architecture.

RPCs became successful because they did not require developers to learn a new language or programming paradigm, but instead used the familiar concept of procedures. Distributed systems could be designed and implemented more efficiently.

RPCs are currently used in many different configurations – they can be stored procedure calls in a database system, or in XML messages for web services.

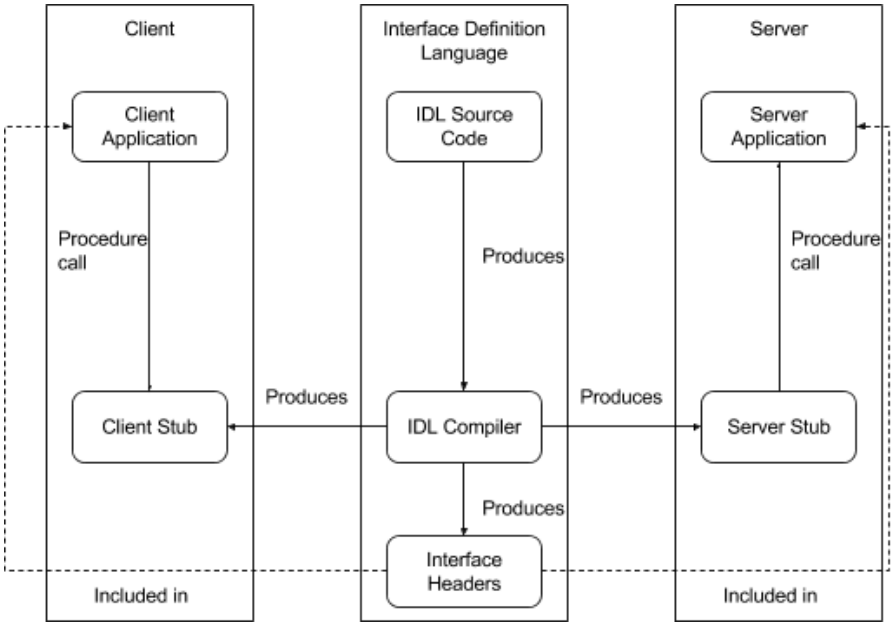
Basics of RPC

The client, the server, and the interface language definition (IDL) are the three primary components of an RPC.

The IDL is the first component implemented. It defines what procedures on

the server are available to the client. It also describes the input parameters, as well as the returned response. In other words, the IDL is the specification for remote procedure calls. It tells the client what remote services are available, how they are accessed, and what the server will respond with.

Once the IDL is compiled, the client and server stubs are produced. They perform the "heavy lifting" with interface headers.



The client stub in particular acts as a proxy for the procedure call. It is responsible for:

- Establishing the connection with the server through a process known as binding
- Formatting the data to a standardized message structure, such as XML
- Sending the remote procedure call
- Receiving the server stub's response

Server Stub

The server stub in an RPC receives the call, and invokes the desired procedure. The server stub also contains the code for receiving the remote call, translates the standardized message into a data format the server recognizes, and sends the server's response back to the client stub.

Stubs are compiled and linked directly to the client or server component. When a client makes a remote procedure call, the invocation will act like it is a local procedure call to the client because the client stub is in the same

address space. In order to create client components in environments that are different than the server, it is possible to create more complex server stubs. For example, complex stubs can allow client components operating on a Windows machine to communicate with server components that are running on Unix.

IDL allow stubs to be generated because the IDL maps the concrete programming languages to the intermediate representation in the stub. This is preferable to manual development of client-server stub pairs that can communicate with each other, as there are so many programming languages and operating platforms.

Interface Headers

The interface headers are a collection of code templates and references that are used to define what procedures are available at compile time. These files are used in the development of the client and server components.

This table summarizes the basic types of interfaces provided to make a remote procedure call.

Type of Interfaces	Description
Procedure registration	Tells the client what procedures are remotely accessible on the server.
Procedure call	The actual procedure that is being invoked on the server.
Procedure call by broadcast	The same thing as a procedure call but these procedures are invoked by broadcast.

The stubs abstract all the networking details, which can be kept hidden from the developer so that they don't have to worry about them. RPC does allow configuring of connection settings if the developer wants more control over the details of the client-server connection.

How RPCs are Performed

There are seven steps to performing RPCs.

No.	Step	Description

1	Client component invokes the procedure.	The client component makes the procedure call, and passes the arguments to the client stub. Since the client stub is linked to the client, no network connection needs to be made. This step is exactly like a standard procedure call.
2	Client stub marshalls the parameters.	The client stub converts the parameters to a standardized message format and copies them into the message through a process called "marshalling". The parameter data is transformed into a format that is suitable for communicating with another application. The format used is determined by the IDL that is used to compile the stub.

3	The message is sent to the server.	The client stub sends the message to the server using the **binding** information that it is given. Binding is the process in a client that connects to a server. It can be done statically or dynamically. Static binding uses hard coded binding information. With static binding, if the server goes offline, then clients will not be able to establish a connection. Static binding also does not allow for server redundancy, since clients will always connect to a specific one. Dynamic binding is more complex and adds another layer to your tiered architecture. This added layer is referred to as the name and directory server. It is responsible for keeping track of which servers have been bound, and balancing the load between all servers. This layer can also keep track of servers and change the binding information if the servers change. The client stub is responsible for the static binding or communicating with the dynamic binding layer, so the developer of the client component does not need to worry about binding.

4	The server stub receives and unmarshalls the message.	The message is received by the server stub. Since the arguments have been marshalled, it must be translated back to a format that is usable for the server-side procedure call. This is done in a process called "unmarshalling". Once the arguments have been converted back to the proper format, the server stub will invoke and pass the arguments to the procedure in the server component. Just like on the client side, this procedure call looks like a normal procedure call to the component, because the server stub is linked to the server-side component.
5	The server component executes the procedure and returns the result to the server stub.	Once the procedure finishes execution, the server component will return the results back to the server stub. In order for the results to be returned back to the client, the server stub needs to marshall the results into a standardized format.
6	The results are returned back to the client stub.	The server stub does not need to bind to the client because the connection is already established by the client. All the server stub needs to do is return the message back through the connection.
7	The client stub receives and processes the results.	The client stub unmarshalls the results in the message, and then returns it to the client component. The client can now close the connection to the server.

Synchronous RPCs

Originally, RPCs were designed to be synchronous. During a remote procedure call, the client component pauses its execution while it waits for a response. This is also known as **blocking**, since the client cannot perform any other task until the server returns a result.

Waiting for a response can introduce a number of issues:

- If the server never returns a response, the client can end up waiting

indefinitely.

- It may not be clear how long to wait for a server to response. Some procedures may take longer than others, or the server may take longer to create a response under a heavy load.
- The RPC may need to be re-transmitted, and this can be difficult to determine.

Asynchronous RPCs

Modern systems need to be able to handle remote procedure calls in an asynchronous manner. Asynchronous systems are considered **non-blocking**, because clients do not need to wait for a server response before moving onto another task. This allows components of a distributed system to work independently.

Systems are also able to perform different tasks in parallel with each other because they do not need to wait for one task to end before starting another.

Asynchronous behaviour adds more complexity to a system, because how the system allocates resources for various pending tasks needs to be managed. Note that overloading a system with asynchronous tasks can also reduce the system's overall performance.

Object Brokers

Systems and their standards must evolve as new programming paradigms are introduced. However, it is expensive to design and create new systems from scratch, particularly as the design of older systems can still be applicable and relevant. In some cases, it may be more efficient to update existing architectures using the new paradigms.

Remote procedure calls evolved in this manner. Instead of being relegated to be a legacy system, they have been used as the basis for a middleware architecture based on **object brokers**.

Object brokers combine the distributed computing aspects of remote procedure calls with object-oriented design principles. By including object-oriented programming, object brokers simplify and allow distributed systems to use an object-oriented approach.

History of Object Brokers

As explored in the first course of this specialization, the object-oriented paradigm introduced object-oriented programming to the world in the late 1970s.

Object-oriented programming simplified data abstraction by allowing programmers to define an abstract data type using a class. Data encapsulated by a class are used to define that class' properties, and are called attributes. Procedures that a class is able to perform are what describe a class' behaviour, and are called methods.

Non-static classes are constructed into object through a process known as instantiation. This allows programs to dynamically create new instances of objects at run time. Additionally, a class can have multiple, unique instances existing at the same time.

Object-oriented programming also introduced two concepts that changed the way methods are identified: inheritance and polymorphism. These allow different classes to have different implementations of the same method signature. In previous programming paradigms, the signature of a procedure in a program was unique, which means that only a single implementation was possible.

Allowing for different implementations of the same method signature caused issues for middleware. Middleware provides services for each object, and so middleware had to be able to take inheritance and polymorphism into account. Remote procedure calls therefore naturally evolved towards object brokers. Beginning in the 1990s, middleware began to be able to address handling object-oriented programming.

CORBA

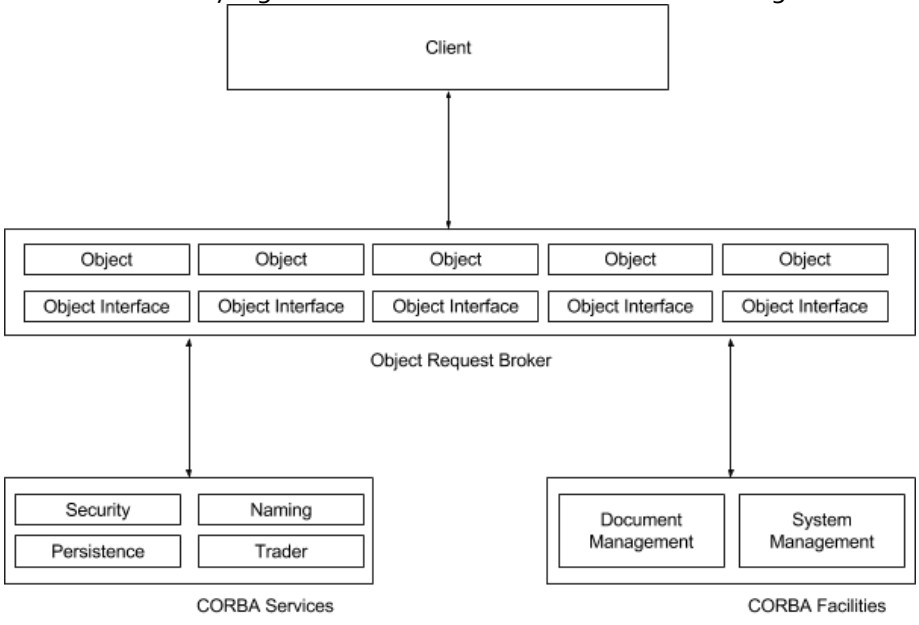
The most common object broker architecture is the **Common Object Request Broker Architecture (CORBA)**, although it can be more closely compared to a set of standards. CORBA provides an outline of what should be included in object brokers, although it is not a step-by-step guide for implementation. As numerous object-oriented languages exist, the primary goal of CORBA is to create a specification that allows object brokers to be independent of the programming languages used to implement clients and servers. Client-side and server-side operating systems are different because quality requirements are different.

CORBA ensures that developers are not restricted to a language or operating system (OS) requirements if middleware is required for a system. Similar to procedures in an architecture based on RPC, objects can be distributed among a network of computers, which allows them to reside in different physical address spaces. Alternately, objects can be distributed throughout different processes on the same computer, which lets them exist in different virtual address spaces.

CORBA allowed for the foundation of Microsoft's **.NET** framework and the **Java 2 Enterprise Edition** framework offered by various vendors.

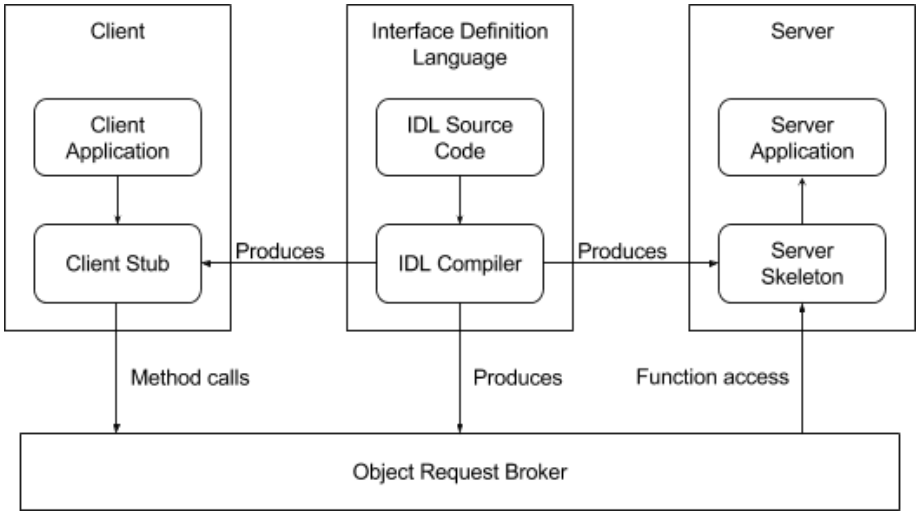
CORBA contains three main components:

1. The **object request broker,** which provides object interoperability to the client and the server. An object must declare its interfaces before it can be accessed by a client or server through the broker. The broker is responsible for marshalling and unmarshalling all the method calls that it receives. This allows the client or server to be modified without affecting the other.
2. The **CORBA services** are services provided by the middleware "to" the objects being used by the client and server. Services provided vary and may include functions needed for persistence to functions related to object security. All CORBA services are accessible through the CORBA standardized application programming interface (API).
3. The **CORBA facilities** provide servers at an application level. These are usually high-level functions such as document management.



As object brokers are an extension of the RPC architecture, the architectural design of using an IDL, can be used to generate the stubs used to facilitate communications within the system. For more information on IDL and stubs, please return to the RPC section of this lesson.

This lesson will focus on the enhancements that object brokers have added to IDL and stubs.



In CORBA, IDL is enhanced over IDL in an RPC-based system, because the IDL is capable of supporting inheritance and polymorphism. All object interfaces must be declared in the interface definition language in order for them to be presented to the client.

The CORBA IDL also has a standardized mapping to multiple object-oriented languages, which allows clients and servers to be developed in different programming languages. Standardization also allows the IDL to be portable across brokers by different vendors.

The CORBA IDL compiler is responsible for constructing the client stub and server skeleton. The client stub, and server skeleton have similar functionality to their RPC counterparts. However, as CORBA deals with objects, the stub and skeleton need to be built upon in order to be able to handle objects.

The client stub acts as a proxy object. It hides all the objects that are distributed and being served by the brokers. Since the objects are hidden and the stub is linked to the client application, all method calls to the objects being served appear as if they are local invocations. Further, as the client accesses the remote methods through the stub, the stub must also contain the method declarations of all the objects distributed throughout the brokers.

The server skeleton hides the object distribution from the server application. The skeleton acts as a proxy object, since the remote objects are making functions call to the server. This makes it appear as if the calls are coming from a local object.

This architectural design means that the client needs to know the server's IDL in order for the two to communicate with each other. Without this

knowledge, your client will not be able to access the methods that are being served by the object broker.

Static and Dynamic Interoperability

Interoperability can only be achieved if the client binds to at least one broker. Without binding, the client cannot access the behaviours of the objects.

Binding can occur statically or dynamically.

- Static binding occurs when the client stub is created. The service that is used for static binding to a broker is handled by the IDL. When the IDL compiler generates the stub, it will be statically bound to the broker that the IDL compiler belongs to.
- Dynamic binding is also specified in CORBA. It allows clients to dynamically search for new objects, retrieve their interfaces, and instantiate the discovered objects.

In dynamic binding, the IDL compiler does not need to generate a stub, but requires two additional components instead.

1. The **interface repository,** which is used to store the IDL definitions for "all" the objects being served by a broker.
2. The **dynamic invocation interface,** which provides all the necessary operations that a client needs for browsing the interface repository, and dynamically constructing methods based on what the client discovers. Object references are provided by two services in the dynamic invocation interface component, allowing for two ways for clients to search for and discover new methods.
    a. The naming service allows the client to retrieve objects using the "name" of the interface that the client needs.
    b. The trading service lets the client search for objects based on object attributes.

Building dynamic invocation interfaces is semantically difficult. It is key to put effort into building the logic for the component. The searching semantic is difficult to implement – the client needs to understand the meaning of each service property in order to properly search the repository for services.

A naming convention therefore needs to be implemented so that both the client and the dynamic invocation interface can understand each other. For example, the term "get" can have synonyms such as fetch, retrieve, and obtain. The client and the dynamic invocation interface needs to be able to identify that all those words mean the same thing.

The client also needs to know how to interact with the services that are discovered, or it will have a hard time determining what the behaviour of a newly discovered service is, what the parameters mean, or in what order services need to be invoked so that the desired effect is achieved.

The disadvantages of implementing semantic rules for dynamic invocation interface generally outweighs the advantages. This is especially true in large and robust object brokers because there are more semantic ties to discover and implement.

Benefits of CORBA

Using CORBA as a middleware system presents many benefits. These include:

- Being able to handle distributed computing in an object-oriented paradigm.
- Objects are essentially independent of the physical and virtual address, as they can move around but still be referenced. The broker keeps track of and maintains both the references made to the objects and the actual address of the objects, while a client only needs to make references to the objects. If an object moves, the client will still have access to it. This benefit allows development of clients and servers to occur without restrictions from programming languages and operating systems.
- CORBA provides a variety of options in regards to data. Data can be strongly or dynamically typed. It can also be marshalled into binary form or compressed in order to reduce the size of the data that is sent.
- CORBA provides standards for optimization by allowing developers to manage threads and network connectivity settings.

Criticisms of CORBA

As with any architectural standards, CORBA has disadvantages. Most of the problems that arise from use of CORBA are not related to the standards themselves, but instead result from poor implementation of those standards.

- Any system that does not implement the CORBA standards correctly will suffer. Issues will usually occur in the broker of the system, which can lead to complex and incoherent API, difficult to use object brokers, and overall poor performance. Poor implementation can occur because of inexperience in development, or the need to reduce cost.

- Although CORBA is advantageous in that it provides objects that are not restricted to a physical or virtual address, this design also causes **location transparency**, which means that even if objects that interact exist in the same physical or virtual address space, they are still treated as if they were in different spaces. Because of this, method calls are always made in the most complex way possible regardless of where objects are located.

Module 2: Web Service

Upon completion of this module, you will be able to:

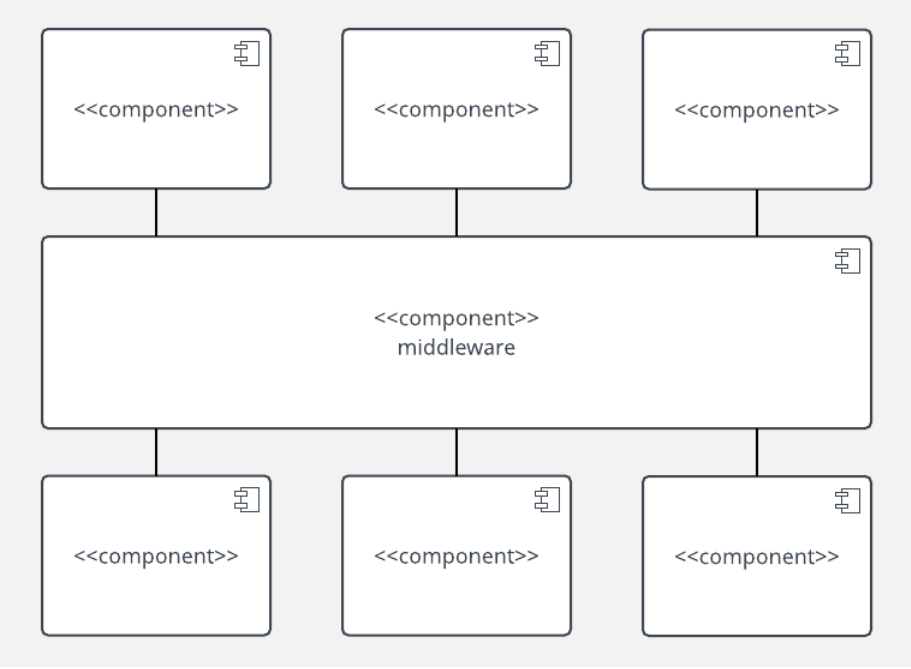
(a) Define a "first generation" web service
(b) Define what a web service is
(c) Explain how to use a web service
(d) Describe the standards for how web services are invoked, described, published, discovered, and composed

Introduction to Web Services

In order for services to be able to be used by other processes, there must be some way of "exposing" the service. In other words, services have interfaces that can be used by some service requester. It follows then that a web service is exposed and accessible using web technologies.

Before web services became commonly used, building integrated systems was a difficult task. Many different pieces of software might be involved, with even more combinations of connections between those software, which is work-intensive to implement, maintain, and extend.

Enterprise Application Integration (EAI) is an enterprise-level solution for integration problems in integrated systems. The exact architecture for EAI varies, but it does use middleware. In the previous module, we learned that middleware is software that is located between other software and facilitates interactions between them. Middleware helps manage complexity in an organization.



Rather than deal with multiple combinations of connections, middleware ensures that a system has fewer interfaces to implement and maintain. However, implementation in business to business (B2B) interactions, where interactions are between business instead of within them, is not always clear. For example, identifying which business implements the middleware, managing security, and protecting data from outside influence can complicate B2B implementation of EAI. Consequently, EAI is not used for B2B interactions.

To solve these issues, web services are implemented for interactions between businesses. Web services are usually implemented with a specific set of standards and protocols for implementing services over the web. Web services are defined by the World Wide Web Consortium (W3C) as "a software system designed to support interoperable machine-to-machine interaction over a network".

The table below summarizes the technologies and standards that make up web services.

Technology or Standard	Description

Web Infrastructure	Web services are built on top of web infrastructures. They start with TCP, the networking protocol responsible for reliable, ordered, connection-oriented communication. On top of that is HTTP, which web services use to send information and interact with their clients. Other transfer protocols exist as well as HTTP. HTTP however is compatible with nearly every machine and provides a strong foundation for platform independence and interoperability. HTTP is the basis for web services such as RESTful web services, which will be discussed later in this course.
Invoking (SOAP)	In order for a service requester to use a particular service, it must invoke it. Invoking is like a method call in an object-oriented language, except that it must be done through a request in XML. Invocation will include which operation is requested along with the parameters and data. In web services, invocation is done with SOAP, a protocol specification that is based on XML and allows services to send information to one another. Service requesters and service providers use these SOAP messages to send each other information. Since this is done through XML, systems coded in different languages and on different platforms can easily communicate.

Describing (WSDL)	Services must know how to interact in order for interaction to take place. **Web Service Description Language (WSDL)** is the standard protocol for describing the interface of a service. WSDL are written in XML. A WSDL description will describe the interface of a service in a machine-readable fashion, so that a service requester can bind itself to this interface. Binding is the act of generating the necessary code to interact with a service so that a service requester can begin invoking it. If a service interface is described unambiguously with WSDL, binding can be done by generating the necessary code automatically.
Publishing and Discovery (UDDI)	Service requesters and service providers need ways to come into contact. **Universal Description, Discovery and Integration (UDDI)** is used by service providers to publish descriptions of their services. Service requesters looking for a service can search by the WSDL descriptions or other aspects of the service. This is called **discovery**.
Composition (WS-BPEL)	Various standards can be built on the foundational standards of SOAP, WSDL, and UDDI for web services. These standards usually have the prefix WS, such as WS-Security for adding security functions, or WS-Coordination for coordinating the activities of many services. WS-BPEL is one of these standards, for Business Process Execution Language (BPEL). BPEL facilitates **service composition** as it allows developers to combine existing services into new, composite services. Composite services are services built with other services, which can be basic services which do not rely on other services, or can be other composite services.

Together, SOAP, WSDL, and UDDI are the three foundational standards of web services. They allow for web services to be invoked by service

requesters, to describe themselves, and to be published to registries where they can be discovered. All of them rely on web infrastructure.

> Instructor's Note: In this module, web services will interact through the use of XML-formatted documents, usually in request-response messaging design.

The standardization of how web services invoke, describe, and publish means that their internal implementation does not matter. Service requesters and services can effectively interact despite being on different platforms and in different languages. However, the commands and parameters of the standards must be supported by the service provider.

The use of web services makes building interfaces easier, because communication between services and service requesters is standardized. Although interactions are pair-wise, they are implemented with web service standards, so applications can be developed in any language and on any platform that supports standard web technology.

Web services can take on the role of middleware, and facilitate interactions between processes for business-to-business interactions. Businesses can control how they interact by choosing how to expose their services. They also do not need to provide a different interface for each outside company.

Service Invocation (SOAP)

Web services use a form of XML messages for communication between service requesters and service providers. The request-response messaging pattern is the basis of all interactions between web services and the software that uses those services.

These XML messages conform to SOAP, a standard developed at Microsoft. SOAP allows a service requester to invoke services. It provides standardization to interactions between service requesters and service providers, or clients and servers. The way that XML defines and organizes requests and responses must be standard so that both parties can understand it.

Standardization therefore allows for interoperability between platforms and languages. SOAP also provides instructions for how the message is bound to the underlying transport protocol.

An SOAP message is meant to solicit an operation from a remote web service, and is in an XML-formatted document. Consider the example below to understand the structure of a SOAP message for a web service

that determines the exchange rate between two currencies.

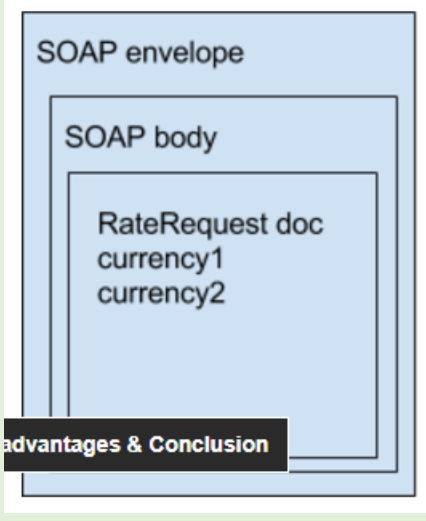
```
<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">
  <soap:Header>
  ...
  </soap:Header>
  <soap:Body>
  ...
  </soap:Body>
</soap:Envelope>
```

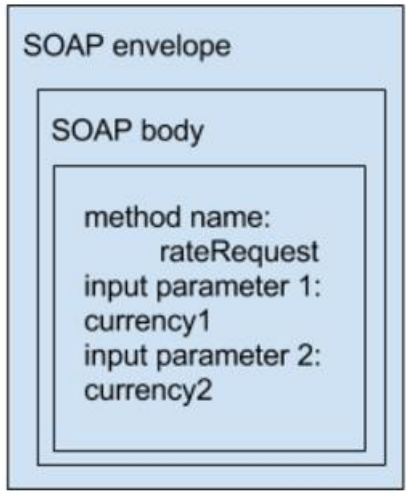
The XML document is identified as a SOAP message by the envelope. In this example, a header is used to also provide contextual information like information about the client or routing information. An envelope is required for SOAP messages, while headers are not.

Additionally, SOAP messages have a body. The body contains the information that the service provider needs to determine which service to provide and the service's input. The body is required.

Two "styles" of SOAP messaging exist. SOAP does not dictate which of the two styles to use. Both styles are commonly used.

Style of SOAP Messaging	Description

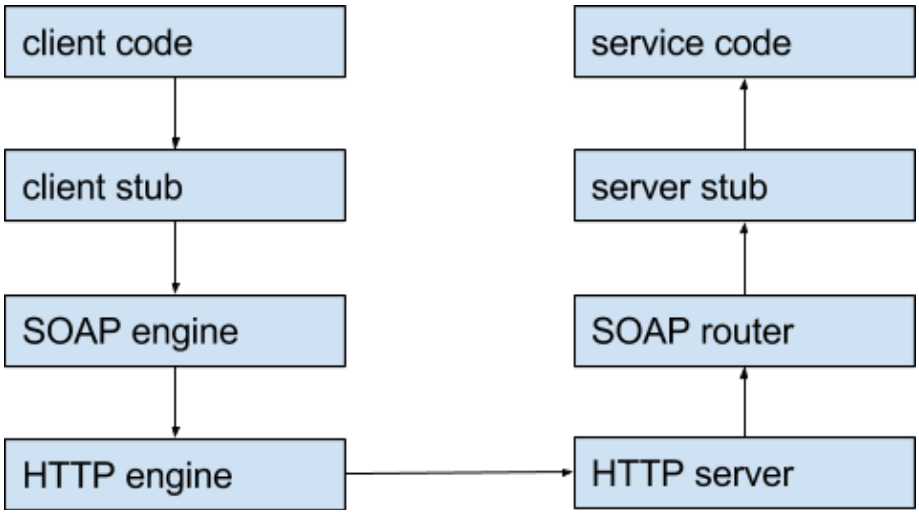
Document Style SOAP	In document style SOAP, a SOAP message is a structured document that contains a request that will be understood by both parties. Below is an example of a document style SOAP message for a currency exchange rate web service. The type of request is determined by the type of document. The two currencies that the service requester wants to compare are fields in the document. 

RPC Style SOAP	In RPC style SOAP, the body of the message is similar to a method-call, that consists of an operation and input parameters. Below is an example of an RPC style SOAP message for a currency exchange rate web service. In this example, there are two input parameters, the two currencies to compare. 

SOAP messages are sent over a transport protocol. For example, Simple Mail Transfer Protocol (SMTP) is used for email. This lesson will focus on using HTTP. SOAP messages are sent using HTTP POST. HTTP determines where to send the request, since this information is not directly included in the SOAP message itself. Since HTTP must acknowledge a POST request, it could return the response, or a simple acknowledgement that the request has been received.

Messaging is **synchronous** if the service request waits for a response before continuing. A program might be left doing nothing while waiting for a response, particularly if the availability or response time of a web service is an issue.

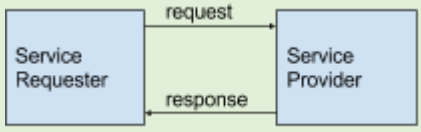
Messaging is **asynchronous** if interactions allow the code to keep executing. This means that when a message returns from the service provider, the code can process it.



A simple implementation of an SOAP request starts when the client code makes a local call to a stub. The stub then converts the request into a SOAP message. This SOAP message is packaged into HTTP and sent to the service provider. When it arrives, the HTTP server passes the content to a SOAP router. The SOAP router determines the appropriate server stub and delivers the message. The server stub uses the information of the SOAP message to make an appropriate method call. Once the service code handles the request, the process works in reverse to send this response.

Messaging Patterns

Four basic messaging patterns exist for SOAP. More complicated messaging patterns exist and are required for meaningful interactions. Since SOAP messages are stateless, these interactions are implemented by relating messages another way, like storing the interaction state on the client and/or the server, or by using extensions to web services like WS-Coordination.

Messaging Pattern	Description
Request-response	In a request-response pattern, the server requester first sends a message, then receives a reply from the service provider.  This process is synchronous and can be implemented over HTTP.

Solicit-response	In a solicit-response pattern, the service provider makes a request to the service requester.  This process is synchronous, and is often a confirmation.
One-way	In a one-way communication pattern, the service requester sends a request to the service provider, but no response is expected. This could be a simple notification that the service requester is up and running.  This process is asynchronous.
Notification	In the notification messaging pattern, the service provider sends a notification to the requester without expecting a response. This model is well-suited to event-based systems where there are publishers and subscribers.  This process is asynchronous.

SOAP messages have certain disadvantages, which fall beyond the scope of this lesson. Some of these disadvantages include the fact that XML encoding and decoding adds overhead and does not easily accommodate some data types. These disadvantages have resulted in SOAP being superseded in many applications by methods that use HTTP more directly, such as RESTful web services.

SOAP and its related web service infrastructure were the basis of the first major consensus on web services. SOAP's neutrality allowed systems on different platforms and in different languages to interact and pass data.

The XML-based structure allowed for machine-readable data that could be manipulated by any machine connected to the internet.

Service Description (WSDL)

In web services, **Web Service Description Language (WSDL)** is a standard used to describe the interface of a web service. This helps SOAP messages find services, and understand how to interact with services, including parameters. WSDL descriptions can be read by potential service requesters, either programmatically or by developers. WSDL was created by Microsoft, IBM, and Ariba by combining various attempts at standardization. WSDL descriptions are written in XML, and can be compared to method signatures in object-oriented programming.

WSDL descriptions include how to structure a request, the input parameters required, the data the service will output, the location where the service requester will send SOAP messages, the transport protocol it will be sent on, and more.

The WSDL description helps the service requester determine how to request the service. WSDL descriptions are machine-readable, which allows the service requester to generate necessary code to interface with a service provider automatically. This process is known as **binding**, whether or not it is done automatically or by the developer. Only after binding can the service requester invoke the service using SOAP messages that they structured with the help of WSDL descriptions.

WSDL Description

Let us examine the general form of WSDL descriptions. WSDL is modular, which means that it is made up of different components that can be combined to define the desired interface or multiple interfaces. WSDL descriptions can even import other WSDL specifications by importing their descriptions and using them to combine into new interface descriptions.

There have been several versions of WSDL. The previous versions used different terminology and structure, although concepts remain largely the same. Some of the most important part of a WSDL 2.0 description are:

- **Types,** which describe the data types that are used. Developers can define abstract data types in XML. If only basic data types already available in XML are used by interactions, then this part is not needed.
- **Interfaces** describe interfaces to the services provided in terms of what operations can be performed and in what order. The order of operations can be described by the message exchange patters of

request-response, solicit-response, one-way, and notification. Interfaces were formally called portTypes in WSDL 1.2.

Interfaces are abstract. They cannot be used until they are bound to the concrete implementations that are needed to use web services.

The categories used to bind interfaces to concrete implementations are:

- **Bindings**, which determine how the SOAP message is translated into XML, and dictate the form of the messages. This includes specifying between document-style and RPC-style interactions, as well as specifying the transport protocol on top of which the SOAP messages are sent.
- **Services**, which bring together interfaces and bindings, and assign them to **endpoints** or **ports**. These are located with URIs.

WSDL provides a robust, modular, and extensible service description language. WSDL description enables reuse because WSDL descriptions are broken into very fine descriptions which allow for the reuse of parts of WSDL specifications in different ways. WSDL documents can also import other WSDL documents, gaining them access to data types in the imported WSDL description or to interfaces.

WSDL descriptions, as an XML-based approach, like other aspects of web services, allow for interoperability between different platforms over the Internet. WSDL can also describe non-XML based services. However, like SOAP messages, encoding and parsing XML imposes computational costs. This is the major disadvantage of XML-based standards.

Service Publication and Discovery (UDDI)

Web services need to be **published** and **discovered**. Developers need to be able to find services to build apps, and there needs to be a way to get people to use these created services.

The advent of the Internet helped customers find services through search engines. However, it is important to advertise web services. This is known as publishing. The first framework for publishing was **Universal Description, Discovery, and Integration (UDDI)**.

UDDI was created in 2000 by Ariba, Microsoft, and IBM, but it is now managed by the Organization for the Advancement of Structured Information Standards (OASIS), which is a non-profit organization that also manages a number of other open standards. UDDI was intended to be used to specify a universal registry and broker of web services, using XML and

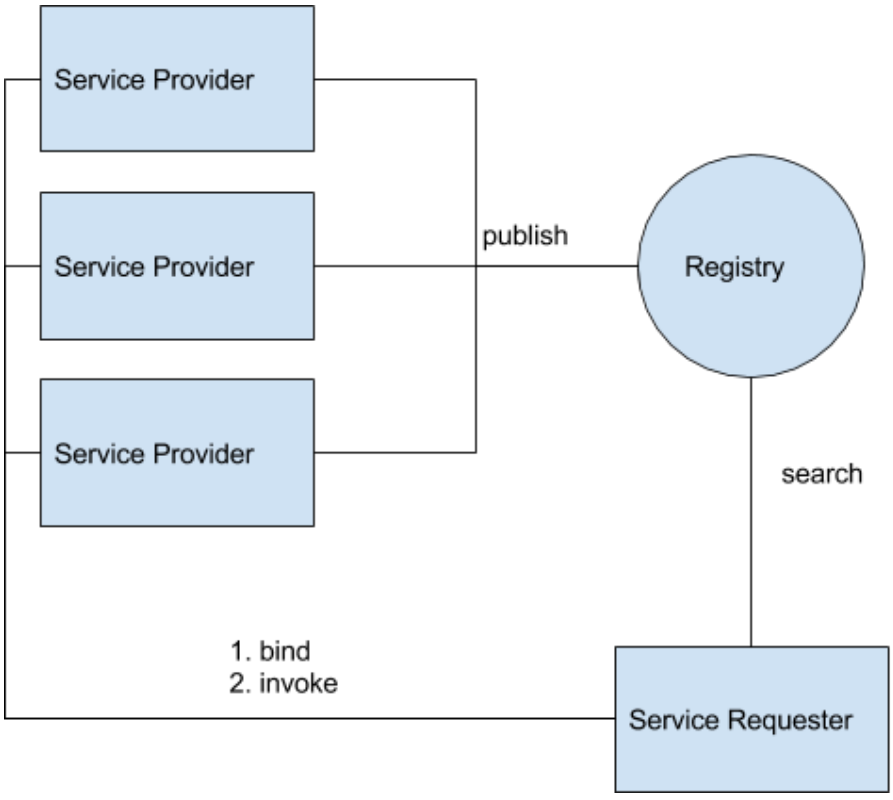
WSDL to structure data about the web services and how they were provided.

UDDI is a protocol for publishing and discovery, and not a necessary component of implementing web services, it is not tied to a specific registry. If the web services to use and the binding is known, then a discovery mechanism is not needed. Conversely, if a service is developed for use with a few specific applications, and it can be bound to those applications directly, then the web service does not need to be published. However, UDDI, is a useful standard for bringing service requester and service providers together.

Overview of UDDI

UDDI allows service providers to publish their services to a UDDI registry. Once they are published, potential service requesters can search the registry and discover the service they need. This is done by searching elements of the WSDL description, or other descriptions of the service or service providers. Bringing service providers and service requesters together is an essential part of creating an effective service ecosystem.

Once the service that the service requester wants to use is found, the service requester can bind to it using the WSDL descriptions. After the service requester is bound to the service by generating the code for the necessary messaging patterns, the service requester can **invoke** the service.



Publishing

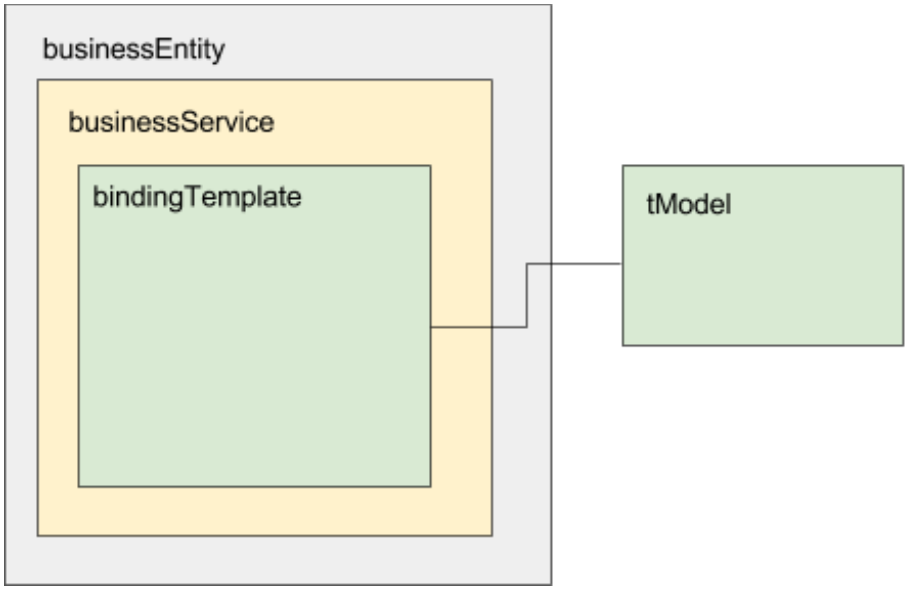
Publishing registers information about the service with a UDDI registry, which includes information about the service provider, the service itself, and various technical descriptions of the service. A **uniform resource identifier (URI)** is assigned to the service by the UDDI registry. The URI is a unique reference used to invoke the service.

Information encapsulated in the UDDI standard is contained in three categories. These are:

1. White pages, where information such as the business name, a short description, contact information, and unique business identification numbers is stored.
2. Yellow pages, which contain information about the service or industry that the business is in, including hierarchical information about the business. For example, exchange rates are a subset of currency services.
3. Green pages, which contain the technical details of how to use the service.

Information encapsulated in UDDI also falls into four data structures. These are:

1. businessEntity, which contains information about the business. This roughly corresponds to the "white pages".
2. businessService, which describes the services that the business is offering. A business can have many services, but a service can only have one business that owns it. This roughly corresponds to the "yellow pages".
3. bindingTemplate, which gives the necessary information needed to invoke the service in question. It is a description of the interface needed for communication. A service may have more than one bindingTemplate. This, along with the tModel, roughly corresponds to the "green pages".
4. tModel, which gives more detailed information about the service. This may include references to the protocols used by the service or detailed technical specifications of the service. The tModel is a flexible data structure; it can represent any abstract concept, and tModels can be nested to provide various meanings. For web services, one of these tModels will reference a WSDL description of the service. This with the bindingTemplate, roughly corresponds to the "green pages".



Publishing and Discovery as Services

UDDI specifies web services for publishing. Service providers publish, including adding, deleting, and modifying entries to a registry through SOAP messages. Operations could be *save_business*, *save_service*, *save_binding*, *save_tModel*, or delete commands for these same elements.

UDDI also specifies web services for discovery. These are accessed by SOAP messages. Commands to search for services include *find_business*, *find_service*, *find_binding*, and *find_tModel*. Information can be requested with commands such as *get_businessDetail*, *get_serviceDetail*, etc.

Service requesters use these commands to find and get information about the service interface. Once the service requester has information about the service interface, it can generate the necessary code to access the interface for the service. In other words, it can dynamically bind to it.

Publishing and discovery are often hidden from the developer. Instead, developers often use web portals to search for services.

Dynamic Binding

Binding can be a highly dynamic, run-time activity, although this has repercussions for a developer. For example, it may prevent a developer from knowing what errors might occur or what exceptions may be generated, and thus from developing robust code. Or a web service provided by a business may require contracts or agreements, which are managed by people and not programs. Consequently, service discovery is usually a design-time activity. Binding can still be automated, although this may be challenging as it would require the interface description to be completely unambiguous.

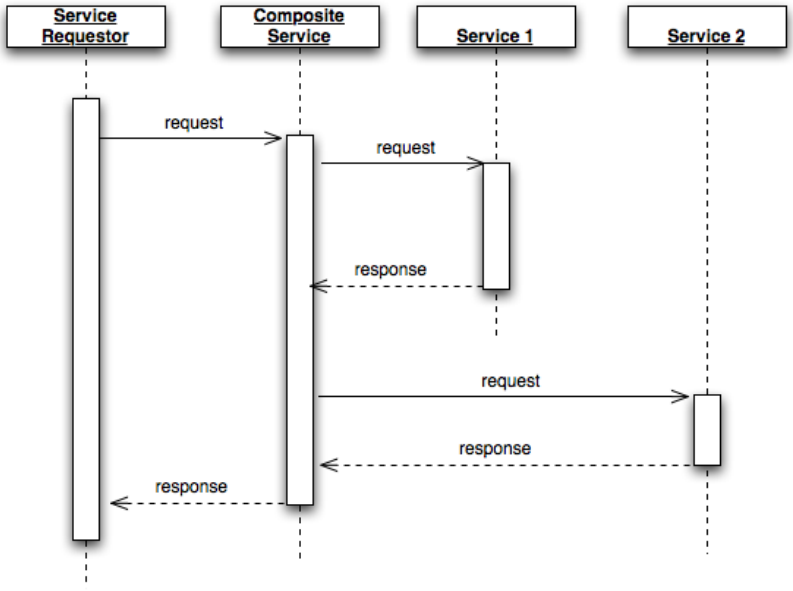
Discovery is usually performed by a human developer, while binding is at least monitored and tested by a human developer.

Service Composition (BPEL)

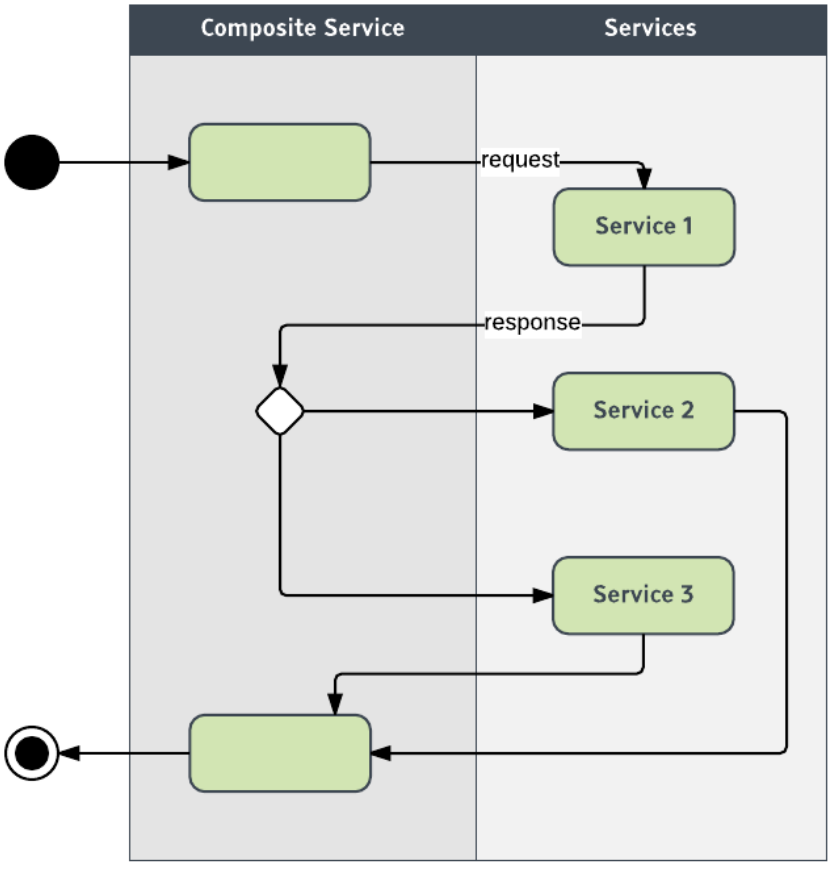
Composition is the action of coordinating several services and providing an interface. A service is composed when it is made up of other services. Further, a service that is composed of other, lower-level services may be used to compose higher-level services. This is comparable to the composing objects principle in object-oriented design. New functionality can be created by combining existing functionality, and encapsulating the new functionality as a service. However, web services are not usually compiled and run in the same physical location like objects.

Composing services involves invoking services in a certain order and handling exceptions that may arise. Services that comprise a composite service still remain separate from that service. A composite service could be like a "script" that uses other services according to some pre-determined order.

The UML sequence diagram below illustrates this:



Service composition can also be captured through a UML activity diagram, as requests made to a service or the service called may depend on response from another service.



Middleware services can be difficult to compose, because their interfaces are not standardized. By extension, wrappers had to be developed as needed, so that components could interface with each other or with the middleware.

Web services, on the other hand, can be easily composed. This is because web services are accessed in similar ways. However, the problems that come with combining services do not go away with web services, they are just easier to manage. This module has examined services invoked with SOAP messages, described by WSDL, and catalogued by UDDI.

Composing a service from other services is similar to a business process. Business processes are easily mapped to activity diagrams. The beginning and end of a process are the points where the composite service is exposed to service requesters.

Basic services often have low-level, basic functionality. This functionality can be combined into higher level functionality with Business Process Execution Language (BPEL), the standard high-level composition language for web services, also known as WS-BPEL. WS-BPEL can then expose this

higher-level functionality as a service which can also, in turn, be composed into higher-level services. Lower-level details of inter-service communication are dealt with by protocols such as SOAP and WSDL.

BPEL allows developers to compose compatible services into a business process that is itself exposed as a service. These services can be from external sources, on the Internet, or internal to a company or organization. Essentially, BPEL allows developers to create new services by composing them with existing services.

BPEL supports basic operations like "if-then-else" decisions, or other logic from various program languages and wrappers.

In addition to composition, web services are also associated with **coordination**. Coordination is when a process coordinates the activities of two or more services. Composition is distinguished from coordination because it exposes the collection of actions as another service.

This concludes the module on web services for this Specialization. The next, and final module will focus on REST architecture.

Module 3: REST Architecture for SOA

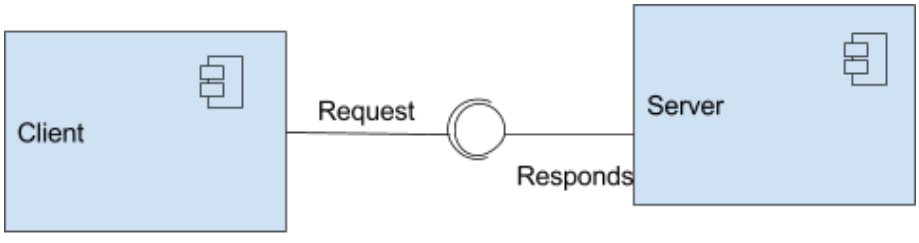
Upon completion of this module, you will be able to:

(a) Explain how REST services are used in service oriented architecture
(b) Describe what a RESTful web service looks like
(c) Explain how a RESTful web service works
(d) Explain microservices

Introduction to REST

In the previous module, we learned about web services and service-oriented architecture (SOA). This module builds on this knowledge, by exploring Representational State Transfer (REST), a client-server architecture based on a request response design. REST is used in distributed applications by using HTTP to send messages to communicate between components.

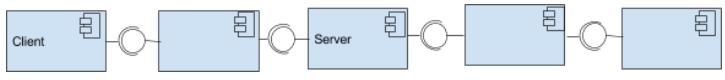
The client sends a request and the server responds, but in REST, the communication is resource-based. This means that messages are sent as representations of resources. **Resources** can be any pieces of information that are self-contained. This can include documents, images, object representations, etc.



REST Restraints

REST is defined by five constraints. These are summarized below.

Constraint	Description

Client-server architecture	A client-server architecture applies separation of concerns by having client and server roles with specific responsibilities that interact with each other. Servers provide services to clients, and clients provide users with a user interface to access services. Client-service architecture allows REST applications to be highly scalable and allows client and server development to occur independently. The client can be improved to provide users with simple and fast user interface without affecting the server, while the server can manipulate larger sets of data because it is freed from having to implement any client responsibilities.
REST is a layered system	REST can consist of multiple architectural layers of software or hardware called by the client and server. These layers can be used to improve performance, translate messages, and manage traffic.  This helps to improve reusability of REST web services because layers can be added and removed based on the needed services of the client.
Interactions must be stateless	In REST architecture, interactions must be stateless. This means that the server does not save information about the current client state or previous requests made by the client. Servers are only aware that the client exists when a request is made. All necessary information for the server to understand and respond to the request comes through with the request, and requests are contained from one another. This improves the performance of web services, as servers do not have to remember the current states of clients in the system. The trade-off, however, is that this imposes significant restrictions on the way a client and server communicate. Every time a client sends a request to a server, it must provide and store information about its current state.

Clients can cache responses	This constraint means that clients can keep a local copy of a server response to use for later requests, depending on what information the servers add to the response to label it as cacheable or non-cacheable. This can help improve performance by reducing the number of requests for the same resources, and help to ensure that the client does not keep redundant or useless data.
There must be a uniform interface for communication between the client and server	This constraint has certain impacts. The first is that there are specific methods that can be understood. REST uses the common HTTP methods, GET, PUT, POST, and DELETE, to communicate different actions the client wants to perform on the resources. The second is that the resource must be identified in the request with a specific Uniform Resource Identifier (URI). Finally, the representations of the resources are uniform. Responses have specific headers, and the resource is written in three specific ways: XML, JSON, or simple text.

These constrains make REST a flexible, high-performance architectural style for building service-oriented systems based on web standards. REST provides benefits including high scalability, reusability, and loose coupling that allow it to meet the needs of modern applications with millions of users.

REST Example

Consider the following example to help understand a request and response. Imagine a client application makes a request to add coffee to an online shopping cart in the form of XML data.

The request uses the HTTP method PUT to identify the method to call in the server. In this example, the client is asking for the shopping card to be updated with a new item by the server. The request identifies the resource in the URI and the XML tells the server the specific shopping cart to add the new item.

The request is stateless, as it includes the information needed to add the coffee item to cart, and the information about what the shopping cart previously included.

REQUEST

```
PUT /web/rest/shoppingcart
<?xml version="1.0"?>

<item>
  <id>4</id>
  <name>Coffee</name>
  <price>5.00</price>
  <qty>1</qty>
</item>

<shopping cart>
  <cid>1234</cid>
  <item>
    <id>5</tid>
    <name>Tea</name>
    <price>4.50</price>
    <qty>2</qty>
  </item>
  <item>
    <id>100</id>
    <name>Milk</name>
    <price>7.50</price>
    <qty>1</qty>
  </item>
</shopping cart>
```

A standard HTTP response in the form of a JSON object might be generated. In the header that there is a specific section called "Cache-Control", which determines if the information should be cached on the client side. The parameter, "max-age=30" instructs the client to cache the information for 30 seconds. There are five items in this shopping cart, with a total of $70.51.

RESPONSE

```
HTTP/1.1 200 OK
Cache-Control: max-age=30, public
Access-Control-Allow-Origin: *
Date: Fri, 06 Jan 2017 18:04:32 GMT
Content-Type: application/json;charset=utf-8

{
  "cart": {
    "id": "1234",
    "qty": 5,
    "subtotal": "$70.51"
  }
}
```

Designing a REST Service

This lesson will explore best practices to follow to create a well-designed RESTful API. These include:

- Use only nouns for a URI;
- GET methods should not alter the state of resource;
- Use plural nouns for a URI;
- Use sub-resources for relationships between resources;
- Use HTTP headers to specify input/output format;
- Provide users with filtering and paging for collections;
- Version the API;
- Provide proper HTTP status codes.

Let us examine each of these in turn.

Use Only Nouns for a URI

URIs should not be based on verbs like "/getAllTeachers". These kinds of URIs are not very RESTful APIs because they are not resource based. A good URI directs the client specifically to the resource to access. The action that needs to be communicated will come with the HTTP method in the request. This best practice also helps keep APIs consistent.

The table of URIs below illustrates this best practice for an example, where a server is created for a university system with students and teachers. In this example, a student is identified through a numeric SID and a teacher is identified through a numeric TID. Note that the URIs end with /students, /students/SID, /teachers, or /teachers/TID, and not with verbs.

Resource	**GET** read	**POST** create	**PUT** update	**DELETE** delete
/students	Returns a list of students	Create a new student	Bulk update of students	Delete all students
/students/*SID*	Returns a specific student with SID	Method not allowed (405)	Updates a specific student with SID	Deletes a specific student with SID
/teachers	Returns a list of teachers	Create a new teacher	Bulk update of teachers	Delete all teachers
/teachers/*TID*	Returns a specific teacher with TID	Method not allowed (405)	Updates a specific teacher with TID	Deletes a specific teacher with TID

GET Methods Should Not Alter the State of Resource

GET methods should only get resources, but not alter the state of them. Methods used to manipulate resources are PUT, POST, and DELETE. This keeps RESTful APIs consistent with other developers and easy to follow for future client applications.

Use Plural Nouns for a URI

Following this principle keeps APIs simple. Mixing and changing references to a noun between singular and plural can cause confusion. Consistently using a single way of referring to resources helps developers use a REST service. For example, use /students to retrieve all students, and /students/3 to get a specific student 3.

Use Sub-Resources for Relationships Between Resources

This practice means that when a resource is related to another resource, the connection can be demonstrated in the URI. In the following example, the first API can be used to get all the courses of student 3. The second API can be used to get the information on course 2 connected with student three. Sub-resources can make your API easy to use and follow because the relationships are transparent in the URI.

```
GET /students/3/courses/
GET /students/3/courses/2
```

Use HTTP Headers to Specify Input/Output Format

Headers can be used to specify different properties. Sometimes, this can lead headers to be left ambiguous by developers. Two of these are "**Content-Type**" and "**Accept**". Making these specific helps make APIs easier to use.

Content-Type defines the format of the message.

Accept defines a list of acceptable formats that can come as a response.

Consider the example below:

REQUEST

```
POST /students HTTP/1.1
Host: www.example.com
Accept:application/json, text/javascript
Accept-Language:en-US,en;q=0.8
cache-control:no-cache
Connection:keep-alive
Content-Type:application/json;
Content-Length: 9

{
  "id":8,
}
```

RESPONSE

```
HTTP/1.1 200 OK
Cache-Control: max-age=30, public
Access-Control-Allow-Origin: *
Date: Fri, 06 Jan 2017 18:04:32 GMT
Content-Type: application/json;
Content-Length: 21


{
  "id": 8,
  "Name" : "John Doe"
}
```

In this example, the request is formatted as JSON, and accepts formats of JSON or text/Javascript. The response content is in the format of a JSON object, an acceptable format by the requesting client.

Provide Users with Filtering and Paging for Collections

Good APIs provide filters to help sort through large data sets. This can be done by allowing parameters to be passed in after the question mark symbol.

The following example shows an API that could be used to filter through courses, and only return those that are in the department of computing science. This is a query parameter.

```
GET /courses?department=computing+science
```

Another set of parameters that can help handle large sets of data are offset and limits. These can provide filtering and paging for APIs, which reduces the overhead of parsing through large amounts of data. In turn, this makes responses quicker, because only needed information is sent in a response.

```
GET /courses?offset=10&limit=5
```

Version the API

Web services can be used by millions of users. Any changes to APIs can break existing applications or services that call the APIs. Version numbers help prevent issues, and clearly describe where changes occurred. In the example below, "v2" is used to specify the version number.

```
http://api.yourservice.com/v2/students/34/courses
```

Provide Proper HTTP Status Codes

There are many different HTTP status codes that can be returned as a response to a request. The most common and wanted response is status 200, which means that everything is working and functioning on the server side.

RESPONSE

```
HTTP/1.1 200 OK
Cache-Control: max-age=30, public
Access-Control-Allow-Origin: *
Date: Fri, 06 Jan 2017 18:04:32 GMT
Content-Type: application/json;
Content-Length: 77

{
  "cart": {
    "id": "1234",
    "qty": 5,
    "subtotal": "$70.51"
  }
}
```

There are many different kinds of status codes. Some of these include:

- 201 which means that a new resource has successfully been created,
- 204 which means that a resource has successfully been deleted.

Using the correct status code during RESTful API responses can help developers understand and use APIs better. The supplementary resources section provides more information on HTTP status codes.

Example of REST service

This section will examine the proper steps and decisions necessary for creating a RESTful web service with an example of a service that stores, retrieves, and deletes information about students and their courses. This example will build on the previous APIs demonstrated in this lesson.

In this example, the web service is focused on students and courses. There should be ways to retrieve information about the students, the courses, and

information about which courses students are taking. The GET APIs for this web service will be as below:

Resource	**GET** read
/students	Returns a list of students
/students/*SID*	Returns a specific student with SID
/courses	Returns a list of courses
/courses/*CID*	Returns a specific course with CID
/students/*SID*/courses	Returns all courses of student SID

It will also be necessary to provide services to create, update, and delete resources. Below is a complete table on the APIs that need to be developed:

Resource	GET read	POST create	PUT update	DELETE delete
/students	Returns a list of students	Create a new student	Bulk update of students	Delete all students
/students/*SID*	Returns a specific student with *SID*	Method not allowed (405)	Updates a specific student with *SID*	Deletes a specific student with *SID*
/courses	Returns a list of courses	Create a new course	Bulk update of courses	Delete all courses
/courses/*CID*	Returns a specific course with *CID*	Method not allowed (405)	Updates a specific course with *CID*	Deletes a specific course with *CID*
	Returns all	Adds a course to	Method not allowed	Delete all courses

With the APIs established, a resource representation class for a student and for a course can be created. Below is the representation of the student class. This Java file will have constructors and accessor methods for the id, full name, and department for a student object.

```
public class Student {

        private long id;
        private String fullname;
        private String department;

        public Student(long id, String fullname, String department) {
                this.id = id;
                this.fullname = fullname;
                this.department = department;
        }

        public long getId() {
                return id;
        }

        public String getFullname() {
                return fullname;
        }

        … other methods

        private void save(){
                //code that saves to db
        }

}
```

Next, the Java file that handles the HTTP requests can be created. This can be done through many pre-built Java libraries that provide the client library and specific syntax to communicate with a RESTful service. These include examples such as Restlet, Spring, Jersey, and RESTEasy. This lesson will use Jersey.

Below is an example of how to create a POST service to create a student:

```
import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.HEAD;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.core.Response;

@Path("/studentcourseapi")
public class SimpleService {

    @POST
    @Consumes("application/json")
    @Path("/students")
    @Produces(MediaType.TEXT_XML)
    public void create(final MyJaxBean input) {
        String id = //TODO: call database for next ID
        String full_name = input.param1;
        String department = input.param2;
        Student student = new Student(id, full_name, department);
        student.save();
        return "<?xml version=\"1.0\"?>" + "<student><fullname>" + full_name + "</fullname><department>" +
department + "</department></student>";
    }

    @DELETE
        public Response putMsg(@PathParam("/students") String msg) {
            //TODO: Call methods that delete all students from database
    }
 ... more methods
}
```

After importing the necessary Java libraries, the path for the REST service is written, as "/studentcourseapi". All the APIs will start with this path. Jersey allows for **Java API for RESTful Web Services (JAX-RS)**. This provides pre-defined annotations that help map a resource class as a web resource that can be accessed through HTTP requests. Jersey provides developers with easy-to-use methods that help create web services by simplifying and extending JAX-RS.

In this example Jersey requires the acceptable HTTP method for this API to be stated in an annotation. The @POST specifies a POST method. @Consumes determines the acceptable format that the API will take. In

this example, it will state that this will only take JSON format, although in theory, various different formats like XML and plain text could be specified. @Path specifies the URI of the API. "/students" specifies that the API can be accessed at "/studentcourseapi/students".

It is also possible to state the returning format of the response. In this case, it will produce an XML message in the @Produces field.

After those specific Jersey annotations, we write the method that occurs once the API has been called. In this example, the API should create a new student, so a new Student object is created, and the post parameters are received from the request, which determines the name and department. MyJaxBean has been used in this example to access the input that will be sent from a request. The new student object is saved, and returns an XML message that says the properties of the new student object just created. Note that other APIs like the GET and PUT students and courses will also need to be created.

Once all the API methods have been written, and the Java objects and files have been finished, it needs to be deployed by using Apache or any other web server software.

That concludes creating a Java REST service.

Once everything has been deployed, the API should be accessible from a similar URI:

```
http://your_ip_address/studentcourseapi/
```

The REST service can be tested either through the creation of a client application that calls the URIs with the specific content for the APIs created, or by sending an HTTP request through the command line. Both create an HTTP request with a JSON object that includes the information of the student we want to create.

A tool available for use is curl, which can transfer data from or to a server.

The following curl command sends an HTTP request to the server and invokes the POST API created in this lesson with the JSON object containing a single student named James Dean in the department of computing science.

```
curl -I
"http://your_ip_address/studentcourseapi/students"
-d "{'fullname': 'James Dean', 'department':
'Computing Science' }"
```

The HTTP request may look like this and is sent through the web:

REQUEST

```
POST /studentcourseapi/students HTTP/1.1
Host: example.com
Accept: application/xml
Content-Encoding: identity
Content-Length: 72
Content-Type: application/json

{"fullname":"James Dean",
 "department":"Computing Science"}
```

Once the server receives the message, it should call the method created earlier and create a new student with the properties that were sent in the JSON object. The response sent back to the client may look like this:

RESPONSE

```
HTTP/1.1 200 OK
Cache-Control: max-age=30, public
Access-Control-Allow-Origin: *
Date: Fri, 06 Jan 2017 18:04:32 GMT
Content-Type: application/xml;
Content-Length: 126
<?xml version=\"1.0\"?>
<student>
  <fullname>James Dean</fullname>
  <department>Computing Science</department>
</student>
```

Introduction to Microservices

In the early days of software development, most applications were large and monolithic. These applications were generally developed by a large team all working on the same code base and were made of code written from the ground up, specifically for the application. However, these applications were hard to maintain and had limited scalability. As they became more complex, they often suffered from performance issues.

As reviewed earlier in this course, Service-Oriented Architecture (SOA) evolved as a response to these limitations of software development. SOA

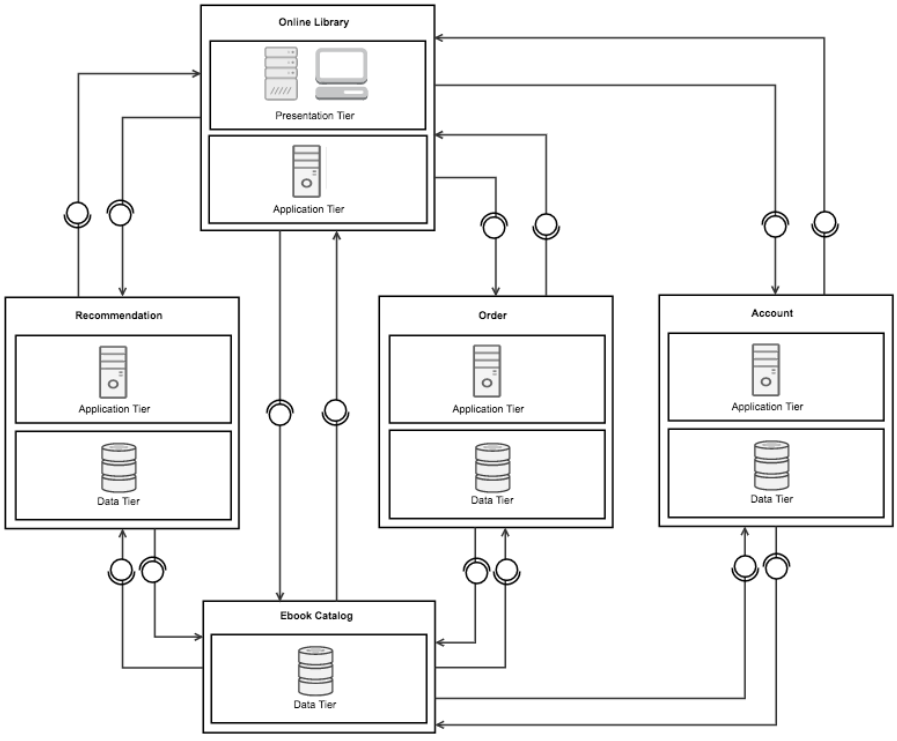
provides principles to guide developers to break down the functionality of large applications into smaller more manageable, modular services. These services are loosely coupled and strictly encapsulated. This modularization allows the smaller services to be tools that can be used to perform a task by internal or external organizations.

Microservices are a kind of variation of SOA, but on an application scale rather than an enterprise scale. The **microservice architectural style** is a way of composing microservices to produce complex applications. Further, in microservices, some SOA principles have been refined to better support application scale systems.

A **microservice** is a process that is responsible for performing a single independent task. A microservice typically is built to perform a specific business capability. Although microservices are developed and exist independently, ultimately they are composed together to provide the overall functionality of an application.

Each microservice often does not obey a full layered architectural style. As microservices are composed with other microservices and are not always intended for end users, presentation and application layers may not always be present. However, usually each microservice controls and manages its own data.

Consider the following example for an online library application. Each microservice has a well-defined interface or API that informs other microservices how it can be used and communicated with. Communication is generally done through standards and protocols such as HTTP and XML.



REST interfaces are used to keep communication between microservices stateless. It is desirable for each request-response to be independent of any other request-response.

Advantages of Microservices

A number of advantages have contributed to the use of microservices in web applications. These include:

- As microservices are independent of the implementation of each microservice, they can use languages, frameworks, and architectures best suited to the service, even if they are different from what other microservices in the same application are using.
    - Developers can create a service using the most appropriate tools for the job.
    - The opportunity for developers to try out new technologies without making an application-wide commitment.
- Microservices make applications easier to scale.
    - A particular microservice can easily be scaled by replication. If there are multiple copies of the same microservice, then multiple requests to the original microservice can be handled in parallel.
- Microservices make applications more resilient to failure.
    - Multiple copies of the same microservice means that if one

instance of the microservice fails, the other instances can continue functioning. Throughput of the system might decrease if there is one less microservice instance processing, but users of the system would be unaware of the failure.

- Microservices can be scaled independently.
    - Microservices have loose coupling. This allows them to be scaled independently, which is important because not all microservices within an application need to scale at the same rate.
- Microservices offer modular maintenance.
    - As microservices have loose coupling, they are easier to maintain. When an application needs to be updated, repaired, or replaced, this can be achieved one microservice at a time, without affecting the rest of the application.
- Microservices can be developed quickly, and deployed and maintained by a small, independent team.
    - Because a small team is responsible for a small piece of functionality of the entire application, the team does not need to be familiar with the whole application to be able to do their jobs. Microservices can thus be developed quickly and in parallel.

Disadvantages of Microservices

Although microservices offer many advantages, there are also certain disadvantages to keep in mind.

- Some centralized management of all microservices will be required to coordinate all the microservices.
    - An application made up of microservices is a distributed system, that is enabled through asynchronous communication. So microservices will need to be coordinated through some central method of management, or they may become inconsistent and result in errors.
- Transactions may span multiple microservices.
    - Databases will likely be distributed over multiple microservices, so transactions may span multiple microservices. Again, centralized management is needed to prevent inconsistencies and errors.
- Testing is complex.
    - Test conditions change, which result in making it harder to test a distributed system. It can be difficult to reproduce bugs that come about from complex interactions between microservices.

- All microservices in the application must be robust to handle failure of any other microservice.
    - It is important to consider how an application will cope when a microservice fails, and there is no other instance of the microservice to take its place. Any other microservices in an application must be robust enough to handle any failure in a microservice.

Using Microservices

The microservice architectural style can be used to create entirely new applications, particularly applications that can be broken down into a collection of tasks or business capabilities. Since most applications can be broken down this way, particularly large applications, microservices have a wide reach of applicability. As tasks can be separated into compartmentalized microservices, the functionality of these services can be easily composed and recomposed to suit the needs of the application. This facilitates code reuse, and keeps code understandable and manageable.

DID YOU KNOW?

The microservice architectural style has one exception in its wide applicability, which is for older, monolithic applications that rely on pre-compiled binaries. Code for such applications becomes rigid and brittle because it is easy to introduce bugs when making small or simple changes.

Modernizing a code base like this may require much of the application to be wrapped.

Microservices can be local, remote, or some combination of the two.

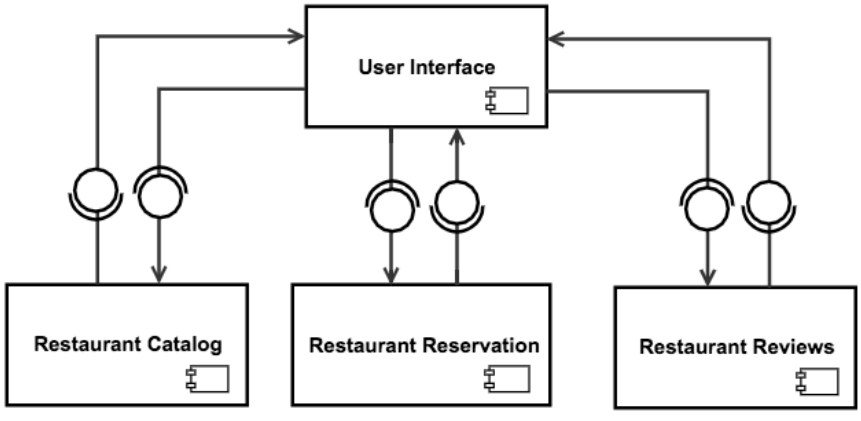
When using microservices, it is important to consider the amount of communication that will be required between microservices in an application. Messaging between microservices have an overhead cost, no matter what communication standards and protocols (HTTP, XML, etc.) is used. Communication between microservices is also stateless. Depending on the application, however, it may be desirable to track the behaviour of a user and their interactions. Web applications can do this with cookies, but this potentially increases the amount of data to transfer between microservices. The overhead cost of communication must be taken into account.

Let us consider microservices through an example. Imagine a web application that allows users to find nearby restaurants, place a reservation, and review a restaurant.

The following microservices for this application can be broken up as follows:

- A user interface microservice that allows the user to interact with the application.
- A restaurant catalogue microservice that provides all restaurants in the system.
- A restaurant reservation microservice that places a reservation with the selected restaurant.
- A restaurant review microservice to access and make restaurant reviews.

For this example, we will assume all communication between microservices is HTTP and REST based. Let us represent the microservices in a diagram.



As demonstrated above, when a user visits the website for this application, the user interface microservice prompts the user to enter their location. Once the address is entered, the user interface microservice communicates with the restaurant catalogue microservice to determine nearby restaurants. This information is communicated back to the user interface microservice, and displayed to the user.

When a user chooses to view more details about a particular restaurant, the user interface microservice communicates with the restaurant review microservice to access the review of the restaurant. This is then displayed to the user, and it provides an option to write a review, or to place a reservation for a particular time.

If a user decides to review the restaurant, the user interface microservice communicates the review to the restaurant review microservice so it can be

saved.

If a user decides to place a reservation, the user interface microservice communicates with the restaurant reservation microservice to inform the restaurant. This returns a confirmation message provided by the reservation microservice.

The microservice architectural style allows applications to be developed in a way that makes applications easy to update and to scale. This helps businesses stay relevant, and to adapt to the rapidly changing technological landscape.

Course Resources

Course Readings

- Disqus. (2017). Disqus. Retrieved from https://disqus.com/how/
- Disqus. (2017). Universal Code. Retrieved from https://disqus.com/admin/universalcode/
- HTTP Status Codes

Glossary

Word	Definition
.NET	A Microsoft framework that serves as middleware.
Accept	A header that defines a list of acceptable formats that can come as a response.
Aggregation	A type of relationship of the design principle of aggregation. Aggregation indicates a "has-a" relationship, where a whole has parts that belong to it. Parts might be shared among wholes in this relationship.
Anonymous function	An un-named function.
Application layer	A typical layer in a web-based system, whose function is to ensure the function or service provided by the system is performed. This layer is usually under the web server layer.
Asynchronous	A type of request-response relationship, where a client sends a request, but control returns right away so that it can continue its processing on another need.
Binding	The process in a client that connects to a server.
Binding(1)	The act of programming a service requester to interact with a service provider. This can be done programmatically or manually by using the WSDL description.
Binding(2)	In WSDL, the bindings section gives detail about the protocols and standards that are mapped to the interaction.

Business Process Execution Language (BPEL/WS-BPEL)	A design language for composing web services.
Cacheable	In REST architecture, where clients keep a local copy of a server response to use for later requests.
Cascading style sheet (CSS)	A style that references HTML tags in an HTML file, to apply specific styles to text within those tags, such as fonts and colours.
Client stub	A proxy for the procedure call that establishes a connection with the server, formats the data to a standardized message structure, sends the remote procedure call, and receives the server stub's response.
Client-server architecture	An architecture where the client sends a request, and the server responds.
Common Object Request Broker Architecture (CORBA)	A common object broker architecture, that acts as a set of standards. CORBA is meant to provide an outline of what should be included in object brokers. It is not a step-by-step guide for implementation. Its goal is to create a specification that allows object brokers to be independent of the programming languages used to implement clients and servers.
Composition (2)	In web services, composition indicates the creation of a new service by combining other services.
Content Type	A header that defines the format of the message.
Coordination	When a process coordinates the activities of two or more services.
CORBA facilities	A component of CORBA. CORBA facilities provide servers at an application level.
CORBA services	A component of CORBA. CORBA services are services provided by the middleware "to" the objects being used by the client and server.
Data layer	A typical layer in a web-based system, whose function is to store, maintain, and manage data. It is usually in the form of a database or filesystem, and is the bottom layer.
Discover	The ability to find web services.
Discovery	When service requesters look for a service, they can search by the WSDL descriptions or other aspects of the service. This is known as discovery.

Distributed computing	A system architecture in which computers on a network are able to communicate and coordinate their actions by passing messages through the network.
DOCTYPE	The first line in any HTML file. It is what denotes to the browser that the file is HTML.
Document-style	In SOAP, a document-style interaction is one in which the type of document determines which service is requests and which fields in the document comprise the input.
Dynamic invocation interface	A component of the IDL compiler in dynamic binding for CORBA. It provides all the necessary operations that a client needs for browsing the interface repository, and dynamically constructing methods based on what the client discovers.
Dynamic web pages	Web pages that are generated by an application at the time of access. Dynamic web pages do not exist on the server beforehand.
eXtensible Markup Language (XML)	A markup language that was meant to store and transport data.
GET method	A method that retrieves the resource given by URI provided in the request-line.
HTML DOM	When a web page is loaded in a web browser, the HTML document becomes a document object. This object can be used to obtain and modify content on the web page.
HTTP request methods	Methods that indicate to the web server the intent of your request.
Hyperlinks	Link together multimedia resources such as images, videos, gifs, text, and audio, or documents containing any combination of these.
Hypertext	A document embedded with hyperlinks, which when clicked, will take you to the intended document or resource.
Hypertext Markup Language (HTML)	A markup language that is used to display data and express content. Used to create websites. HTML does not handle the styling of data, however.
Hypertext Transport Protocol (HTTP)	A web standard and protocol that allows web browsers to interact with remote web servers by making requests, receiving responses, and communicating content.

Inheritance	According to the principle of generalization, repeated, common, or shared characteristics between two or more classes are taken and factored into another class. Subclasses can then inherit the attributes and behaviours of this generalized or parent class.
Instantiation	A process during which non-static classes are constructed into objects.
Interface	In WSDL, the interfaces section describes abstractly the potential interactions as a series of input and output operations.
Interface definition language (IDL)	The language through which client and server communicate in an RPC.
Interface headers	A collection of code templates and references that are used to define what procedures are available at compile time.
Interface repository	A component of the IDL compiler in dynamic binding for CORBA. It is used to store the IDL definitions for "all" the objects being served by a broker.
Internet	A network of networks that allow data to be sent between networks, on a global scale.
Interpreted language	A programming language where the web browser interprets the code at runtime.
Java 2 Enterprise Edition	A framework offered by various vendors that serves as a middleware.
Java API for RESTful Web Services (JAX-RS)	An API that provides predefined annotations that help map a resource class as a web resource that can be accessed through HTTP requests.
Javascript	A programming language that can be embedded into HTML documents to make web pages interactive.
JavaScript Object Notation (JSON)	A format that can be used to store and transport data.
Layer	A collection of components that work together towards a common purpose.
Layered	An architecture where there are multiple layers of software or hardware that can be called by the client and server.
Location transparency	The use of names to identify network resources, rather than their actual location.

Markup languages	Markup languages are designed to define, present, and process text in a machine and human readable way.
Microservice	A process that is responsible for performing a single independent task.
Microservice architectural style	A way of composing microservices to produce complex application.
Middleware	A type of architecture used to facilitate communications of services available and requests for these services between two applications that are operating on environmentally different systems.
Non-blocking	When the client component of an RPC does not need to wait for a server response before moving onto another task.
Notification	In SOAP, refers to a one-way message from server to client.
Object brokers	Object brokers combine the distributed computing aspects of remote procedure calls with object-oriented design principles. They simplify and allow distributed systems to use an object-oriented approach.
Object Management Group (OMG)	The group that devised CORBA.
Object request broker	A component of CORBA that provides object interoperability to the client and the server.
One-way	In SOAP, refers to a one-way message from client to server.
Polymorphism	In object-oriented languages, polymorphism is when two classes have the same description of a behaviour but the implementation of the behaviour may be different.
POST method	A method used to add or modify the resource located in the message body of the request, on the host specified in the URI of the request.
Procedure call	An interface provided to make an RPC, which is the actual procedure that is being invoked on the server.
Procedure call by broadcast	An interface provided to make an RPC, which is the same thing as a procedure call but these procedures are invoked by broadcast.
Procedure registration	An interface provided to make an RPC that tells the client what procedures are remotely accessible on the server.

Publish	The process of advertising a web service.
PUT method	A method that takes the object provided in the message body of the request and creates or updates the object at the location specified in the URI of the request.
Remote procedure call (RPC)	An example of middleware, for networked systems. It allows clients to invoke procedures that are implemented on a server.
REpresentational State Transfer (REST)	A client-server architecture based on a request response design. The client sends a request and the server responds, but the communication is resource based.
Request-response	In SOAP, a messaging style in which the service requester requests a service, then the service provider responds.
Resource	Any piece of information that is self-contained. Examples may include documents, images, object representations, etc.
RPC-style	In SOAP, an RPC-style (remote procedure call) is one in which the SOAP response is similar to a method call; an operation and parameters are specified explicitly.
Server stub	Receives the call and invokes the desired procedure in an RPC.
Service	A valuable action that helps fulfill a demand. In software, deploying a service means providing a tool that other software can use. Services are external to the software requesting them.
Service Composition	The act of combining services into new services.
Service requester	A software requesting a service.
Service-oriented architecture	The process of building, using, and combining services.
Servicer provider	A software which fulfills a request.
Services	In WSDL, the services section is the final step to describing the concrete implementation of a service and its location.
SOAP	A protocol specification based on XML that allows services to send information to one another.
Solicit-response	In SOAP, this is a pattern that is similar to request-response, only the server sends the client a request, and the client responds, often with a simple confirmation.

Stateless	In REST architecture, where the server does not save information about the current client state or previous requests made by the client.
Static web pages	Web pages that are stored on the server as a separate HTML file. Static web pages require manual developer intervention when an update is needed, because changes for the webpage must be applied to the corresponding HTML document.
Synchronous	A type of request-response relationship, where a client sends a request, then awaits the server's response before continuing execution.
Tags	Markup languages use tags to define how certain pieces of text are interpreted.
Transmission Control Protocol (TCP)	TCP allows for reliable, ordered, connection oriented communication. HTTP relies on TCP.
Types	In WSDL, the types section defines abstract data types in XML to facilitate information exchange.
Uniform interface	In REST architecture, this a constraint that simplifies communication by identifying resources.
Uniform Resource Identifier (URI)	A string of characters used to identify a resource.
Universal Description, Discovery, and Integration (UDDI)	A standard for connecting service providers with potential service requesters.
Universal Description, Discovery, and Integration (UDDI)	UDDI is used by service providers to publish descriptions of their services. It is used to build public or private registries of web services.
Universal Resource Identifiers (URI)	Address used to identify a resource.
Universal Resource Locators (URL)	Used to identify the resource and tell the protocol how locate and access the resource.
Web applications	Applications that provide graphical user interfaces that allow users to interact with them, but which are run in web browsers and stored on remote web servers, instead of locally.

Web browser layer	A typical layer in a web-based system, whose function is to display information to a user. This layer is usually the topmost of the system.
Web server layer	A typical layer in a web-based system, whose function is to receive the request from the web browser, obtain the requested content, and relay it back to the browser. This layer is usually directly under the web browser layer.
Web service description language (WSDL)	A formal standard for describing services.
Web Service Description Language (WSDL)	The standard protocol for describing the interface of a service.
Web services	Services offered on the Internet.
World Wide Web	A web management system inspired by hypertext and built on top of the Internet.

Sources

- Admin. (2012, October 15). Difference Between Static and Dynamic Web Pages. Retrieved from http://www.differencebetween.com/difference-between-static-and-vs-dynamic-web-pages/

- Alonso, G., Casati, F., Kuno, H., & Machiraju, V. (2004) *Web Services: Concepts, architectures and applications* (94-97). New York: Springer Berlin Heidelberg.

- Capehart, B. L, Capehart, L.C. (eds.). (2005). *Web Based Energy Information and Control Systems: Case Studies and Applications*. Lilburn, GA: The Fairmont Press, Inc. Retrieved from https://books.google.ca/books?id=2cqNN-iho8cC&pg=PA371&lpg=PA371&dq=static+web+page+presentation+layer&source=bl&ots=kzf74cQ5sT&sig=LItNDDcdkViNrIfAd_OfOTt629g&hl=en&sa=X&ved=0ahUKEwjM4Lv80ZjUAhV050MKHZCwDt8Q6AEIRTAD#v=onepage&q=static%20web%20page%20presentation%20layer&f=false

- Disqus. (2017). Disqus. Retrieved from https://disqus.com/how/

- Disqus. (2017). Universal Code. Retrieved from

https://disqus.com/admin/universalcode/

- Gurugé, A. (2004). *Web services: theory and practice.* Burlington, MA: Digital Press.

- Haldar, M. (2017). RESTful API Designing guidelines – The best practices. Retrieved from https://hackernoon.com/restful-api-designing-guidelines-the-best-practices-60e1d954e7c9

- IBM Knowledge Center. (2017). Publishing a web service to a UDDI registry. Retrieved from https://www.ibm.com/support/knowledgecenter/SSAW57_9.0.0/com.ibm.websphere.nd.multiplatform.doc/ae/tjw_uddi_pub.html '

- Indika. (2011, June 19). Difference Between Web Service and Web application. Retrieved from http://www.differencebetween.com/difference-between-web-service-and-vs-web-application/

- Living without sessions. (2006, July 19). Retrieved from http://www.peej.co.uk/articles/no-sessions.html

- Schneidenbach, S. (2016, Feb 24). RESTful API Best Practices and Common Pitfalls. Retrieved from https://medium.com/@schneidenbach/restful-api-best-practices-and-common-pitfalls-7a83ba3763b5

- Spring. (n.d.). Building a RESTful Web Service. Retrieved from https://spring.io/guides/gs/rest-service/

- tfredrich. (n.d.). HTTP Status Codes. Retrieved from http://www.restapitutorial.com/httpstatuscodes.html

- Tutorialspoint. (n.d.). HTTP – Requests. Retrieved from https://www.tutorialspoint.com/http/http_requests.htm

- Tutorialspoint. (n.d.). HTTP – Responses. Retrieved from https://www.tutorialspoint.com/http/http_responses.htm

- Vogel, L. (2017). REST with Java (JAX-RS) using Jersey – Tutorial. Retrieved from http://www.vogella.com/tutorials/REST/article.html

- W3schools. (n.d.). HTML URL Encoding Reference. Retrieved from https://www.w3schools.com/tags/ref_urlencode.asp

- W3schools. (n.d.). JavaScript HTML DOM. Retrieved from

https://www.w3schools.com/js/js_htmldom.asp

- W3schools. (n.d.). JavaScript Introduction. Retrieved from https://www.w3schools.com/js/js_intro.asp
- Wikipedia. (2017). Web services discovery. Retrieved from https://en.wikipedia.org/wiki/Web_Services_Discovery
- Wikipedia. (2017, July 15). Hypertext. Retrieved from https://en.wikipedia.org/wiki/Hypertext