Contents

[JavaServer Faces (JSF) 4](#_Toc417232749)

[What is JSF? 4](#_Toc417232750)

[JSF Pages 5](#_Toc417232751)

[What JSF Looks Like 5](#_Toc417232752)

[Why JSF? 5](#_Toc417232753)

[The JSF Life Cycle 6](#_Toc417232754)

[The Lifecycle Phases 6](#_Toc417232755)

[JSF Request Processing 7](#_Toc417232756)

[JSF Lifecycle 7](#_Toc417232757)

[How JSF Works 8](#_Toc417232758)

[Facelets 8](#_Toc417232759)

[Event Handling 9](#_Toc417232760)

[Component Trees 9](#_Toc417232761)

[Application Scopes 9](#_Toc417232762)

[How Scopes Work 9](#_Toc417232763)

[Application 10](#_Toc417232764)

[Session 10](#_Toc417232765)

[Request 10](#_Toc417232766)

[View 10](#_Toc417232767)

[Beans 10](#_Toc417232768)

[What a Bean Is 10](#_Toc417232769)

[Managed Entities 10](#_Toc417232770)

[Binding to Managed Entities 11](#_Toc417232771)

[SessionMap, ApplicationMap 11](#_Toc417232772)

[Using JavaScript in JSF 11](#_Toc417232773)

[Using CSS in JSF 11](#_Toc417232774)

[Ajax Support (With JSF 2.0) 11](#_Toc417232775)

[Developer Roles 12](#_Toc417232776)

[Page Author 12](#_Toc417232777)

[Component Writer 12](#_Toc417232778)

[JDBC, Data Sources 13](#_Toc417232779)

[Loading the Driver 13](#_Toc417232780)

[Getting a Connection 13](#_Toc417232781)

[Preparing SQL Statements 13](#_Toc417232782)

[Running the Statement 13](#_Toc417232783)

[Getting the Results 13](#_Toc417232784)

[Using a data source to access and commit changes to a database 13](#_Toc417232785)

[Servlets 14](#_Toc417232786)

[Why servlets? 14](#_Toc417232787)

[GET Method & doGet 14](#_Toc417232788)

[POST Method & doPost 14](#_Toc417232789)

[XML 14](#_Toc417232790)

[Why XML? 14](#_Toc417232791)

[Parts of XML 15](#_Toc417232792)

[Tags 15](#_Toc417232793)

[Elements 15](#_Toc417232794)

[Attributes 15](#_Toc417232795)

[Well-Formedness vs. Validity 15](#_Toc417232796)

[Difference between HTML and XML 15](#_Toc417232797)

[XML Schema 16](#_Toc417232798)

[XML vs. JSON 16](#_Toc417232799)

[Benefits of XML 16](#_Toc417232800)

[Web Services 17](#_Toc417232801)

[What is a Web Service? 17](#_Toc417232802)

[Why Use Web Services? 17](#_Toc417232803)

[Problems & Concerns 17](#_Toc417232804)

[SOAP vs. RESTful Web Services 17](#_Toc417232805)

[SOAP 17](#_Toc417232806)

[Anatomy of a SOAP Message 18](#_Toc417232807)

[Web Services Description Language (WSDL) 18](#_Toc417232808)

[Universal Description, Discovery, and Integration (UDDI) 18](#_Toc417232809)

[The History of the Web Service 18](#_Toc417232810)

[Web Services Life Cycle 19](#_Toc417232811)

[Additional Terminology 19](#_Toc417232812)

[Context Dependency Injection 19](#_Toc417232813)

[Cross-Site Scripting 20](#_Toc417232814)

[Data Access Object (DAO) 20](#_Toc417232815)

[Model-View-Controller (MVC) 20](#_Toc417232816)

[Code Examples 21](#_Toc417232817)

[Ajax 21](#_Toc417232818)

[The f:ajax Tag 21](#_Toc417232819)

[Hashmap 21](#_Toc417232820)

[JavaMail 22](#_Toc417232821)

[JDBC 23](#_Toc417232822)

[Creating a Connection (DBHelper.java) 23](#_Toc417232823)

[SQL Query 24](#_Toc417232824)

[JSF Guessnumber Program 25](#_Toc417232825)

[index.xhtml 25](#_Toc417232826)

[index\_2.xhtml 25](#_Toc417232827)

[NumberBean.java 26](#_Toc417232828)

[Prerender View (Page Re-Direction) 27](#_Toc417232829)

[Code Embedded in XHTML 27](#_Toc417232830)

[SOAP Request (HTTP) 27](#_Toc417232831)

[SOAP Response (HTTP) 27](#_Toc417232832)

[SQL Injection 28](#_Toc417232833)

[Base Code 28](#_Toc417232834)

[Solution 28](#_Toc417232835)

[Templating 28](#_Toc417232836)

[Facelets Template 28](#_Toc417232837)

[Facelets Client (uses Template) 29](#_Toc417232838)

[HTML Output 29](#_Toc417232839)

[WSDL Example 29](#_Toc417232840)

[XML 30](#_Toc417232841)

[Schema 30](#_Toc417232842)

# JavaServer Faces (JSF)

## What is JSF?

JSF, or Java Server Faces, is a server-side user interface component framework for Java-based web applications. Basically, it’s the web UI that can be developed for web-based Java applications. JSF is a **specification** and **reference implementation** for a web application development framework. It is made up of:

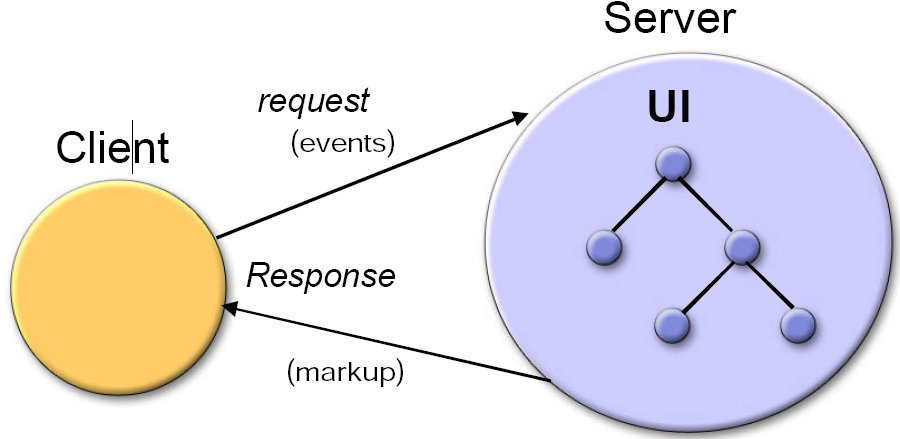
* Components
* Events
* Validators
* Back-end-Data Integration

JSF is designed to be leveraged by a variety of tools, such as NetBeans, RAD, Eclipse, etc. JSF makes use of an **events-based** interaction model, instead of the traditional request/response model, and provides a built-in **validation framework**, basic page **navigation support**, and many other features.

### JSF Pages

A JSF Page is represented by a tree of UI components, called a **view**. It’s easy to use,

### What JSF Looks Like



## Why JSF?

Why use JSF? Because it uses the **model-view-controller** architectural pattern, it’s easy to use, supports **client device independence**, also known as being **responsive** to the client’s device, regardless of if it’s a phone or tablet or TV, it has a huge vendor and industry support, and has a built-in UI component model. JSF also offers a finer-grained **separation of behavior and presentation** than other alternatives, treating UI elements as **stateful objects**. It even works with any presentation technology, and doesn’t restrict you to a particular scripting technology or markup language.

## The JSF Life Cycle

The JSF “life cycle” begins when a client makes a request for a page. The requested **view** is then built by JSF, taking into consideration changes of state that were saved from the previous **postback**. Once **postback** is performed, the JSF implementation must perform **validation** and **conversion**.

### The Lifecycle Phases

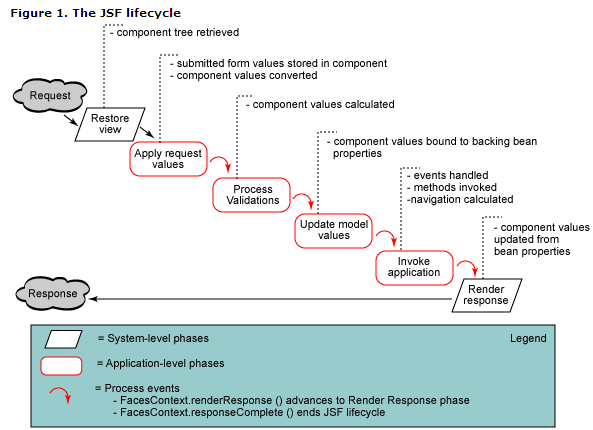
* **Restore View (Reconstitute Component Tree)**
  + JSF begins the restore view phase as soon as a link or a button is clicked and JSF receives a request. During this phase, the JSF builds the view, wires event handlers and validators to UI components and saves the view in the FacesContext instance. The FacesContext instance will now contains all the information required to process a request.
* **Apply request values**
  + After the component tree is created/restored, each component in component tree uses decode method to extract its new value from the request parameters. Component stores this value. If the conversion fails, an error message is generated and queued on FacesContext. This message will be displayed during the render response phase, along with any validation errors.
  + If any decode methods / event listeners called renderResponse on the current FacesContext instance, the JSF moves to the render response phase.
* **Process validations**
  + During this phase, the JSF processes all validators registered on component tree. It examines the component attribute rules for the validation and compares these rules to the local value stored for the component.
  + If the local value is invalid, the JSF adds an error message to the FacesContext instance, and the life cycle advances to the render response phase and display the same page again with the error message.
* **Update model values**
  + After the JSF checks that the data is valid, it walks over the component tree and set the corresponding server-side object properties to the components' local values. The JSF will update the bean properties corresponding to input component's value attribute.
  + If any updateModels methods called renderResponse on the current FacesContext instance, the JSF moves to the render response phase.
* **Invoke application**
  + During this phase, the JSF handles any application-level events, such as submitting a form / linking to another page.
* **Render Response**
  + During this phase, the JSF asks container/application server to render the page if the application is using JSP pages. For initial request, the components represented on the page will be added to the component tree as the JSP container executes the page. If this is not an initial request, the component tree is already built so components need not to be added again. In either case, the components will render themselves as the JSP container/Application server traverses the tags in the page.
  + After the content of the view is rendered, the response state is saved so that subsequent requests can access it and it is available to the restore view phase.

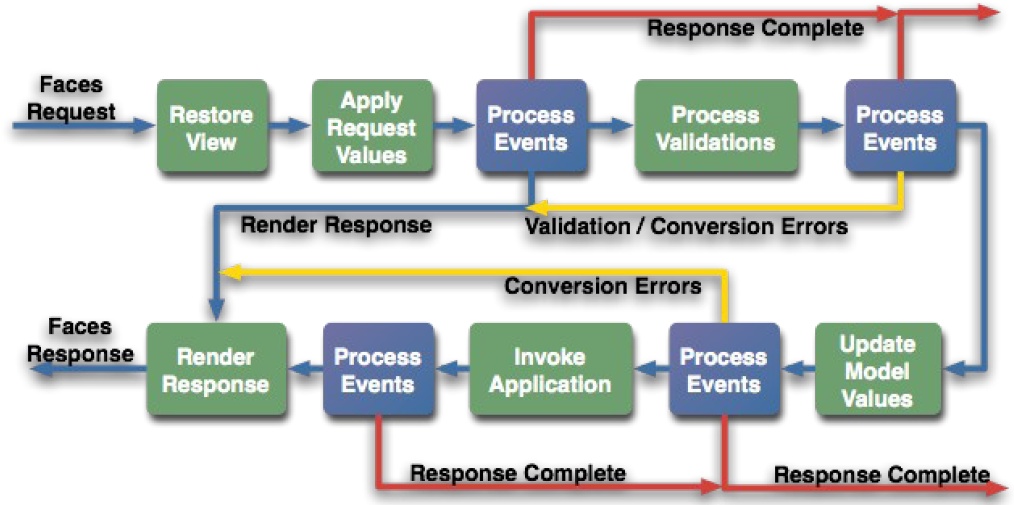
### JSF Request Processing

The JSF Life Cycle handles two types of requests, the **initial request** and **postback**.

* **Initial Request** – This occurs when a user requests the page for the first time. At this point, the lifecycle only executes the **restore view** and **render response** phases. It doesn’t execute the rest because there’s no need for them. The user hasn’t submitted any additional information to be processed.
* **Postback** – **Postback** occurs when a user submits a form contained on a page, which was previously loaded into the browser as a result of an **initial request**. At this point, the lifecycle executes all the phases.

### JSF Lifecycle





## How JSF Works

### Facelets

Facelets is the lightweight page declaration language that’s used to build JSF **views** using HTML-style **templates**, which are then used to build **component trees**. Facelets use **XHTML** to create web pages, and have their own “tag” libraries, in addition to JSF and JSTL tag libraries. It allows for templating for various components and pages.

#### Advantages

Facelets reduce the amount of time and effort that’s needed for development and deployment. Templating and composite components make it easy to re-use code, components are extensible, and it has a very fast compilation time, compile-time EL validation, and high-performance rendering.

#### Templating & Inclusion

JSF offers two features that streamline the creation of new pages; **templating** and **inclusion**. Templating is a way of establishing a common theme or layout throughout your pages, by re-using a **template,** often used in headers, footers, sidebars, or any other common elements that are re-used often in a site. There are four facelet tags that are used to build a page from a template:

1. ui:insert – This tag is used to declare a named content element that will be later defined by another facelet. In essence, it creates the structure, without content. The ui:define tag relies on this tag to tell it what content gets filled in.
2. ui:define – This templating tag defines the content that will be inserted into the template defined by ui:insert.
3. ui:include – This tag is used on the server-side to “**include**” files within a page as it is rendered. An example of this would be loading an additional html file within a page, for a page-within-a-page.
4. ui:composition – This tag defines content that will be included in another Facelet. Content that’s outside of this tag set will be ignored by the Facelets view handler.

### Event Handling

#### Page Forwarding

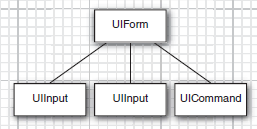
Page forwarding is when you have an action attribute directing the browser directly to the page. The end result of this is that **the URL will not change.**  
<h:form>  
 <h:commandButton action="page1" value="Page1" />  
</h:form>

#### Page Redirection

Page redirection is when you specify within an action’s attribute value “faces-redirect=true”

<h:form>  
 <h:commandButton action="page1?faces-redirect=true" value="Page1"   
/>  
</h:form>

### Component Trees

A component tree is a structure that’s generated when a page is read; it is at this point that tag handlers are executed, and collaborate together. The component tree is a data structure that contains Java objects for all user interface elements on the JSF page. For example, the two UIInput objects correspond to the h:inputText and h:inputSecret fields in the JSF file. Once this is built, the page is rendered.

## Application Scopes

### How Scopes Work

As long as a user stays on a page, the values of components are remembered, even when the page redisplays. When the user leaves the page, component values then disappear. In order to make values available to other pages, or to make the values available to the same page if the user returns later, the values must be stored.

When you create a project from the IDE, the IDE creates three managed beans for storing values:

* RequestBean1
* SessionBean1
* ApplicationBean1

For more on Beans, see page 11

### Application

Application scope lasts until the server stops the application. Values stored in an application bean are available to every session and every request that uses the same “application map”. It’s generally used for things that are used by all users, or track across multiple users (like a counter tracking users, or a drop-down list of measurement types that all use).

### Session

Session scope begins when a user first accesses a page in the web app, and ends when the session either times out from inactivity, or the web app invalidates the session (by calling session.invalidate()). This would see use in storing login info, or a shopping cart.

### Request

Request scope begins when the user submits the page, and ends when the response is fully rendered. This would be used for things that are only needed in a single request/response cycle, like a timestamp.

### View

A @ViewScoped bean will live as long as you return null or void from the action methods (and thus navigate back to the same view). A @RequestScoped bean will be garbaged by end of every request and recreated on every new request, hereby losing all original properties. A @ViewScoped bean is particularly more efficient in rich Ajax-enabled views. A @RequestScoped one would be recreated on every Ajax request.

## Beans

### What a Bean Is

A JavaBeans object is an object in Java that has all private properties (meaning getters and setters are used), a public, no-argument constructor, and implements “Serializable”. The last part means that the bean can be written to streams, things like files, databases, anything.

### Managed Entities

Managed beans are classes that are managed by Java, instead of manually by you. This means that when a class is called, Java instantiates an instance of the class to be used automatically. These can be beans, controllers, what-have-you. You can specify the name of a managed bean by using:   
@Named(value=”newName”)

### Binding to Managed Entities

Binding refers to the link between elements in JSF views, and how they are created. These are reflected in the code below:  
<h:inputText id=”inputValue” value=”#{newController.newBean.value}”> /\* This calls the setter of value. \*/  
<h:outputText id=”inputValue” value=”#{newController.newBean.value}” /> /\* This calls the getter of value. \*/  
In the above, immediately following the pound comes the managed entity, and after that can follow any beans, values, methods, etc. This is how the bean handles getters and setters, automatically.

<h:commandbutton id=”newButton” action=”#{managedEntity.method()}” /> /\* An example of a method being called from a controller/other managed entity. \*/

## SessionMap, ApplicationMap

## Using JavaScript in JSF

Nothing changes, as far as we can tell.   
<h:head>  
<script src=”./scripts/script.js” type=”text/javascript”></script> /\* One way of closing a tag \*/  
</h:head>

## Using CSS in JSF

<h:head>  
<link href=”./css/styles.css” rel=”stylesheet” type=”text/css” /> /\* Other way of closing tag \*/  
</h:head>

## Ajax Support (With JSF 2.0)

You can handle most of the common Ajax use cases—such as field validation and progress indicators—with a tag from JSF’s core library: f:ajax. Like other tags from JSF’s core library, such as f:validator and f:converter, f:ajax attaches a behavior to a component. For example, here’s how you would attach an Ajax behavior to a text input:

<h:inputText value="#{someBean.someProperty}">

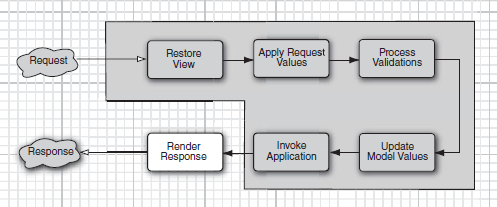
<f:ajax event="keyup" render="someOtherComponentId"/>

</h:inputText>

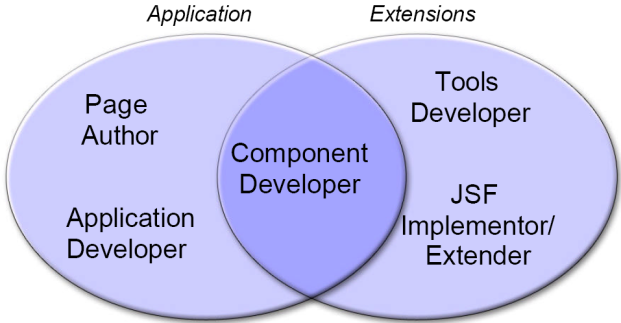
For each keyup event in the text input, JSF makes an Ajax call to the server, and processes the input’s value. When the Ajax call returns, JSF renders a component with the identifier someOtherComponentId. The steps involved in this process are as follows:

1. onkeyup event is detected.
2. Ajax request is sent to the server.
3. The server processes the request sent by Ajax.
4. The Ajax response is passed back.
5. The message is then rendered.

JSF 2.0 splits the JSF life cycle into two parts: **execute** (where components are executed), and **render** (where things are rendered on the client end). On any given Ajax request, you specify a set of components that JSF executes, and another set of components that it renders. The steps are carried out as seen below:



## Developer Roles



### Page Author

A page author creates the user interface of the web application, and should generally be familiar with the markup languages that are being used. They assemble prebuilt components.

### Component Writer

The component writer creates reusable components, renderers, and libraries. **Components** are render-independent properties. **Renderers** are render-dependent properties.

# JDBC, Data Sources

## Loading the Driver

A connection is made using the DriverManager.getConnection method. DriverManager.getConnection takes three variables, the database, the username, and the password. Check the JDBC code examples on <pageno> for a simple method.

## Getting a Connection

There are various methods of obtaining a connection. The most elegant one is to use a data source*.*

DataSource source = . . .;

Connection conn = source.getConnection();

Once you have the Connection object, you create a Statement object that you use to send SQL statements to the database. You use the executeUpdate method for SQL statements that update the database and the executeQuery method for queries that return a result set:

Statement stat = conn.createStatement();

stat.executeUpdate("INSERT INTO Credentials VALUES ('troosevelt', 'jabberwock')");

ResultSet result = stat.executeQuery("SELECT \* FROM Credentials");

## Preparing SQL Statements

This is pretty self-explanatory. An SQL statement will have to be a string, formatted as you’d expect SQL to be.

## Running the Statement

Running the SQL statement relies on the Statement and ResultSet Java class, as follows:  
Statement stmt = DBConn.createStatement();

ResultSet rs = stmt.executeQuery(query);

## Getting the Results

Finally, getting the results of an SQL query involves using the ResultSet class. Using the ResultSet.next(<string>) method, you can then pass the column name through the string to get the value of that column on the row currently loaded in the ResultSet.

## Using a data source to access and commit changes to a database

By default, a database connection is in autocommit mode, and each SQL statement is committed to the database as soon as it is executed. Once a statement is committed, you cannot roll it back. Turn off this default when you use transactions:

conn.setAutoCommit(false);

Execute queries and updates in the normal way. If all statements have been executed without error, call the commit method:

conn.commit();

However, if an error occurred, call:

conn.rollback();

# Servlets

A Java servlet is a Java programming language program that **extends the capabilities of a server**. Although servlets can respond to any types of requests, **they most commonly implement applications hosted on Web servers**.

## Why servlets?

## GET Method & doGet

The GET method involves **sending the query inside of the URL, in name/value pairs**. Other facts:

* GET requests can be cached
* GET requests remain in the browser history
* GET requests can be bookmarked
* GET requests should never be used when dealing with sensitive data
* GET requests have length restrictions
* GET requests should be used only to retrieve data

## POST Method & doPost

With the POST method, the query string is sent in the HTTP message body of the request, again in name/value pairs. Other facts:

* POST requests are never cached
* POST requests do not remain in the browser history
* POST requests cannot be bookmarked
* POST requests have no restrictions on data length

# XML

## Why XML?

HTML was designed with humans in mind, and because of this, machines don’t get as much from it as they could otherwise. XML uses tags that specifically tell the browser what the information contained in it actually is, which allows for a lot of neat stuff to happen. More importantly, it’s become a standardized method of communicating data between sources.

## Parts of XML

### Tags

A tag is the text between the left angle bracket ( <) and the right angle bracket ( >). There are starting tags (such as <name>) and ending tags (such as </name>)

### Elements

An element is the starting tag, the ending tag, and everything in between. In the sample above, the <name> element contains three child elements: <title>, <first-name>, and <last-name>.

### Attributes

An attribute is a name-value pair inside the starting tag of an element. In this example, state is an attribute of the <city> element; in earlier examples, <state> was an element

## Well-Formedness vs. Validity

**Well-formedness** and **validity** are determined in-part by the schema that the XML is using. **Well-formed** checks for syntax errors against the XML standard. The standard has many syntax, grammar, and structure rules:

* The document must have a single root element.
* Elements must be properly nested.
* Tag names should not begin with a number, or contain certain characters, etc.

**Valid** branches off from well-formed, as it is not possible to be valid and not well-formed. An XML document is said to be valid if it is associated with a DTD or XML schema, and complies with the constraints specified. Any namespaces used must be specified, and if the structure created in the DTD or schema is not completely reflected in the XML, including following data formats and value restrictions, it is no longer valid, but can still be well-formed.

## Difference between HTML and XML

One of the differences between HTML and XML is that, unlike HTML, XML’s tags specify not only the format of the contents of a tag, but **what the contents inside are as well**. Another of the major differences between HTML and XML is that XML is far stricter in its adherence to schema. HTML will give the coder a bit of wiggle room and do its best to interpret what it’s given; **XML will just fail**.

## XML Schema

XML schema are a method of defining what an XML document will look like, by defining the tags, elements, and attributes that must be present. **XML schema are themselves written in XML**, which can be interesting.

## XML vs. JSON

* XML is human readable
  + JSON is much easier for human to read than XML. It is easier to write, too. It is also easier for machines to read and write.
* XML can be used as an exchange format to enable users to move their data between similar applications
  + The same is true for JSON.
* XML provides a structure to data so that it is richer in information
  + The same is true for JSON.
* XML is easily processed because the structure of the data is simple and standard
  + JSON is processed more easily because its structure is simpler.
* There is a wide range of reusable software available to programmers to handle XML so they don't have to re-invent code
  + JSON, being a simpler notation, needs much less specialized software
* XML separates the presentation of data from the structure of that data.
  + XML requires translating the structure of the data into a document structure. This mapping can be complicated. JSON structures are based on arrays and records. That is what data is made of. XML structures are based on elements (which can be nested), attributes (which cannot), raw content text, entities, DTDs, and other meta structures.
* A common exchange format
  + JSON is a better data exchange format. XML is a better document exchange format. Use the right tool for the right job.
* Many views of the one data
  + JSON does not provide any display capabilities because it is not a document markup language.

# Web Services

## What is a Web Service?

A web service is a program that can interact with another program, using (going through) standardized web protocols, such as HTTP. It allows for interoperable machine-to-machine interaction over a network. It uses an interface described in a **machine-processable** format (in this case, **WSDL**). The most commonly-known web service is the humble API.

## Why Use Web Services?

A web service allows for the sharing of information through the web without a terrible amount of work. A good example of it being used well is with Amazon, where users of their web services can allow their users to view content and buy products from Amazon without ever actually visiting Amazon.

Beyond all of that, web services allow for **integration and interoperability** at low cost, reducing the requirements for communication down to the “lowest common denominator”, being **XML-based protocols**. Web services are **relatively easy to use**, and are **reusable.** They are **ubiquitous**, able to be used regardless of where your system is.

## Problems & Concerns

* Reliability / Consistency
* Security
* Authentication
* Privacy
* Billing
* Reuse
* Performance
* Incompatible implementations of standards

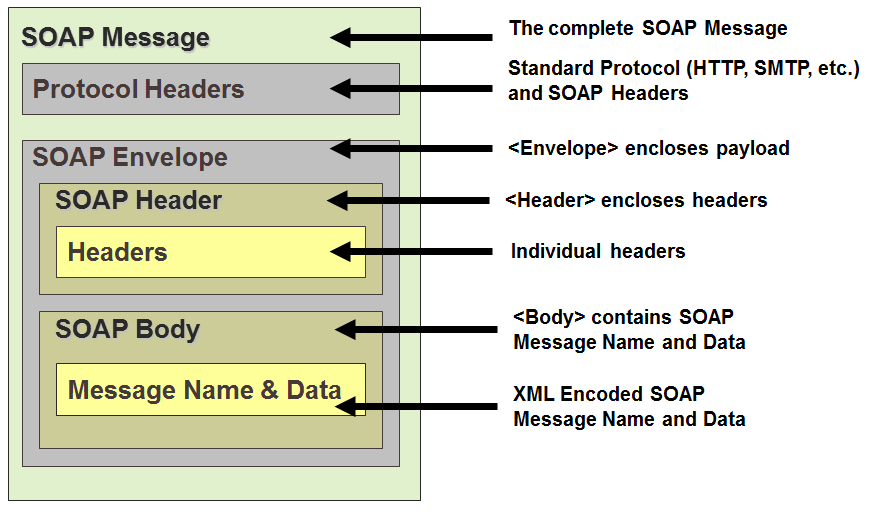
## SOAP vs. RESTful Web Services

### SOAP

SOAP stands for **Simple Object Access Protocol**, and is a **remote procedure call**-based (**RPC**) method. Systems can interact with a web service using SOAP-based messages, which is usually described in the service’s description. This description is usually conveyed using HTTP, with an XML serialization. SOAP uses a lightweight, XML-based messaging format, and builds on W3C XML standards, as well as the IETF HTTP standard. **SOAP can work with any operating system, programming language, and platform**.

<todo> Lim has a youtube link, check it out.

### Anatomy of a SOAP Message



### What is a REST Web Service

The acronym REST stands for Representational State Transfer, this basically means that each unique URL is a representation of some object. You can get the contents of that object using an HTTP GET, to delete it, you then might use a POST, PUT, or DELETE to modify the object (in practice most of the services use a POST for this).

### Who's using REST?

All of Yahoo's web services use REST, including Flickr, del.icio.us API uses it, pubsub, bloglines, technorati, and both eBay, and Amazon have web services for both REST and SOAP.

### Who's using SOAP?

Google seems to be consistent in implementing their web services to use SOAP, with the exception of Blogger, which uses XML-RPC. You will find SOAP web services in lots of enterprise software as well.

### REST vs. SOAP

The main advantages of REST web services are:

Lightweight - not a lot of extra xml markup Human Readable Results Easy to build - no toolkits required SOAP also has some advantages:

Easy to consume - sometimes Rigid - type checking, adheres to a contract Development tools for consuming web services, it’s sometimes a tossup between which is easier. For instance Google's AdWords web service is really hard to consume (in CF anyways), it uses SOAP headers, and a number of other things that make it kind of difficult. On the converse, Amazon's REST web service can sometimes be tricky to parse because it can be highly nested, and the result schema can vary quite a bit based on what you search for.

Whichever architecture you choose make sure it’s easy for developers to access it, and well documented.

## Web Services Description Language (WSDL)

Web Services Description Language, or WSDL, is a method by which web services describe **what they are**, **where they can be found**, and **how they should be used**. This is the primary interface of a web service.

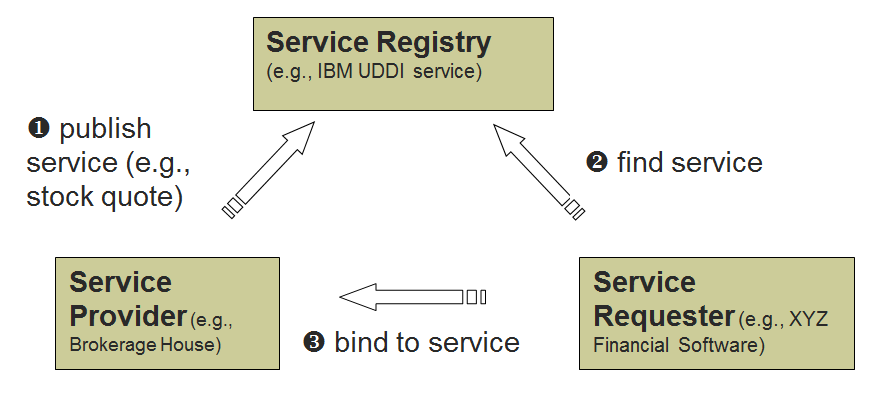
## Universal Description, Discovery, and Integration (UDDI)

Universal Description, Discovery, and Integration (UDDI) allows companies to find publically available web services on the internet, or on corporate intranets. Basically, think of them as yellow pages, making it easier to share business information across the web.

## The History of the Web Service

* 2002 (within the firewall)
  + Simplified app integration
  + Increased developer productivity
* 2004 (contained external users)
  + Simplified business-partner connectivity
  + Richer app functionality
  + Subscription-based services
* 2006 to 2008 (fully dynamic search and use)
  + Casual / ad-hoc use of services
  + New business models possible
  + Commoditization of software
  + Pervasive use in nontraditional devices

## Web Services Life Cycle



# Additional Terminology

## Context Dependency Injection

Dependency injection is a design pattern where external dependencies are “injected” into components rather than baked in. This is where Dr. Lim’s having-a-car-but-needing-a-minivan scenario comes into play. With CDI, instead of you being stuck having to always use your car for your trips, you will be given the correct car to use based on the circumstances. A basic, non-dependency-injected class would look like this:

public class You {

private Car myCar = new Car();

public void Drive(IList<Person> passengers, Location destination)

{

foreach (var passenger in passengers)

myCar.AddPassenger(passenger);

myCar.Drive(destination);

}

}

With the above example, you’d always be using the same car, no matter what. With dependency injection, the class would look like this:

public class You {

private ICar myCar;

public You(ICar car)

{

myCar = car;

}

public void Drive(IList<Person> passengers, Location destination)

{

foreach (var passenger in passengers)

myCar.AddPassenger(passenger);

myCar.Drive(destination);

}

}

In this case, **it doesn’t matter what car is being used**; **the car itself is given to you by an outside entity, via the constructor**. So, why does this matter? Dependency injection allows you to create “**loosely coupled**” classes, which are classes that have less knowledge about their dependencies. This allows you to quickly and easily substitute out dependencies, which makes code more flexible. It also makes unit testing easier, allowing for “**mocking**”, where “mock objects” are injected, allowing for you to test without having to worry about if the dependencies are working or not.

## Cross-Site Scripting

Cross-site scripting is a form of attack that entails an attacker inserting his own malicious javascript into a legitimate website, and then wait for other visitors to visit the site; when they do, the malicious code is downloaded to their computers, and executed. This generally happens when a site allows users to leave comments or content that is **directly added** to the HTML, keeping any code, instead of sanitizing it (like on a forum or message board). The next time someone visits, the page is downloaded by their browser, and the code is triggered.

## Data Access Object (DAO)

A data access object (DAO) is an object that provides an abstract interface to some type of database or other persistence mechanism. By mapping application calls to the persistence layer, DAO provide some specific data operations without exposing details of the database.

## Model-View-Controller (MVC)

Model-view-controller is a style of software architecture used in implementing user interfaces. It divides a given software into three interconnected parts, in order to separate internal representations of information from the ways that information is presented to or accepted from the user. There are three components, the **controller**, which sends commands to the model to update the model’s state (like editing a document). It can also send commands to its associated view, to change the view’s presentation of the model. There is the **model**, which notifies its associated views and controllers when there’s been a change of state. This notification allows the views to produce updated output, and the controllers to change the available set of commands. Lastly, there’s the **view**, which requests information from the model that it uses to generate an output representation to the user.

# Code Examples

## Ajax

<h:inputText id="name" value="#{user.name}">

<f:ajax event="keyup" execute="@this" render="echo"/>

</h:inputText>

...

<h:outputText id="echo" value="#{user.name}"/>

The preceding markup attaches an Ajax behavior to the input. That behavior echoes whatever the user types in the name field. For each keyup event fired by the input, JSF makes an Ajax call to the server. On the server, the Ajax call executes the name component (signified by the built-in @this keyword), and when the Ajax call returns, JSF renders only the echo component on the client.

The echo output text echoes the name input because, when JSF executes the name input on the server, it copies the name into the name input’s associated backing bean property, the name field of a managed bean named user. Subsequently, JSF renders the echo field, which displays the newly updated user.name.

## Hashmap

import java.util.\*;

public class HashMapDemo {

public static void main(String args[]) {

// Create a hash map

HashMap hm = new HashMap();

// Put elements to the map

hm.put("Zara", new Double(3434.34));

hm.put("Mahnaz", new Double(123.22));

hm.put("Ayan", new Double(1378.00));

hm.put("Daisy", new Double(99.22));

hm.put("Qadir", new Double(-19.08));

// Get a set of the entries

Set set = hm.entrySet();

// Get an iterator

Iterator i = set.iterator();

// Display elements

while(i.hasNext()) {

Map.Entry me = (Map.Entry)i.next();

System.out.print(me.getKey() + ": ");

System.out.println(me.getValue());

}

System.out.println();

// Deposit 1000 into Zara's account

double balance = ((Double)hm.get("Zara")).doubleValue();

hm.put("Zara", new Double(balance + 1000));

System.out.println("Zara's new balance: " +

hm.get("Zara"));

}

}

## JavaMail

public class JavaMailApp {

public static void main(String[] args) {

// Recipient's email ID needs to be mentioned.

String to = "msabu@ilstu.edu";

// Sender's email ID needs to be mentioned

String from = "msabu@ilstu.edu";

// Assuming you are sending email from this host

String host = "smtp.ilstu.edu";

// Get system properties

Properties properties = System.getProperties();

// Setup mail server

properties.setProperty("mail.smtp.host", host);

properties.setProperty("mail.user", "yourID"); // if needed

properties.setProperty("mail.password", "yourPassword"); // if needed

// Get the default Session object.

Session session = Session.getDefaultInstance(properties);

try {

// Create a default MimeMessage object.

MimeMessage message = new MimeMessage(session);

// Set From: header field of the header.

message.setFrom(new InternetAddress(from));

// Set To: header field of the header.

message.addRecipient(Message.RecipientType.TO,

new InternetAddress(to));

// Set Subject: header field

message.setSubject("Congratulations!");

String messageBody = "<img src=\"../../web/resources/images/logo.gif\">";

// Send the actual HTML message, as big as you like

message.setContent(messageBody+"<h1>You have been\" promoted to the VP!</h1>", "text/html");

// Send message

Transport.send(message);

System.out.println("Sent message successfully....");

} catch (MessagingException mex) {

mex.printStackTrace();

}

}

}

## JDBC

### Creating a Connection (DBHelper.java)

package dao;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;

public class DBHelper {

/\*\* Creates a new instance of DBHelper \*/

public DBHelper() {

}

public static void loadDriver(String driverStr) {

try {

Class.forName(driverStr);

} catch (ClassNotFoundException e) {

System.err.println(e.getMessage());

}

}

public static Connection connect2DB(String connectStr, String userName, String password) {

String myDB = connectStr;

Connection DBConn = null;

try {

DBConn = DriverManager.getConnection(myDB, userName, password);

} catch (SQLException e) {

System.err.println(e.getMessage());

}

return DBConn;

}

}

### SQL Query

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.sql.Statement;

import java.util.ArrayList;

public int findAccount(UserBean aLogin) {

String userName = aLogin.getUserName();

String password = aLogin.getPassword();

String query = "SELECT COUNT(\*) AS USERCOUNT FROM IT353.ACCOUNT ";

query += "WHERE ulid = '" + userName + "' AND password = '" + password + "' AND accountstatus = 'approved'";

int accountsCount = 0;

try {

DBHelper.loadDriver("org.apache.derby.jdbc.ClientDriver");

String myDB = "jdbc:derby://localhost:1527/IT353";

Connection DBConn = DBHelper.connect2DB(myDB, "itkstu", "student");

// With the connection made, create a statement to talk to the DB server.

// Create a SQL statement to query, retrieve the rows one by one (by going to the

// columns), and formulate the result string to send back to the client.

Statement stmt = DBConn.createStatement();

ResultSet rs = stmt.executeQuery(query);

while (rs.next()) {

accountsCount = Integer.parseInt(rs.getString("USERCOUNT"));

}

System.out.println("approved account count=" + accountsCount);

DBConn.close();

} catch (SQLException e) {

System.err.println("ERROR: Problems with SQL select in findAccount()");

System.err.println(e.getMessage());

}

return accountsCount;

}

## JSF Guessnumber Program

You are to develop a Web-based Number Guessing Game, i.e., one that allows a player to guess a number between a high and a low number, with the system guiding the player by indicating whether the number is too high or too low after a guess. This process is repeated until the player correctly guesses the number or simply leaves your page. Assume the magic number to be guessed is always 88.

1. Implement the game described above using JSF.
2. If you wish to use the static method random() from the Math class to randomly generate a number for the user to guess, how would you change part (a)?

### index.xhtml

<?xml version='1.0' encoding='UTF-8' ?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"

"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"

xmlns:f="http://java.sun.com/jsf/core"

xmlns:h="http://xmlns.jcp.org/jsf/html">

<h:head>

<title>Facelet Title</title>

</h:head>

<h:body>

<h:form>

Guess a Number: <h:inputText id="numberID" required="true"

value="#{numberBean.number}">

</h:inputText> <br/><br/>

<h:commandButton id="submitID" value="Submit"

action="#{numberBean.authenticate()}"/> <br/><br/>

<h:outputText id="resultStr" value="#{numberBean.result}" />

</h:form>

</h:body>

</html>

### index\_2.xhtml

<?xml version='1.0' encoding='UTF-8' ?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"

"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"

xmlns:f="http://java.sun.com/jsf/core"

xmlns:h="http://xmlns.jcp.org/jsf/html">

<h:head>

<title>Facelet Title</title>

</h:head>

<h:body>

<h:form>

Generate a number<h:commandButton id="generatetID" value="Generate"

action="#{numberBean.authenticate()}"/>

<br/><br/>

<h:outputText id="resultStr1" value="#{numberBean.result}" />

</h:form>

</h:body>

</html>

### NumberBean.java

package my\_beans;

import javax.inject.Named;

import javax.enterprise.context.RequestScoped;

@Named(value = "numberBean")

@RequestScoped

public class NumberBean {

private static int number;

private static String result;

public NumberBean() {

number=0;

result=null;

}

public int getNumber() {

return number;

}

public void setNumber(int number) {

this.number = number;

}

public String getResult() {

return result;

}

public void setResult(String result) {

this.result = result;

}

public static int generateNumber(){

return (int)(Math.random()\*100);

}

public static void authenticate() {

// number = generateNumber(); //uncomment while executing index\_2.xhtml (part b)

System.out.println(number);

if (number < 1) {

result = "The number guessed is too low";

} else if (number > 100) {

result = "The number guessed is too high";

} else if (number != 88) {

result = "Wrong number guessed. Keep trying!";

} else {

result = "Great guessing";

}

}

}

## Prerender View (Page Re-Direction)

### Code Embedded in XHTML

<f:event listener="#{user.isAdmin}" type="preRenderView" />

<h:body>

<h1>JSF 2 protected page example</h1>

</h:body>

Bean

public String isAdmin(ComponentSystemEvent event) {

String navi = null;

if (!role.equals("admin")) {

FacesContext fc = FacesContext.getCurrentInstance();

ConfigurableNavigationHandler nav = (ConfigurableNavigationHandler)

fc.getApplication().getNavigationHandler();

nav.performNavigation("access-denied?faces-redirect=true");

}

return navi;

}

## SOAP Request (HTTP)

POST /StockQuote HTTP/1.1   
Host: www.stockquoteserver.com   
Content-Type: text/xml   
Content-Length: 323   
SOAPAction: “www.stockquoteserver.com/GetLastTradePrice”

<?xml version=“1.0” encoding=“utf-8”?>

<SOAP-ENV:Envelope   
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"   
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">

<SOAP-ENV:Body>  
 <m:GetLastTradePrice xmlns:m="Some-Namespace-URI">  
 <symbol>DIS</symbol>  
 </m:GetLastTradePrice>  
 </SOAP-ENV:Body>  
</SOAP-ENV:Envelope>

## SOAP Response (HTTP)

HTTP/1.1 200 OK  
Content-Type: text/xml; charset=utf-8

Content-Length: nnnn

<?xml version=“1.0” encoding=“utf-8”?>

<SOAP-ENV:Envelope  
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"  
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">

<SOAP-ENV:Body>  
 <m:GetLastTradePriceResponse   
 xmlns:m="Some-Namespace-URI">  
 <Price>24.5</Price>  
 </m:GetLastTradePriceResponse>  
 </SOAP-ENV:Body>  
</SOAP-ENV:Envelope>

## SQL Injection

### Base Code

String queryStr = "Select count(\*) from IdPassword where Id = " +   userName + " and Password = ‘ " +

password + " ‘ ";

rs = stmt.executeQuery(queryStr);

// if login info is invalid, rs will have a row and the count will be 0.

// Else, login is good.

rs.next(); // get the count

if (rs.getInt(1) == 0)

outStr += "Your login info is incorrect. Try again.";

else

outStr += "Welcome back," + userName + ". Please buy something this time :)";

### Solution

String queryStr = “Select count(\*) from IdPassword where Id =? and Password = ?”;

PreparedStatement stmt = DBConn.prepareStatement(queryStr);

stmt.setString(1, aUser.getUserName());

stmt.setString(2, aUser.getPassword());

ResultSet rs = stmt.executeQuery();

rs.next(); // get the count

if (rs.getInt(1) == 0)

outStr += "Your login info is incorrect. Try again.";

else

outStr += "Welcome back," + userName + ". Please buy something this time :)";

## Templating

### Facelets Template

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"

    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"

      xmlns:h="http://java.sun.com/jsf/html"

      xmlns:ui="http://java.sun.com/jsf/facelets">

<h:head>

<title><ui:insert name="title" /></title>

</h:head>

<body>

<h2><ui:insert name="header" /></h2>

<ui:insert name="message" />

</body>

</html>

### Facelets Client (uses Template)

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"

    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"

      xmlns:ui="http://java.sun.com/jsf/facelets">

<h:head />

<body>

<ui:composition template="template.xhtml">

  <ui:define name="title">Welcome</ui:define>

  <ui:define name="header">Hello World</ui:define>

  <ui:define name="message">How are you today?</ui:define>

</ui:composition>

</body>

</html>

### HTML Output

<html>

  <head>

    <title>Welcome</title>

  </head>

  <body>

    <h2>Hello World!</h2>

    How are you today?

  </body>

</html>

## WSDL Example

<?xml version="1.0"?>  
<serviceDescription>  
 <soap >  
 <service>  
 <addresses>  
 <address uri="http://localhost//HelloWorld.asmx"/>  
 </addresses>  
 <requestResponse name="HelloWorld" soapAction="http://tempuri.org/HelloWorld">  
 <request ref="s0:HelloWorld"/>  
 <response ref="s0:HelloWorldResult"/>  
 </requestResponse>  
 </service>  
 </soap>  
</serviceDescription>

## XML

### Schema

<?xml version="1.0" encoding="UTF-8"?>

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<xsd:element name="address">

<xsd:complexType>

<xsd:sequence>

<xsd:element ref="name"/>

<xsd:element ref="street"/>

<xsd:element ref="city"/>

<xsd:element ref="state"/>

<xsd:element ref="postal-code"/>

</xsd:sequence>

</xsd:complexType>

</xsd:element>

<xsd:element name="name">

<xsd:complexType>

<xsd:sequence>

<xsd:element ref="title" minOccurs="0"/>

<xsd:element ref="first-Name"/>

<xsd:element ref="last-Name"/>

</xsd:sequence>

</xsd:complexType>

</xsd:element>

<xsd:element name="title" type="xsd:string"/>

<xsd:element name="first-Name" type="xsd:string"/>

<xsd:element name="last-Name" type="xsd:string"/>

<xsd:element name="street" type="xsd:string"/>

<xsd:element name="city" type="xsd:string"/>

<xsd:element name="state">

<xsd:simpleType>

<xsd:restriction base="xsd:string">

<xsd:length value="2"/>

</xsd:restriction>

</xsd:simpleType>

</xsd:element>

<xsd:element name="postal-code">

<xsd:simpleType>

<xsd:restriction base="xsd:string">

<xsd:pattern value="[0-9]{5}(-[0-9]{4})?"/>

</xsd:restriction>

</xsd:simpleType>

</xsd:element>

</xsd:schema>