**Java Enterprise Edition (JAVA EE)**

**JavaServer Faces Technology**

Group 11: Khai Phan Thanh, Adrian Recchi, Ty Unser

Object-Oriented Software Development - Java (BTP400 - SAB)

Instructor: Prof. Eden Ronald Burton

Word Count: 5036

****

**Content Page:**

1. **Introduction** 3

1.1 Tier Classification 3

1.2 The Containers 3

1. **JavaServer Faces Technology**  4

2.1 Introduction 4

2.2 MVC Design Pattern 5

2.3 Life Cycle 6

2.4 Managed Beans 6

2.5 Navigation 9

2.6 JSF Tag 10

2.6.1 The Header 10

2.6.2 Basic Tag 11

2.6.3 Facelet Tag 12

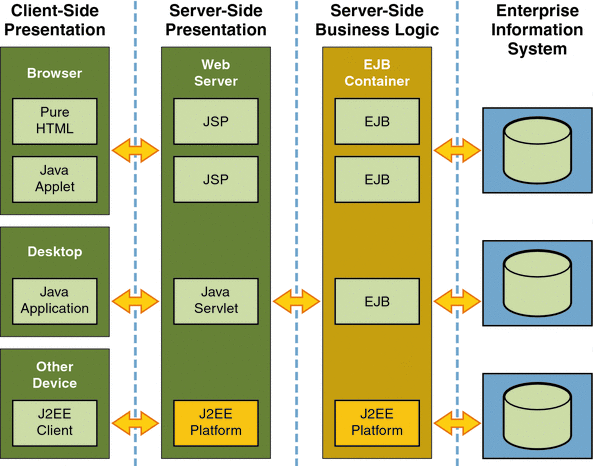
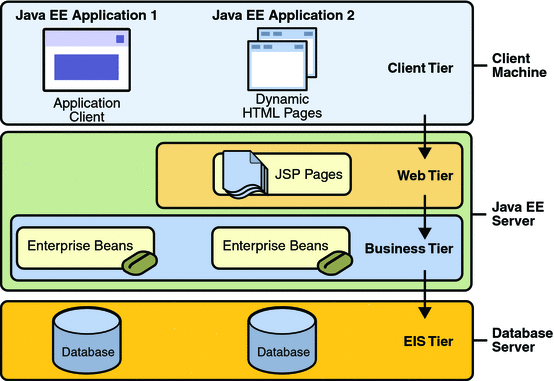
2.6.4 Convertor and Validator Tag 15

1. **Conclusion** 17

**Introduction:**

Java Enterprise Edition is developed by Oracle in order to simplify the coding process and modernization with the development of technology that requires better software development skills. It is designed to help developers to manage the system in a large-scale base, such as company software or corporation software. Besides management, Oracle understands that any system going from small-scale to large-scale will require lots of rework in different levels, for example, security and maintenance. Developers used to required to make code that contains high-security level and easy level of maintenance. Since the Java Enterprise Edition (Java EE) was born, software technology has improved such as high definition level of security is implemented, provides sufficient materials to cover large scale project and also allows developers to concentrate on other important components. Before getting started with Java EE, it is requires that the user to be familiar in building code application as well as building a user interface. It needs for both client and server machine to have JDK extension in order to support Java framework handling problems.

In Java EE, the application is divided into 3 tier: Client Tier, Middle Tier and Server Tier (or known as Enterprise Information System Tier). Each tier has its own purpose (that according to the name) and have different supportive classes and functions. The Client Tier is responsible for controlling all the request from the User Interface (UI) which interact with the user. The code for the Client Tier is located in the client machine and it makes the request via a method (a web browser or a stand-alone application). The Middle Tier is divided into sub-tier: the Web Tier (sometimes known as the Presentation Tier) and Business Tier (sometimes referred to Logic Tier), which handles the client’s request and processes the appropriate data needed for respond. Finally, the last tier, Enterprise Information Systems Tier (EIS or known as the Data Tier) is made up of mainframes, database servers, and other resources which can be accessed by the Business Tier. The Java EE server implements API across all tier, platform support and provides standard services, just like an application service. Base on accessing device type, the Middle Tier will react differently to the request of UI and to the response from the server to ensure data is being transferred can be compatible under the requested using purpose. There are 3 supporting containers used within the Java EE server: the Web Container, used for supporting web components; the web server, the EJB container, allows the use of logic to be used within the platform, and the Application Client Container runs on the client machine and acts as the mediator between the Java EE server and the client application. It will pass the data from client side to the server, the server will then automatically convert into the server format and make the data ready upon user request in the future.

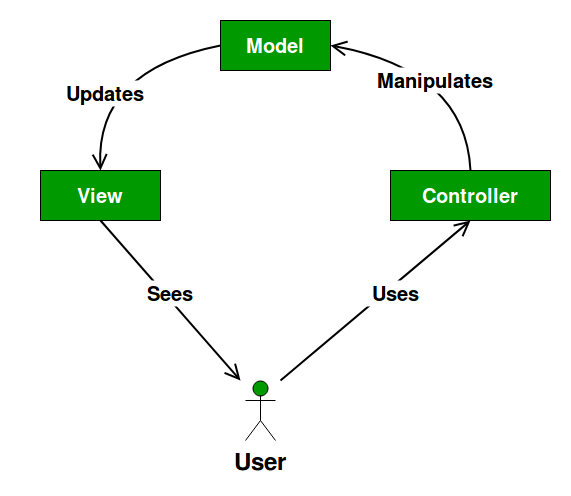


*Figure 1: List of Java EE tiers Figure 2: List of request/response made in different machines and how it handles*

The middle tier, especially the Web Tier, has the most important job in the sequence. It handles any request from Client-side and response by using Business Tier linked with the Server Tier. Therefore, JavaServer Face Technology was created and implemented specifically into Web Tier to help developers reduce the process of coding and enhance the compatibility as well as performance. This report will explain and analyze deeply about this technology and how it helps to reduce workload for users in the future.

**JavaServer Faces Technology:**

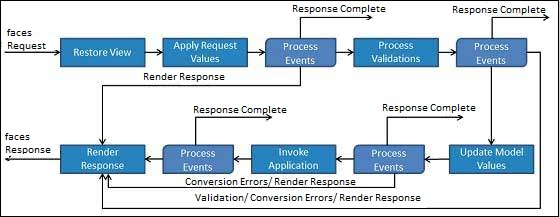
JavaServer Faces Technology is developed by using the XHTML type file. The function is the same with HTML type file, with more built-in advanced functions to process different types of request from the user and also different types of response from the server. In the past, most developers create an HTML for the user interface and a coding application for the server and handling process. The HTML is used to handle all the input/request and the coding start to do calculating, conversion, syntax, transferring data to the server and fetch, etc. separately and return to HTML once all the work are done to display to the user. To reduce this hassle, JSF was developed to use XHTML and interact with the Java file in the same old way, however, it works in a new method and create more efficient. With XHTML, all the basic function could be done easily. This functionality requires a tag and it will be introduced later on. Another special feature of JSF is that it was designed based on MVC Design Pattern. The user interpret the controller by an event handler, the controller then request the model, which responsible for all the data coming in and coming out of the server depends on request (ex. create new data or display current data), check authentication and many others factors. Assumes all factors have been accessed carefully, the model will then response to the controller about the event handler act and update the view, which is the user interface, to either the system have handled the event successfully or if here is a problem either from the user side or server side and display the result. This design pattern is very efficient since this architecture can separate the logic in each side but still connecting to each other. That means, when the developer decides to change the view of the user, it should not affect the controller or model, and the same logic applies to the model. However, if the controller handle a different event, the model might require a bit of change to satisfy with the event and the view might need to change to display a different type of event. But overall, this theory should make developers interact less with their code than the past.



*Image 3: MVC Design Pattern*

Those are the basic theory concept from JavaServer Faces, and before moving on to the details of functionality, or any implementation, it is necessary to know how JSF Life Cycle works.

Understand the JavaServer Face Life Cycle will help improve in the debugging stage by figure out how the technology works. If the developer knows what is happening in each step, they will learn how to fix most of the error related to syntax or logic. There are total of 6 steps between accepting the request, handling the request to response to the request. The first step is the Restore View, that whenever the user clicks on any link/button, the cycle will be triggered to start and JSF will start building the view of the request with a FacesContext instance to hold any error message if there is any inaccuracy data transfer occur during the process. This instance will be saved temporarily and able to pull back upon user request (more like a built form). After the view is constructed, it will extract (or replace if there is already an existing one) all the data from the request to the component tree. In case that the system notice any error during the extraction, it will store them into FacesContext and display the errors after all the steps (supposing that the error is minor so it will skip or prompt the user right away if the error is required attention). If all the value has been successfully extracted to the local component tree, it will use the component attribute value to check the value for any invalid field and if there is any error, it will store the error message to the FacesContext attribute. The cycle will then skip all the remain steps and display the error for the user since validation error is critical. If there is no error, it will update the bean properties in the application file and set the local component tree into the server. Also it will fetch the data from the server into local component tree depends on the event calling. At this stage, the error usually occurs the most when transferring data between XHTML to the application file since the data could not transfer properly into the correct bean properties, the wrong type of bean has been set the value into the wrong application file.



*Image 4: JSF Life Cycle and Processing*

**Managed Beans**

An important part of JSF is the use of managed beans. Managed beans are similar to regular Java Beans, although registered to JSF and managed through the JSF framework. These beans, like regular Java Bean classes contain getter and setter methods, business logic and potentially even backing beans. Backing beans are Java Beans that are used mainly to provide UI logic and to manage data between the web tier and the business tier of an application, similar to how a data transfer object would. The managed bean contains all of the HTML form values.



*Image 5: Basic Managed bean*

In JSF version 1.2 to version 2.0, beans were registered through XML, however since version 2.0 and on this is done using annotations. The main annotation to register your managed bean as a resource to JavaServer Faces is @ManagedBean. The @ManagedBean annotation can take two attributes, the first is name, and the second is eager.



*Image 6: Managed bean annotation*

The name attribute is pretty self explanatory, it sets the name of the bean to the value set in name. If no name for the bean is specified, the bean will take the name of the class as its name by default. The Second attribute for a managed bean is the eager attribute. Managed beans are “lazily-instantiated”, which means that they will be instantiated when a request is made from the application. In the situation you want your bean to be created in the application scope (to be elaborated on later) you would set the eager attribute to true. This will force the bean to be instantiated when the application starts rather than when the bean is first requested.

For the managed beans, there are multiple different scopes that they may be placed under. Managed beans use various annotations to define which scope the bean will be stored in. The following scopes can be specified for a bean:

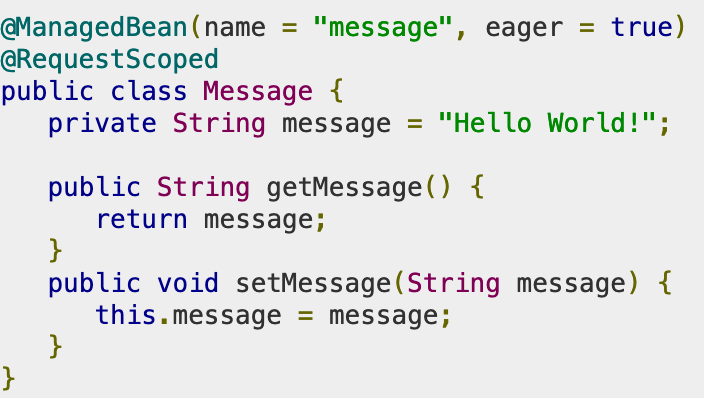
* Application (@**ApplicationScoped**) - Session (@**SessionScoped**)
* View (@**ViewScoped**) - Request (@**RequestScoped**)
* None (@**NoneScoped**) - Custom (@**CustomScoped**)

When a bean is created but no scope annotation is set, the default scope is the request scope. Scope is described briefly below:

|  |  |
| --- | --- |
| Scope | Scope Description |
| Application  (@**ApplicationScoped**) | Application scope lives as long as the web application lasts. The bean gets created with first HTTP request from application (or if eager=true, when application starts) and is destroyed when the web application shuts down. |
| Session  (@**SessionScoped**) | Session scope lives across multiple HTTP requests of an application and so long as the HTTP session is alive Bean is created with the first HTTP request requiring it and destroyed when that session is invalidated. |
| View  (@**ViewScoped**) | View scope lives as during the interaction of a single web page of the web application. The bean gets created upon the first HTTP request requiring it and is destroyed when the user goes to a different view. |
| Request  (@**RequestScoped**) | Request scope lives during a single HTTP request of the application. The Bean gets created upon an HTTP request and is destroyed when the associated response from the HTTP request is finished. |
| None  (@**NoneScoped**) | Indicated that the scope for the bean is not defined for the application. It will live during a single EL evaluation and is destroyed immediately following the evaluation. \*1 |
| Custom  (@**CustomScoped**) | A user defined scope. Bean lives as long as customly mapped. Custom maps must be configured as java.util.Map. (NOTE: Custom scopes are used infrequently used.) |

\*1: An EL evaluation connects the JSF view (usually XHTML) to the java-based back-end.

The last major annotation that can be used for a managed bean is the @ManagedProperty annotation. As JSF is a simple static Dependency Injection (DI) framework, we can use @ManagedProperty to inject a bean in another managed bean.



*Image 7: The Message class is injected into HelloWorld with the @ManagedProperty annotation. Image 8: The class to be injected, Message*

To use the bean in an a web application, we would do so using the XHTML file. For the example of above, this would be the associated XHTML file:



*Image 9: XHTML file that uses the bean*

**Navigation**

Now that we have looked at managed beans, let’s take a look at a practical use for them. Like in other web applications, an important part of a complete app is navigation. In JSF this is done by defining a set of rules, either implicitly or explicitly. Each of these rules will define the routing from page to page based on which page is currently displayed.

Implicit navigation rules are used when there are no rules configured for a page. JSF 2.0 provides this in the form of its auto view page resolver mechanism. With implicit navigation, you only need to the view name in the action attribute for JSF to search for it. For example, we can use the code to add a button to a page.



*Image 10: JSF UI component for a button*

By default, the code above will try to locate a page named “response.xhtml” within the application and navigate there. When the proper navigation rule is selected, the selection of the next page is based on two factors: the action method invoked when clicked, or the logical outcome of a component’s tag, or returned from a function. Some of the common outcomes for a web application could be success, where the application goes to the next page, a failure, where the application goes to an error page or a login page requesting login.

Using managed beans we can create methods that will work with navigation elements. Say we have an application with either multiple buttons or multiple links that we want the user to be able to use to navigate, this can be accomplished via the beans and the JSF UI components. We can also use conditional navigation this way, which allows us to switch pages based on a values. Let’s say there are three links, one for Page1, one for Page 2, and the third for a home page.



Image 11: Managed bean with conditional navigation

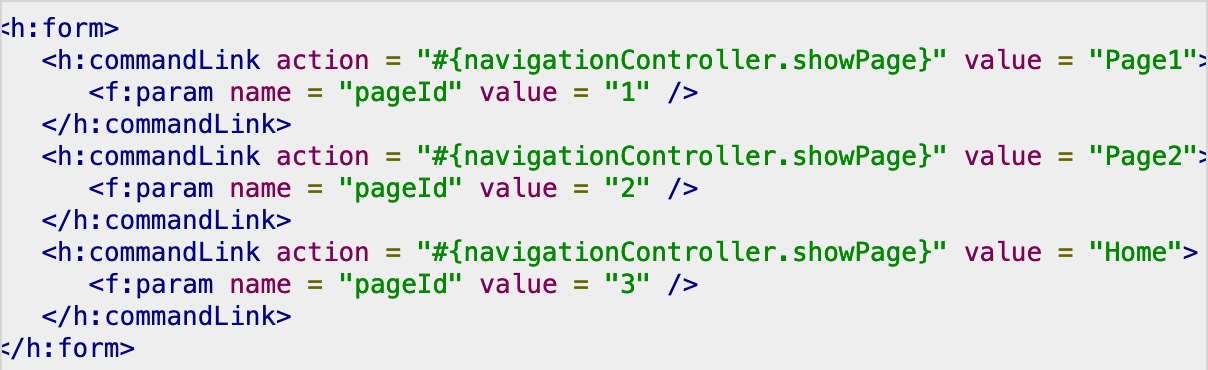


Image 12: JSF UI component with conditional navigation

In the example, we see that each link calls the showPage() method, and depending on the value of pageId, the corresponding page will be rendered to the user.

**JavaServer Tag**

We have discovered how beans can interact with Java coding file to ensure when the user typing something into the XHTML UI, the corresponding value will be stored in the variable/attribute in the coding file. However, to make XHTML file to acknowledge the user input and remember where to store the value to, it requires some “tag" coding like HTML, with more advanced and logical functions. There are 4 types of tag coders can experiences in a XHTML file: Basic Tag, Facelet Tag, Converter Tag and Validator Tag. As the name shows the functionality of the tag going to do and support the developer during the process. These are the most common functionality tag that most developers would need to simplify their work. However, whenever the developer performs other advance and challenging tasks, they would have to pass all the necessary value into the Java coding file and work there, then return into XHTML for display or any other mandatory tasks.

1. **The Header**

This is the most important part of XHTML before doing anything else. The header works just like include<> in C++ or import in Java, to include the source material that would handle the function that will be use. Most of the XHTML use the same header sources and its format is easy to set up. The only difference is that for each type of functional tag, it requires another header source which each header belongs to a different source.

General Header Format for all XHTML file:

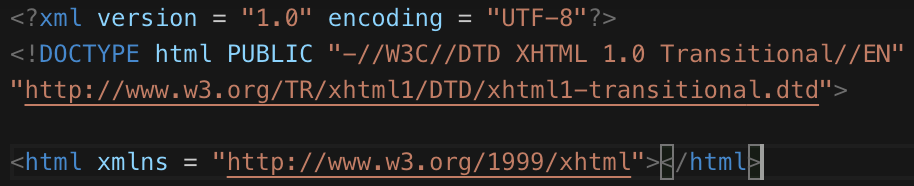


Image 12: Every XHTML header requirement

If developers planning to use one of the implemented tag prepared by JSF, it needs to add an extra source (after the xmlns:h source, before the enclosure ‘>’), depends on which tag type the developer used.

|  |  |
| --- | --- |
| **Basic Tag** | xmlns:h = "http://java.sun.com/jsf/html" |
| **Facelet Tag** | xmlns:ui = "http://java.sun.com/jsf/facelets" |
| **Converter Tag** | xmlns:f = "http://java.sun.com/jsf/core" |
| **Validator Tag** | xmlns:f = "http://java.sun.com/jsf/core" |

1. **Basic Tag**

This is the most basic tag that most XHTML file will use. Most of the tag in this category perform simple tasks such as receiving input from the user (textarea, password, checkbox, select listing, output, rendering an image,...). This basic tag tends to be similar with simple tags used in HTML, but it allows the system to recognize those value stored in these tag might be passing into the application file so as soon as the system receive the data, it will set the value ready to be pass. References to the table below for list of supporting tag in Basic Tag, its functionality and an example of how to implement them accurately.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | **Tag Type** | **Tag Functionality** | | **<h:inputText />** | create a text box and rendering data in the box | | **<h:inputSecret />** | create a password field | | **<h:inputTextarea />** | create a text area | | **<h:inputHidden />** | create an input of type hidden | | **<h:selectManyCheckBox />** | renders multiple checkboxes | | **<h:selectOneRadio />** | render a radio button | | **<h:selectOneListBox />** | renders a single list box | | **<h:selectManyListBox />** | renders multiple list box | | **<h:selectOneMenu />** | create a combo box | | **<f:param />** | pass a param to JSF UI component | | **<f:attribute />** | pass an attribute to JSF UI component | | |  |  | | --- | --- | | **Tag Type** | **Tag Functionality** | | **<h:outputText />** | displaying a text | | **<h:outputFormat />** | displaying a text, can take parameters | | **<h:graphicImage />** | render an image | | **<h:outputStylesheet />** | create a CSS style sheet for web decoration | | **<h:outputScript />** | display a script | | **<h:commandButton />** | render a button of type event trigger | | **<h:Link />** | Render an HTML link | | **<h:commandLink />** | Render an HTML link | | **<h:outputLink />** | Render an HTML link | | **<h:panelGrid />** | renders an HTML table in form of grid | | **<h:message />** | render a message for JSF UI component | | **<h:messages />** | render all available message for JSF UI component | | **<f:setPropertyActionListener />** | Set value of a managed bean property | |

An example of how to use Basic Tag correctly:

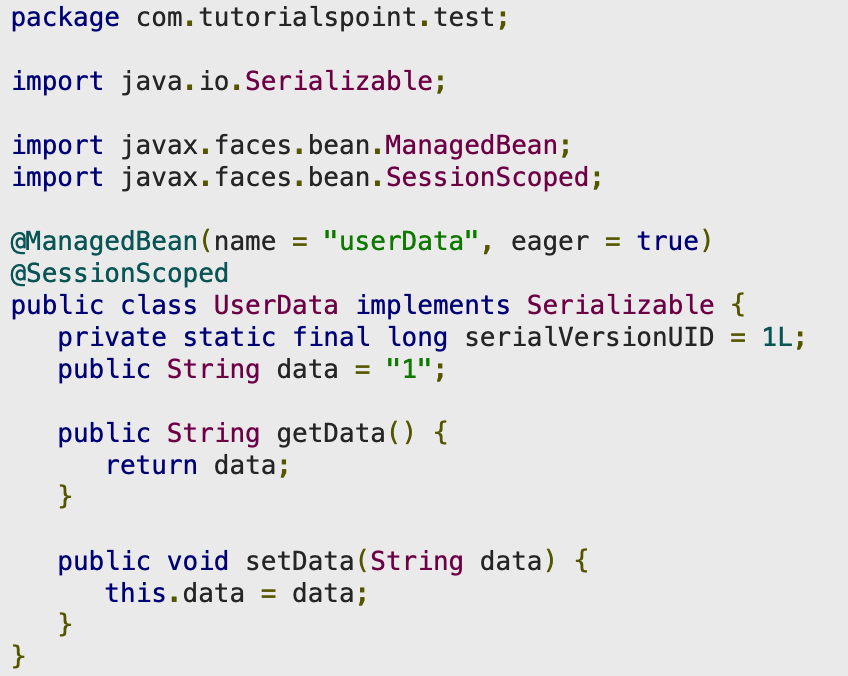
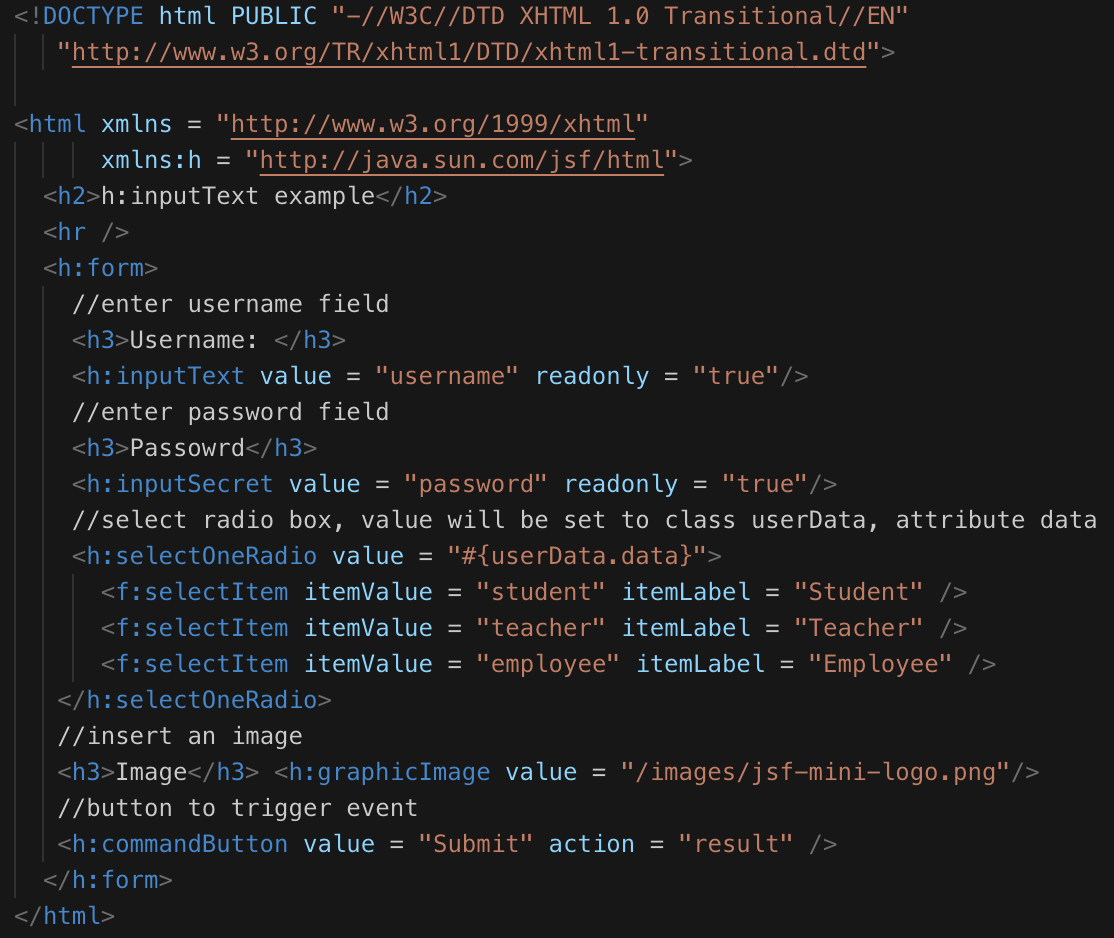
Image 12: Example of XHTML using Basic Tag Image 13: Example of a Java application file use XHTML

Image 12 is an example of XHTML code that creates a simple form, accepts input for username and password, along with a radio box to identify if the user is a student, teacher or employee. It can render an image from an online source and display it in the UI, and a submit button to manipulate the event. Image 13 is an example of a Java application file with added Bean and Scoped to make the class interact with the data in XHTML. The class userData is under control of *@ManagedBean* (the entire class attribute(s) is(are) ready to be assign a value) and *@SessionScoped* is used to set the availability of the class across the HTTP platform. In the image 12 & 13, the attribute data in class userData was set into a radio button which means when the user clicks on the command button to trigger the event, if no error is store in the JSF Life Cycle, it will store the chosen value from the radio button into the attribute “public String data” (which was set default to 1). The examples above shows some basic tags and for the other tags, the coding format is similar and the developer can reference online for more syntax detail as well as purpose of using.

1. **Facelet Tag**

Unlike other Tag, Facelet Tag is more about customize the UI design, inserting content from other XHTML(s) and design the layout of the UI. This is a good example could apply as group work: one person is responsible for the header, other member for the content in the middle, another person to finishing up in the footer and the last person to combine them together. Below is a table with all Facelets tags and description

|  |  |  |
| --- | --- | --- |
| **Type** | **Format** | **Description** |
| **Templates** | **<ui:insert> <ui:insert />** | defines the position to place the template in the main file (header, content, footer) |
|  | **<ui:define> <ui:define />** | defines the content to be inserted into the main template |
|  | **<ui:include> <ui:include />** | includes the content of another xhtml into the main template |
|  | **<ui:composition> <ui:composition />** | load a template using template attribute |
| **Parameters** | **<ui:param> <ui:param />** | accept a parameter as an attribute like application file |
| **Custom Tags** |  | to create a tag that can render custom contents, usually for commandButton event |
| **Remove Tags** | **<ui:remove> <ui:remove />** | to temporary remove a tag that is no longer require, mostly use to remove any custom tag |

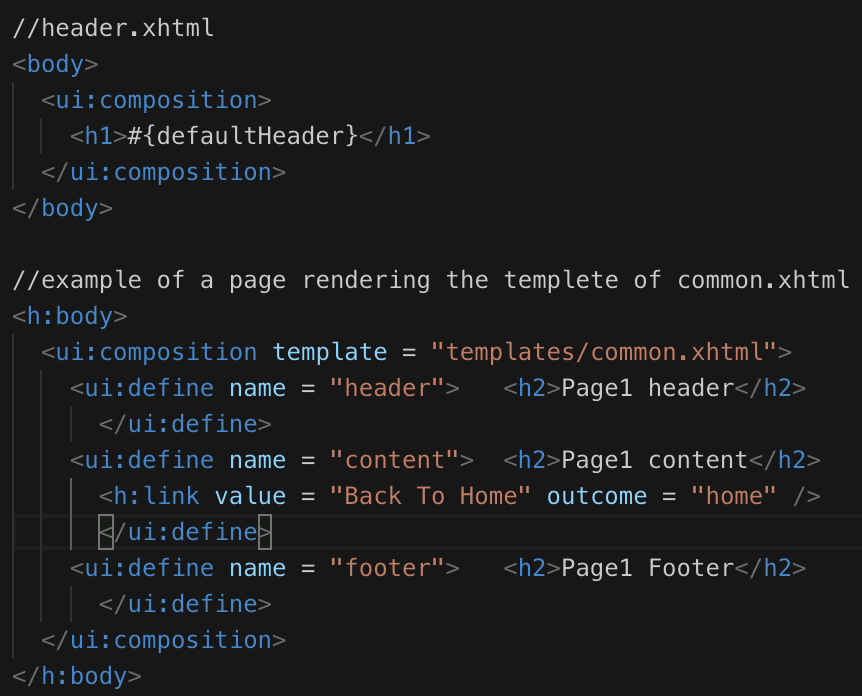
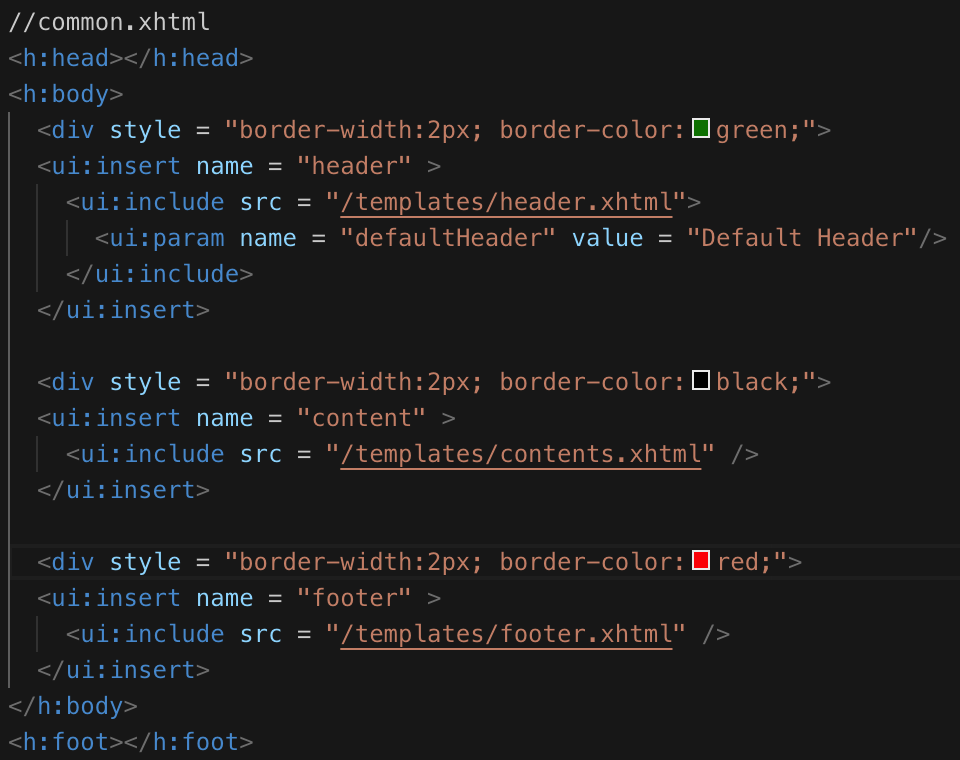


Image 14: Example of a simple UI Facelets tag Image 15: Example of a header XHTML and rendering the common template

This one can be considered as one of the most difficult tags to use because it is a file containing another file which also contains sub-files and each file requires a bit of coding to set up the layout. Basically, there are 3 simple files (and more if your template require more components), header.xhtml, content.xhtml and footer.xhtml. These files will be included in another XHTML file to complete a page design or called template. Then there is another XHTML file(s) who wish to use the content of that template can easily “import" or composition the template and render it. Above is an example (assume that the code already has correct header import). Assume that the content.xhtml and footer.xhtml has the same code with header.xhtml. The header is taking a parameter of #{defaultHeader}, which the value will be pass from common.xhtml. According to common.xhtml, the value being passed to defaultHeader is “Default Header". Each section of the page has been laid out pretty straightforward. It has the style to create a simple style (instead of creating a new CSS stylesheet), an insert section to specify the section of the content in the page layout, an include with a source (this has to be another XHTML file, cannot take other formats). In the example of a page referencing or rendering to a template, there is a bit different in the using composition and define while the common.XHTML doesn't have that. It is because those 2 tags are only used for templating. What the composition do, as described above, will take a reference from an existing template and use that to save time (as the template already have a default style display and a layout component). Another reason for the developer to considering using this code is, for example, take a look at the examples below:

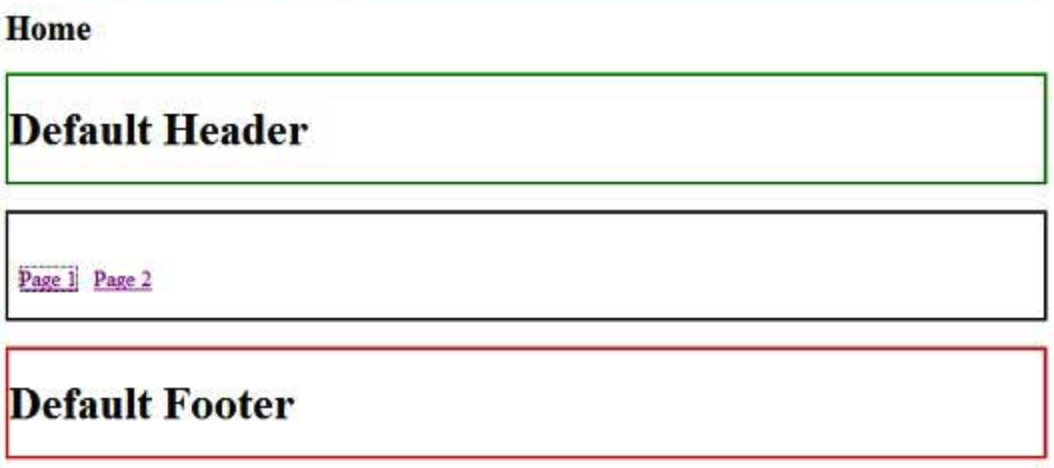
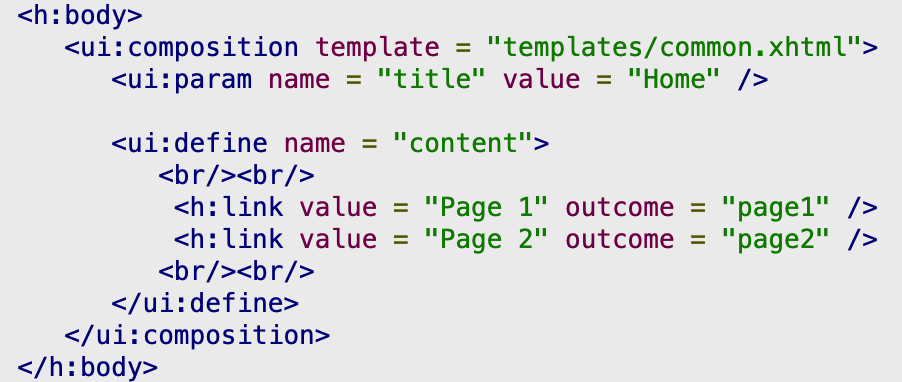


Image 16: Example of an XHTML rendering a template of another XHTML Image 17: Output of the code

In the example above, there is another XHTML page wants to render the template of common.xhtml. Different from the previous template, this template only defines the content section, which is the middle part (according to the template layout). What happens here is, if there is a page using a template that already contents a default value (have to set default), it will display that template to the XHTML that is rendering it from unless that XHTML is specifying they want to keep the format but change the content. In image 16, the developer only wants to change the setting of the content section so as the result in image 17, only the middle part is being changed. While in image 15, all of the section has been changed.

Since Facelets Tag is the most difficult tag amongst other tags, it also provides more functionality to support the developer and one of them is the ability to create a custom tag. For customers, developers are required to add the following code into the XHTML destination folders:

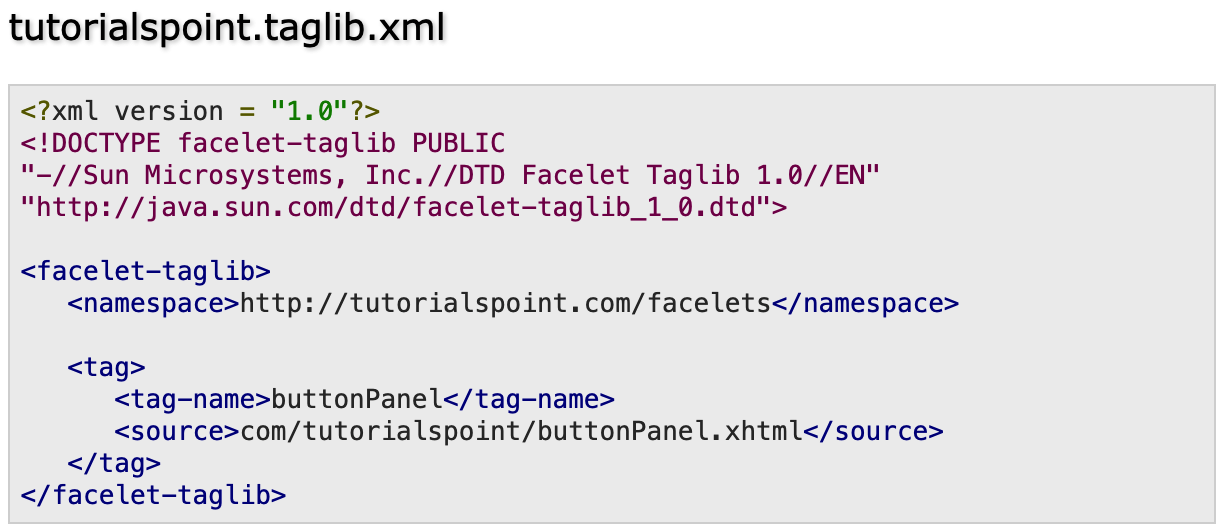


Image 18: web.xml type example Image 19: taglib.xml type code

The above XML file will be used to enhance the ability of custom code, for example, button where it is not supported in Facelets default. and after the developer finish with all the new file, it is time to update the old file

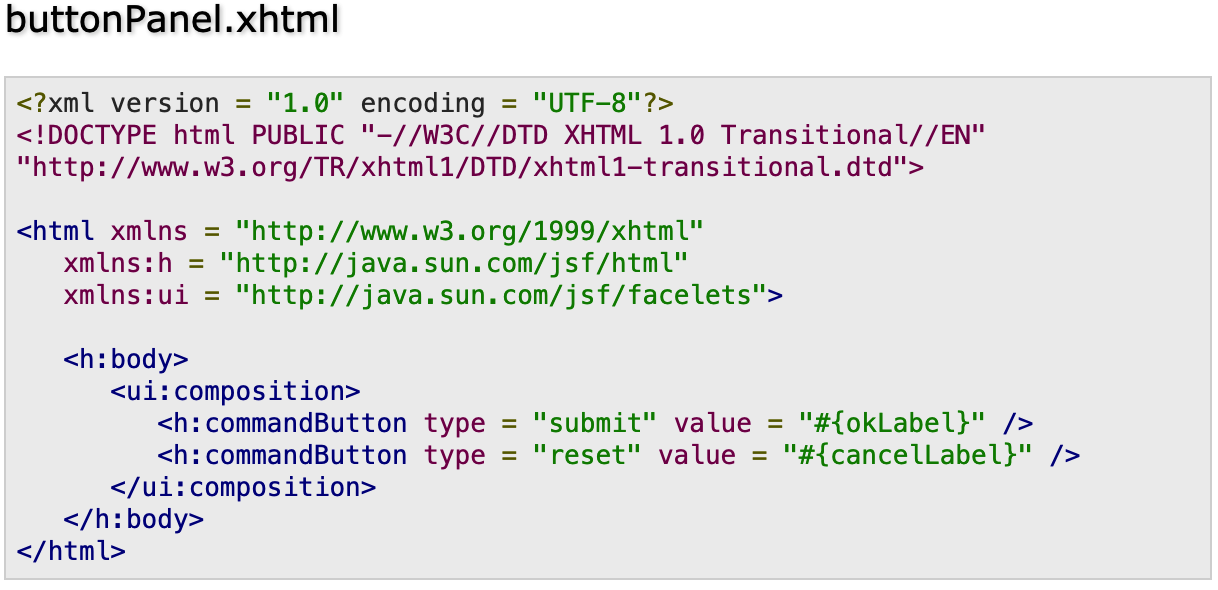


Image 20: create button in xhtml Image 21: display the button

The code is pretty straightforward, instead of using designed templated, they use commandButton in Basic Tag to create a button type, set it value as the parameter to reduce the amount of code if there are multiple files requires the same button. For delete a code it is very simple, all it takes is a single line at home.XHTML, and it will find all the matching value and comment them out, without creating any error. The syntax is: <ui:remove> <h:commandButton value = "Ok" /> </ui:remove>.

1. **Convertor and Validator Tag**

Since these 2 tags provide about the same implementation format, just a bit different in functionality and using purpose, it is fine to combine them and make it short, simple and clear. Take a look at the list of built-in function it provides and understands how it works.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Convertor Tags**   |  |  | | --- | --- | | **Tag Type** | **Tag Functionality** | | **<f:convertNumber> <f:convertNumber />** | convert a string into number format | | **<f:convertDateTime>**  **<f:convertDateTime />** | convert a string into a date/time format | | **Custom Tag** | create a custom tag for conversions | | **Validator Tags**   |  |  | | --- | --- | | **Tag Type** | **Tag Functionality** | | **<f:validateLength> <f:validateLength />** | check the length of incoming value | | **<f:validateLongRange> <f:validateLongRange />** | checking a range of an integer value | | **<f:validateDoubleRange> <f:validateDoubleRange />** | checking a range of a double value | | **<f:validateRegex>**  **<f:validateRegex />** | validates JSF components with regular expression | | **Custom Tag** | create a custom tag for validation | |

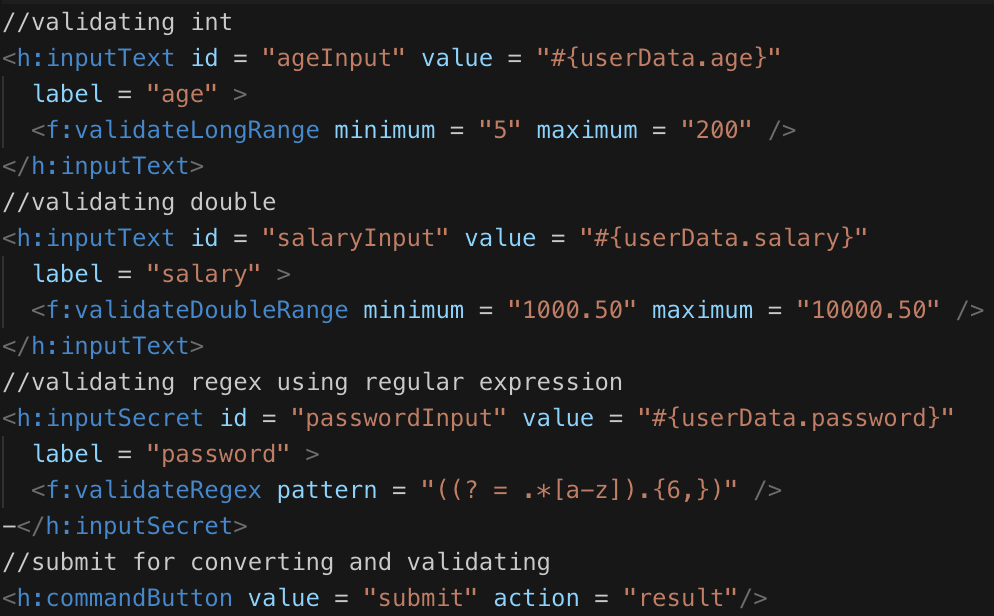
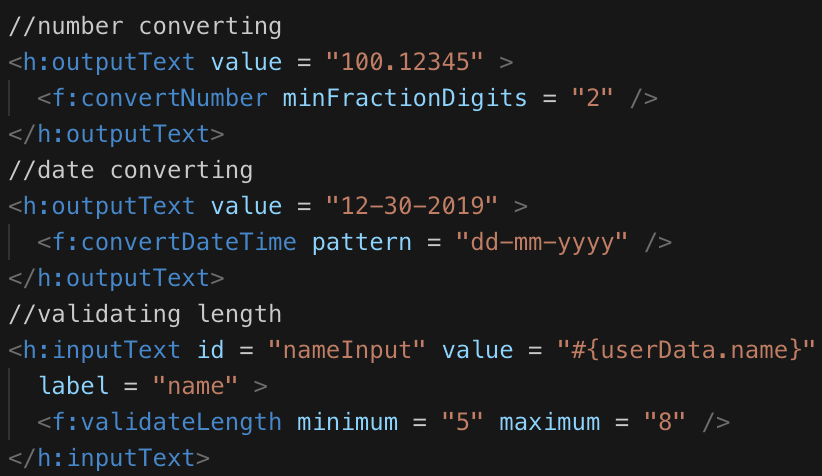


Image 22: Example of tag converting and validating (1) Image 23: Example of tag converting and validating (2)

Examples in image 22 and 23 have pretty much all information requires for converting and validating tag. With just a small change for the output format (ex: convertDateTime set new pattern into “mm-dd-yyyy" or validateRegex set new pattern to “({[a-zA-Z]})”), developers can use these tag to help them with the data right on the XHTML file, without have to transfer into the application file, convert it and return the new value back. The Converter Tag and Validator Tag have pretty much most of the convert and validate function to sufficiently help developers save their time and make the code more efficient. There is some rare case, require developers to create a custom converter or custom validator function to make a better conversion and stricter data coming in. However, since the topic of custom converter and validator is very broad and each type requires a different implementation, unfortunately, it is not possible to cover enough in this report. In the other hand, there is plenty of pre-built source code ready to be used for each custom type so take the powerful opportunity of the Internet and built the code.

**Conclusion:**

This is a pretty long report covering from an overview of Java EE, an introduction of both Java EE and JSF, explain the beneficial JSF could help developers as part of the team or large scale project. The report also went in depth about JSF supportive, its implementation and simplification. From going over and details about its life cycle, the managed bean used in Java application as server processing file and functional tags use in XHTML as a user interface to many good coding examples throughout the report to help new developers ready for their tasks. Another advantage for JSF is, it separates the server class and client class in 2 different places (in server sources and client machine, respectively) but when one side requires modification or change, it does not affect the other side's code. JSF is also easier to install than other similar technology. That last reason developer should consider JSF over other technology is the code syntax is pretty simple to understand and implement, which only requires a person with basic knowledge of Java+HTML skills to write this. This is in overall an option developer should choose for their project if the scale is very big, require a big team to work and each person has their own strength about back-end or front-end as well as knowledge level.

# 

# 

# 

**References:**

**Resources:**

* JSF Managed Beans - javatpoint. (n.d.). Retrieved April 12, 2019, from <https://www.javatpoint.com/jsf-managed-beans>
* O. (n.d.). Java Platform, Enterprise Edition (Java EE) 8. Retrieved April 12, 2019, from <https://javaee.github.io/firstcup/toc.html>
* O. (2013, January 01). The Java EE 6 Tutorial. Retrieved April 12, 2019, from <https://docs.oracle.com/javaee/6/tutorial/doc/bnaxf.html#giqwk>
* O. (2013, January 01). The Java EE 6 Tutorial. Retrieved April 12, 2019, from <https://docs.oracle.com/javaee/6/tutorial/doc/bnawq.html>
* Tutorialspoint.com. (n.d.). JSF Tutorial. Retrieved April 12, 2019, from <https://www.tutorialspoint.com/jsf>
* What is Java Server Faces (JSF) and jsf life cycle. (2012, April 16). Retrieved April 12, 2019, from <http://javawebaction.blogspot.com/2012/04/what-is-JSF-java-server-faces-and-jsf-life-cycle.html>

**Images:**

* O. (n.d.). Sun GlassFish Enterprise Server 2.1 Performance Tuning Guide. Retrieved from <https://docs.oracle.com/cd/E19316-01/820-4343/abeau/index.html>
* O. (2007, September 01). The Java EE 5 Tutorial. Retrieved from <https://docs.oracle.com/javaee/5/tutorial/doc/bnaay.html>
* G. (2018, February 08). MVC Design Pattern. Retrieved April 12, 2019, from <https://www.geeksforgeeks.org/mvc-design-pattern/>