AJAX Using Java

Onlinevarsity
your e-way to learning

Find these words

- REGISTER
- PRINT
- LEGAL
- PIRACY
- DOWNLOAD
- USER
- PIRATED

R	E	G	D	R	R	T	Y	K
W	D	A	O	L	E	G	A	L
F	C	K	W	O	G	B	P	W
Z	X	V	N	T	I	P	I	F
B	P	Q	L	U	S	E	R	Y
H	R	A	O	C	T	J	A	O
P	I	R	A	T	E	D	C	L
U	N	A	D	I	R	W	Y	X
S	T	V	U	L	A	P	D	M

Solution

R	E	G	D	R	R	T	Y	K
W	D	A	O	L	E	G	A	L
F	C	K	W	O	G	B	P	W
Z	X	V	N	T	I	P	I	F
B	P	Q	L	U	S	E	R	Y
H	R	A	O	C	T	J	A	O
P	I	R	A	T	E	D	C	L
U	N	A	D	I	R	W	Y	X
S	T	V	U	L	A	P	D	M

AJAX Using Java
Learner's Guide

APTECH LIMITED

Contact E-mail: ov-support@onlinevarsity.com

Second Edition - 2014

Aptech®
Unleash your potential

Aptech™
COMPUTER EDUCATION
Unleash your potential

Dear Learner,

We congratulate you on your decision to pursue an Aptech Worldwide course.

Aptech Ltd. designs its courses using a sound instructional design model – from conceptualization to execution, incorporating the following key aspects:

- Scanning the user system and needs assessment

Needs assessment is carried out to find the educational and training needs of the learner

Technology trends are regularly scanned and tracked by core teams at Aptech Ltd. TAG* analyzes these on a monthly basis to understand the emerging technology training needs for the Industry.

An annual Industry Recruitment Profile Survey# is conducted during August - October to understand the technologies that Industries would be adapting in the next 2 to 3 years. An analysis of these trends & recruitment needs is then carried out to understand the skill requirements for different roles & career opportunities.

The skill requirements are then mapped with the learner profile (user system) to derive the Learning objectives for the different roles.

- Needs analysis and design of curriculum

The Learning objectives are then analyzed and translated into learning tasks. Each learning task or activity is analyzed in terms of knowledge, skills and attitudes that are required to perform that task. Teachers and domain experts do this jointly. These are then grouped in clusters to form the subjects to be covered by the curriculum.

In addition, the society, the teachers, and the industry expect certain knowledge and skills that are related to abilities such as *learning-to-learn, thinking, adaptability, problem solving, positive attitude etc*. These competencies would cover both cognitive and affective domains.

A precedence diagram for the subjects is drawn where the prerequisites for each subject are graphically illustrated. The number of levels in this diagram is determined by the duration of the course in terms of number of semesters etc. Using the precedence diagram and the time duration for each subject, the curriculum is organized.

- Design & development of instructional materials

The content outlines are developed by including additional topics that are required for the completion of the domain and for the logical development of the competencies identified. Evaluation strategy and scheme is developed for the subject. The topics are arranged/organized in a meaningful sequence.

The detailed instructional material – Training aids, Learner material, reference material, project guidelines, etc.- are then developed. Rigorous quality checks are conducted at every stage.

- Strategies for delivery of instruction

Careful consideration is given for the integral development of abilities like thinking, problem solving, learning-to-learn etc. by selecting appropriate instructional strategies (training methodology), instructional activities and instructional materials.

The area of IT is fast changing and nebulous. Hence considerable flexibility is provided in the instructional process by specially including creative activities with group interaction between the students and the trainer. The positive aspects of web based learning –acquiring information, organizing information and acting on the basis of insufficient information are some of the aspects, which are incorporated, in the instructional process.

- Assessment of learning

The learning is assessed through different modes – tests, assignments & projects. The assessment system is designed to evaluate the level of knowledge & skills as defined by the learning objectives.

- Evaluation of instructional process and instructional materials

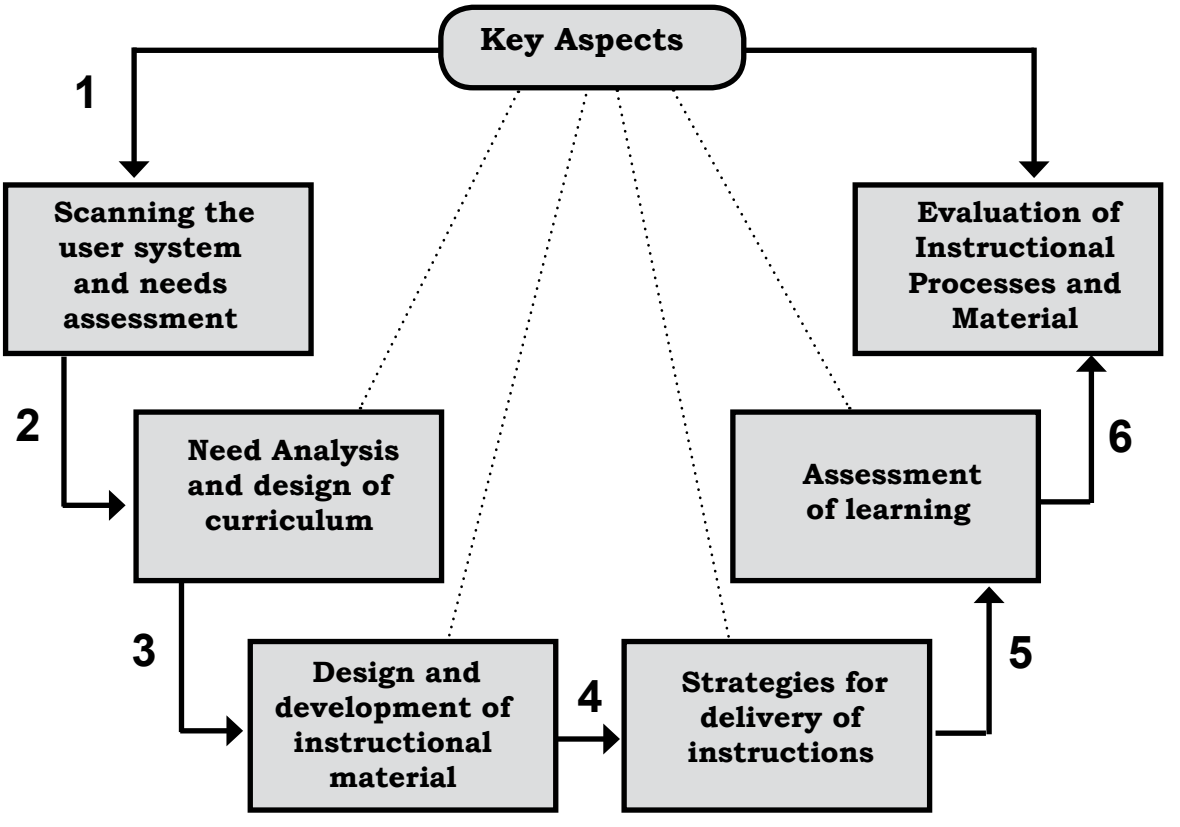
The instructional process is backed by an elaborate monitoring system to evaluate - on-time delivery, understanding of a subject module, ability of the instructor to impart learning. As an integral part of this process, we request you to kindly send us your feedback in the reply pre-paid form appended at the end of each module.

*TAG – Technology & Academics Group comprises of members from Aptech Ltd., professors from reputed Academic Institutions, Senior Managers from Industry, Technical gurus from Software Majors & representatives from regulatory organizations/forums.

Technology heads of Aptech Ltd. meet on a monthly basis to share and evaluate the technology trends. The group interfaces with the representatives of the TAG thrice a year to review and validate the technology and academic directions and endeavors of Aptech Ltd.

Industry Recruitment Profile Survey - The Industry Recruitment Profile Survey was conducted across 1581 companies in August/September 2000, representing the Software, Manufacturing, Process Industry, Insurance, Finance & Service Sectors.

Aptech New Products Design Model



“A little learning is a dangerous thing,
but a lot of ignorance is just as bad”

Preface

A new breed of Web applications are emerging recently that have better, faster and more user-friendly interfaces, almost as good as typical desktop interfaces. These Web applications do not need page reloading unlike traditional Web applications. These Web applications use AJAX extensively.

AJAX is not a new programming language, rather it is a new technique of doing same old things. It is a combination of JavaScript and XML. Besides these two technologies, several other technologies like Java, .NET and a host of open source technologies are used to get the AJAX effect in a traditional Web application.

This book starts with an overview of a typical 'AJAXified' Web application. It then familiarizes you with various AJAX features, technologies, toolkits and frameworks used in these new breed of Web applications.

This book is the result of a concentrated effort of the Design Team, which is continuously striving to bring you the best and the latest in Information Technology. The process of design has been a part of the ISO 9001 certification for Aptech-IT Division, Education Support Services. As part of Aptech's quality drive, this team does intensive research and curriculum enrichment to keep it in line with industry trends.

We will be glad to receive your suggestions.

Design Team

“Nothing is a waste of time if you use the experience wisely”

Table of Contents

Sessions

“Learning is not compulsory
but neither is survival”

Module - 1

Introduction to AJAX

Welcome to the module, **Introduction to AJAX**.

Asynchronous JavaScript and XML (AJAX) refers to a group of Web technologies. AJAX provides Web applications with rich user interface (UI), similar to desktop applications, and improves their response time as well. Reduced request-response time of Web applications makes the Web applications highly responsive.

In this module, you will learn about:

- ➔ Asynchronous JavaScript and XML
- ➔ Document Object Model

1.1 Lesson Overview

In the first lesson, **Asynchronous JavaScript and XML**, you will learn to:

- ➔ Explain the evolution of AJAX.
- ➔ Explain the working of AJAX.

1.1.1 Conventional Web Applications

Conventional Web applications work in a simple manner. When a user clicks a submit button, a link or so on, an HTTP request is triggered by the browser at the client's end. This HTTP request is transmitted to the server.

At the server's end, server-side components process the HTTP request and send the results back to the client browser. On receiving the results from the server, the client browser displays the results after refreshing the Web page.

This mode of working is termed as "click, wait and refresh", as the user clicks, waits and then views the results after the browser refreshes the Web page. The "click, wait and refresh" concept makes its easy to develop Web applications. However, this ease in development comes with a very great price, high request-response latency periods. High request-response latency periods lead to slow responsiveness.

Figure 1.1 shows the conventional Web application model.

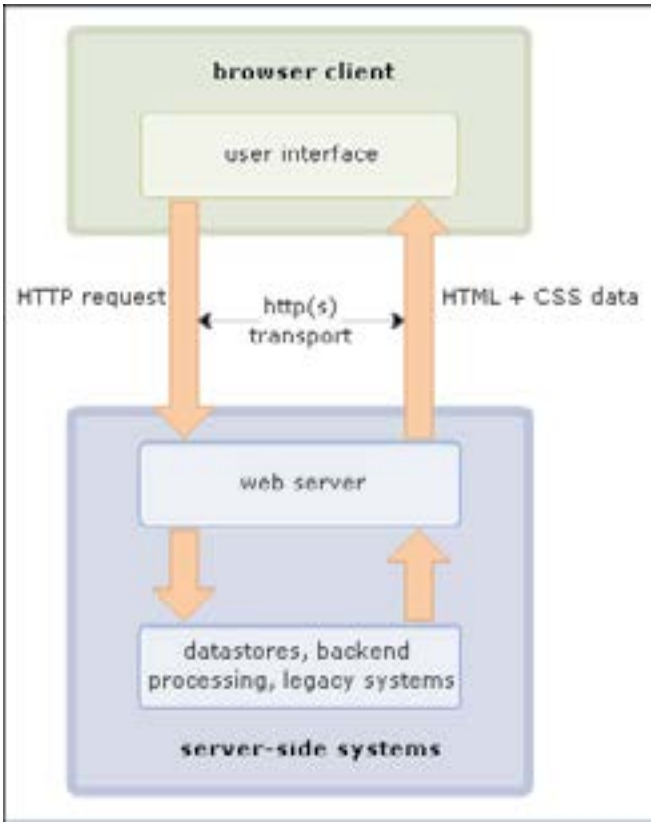


Figure 1.1: Conventional Web Application Model

1.1.2 Drawbacks of Conventional Web Applications

Conventional Web applications communicate with the server synchronously. Once an HTTP request is generated, the user can no longer interact with the application. The user is forced to stay idle while the browser refreshes the Web page to display the results retrieved from the server. The page refresh also means that the existing browser contents too vanish, making the application completely unreadable. This synchronous mode of communication causes the long idle spells.

In addition to the long recurring waiting periods, conventional Web applications also lack rich user interface. This is because Web technologies such as servlets and JSP hardly provide any options to enhance the appearance of standard HTML components such as buttons, links, labels, and so on.

Thus, long waiting periods due to synchronous communication and poor user experience due to the lack of rich user interface are the major drawbacks of conventional Web applications.

Figure 1.2 shows the synchronous model for conventional Web application.

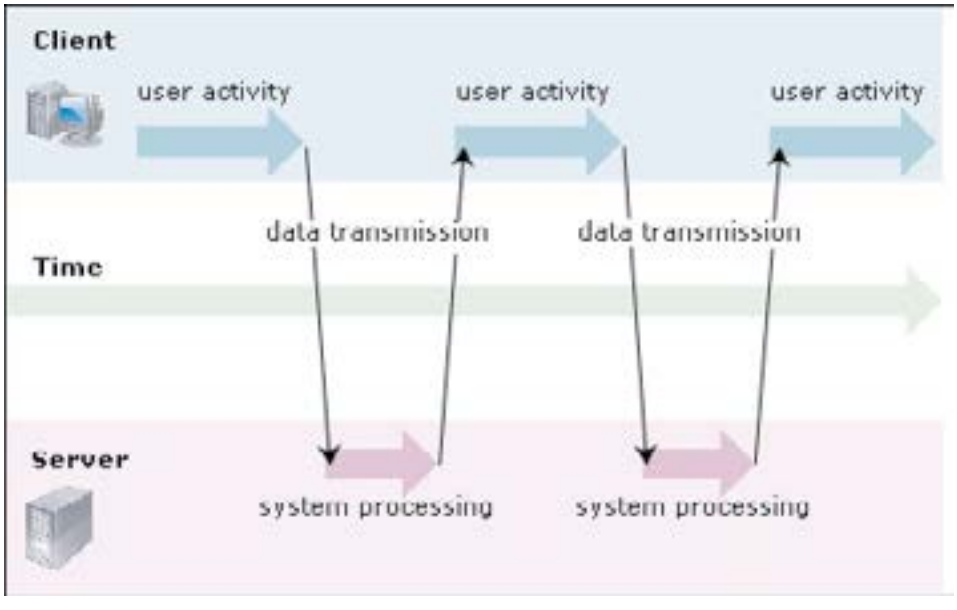


Figure 1.2: Synchronous Model for Conventional Web Application

1.1.3 Birth of AJAX

The aim to overcome the conventional Web application's drawbacks forced Web developers to look for alternative Web development methods. Developers overcame the drawbacks using technologies like JavaScript, XML, DOM, CSS, and so on.

Web developers all over the world adapted to this trend of using disparate technologies. However, Jesse James Garrett was the first to talk about them. In his article, "Ajax: A New Approach to Web Applications", Garrett listed the various technologies involved, discussed their roles and explained how they worked together. Garrett named this group of technologies as Asynchronous JavaScript and XML (AJAX). Garrett's article marked the birth of AJAX.

Figure 1.3 shows the technologies used in AJAX Web application.

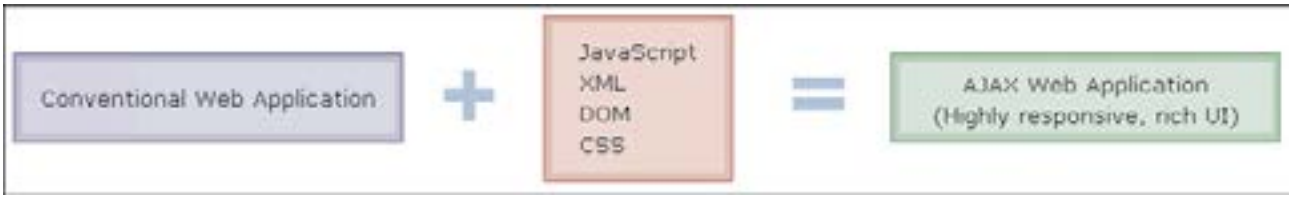


Figure 1.3: Technologies Used in Ajax Web Application

1.1.4 AJAX Technologies

AJAX comprises JavaScript, XML, DOM and CSS. These technologies enable you to develop highly responsive and rich UI-based Web applications.

➔ **JavaScript**

JavaScript facilitates the creation of `XMLHttpRequest` objects. `XMLHttpRequest` objects facilitate asynchronous communication between the client and the server. Asynchronous communication permits the user to continue interacting with the Web page on the client-side while the `XMLHttpRequest` object retrieves data from the server. Thus, the user does not experience a page refresh.

➔ XML

The server-side components process the client request and send a corresponding response back to the client. The response contains the data requested by the client. This data is sent to the client in XML format.

➔ DOM

DOM performs dual role in AJAX. Firstly, DOM allows you to parse the XML response received from the server and extract the data from it. Secondly, it allows you to access the Web page's DOM tree to add or update existing nodes with new data received from the server. In other words, DOM facilitates the update of a Web page.

➔ **CSS**

CSS enables you to add desktop applications-like look-and-feel to Web applications.

1.1.5 Working of AJAX

Consider the Web page of an AJAX-enabled Web application. A user needs to provide a credit card number. The application validates the credit card number asynchronously using the ValidateCCN servlet and a Web service on the server. Based on the validity of the number, a corresponding message is displayed next to the text box.

Figure 1.4 shows the various steps involved in processing the AJAX request and getting the response from the server.

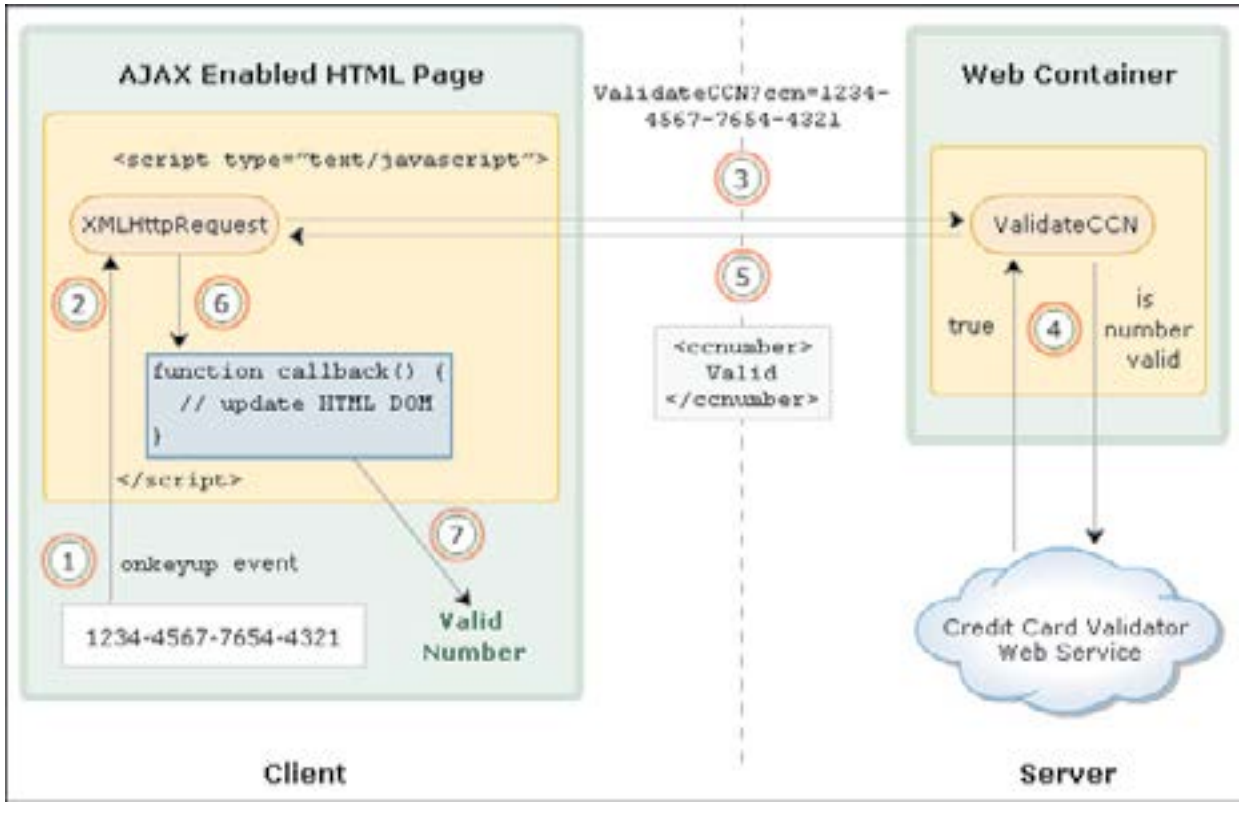


Figure 1.4: AJAX Request and Response Model

➔ **Step 1**

The input text box is associated to an event handler with the help of the `onkeyup` event. Thus, every time the user types a digit in the text box, the `onkeyup` event occurs. Every time the `onkeyup` event occurs, an event handler is invoked.

➔ **Step 2**

The event handler is a JavaScript function that creates an `XMLHttpRequest` object and configures it using the `open()` method. The `open()` method specifies the HTTP method (`GET` or `POST`), the URL of the server-side component that processes the request, and the mode (synchronous or asynchronous) of communication. Note that the credit card number is included as a request parameter in the HTTP request.

➔ **Step 3**

The `send()` method of the `XMLHttpRequest` object is then invoked. This establishes a connection with the server-side component such as a servlet or a JSP page. In this case, this component is a servlet. Therefore, the servlet whose URI is mapped to `ValidateCCN` is executed.

➔ **Step 4**

The processing of AJAX begins. The servlet first retrieves the credit card number from the request object. This number is then sent to a credit card validator Web service. This service verifies the credit card number and accordingly intimates the servlet.

➔ **Step 5**

The servlet then generates an XML response. This XML response is an XML document containing the element `ccnumber` with the text **`Valid`** or **`Invalid`** in it. If the credit card number is valid, the text **`Valid`** is enclosed in the `ccnumber` element. This XML response is sent to the client.

➔ **Step 6**

After the client receives the XML response, a callback function is called. The callback function is usually specified in step 2 while configuring the `XMLHttpRequest` object. The XML response sent by the server is accessible through the `responseXML` property of the `XMLHttpRequest` object.

➔ **Step 7**

The callback function displays a message on the Web page about the validity of credit card number. This is achieved by reserving a div element specifically for displaying such a message. The callback function retrieves this `div` element using DOM API and sets its `innerHTML` property to display the message.

Knowledge Check 1

1. Which of the following statements about the evolution of AJAX are true?

(A)	The “c+lick, wait and refresh” model was slow and hampered user interactivity to some limit.
(B)	Conventional Web applications, while communicating with the Web server, rendered the Web page useless.
(C)	Refreshing the entire page to display request results, added to higher request-response latency.
(D)	Synchronous communication provided fast responses but fared badly when it came to providing rich user interactivity.
(E)	Asynchronous communication is slower but less complicated than synchronous communication.

(A)	A, B, and C	(C)	B, C, and D
(B)	C, D, and E	(D)	B, D, and E

2. Which of the following statements about the working of AJAX are true?

(A)	The callback function is specified in the responseXML property with the help of the send() method of XMLHttpRequest object.
(B)	A client event on the Web page invokes an event handler that updates the Web page instantaneously.
(C)	The callback function is called only after successful receipt of XML response from the server.
(D)	The XMLHttpRequest object's send() method invokes the server-side component that processes the AJAX request.
(E)	The callback function uses DOM to display data on a Web page.

(A)	A, B, and C	(C)	B, C, and D
(B)	C, D, and E	(D)	B, D, and E

1.2 Lesson Overview

In the last lesson, **Document Object Model**, you will learn to:

- ➔ Explain the role of DOM in AJAX.
- ➔ Explain the various HTML DOM methods and properties.
- ➔ Explain the various XML DOM methods and properties.

1.2.1 Document Object Model

DOM is a standard object model. DOM allows you to access and manipulate HTML and XML document content. Thus, DOM facilitates dynamic modification of Web pages.

DOM represents HTML or XML document as a collection of objects referred as nodes. Based on how these objects are placed in the document, DOM connects each of these nodes creating a tree like structure. DOM classifies every node as a specific type of node. For example, tags are classified as element nodes where as text within the tags is classified as text nodes.

Consider the **`Example.html`** file shown in figure 1.5. It has six lines of code. html is the main tag. It has two sub tags, title and body respectively. title contains text while body consists of a sub tag, h1. Again, h1 contains text. Using these observations a tree structure can be created as shown in figure 1.5.

Figure 1.5 shows the tree structure for **`Example.html`** file.

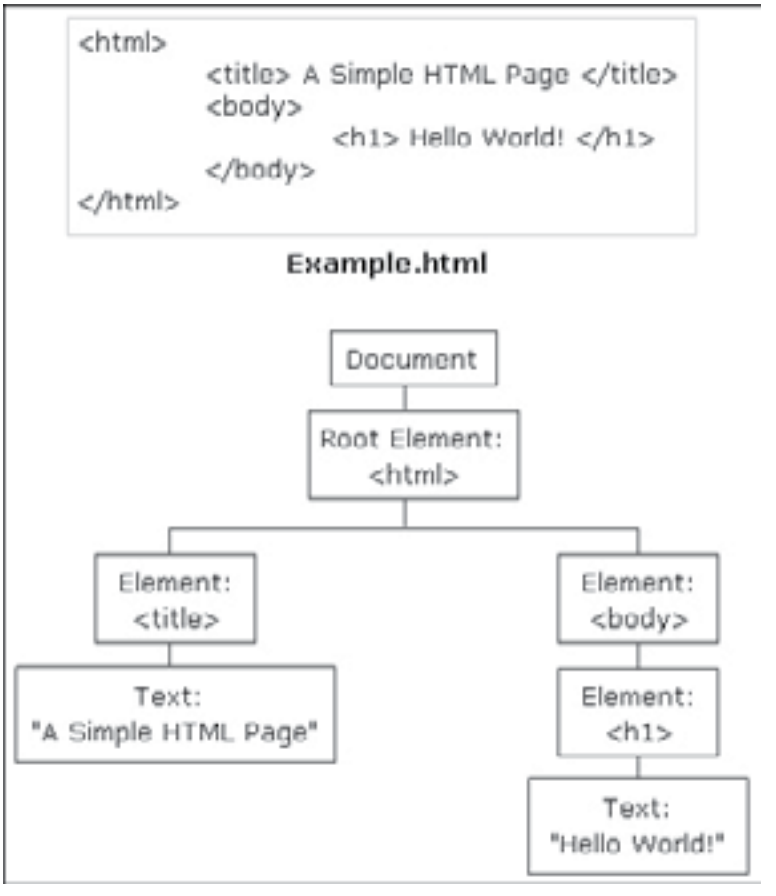


Figure 1.5: Tree Structure of example.html

1.2.2 HTML DOM Methods

HTML Document Object Model (HTML DOM) allows you to access and manipulate HTML documents using built-in methods. These HTML DOM methods are as follows:

➔ **getElementById(id)**

The `getElementById()` method searches for and returns the element whose id is specified as the input parameter.

Syntax:

```
getElementByID(id)
```

where,

`id` - value of the id attribute of an element in an HTML document

Concepts

The code shows the use of the `getElementById()` method.

Code Snippet:

```
<script type="text/javascript">
 x = document.getElementById("one");
 txt = x.innerHTML;
 document.write("Value of p element with id = one: " + txt);
 document.write("<br />");
</script>
```

The code snippet when added to **`Example.html`** results in the following output:

```
Hello World 1!
Hello World 2!
Hello World 3!
Hello World 4!
```

Value of p element with id = one: Hello **World** 1!

➔ **getElementsByTagName(name)**

The `getElementsByTagName()` method searches for and returns all such elements whose name matches to the one specified as the input parameter. It returns the entire list of elements in the form of an array.

Syntax:

```
getElementsByTagName(name)
```

where,

`name` – name of a tag in an HTML document

The code shows the use of the `getElementByTagName()` method.

Code Snippet:

```
<script type="text/javascript">
     x = document.getElementsByTagName("p");
     for (i=0;i < x.length; i++) {
        document.write(x[i].innerHTML);
        document.write("<br />");
     }
</script>
```

The code snippet when added to **`Example.html`** results in the following output:

```
Hello World 1!
Hello World 2!
Hello World 3!
Hello World 4!
Hello World 1!
Hello World 2!
Hello World 3!
Hello World 4!
```

➔ **appendChild(node)**

The `appendChild()` method appends the node specified as input parameter to the tree structure.

Syntax:

```
appendChild(node)
```

where,

`node` - the exact node that needs to be appended

The code shows the use of the `appendChild()` method.

Code Snippet:

```
<script type="text/javascript">
    x=document.getElementsByTagName("p");
    x[0].appendChild(x[3].childNodes[1]);
</script>
```

The code snippet when added to **`Example.html`** results in the following output:

```
Hello World 1! World 4!
Hello World 2!
Hello World 3!
Hello
```

➔ **removeChild(node)**

The `removeChild()` method removes the node specified as input parameter from the tree structure.

Syntax:

```
removeChild(node)
```

where,

`node` – the exact node that needs to be removed

The code shows the use of the `removeChild()` method.

Code Snippet:

```
<script type="text/javascript">
     x=document.getElementsByTagName("p");
     x[2].removeChild(x[2].childNodes[1]);
</script>
```

The code snippet when added to **`Example.html`** results in the following output:

```
Hello World 1!
Hello World 2!
Hello
Hello World 4!
```

1.2.3 HTML DOM Properties

HTML DOM properties provide information about the various nodes in a DOM tree. You can use these properties to access the information stored within the nodes as well.

The HTML DOM object properties are:

➔ **innerHTML**

The `innerHTML` property provides access to the content of an element. Therefore, you can use this property to retrieve or update the text in an element.

The code shows the use of the `innerHTML` property.

Code Snippet:

```
<script type="text/javascript">
     x=document.getElementById("one");
     document.write(x.innerHTML);
</script>
```

The code snippet when added to **`Example.html`** results in the following output:

```
Hello World 1!
Hello World 2!
Hello World 3!
Hello World 4!
Hello World 1!
```

➔ nodeName

The `nodeName` property provides access to the name of the current node. It is used to display the name of the current node.

The code shows the use of the `nodeName` property.

Code Snippet:

```
<script type="text/javascript">
     x=document.getElementById("one");
     document.write(x.nodeName);
</script>
```

The code snippet when added to **`Example.html`** results in the following output:

```
Hello World 1!
Hello World 2!
Hello World 3!
Hello World 4!
P
```

➔ nodeValue

The `nodeValue` property can be used to display a node's content. The nodeValue property when used with text nodes returns the nodes content. The `nodeValue` property when used with element nodes returns the value "null".

The code shows the use of the nodeValue property:

Code Snippet:

```
<script type="text/javascript">
     x=document.getElementById("one");
     document.write(x.nodeValue);
</script>
```

The code snippet when added to **`Example.html`** results in the following output.

```
Hello World 1!
Hello World 2!
Hello World 3!
Hello World 4!
null
```

The `p` node has null value. However, it does have two child nodes: a text node and another element node.

➔ parentNode

The `parentNode` property provides access to the parent node of the current node.

Code Snippet:

```
<script type="text/javascript">
    x=document.getElementById("one");
    document.write(x.parentNode.nodeName);
</script>
```

The code snippet when added to **`Example.html`** results in the following output:

```
Hello World 1!
Hello World 2!
Hello World 3!
Hello World 4!
BODY
```

➔ childNodes

The `childNodes` property provides access to all the child nodes of the given node. You can access the child nodes by using prefixes. For example, `childNodes[1]` will return the second child node of the current node.

The code shows the use of the `childNodes` property.

Code Snippet:

```
<script type="text/javascript">
    x=document.getElementsByTagName("p");
    document.write(x[2].childNodes[0].nodeValue);
</script>
```

The code snippet when added to **`Example.html`** results in the following output:

```
Hello World 1!
 Hello World 2!
 Hello World 3!
 Hello World 4!
 Hello
```

➔ **length**

The `length` property returns the number of nodes present in a node list. The `getElementsByTagName()` method returns a list of all the nodes having a specific tag name. The length property returns the total number of nodes present in the list.

The code shows the use of the `length` property.

Code Snippet:

```
<script type="text/javascript">
     x = document.getElementsByTagName("p");
     document.write(x.length);
</script>
```

The code snippet when added to Example.html (as shown in the image) results in the following output:

```
Hello World 1!
Hello World 2!
Hello World 3!
Hello World 4!
4
```

1.2.4 XML DOM Methods

XML Document Object Model (XML DOM) allows you to access and manipulate XML documents using methods similar to that of HTML DOM.

Figure 1.6 shows an XML document.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
- <students>
  - <record id="1">
      <name>Ashley Andrews</name>
      <gender>male</gender>
      <age>12 yrs</age>
      <grade>7th</grade>
      <division>A</division>
    </record>
  - <record id="2">
      <name>Ashley Mathias</name>
      <gender>male</gender>
      <age>5 yrs</age>
      <grade>1st</grade>
      <division>B</division>
    </record>
  - <record id="3">
      <name>Ashley Waters</name>
      <gender>male</gender>
      <age>6 yrs</age>
      <grade>1st</grade>
      <division>A</division>
    </record>
  </students>
```

Figure 1.6: XML Document

The XML DOM methods are:

➔ **getElementsByTagName(name)**

The `getElementsByTagName()` method searches for and returns elements whose name matches with the input parameter.

Syntax

```
getElementsByTagName(name)
```

`where,`

`name` – name of a tag in an XML document

The code shows the use of the `getElementByTagName()` method.

Code Snippet:

```
<script type="text/javascript">
    xmlDoc=new ActiveXObject("Microsoft.XMLDOM");
    xmlDoc.async="false";
```

Concepts

```
    xmlDoc.load("Student.xml");
    x=xmlDoc.getElementsByTagName("name");
    for (i=0;i<x.length;i++) {
        document.write(x[i].childNodes[0].nodeValue);
        document.write("<br />");
    }
</script>
```

In accordance with the XML content shown on screen, the script generates the following result:

```
Ashley Andrews
Ashley Mathias
Ashley Waters
```

➔ **appendChild(node)**

The `appendChild()` method appends the node specified as input parameter to the tree structure.

Syntax

```
appendChild(node)
```

where,

`node` – the exact node that needs to be appended

The code shows the use of the `appendChild()` method.

Code Snippet :

```
<script type="text/javascript">
    xmlDoc=new ActiveXObject("Microsoft.XMLDOM");
    xmlDoc.async="false";
    xmlDoc.load("Student.xml");

    x=xmlDoc.getElementsByTagName("name");
    x[0].appendChild(x[1].childNodes[0]);
    document.write(x[0].childNodes[0].nodeValue);
    document.write(x[0].childNodes[1].nodeValue);
</script>
```

➔ removeChild(node)

The `removeChild()` method removes the node specified as input parameter from the tree structure.

Syntax

```
removeChild(node)
```

where,

`node` – the exact node that needs to be removed

The code shows the use of the `removeChild()` method.

Code Snippet :

```
<script type="text/javascript">
     xmlDoc=new ActiveXObject("Microsoft.XMLDOM");
     xmlDoc.async="false";
     xmlDoc.load("Student.xml");
     x=xmlDoc.getElementsByTagName("name");
     x[0].removeChild(x[1].childNodes[0]);
     document.write(x[0].childNodes[0].nodeValue);
</script>
```

In accordance with the XML content shown on screen, the script generates the following result:

```
A Runtime Error has occurred.
Do you wish to Debug?
Line: 8
Error: The parameter Node is not a child of this Node.
Yes or No?
```

This clearly indicates the node has been removed.

Note - Unlike HTML DOM, XML DOM does not support the `getElementById()` method.

1.2.5 XML DOM Properties

XML DOM nodes possess a string of properties. These properties enable you to extract data from XML document.

The XML DOM object properties are:

➔ **nodeName**

The `nodeName` property provides access to the name of the current node. It is widely used to display the name of the current node.

The code shows the use of the `nodeName` property.

Code Snippet :

```
<script type="text/javascript">
     xmlDoc = new ActiveXObject("Microsoft.XMLDOM");
     xmlDoc.async = "false";
     xmlDoc.load("Student.xml");
     x = xmlDoc.getElementsByTagName("name");
     document.write(x[0].nodeName);
</script>
```

In accordance with the XML content shown on screen, the script generates the following result:

```
name
```

➔ **nodeValue**

The `nodeValue` property can be used to display a node's value. Not all nodes have values. For example, an element node could either have a text node or another element node as its child node. In either case, the element node will always have a value of null.

The code shows the use of the `nodeValue` property.

Code Snippet :

```
<script type="text/javascript">
     xmlDoc = new ActiveXObject("Microsoft.XMLDOM");
     xmlDoc.async = "false";
     xmlDoc.load("Student.xml");
     x = xmlDoc.getElementsByTagName("name");
     document.write(x[2].childNodes[0].nodeValue);
</script>
```

In accordance with the XML content shown on screen, the script generates the following result:

```
Ashley Waters
```

➔ **parentNode**

The `parentNode` property provides access to the parent node of the current node. You can use the `parentNode` property to access or gain control of the current node's parent.

The code shows the use of the `parentNode` property.

Code Snippet :

```
<script type="text/javascript">
     xmlDoc=new ActiveXObject("Microsoft.XMLDOM");
     xmlDoc.async="false";
     xmlDoc.load("Student.xml");
     x=xmlDoc.getElementsByTagName("name");
     y=x[0];
     document.write(y.parentNode.nodeName);
</script>
```

In accordance with the XML content shown on screen, the script generates the following result:
`record`

➔ **childNodes**

The `childNodes` property provides access to all the child nodes of the given element. It returns a list of all the child nodes present. You can access the various child nodes using prefixes. For example, the second child node is accessed using expression `childNodes[1]`.

The code shows thw use of the `childNodes` property.

Code Snippet :

```
<script type="text/javascript">
     xmlDoc=new ActiveXObject("Microsoft.XMLDOM");
     xmlDoc.async="false";
     xmlDoc.load("Student.xml");
     x=xmlDoc.getElementsByTagName("name");
     document.write(x[2].childNodes[0].nodeValue);
</script>
```

In accordance with the XML content shown on screen, the script generates the following result:

```
Ashley Waters
```

➔ length

The `length` property returns the number of nodes present in a node list. The `getElementsByTagName()` method returns a list of all the nodes having a specific tag name. The length property returns the total number of nodes present in the list.

The code shows the use of the `length` property.

Code Snippet :

```
<script type="text/javascript">
     xmlDoc = new ActiveXObject("Microsoft.XMLDOM");
     xmlDoc.async = "false";
     xmlDoc.load("Student.xml");
     x = xmlDoc.getElementsByTagName("name");
     for(i = 0; i < x.length; i++) {
         document.write(x[i].childNodes[0].nodeValue);
     }
</script>
```

In accordance with the XML content shown on screen, the script generates the following result:

```
Ashley AndrewsAshley MathiasAshley Waters
```

➔ attributes

The `attributes` property returns a list containing the specified node's attributes. If the specified node is not an element, this property returns NULL.

The code shows the use of the `attributes` property.

Code Snippet :

```
<script type="text/javascript">
     xmlDoc = new ActiveXObject("Microsoft.XMLDOM");
     xmlDoc.async = "false";
     xmlDoc.load("Student.xml");
     x = xmlDoc.getElementsByTagName("record")[0].attributes;
     attri = x.getNamedItem("id");
     document.write(attri.value);
</script>
```

In accordance with the XML content shown on screen, the script generates the following result:

1

Knowledge Check 2

1. Which of the following statements about DOM are true?

(A)	DOM is a language-dependent standard object model that represents HTML or XML documents.
(B)	DOM creates the tree structure based on how the document objects are connected to each other.
(C)	AJAX relies on asynchronous communication and DOM equally to achieve high responsiveness.
(D)	Asynchronous communication enables partial refreshing while DOM retrieves the results.
(E)	DOM provides methods and properties to edit the content of Web pages.

(A)	A, B, and C	(C)	B, C, and E
(B)	C, D, and E	(D)	A, D, and E

Summary

In the module, **Introduction to AJAX**, you learnt about:

➔ Asynchronous JavaScript and XML

Synchronous communication between the client and server introduced long delays in the request-response cycle. Technologies like CSS, JavaScript, DOM and XML provided a solution. Not only did they reduce the request-response latency but also lead the way for the development of Web applications that sported desktop application features.

➔ Document Object Model

The Document Object Model is a platform and language-independent standard for representing HTML and XML documents. It creates tree-like structures of objects present in HTML or XML documents. It then classifies these objects as nodes. Each node has properties. Using these properties and some methods you can update HTML documents and extract data from XML documents.

Module - 2

Using Dojo Toolkit

Welcome to the module, **Using Dojo Toolkit**. Dojo is a JavaScript-based toolkit designed to develop AJAX-based applications. It provides a huge set of widgets to design Web pages. These widgets use Dojo styles and themes to give a rich look and feel to Web applications. It includes a JavaScript library to send and receive data asynchronously. In short, it enables rapid development of AJAX applications with minimal JavaScript code.

- Dojo Toolkit
- Dojo Widget Library (Dijit)

2.1 Lesson Overview

In the first lesson, **Dojo Toolkit**, you will learn to:

- Explain Dojo.
- Explain the architecture and working of Dojo.
- Explain the steps to create a button widget using the Dojo toolkit.

2.1.1 Dojo Toolkit

Dojo is an open source JavaScript toolkit. Alex Russell, Dylan Schiemann, David Schontzler, and others started working on Dojo in the year 2004. Later, the open source community joined to improve it.

You can use Dojo to add rich look and feel to Web pages. Additionally, you can enable asynchronous communication using the built-in libraries of Dojo. There are several JavaScript-based toolkits. However, Dojo outperforms them for the following reasons:

- **Quality**

  Dojo widgets use the combination of CSS, DOM, and JavaScript to create a rich User Interface (UI). Therefore, these widgets deliver same output quality across all browsers.

- **Performance**

  Dojo provides tools to handle high-traffic sites. Using Dojo, you can manage big projects without making any change in the code. Dojo also helps in the creation of a custom toolkit that can provide performance similar to the actual toolkit.

- **Community**

  Dojo is an open source community, and hence many organizations and individuals contribute in improving it. The goal of the toolkit is to be as simple as possible so that the end users can use it with ease.

2.1.2 Features of Dojo

The five main features of Dojo that help in designing AJAX-based applications are:

- **Widget**

  Dojo provides many widgets such as menus, trees, tabs, tool tips, date selector, time selector, and so on for designing Web pages. You can create Dojo widgets by using JavaScript, HTML, and CSS style declarations.

➔ **Asynchronous communication**

AJAX applications send and receive data from a server asynchronously by using the `XMLHttpRequest` object. However, you have to write a lot of JavaScript code to implement this functionality. Dojo provides an abstract wrapper method called `dojo.xhrGet()` that allows you to exchange data asynchronously using minimal code.

➔ **Packaging System**

The packaging system in Dojo allows you to list the packages that need to be imported for the application. Thus, there is no need to include a script tag for every script file that is to be loaded. The packaging system ensures that the required package is loaded using the `dojo.require()` function.

➔ **Client-side data storage**

Dojo provides a feature called Dojo Storage for storing client-side data. This feature allows Web applications and existing Web browsers, such as Internet Explorer and Firefox to store data on the client-side securely.

➔ **Server-side data storage**

Dojo implements server-side data storages such as CsvStore, OpmlStore, YahooStore, DeliciousStore, and RdfStore to store data. CsvStore reads tabular data from comma-separated files whereas, OpmlStore reads hierarchical data from OPML format files. YahooStore and DeliciousStore fetch search results from the Yahoo Search Web service and del.icio.us Web service respectively.

2.1.3 Benefits of Dojo

There are many Dojo toolkits available in the market such as script.aculo.us Prototype, MochiKit, and so on. Some of the benefits of using Dojo toolkit are:

➔ **Code Simplification**

Dojo provides wrappers that encapsulate all the functionality required to send and receive AJAX requests. Dojo also handles cross browser incompatibility issues.

➔ **Reusable Code**

Dojo supports reusability of code. In other words, you can use dojo code of one application in another application. Such reusability of code allows you to add new functionalities to existing applications with minimal effort.

➔ **Portable Tools**

Dojo provides several widgets for Web page authors and designers. You can add new functionalities to existing Web pages by using Dojo widgets. A page author need not learn additional programming language to use the Dojo widgets.

Note - Disadvantage of Dojo is that a developer has to depend on the browser support for the Dojo. In addition, there is no way to hide Dojo code in commercial applications.

2.1.4 Architecture of Dojo

Dojo provides a set of standard libraries that are arranged in layers, one above the other.

The Packaging System is the third and the last layer in the hierarchy. This layer helps you to customize distribution of Dojo and develop functionalities using modules, resources, and widget namespaces. Dojo code is divided into logical units called modules. Dojo modules are defined in JavaScript files. The JavaScript files are called resources. A module is usually defined in a single JavaScript file, but sometimes a module definition is split into multiple JavaScript files. Widgets are combined into groups called namespaces. You can create new namespaces and put custom widgets in these namespaces.

The Dojo Event System is the second layer that contains language libraries to process Web application events. These events allow widgets to interact with each other. The Language Utilities is the first layer that improves and simplifies the coding technique of JavaScript developers while developing Web sites.

Figure 2.1 shows the architecture of Dojo.

Figure 2.1: Architecture of Dojo

2.1.5 Dojo Programming Model

Dojo comprises two programming models, namely Declarative model and Programmatic model. The image shows the declarative and programmatic approach of creating widgets.

In declarative model, the Button widget is instantiated declaratively using the tags such as button. The `dojoType` attribute instructs Dojo to render a button on the Web page. The `id` attribute assigns a unique identifier to the widget.

In Programmatic model, the widget `class`, `style`, and `id` are passed as parameters to the constructor. The `btnExit` variable will refer to the instantiated widget. The first parameter refers to the `Button` class that is used by Dijit to initialize the `Button` widget's properties. The second parameter shows the label of the button. The third parameter uses the `dijit.byId()` function to refer the `Exit` button by its `id` name.

Figure 2.2 shows the creation of a Button widget declaratively and programmatically.

Creating a Button widget declaratively:

```
<button dojoType="dijit.form.Button" id="btnExit">
Exit
</button>
```

Creating a Button widget programmatically:

```
var btnExit = new dijit.form.Button('Button',
({label : 'Exit'}), dijit.byId("btnExit"));
```

Figure 2.2: Creation of a Button Widget

2.1.6 Setting Up Dojo

To use Dojo toolkit in Web applications, the first step is to copy the Dojo toolkit libraries to Web application's root folder. The first image shows the Dojo toolkit included in the Web application named `DojoWidgets`. The Dojo toolkit comprises the modules `dojo`, `digit`, `dojox`, and `util` as shown in figure 2.3.

Figure 2.3 shows the modules of the Dojo toolkit.

Figure 2.3: Modules of Dojo Toolkit

Figure 2.4 shows the steps to set up and load Dojo in a Web page.

```
...
<!-- Step 1 -->
<style type="text/css">
    @import "./dojo/resources/dojo.css";
    @import "./dijit/themes/tundra/tundra.css";
</style>

<!-- Step 2 -->
<script type="text/javascript" src="./dojo/dojo.js">
</script>
...
```

Figure 2.4: Steps to Set Up and Load Dojo

The first step imports the cascading style sheet files named `dojo.css` and `tundra.css` by using the `@import tag`. The `dojo.css` file applies Dojo style, whereas, `tundra.css` applies the Tundra theme to the widgets. You can also apply other themes by importing the respective `.css` file.

The second step loads the `dojo.js file`. This file contains the base Dojo script for searching or manipulating DOM, binding events, sending AJAX requests, and applying basic effects. This file is located in the dojo subdirectory.

2.1.7 Creating a Button Widget

The Dojo Widget library, also known as Dijit, contains several widgets for designing Web pages. The most commonly used widget is a button. Figure 2.5 shows the code to create a `Button` widget with the caption `Hello Dojo`!

```
...
<script type="text/javascript" src="./dojo/dojo.js">
  dojo.require("dijit.form.Button");
</script>
...
<body class="tundra">
  <button dojoType="dijit.form.Button">Hello Dojo!
  </button>
</body>
...
```

Output

Hello Dojo!

Figure 2.5: Button Widget with a Caption

To create a Button widget, you first load the Button widget module by using the `dojo.require()` function. In the body section of the HTML document, you create a `Button` widget by using the `button` tag. The `dojoType` attribute instructs Dojo about the type of the widget to be displayed.

2.1.8 Connecting an Event to the Button Widget

Events in JavaScript or Dojo-based applications enable you to add interactivity to Web pages. Figure 2.6 shows the code to implement event handling for a Button widget.

```
...
<body class="tundra">
  <button dojoType="dijit.form.Button"
    id="btnButton">Hello Dojo!
    <script type="dojo/method" event="onclick">
      alert('Welcome to the world of Dojo!!!');
    </script>
  </button>
</body>
...
```

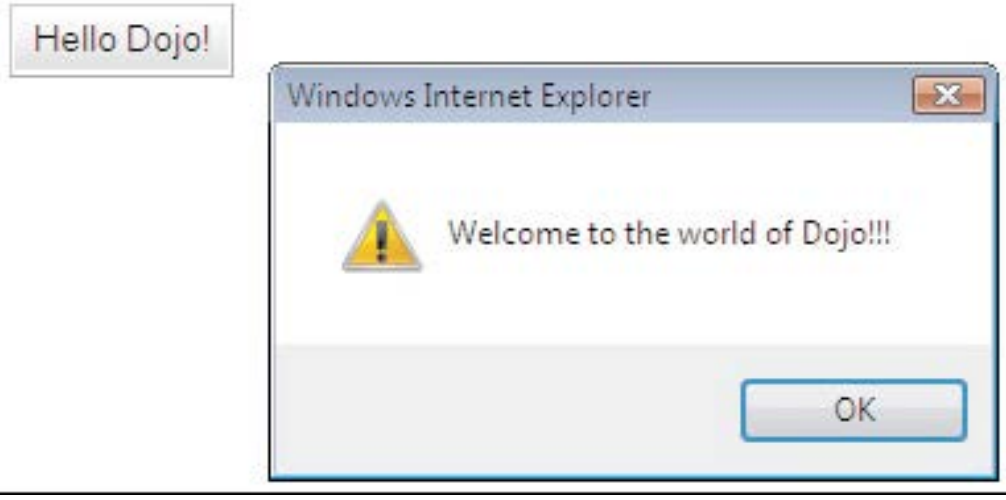


Figure 2.6: Event Handling for a Button Widget

In Dojo, you connect an event handler to a widget through a `script` tag. Note that the script tag is written within the `button` tag. Additionally, the `type` attribute of the `script` tag uses `dojo/method` to indicate that the `script` is a method. This method is bound to the `onclick` event using the `event` attribute. Therefore, when the button is clicked, an alert message is displayed on the page.

Knowledge Check 1

1. Which of the following statements about Dojo are true?

(A)	Dojo is an open source Javascript toolkit designed to develop AJAX-based applications.
(B)	Dojo widgets are a combination of only HTML and CSS style declarations.
(C)	Dojo allows storage of data on the client and the server using server-side data store implementations.
(D)	The Web page author need not learn additional programming language to use the Dojo widgets.
(E)	The Dojo code created for one application can be used in another application.

(A)	A, B, and C	(C)	B, C, and E
(B)	C, D, and E	(D)	A, C, and D

2. Which of the following statements about Dojo architecture and its working are false?

(A)	Dojo provides a set of three layered libraries namely, the packaging system, the event system and the language utilities layer.
(B)	Dojo fails to overcome compatibility issues across major browsers.
(C)	Dojo's programming model allows you to create widgets using tags.
(D)	The Dojo programming model can be used only declaratively.
(E)	The Dojo code is divided into logical units called modules.

(A)	A, B, and C	(C)	B, C, and E
(B)	B and D	(D)	A and C

3. Which of the following code snippet will create a Dojo button widget with the caption Close?

(A)
```
<script type="text/javascript" src="./dojo/dojo.js">
 dojo.require("dijit.form.Button");
</script>
...
<body class="tundra">
 <button dojoType="dijit.form.Button">Close</button>
</body>
```

Concepts

(B)	`<script type="text/javascript" src="./dojo/dojo.js">`
	`dojo.provide("dijit.form.Button");`
	`</script>`
	`...`
	`<body class="tundra" >`
	`<button dojoType="dijit.form.Button">Close</button>`
	`</body>`

(C)	`<script type="text/javascript" src="./dojo/dojo.js">`
	`dojo.require("dijit.form.Button");`
	`</script>`
	`...`
	`<body class="tundra">`
	`<button dojoType="dojo.form.Button">Close</button>`
	`</body>`

(D)	`<script type="text/javascript" src="./dojo/dojo.js">`
	`dojo.require("dijit.Button");`
	`</script>`
	`...`
	`<body class="tundra">`
	`<button dojoType="dijit.button">Close</button>`
	`</body>`

2.2 Lesson Overview

In the last lesson, **Dojo Widget Library**, you will learn to:

- ➔ Explain Dijit.
- ➔ Describe the different widgets used for form.
- ➔ Explain the steps to send AJAX request to a server using Dojo.

2.2.1 Introduction to Dijit

Dijit is an acronym for Dojo widget. Dijit is a widget system that is placed on top of Dojo. You use Digit to design rich Graphic User Interfaces (GUIs) by using minimal code. You can use Dijit either declaratively by using special attributes within regular HTML tags, or programmatically by using JavaScript.

Based on the context, you use the term `Dijit` for a single Dojo widget or the term `Dijits` for the all the widgets in the toolkit.

Some of the widget libraries available include CheckBox, RadioButton, ComboBox, TextBox, TextArea, ValidationTextBox, and so on.

2.2.2 Dijit Layout

Dijit has multiple layout widgets that are combined together in a hierarchy. Figure 2.7 shows that the screen is broadly split into two parts. The top acts as a toolbar. The bottom is again split into a left section and right section. The left section has three panes, and the right section is split into two parts.

Conceptually, a Dijit is a set of container that contains three types of elements. The first type of elements are those containers that display all their children side by side. The second type of elements are those containers that display one child at a time, and the third type of elements are the leaf nodes that contain only the content.

Figure 2.7 shows the hierarchy of Dijit.

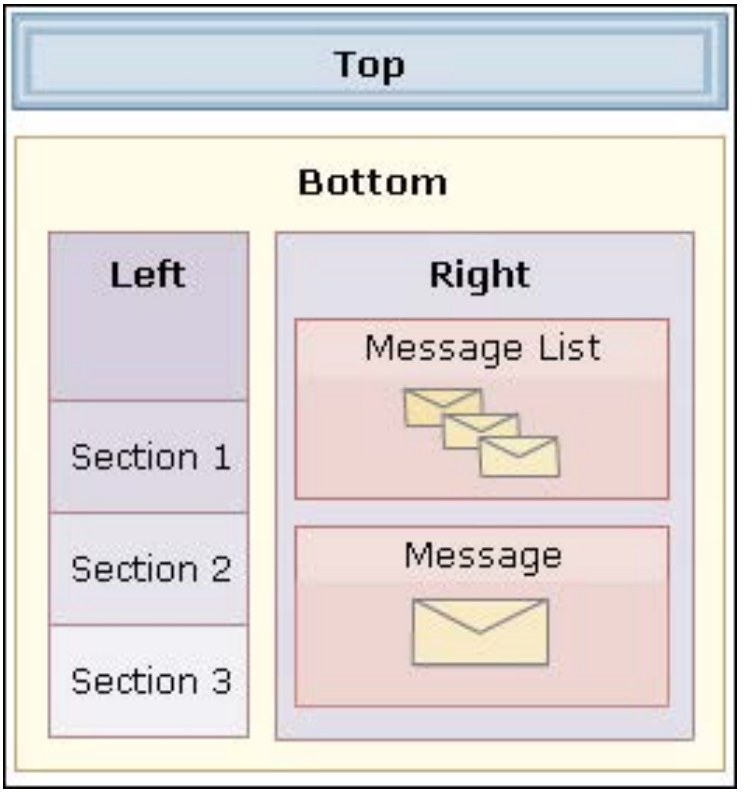


Figure 2.7: Hierarchy of Dijit

2.2.3 Dijit Layout

Dijit's layout widgets are stored in the `dojo.layout` subpackage.

Figure 2.8 shows the layout widgets available in the `dojo.layout` subpackage.

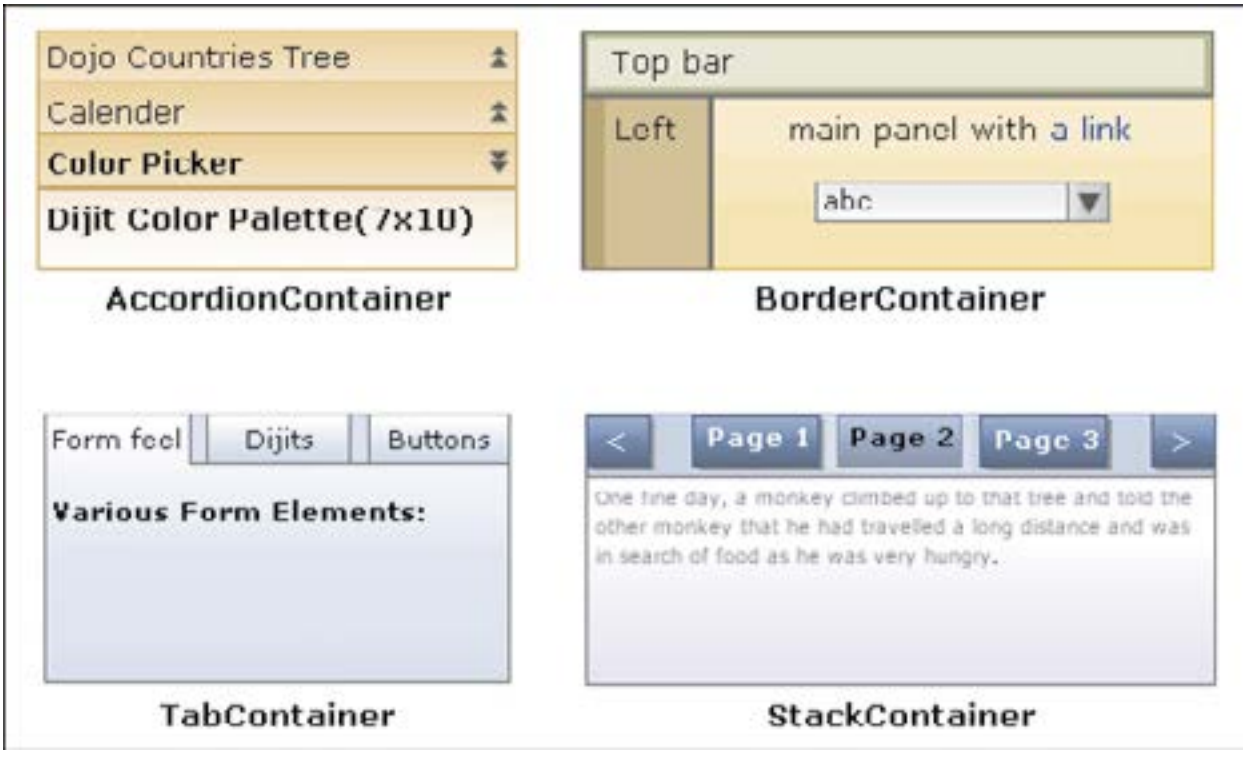


Figure 2.8: Layout Widgets in dojo.layout Subpackage

The different layouts provided by Dijit are:

➔ **AccordionContainer**

The AccordionContainer layout displays multiple panes. You can click a pane's title to pull up or pull down the panes. Only one complete pane is visible at a time.

➔ **BorderContainer**

The BorderContainer layout divides the container into top, left, bottom, right, and center sections. The layout also provides optional splitter controls to allow users to adjust the dimensions. For example, this layout can be used to reserve the top 100 pixels of the screen for title and navigation, and the rest for the displaying some other content.

➔ **ContentPane**

The ContentPane layout resembles an internal frame, but contains additional design features. A ContentPane is the most basic layout container that is placed inside the layout container.

➔ **LayoutContainer**

The LayoutContainer arranges the child nodes in top, left, bottom, right, and client sections. The container has a specific size. It places the child nodes along the edges of the different sections. It then places the child marked as client in the remaining space at the center of the page. For example, this layout easily formats table of contents.

➔ **SplitContainer**

The SplitContainer layout splits the children into many sections. You can adjust the size of each section.

➔ **StackContainer**

The StackContainer layout has multiple children, but shows only one child at a time. This container can be used for slide shows, allowing the user to display one pane at a time.

➔ **TabContainer**

The TabContainer layout resembles a tabbed folder. To display the content of a particular tab, you click the corresponding tab title.

2.2.4 Button Dijit

Dojo handles the HTML forms using the concept of Form Bind. Whenever a user clicks the Submit button, rather than submitting the form in the normal way and reloading the entire page again, Dojo sends the content to `XMLHttp` transport layer. From there, the transport layer sends the result to a callback method and then validates the form.

Button is the most important and widely used form element. The look-and-feel of a Dojo button is far better than an HTML button. You can display text or an icon on the button. You can display an image on the button by using the `img` tag within the `button` tag. Like HTML buttons, the Dojo button automatically resizes itself to fit the caption.

Figure 2.9 shows the creation of button using dijit.

```
...
<script type="text/javascript" src="./dojo/dojo.js">
  dojo.require("dijit.form.Button");
</script>
...
<body class="tundra">
  <h2>Image Button</h2>
  <button dojoType="dijit.form.Button">
    <img src="Open.gif" width="30" height="30">
  </button>
</body>
...
```

Output

Image Button

Figure 2.9: Creation of Button Using dijit

2.2.5 Check box and Radio button Dijits

Check boxes are used when you want to allow a user to select zero or more options from a set of options. Dojo check boxes are similar to HTML check boxes, but the former provides more styling options. To render a selected check box, you can set the value of the `checked` attribute as checked. The `value` attribute returns the value of the selected check box.

Radio buttons are used when there is a list of two or more options and you want to allow the users to select only one option from the list of options. Note that you import the CheckBox and RadioButton classes by using `dijit.form.*`. The `name` attribute defines a common name, `group1`, for all the radio buttons. The `label` tag displays a label for the radio button.

Figure 2.10 shows the creation of check box and radio button using dijit.

```
...
<body class="tundra">
  <h2>Check box</h2>
  <input dojoType="dijit.form.CheckBox" id="cb1"
  name="Designer" checked="checked" type="checkbox"/>
  <label for="cb1"> Are you a Web Designer?</label>

  <input dojoType="dijit.form.CheckBox" id="cb2"
  name="Programmer" type="checkbox" />
  <label for="cb2"> Are you a Programmer?</label>

  <hr><hr> <h2>Radio button</h2>
  <input dojoType="dijit.form.RadioButton" id="val1"
   name="group1" type="radio"/>
  <label for="val1"> Designer</label>

  <input dojoType="dijit.form.RadioButton" id="val2"
   name="group1" checked="checked" type="radio"/>
  <label for="val1">Programmer</label>
</body>
...
```

Output

Figure 2.10: Check Box and Radio Button Using dijit

2.2.6 AutoCompleter Combo box Dijit

Dojo combo box is a combination of a drop-down list and a single-line text box. A user can display the list by clicking the drop-down arrow. As the user moves the pointer over the list, each option under the pointer is highlighted. If the user selects an option from the list, the current selection is replaced with the selected option.

You create a combo box widget by using the `select` and `option` tags as shown in the image, The `dojoType` attribute uses the value `digit.form.FilteringSelect`.

The `autocomplete` attribute is set to `false`, which forces the user to write the entire text in the combo box to confirm its availability in the options provided.

Figure 2.11 shows the creation of AutoCompleter Combo box using dijit.

```
...
<script type="text/javascript" src="./dojo/dojo.js">
 dojo.require("dijit.form.FilteringSelect");
</script>
...
<body class="tundra">
  <h2>Auto Completer Combo box</h2>
  <select dojoType="dijit.form.FilteringSelect"
name="sname"
  autocomplete="false">
    <option value="Alex">Alex</option>
    <option value="Tony">Tony</option>
    <option value="Alice">Alice</option>
    <option value="Steven">Steven</option>
  </select>
</body>
...
```

Output

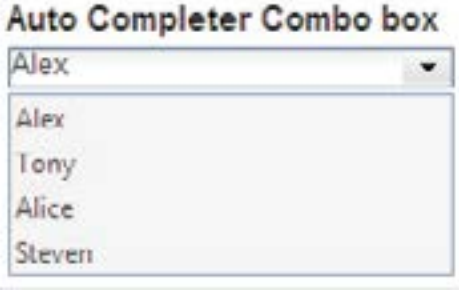


Figure 2.11: AutoCompleter Combo Box Using dijit

2.2.7 Dialog box Dijit

Dialog box is a rectangular GUI window that either requests or provides information to the user. Dojo provides the `digit.Dialog` class to create a dialog box.

Figure 2.12 shows the creation of Dialog box using dijit.

```
...
<script type="text/javascript">
    dojo.require("dijit.form.Button");
    dojo.require("dijit.form.TextBox");
    dojo.require("dijit.Dialog");
</script>
...
<body class="tundra">
    <div dojoType="dijit.Dialog">
      <table>
        <tr>
          <td><label>Username: </label></td>
          <td><input dojoType="dijit.form.TextBox" type="text"></td>
        </tr>
        <tr>
          <td><label>Password: </label></td>
          <td><input dojoType="dijit.form.TextBox" type="password"></td>
        </tr>
        <tr>
          <button dojoType="dijit.form.Button" type="submit">Login</button>
        </td>
      </tr>
    </table>
  </div>
/body>
..
```

Figure 2.12: Dialog box Using dijit

2.2.8 Using dojo.xhrGet() function

Dojo provides a function named `dojo.xhrGet()` to send and receive data asynchronously. The image shows the code to send an AJAX request using Dojo. Note that the `script` tag is enclosed within the `button` tag and bound to the `onclick` event. This ensures that the script is executed on the click of the Dojo button.

Figure 2.13 shows the use of the `dojo.xhrGet()` method.

```
...
<button dojoType="dijit.form.Button">Submit
  <script type="dojo/method" event="onclick">
    dojo.xhrGet({
      url: 'DataServlet',
      handleAs: 'xml',
      load: loginCallback,
      error: loginError,
      content: {nameParam: dojo.byId('name').value}
    });
  </script>
</button>
...
```

Figure 2.13: dojo.xhrGet() Method

➔ **url**

The `url` attribute specifies the name of the server-side component such as a JSP page or a servlet that will process the AJAX request. Here, the servlet named `DataServlet` acts as the server-side component.

➔ **handleAs**

The `handleAs` attribute specifies the MIME type such as text, json, javascript, and xml. Here, the MIME type used is `xml`, as the `DataServlet` will send an XML response.

➔ **load**

The `load` attribute specifies the name of the callback function that will be executed after successful receipt of response. Here, the `loginCallback()` function will be called.

➔ **error**

The `error` attribute specifies the name of the callback function that will be executed in case an error is encountered while processing the request. Here, the `loginError()` function will be called.

➔ **content**

The `content` attribute provides a comma-separated list of name-value pairs of request parameters. Here, the value of a Dojo widget named `name` is retrieved using the `byId()` function. This value is associated with the request parameter named `nameParam`.

2.2.9 Defining the load and error functions

Figure 2.14 shows the code of the functions `loginCallback()` and `loginError()`. Note that both the function accept two parameters, namely `data` and `ioArgs`. The `data` parameter holds the data returned by the server-side component such as `DataServlet`. The `ioArgs` parameter holds the request parameters sent using `xhrGet()` function.

The `loginCallback()` function is used to process the response received from the server. The `DataServlet` returns an XML response containing the request parameter's value enclosed in a `name` element. Therefore, the code retrieves the element named `name` using the getElementsByTagName() function. Next, the value of the element is retrieved using the `nodeValue` property. The value is then displayed in an alert dialog box.

The `loginError()` function is used to display an error message if AJAX request could not be processed.

Figure 2.14 shows the `loginCallback()` and `loginError()` method.

```
...
function loginCallback(data, ioArgs)
  alert(data.getElementsByTagName("name")[0].
  childNodes[0].nodeValue);
}

function loginError(data, ioArgs) {
  alert('Error when retrieving data from the server!');
}
...
```

Figure 2.14: loginCallback() and loginError() Method

2.2.10 Working Server-side Component

After the request is sent to the server, the request is processed using a server-side component such as JSP page or a servlet such as `DataServlet`.

➔ **Server-side Code**

Figure 2.15 shows the `doGet()` method of `DataServlet`. This method will process the AJAX request and send an XML response back to the client.

First, the content type of the response is set to `text/xml` indicating that the response will contain XML data. Next, an instance of `PrintWriter` is created to write data to the response. Then, you retrieve the value of request parameter named `nameParam` and store it in a variable, `param`. Finally, you write the XML data to enclose the value of variable `param` in the `name` element.

Figure 2.15 shows the implementation of the `doGet()` method at the server-side.

```
...
protected void doGet(HttpServletRequest request,
HttpServletResponse response) throws ServletException,
IOException {

  response.setContentType("text/xml");

  PrintWriter out = response.getWriter();

  String param = request.getParameter("nameParam");

  if(param !=null){
    out.write("<name>" + param + "</name>");
  } else
     out.write("Error!!!");

  out.close();
}
...
```

Figure 2.15: doGet() Method

Figure 2.16 shows the output of the code.

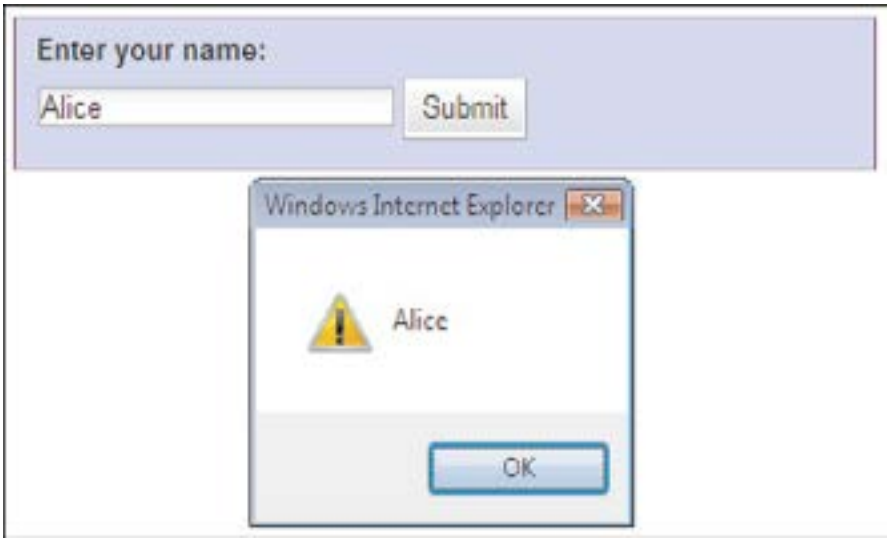


Figure 2.16: Output of Processing Request at Server-side

Knowledge Check 2

1. Which of the following statements about Dijit are true?

(A)	Dijit is a widget system that can be used to build good GUIs using minimal JavaScript code.
(B)	The BorderContainer layout resembles a tabbed folder.
(C)	The AccordionContainer layout consists of panes that are pulled up or down by clicking the pane title.
(D)	The StackContainer layout has multiple children, but shows only one child at a time.
(E)	The ContentPane is placed outside the layout container.

(A)	A, B, and C	(C)	B, C, and E
(B)	C, D, and E	(D)	A, C, and D

2. Which one of the following code snippets will create two radio buttons with the labels Graduate and Post Graduate respectively?

(A)	`<script type="text/javascript" src="./dojo/dojo.js">`
	`dojo.require("dijit.RadioButton");</script>`
	`<body class="tundra">`
	`<input dojoType="dijit.form.RadioButton" id="val1" name="group1" type="radio"/> Graduate`
	`<input dojoType="dijit.form.RadioButton" id="val2" name="group1" type="radio"/> Post Graduate`
	`</body>`

(B)	`<script type="text/javascript" src="./dojo/dojo.js">`
	`dojo.require("dijit.form.*");</script>`
	`<body class="tundra">`
	`<input dojoType="dijit.form.RadioButton" id="val1" name="group1" type="radio"/> Graduate`
	`<input dojoType="dijit.form.RadioButton" id="val2" name="group1" type="radio"/> Post Graduate`
	`</body>`

(C)	<script type="text/javascript" src="./dojo/dojo.js"> dojo.require("dijit.form.RadioButton"); </script> <body class="tundra">br> <input dojoType="dijit.form.*" id="val1" name="group1" type="radio"/> Graduate <input dojoType="dijit.form.*" id="val2" name="group1" type="radio"/> Post Graduate </body>
(D)	<script type="text/javascript" src="./dojo/dojo.js"> dojo.require("dijit.form.*"); </script> <body class="tundra">br> <input dojoType="dijit.form.CheckBox" id="val1" name="group1" type="radio"/> Graduate <input dojoType="dijit.form.CheckBox" id="val2" name="group1" type="radio"/> Post Graduate </body>

3. Consider a scenario where you want to send the value of a Dojo widget, named bookname, asynchronously. A callback function named bookCallback will process the XML response sent by servlet, BookServlet. Which one of the following code snippets will allow you to achieve this?

(A)	dojo.xhrGet({ url: 'BookServlet', handleAs: 'text', load: bookCallback, content: {param: dojo.byId('bookname').value} });
(B)	dojo.xhrGet({ url: 'BookServlet', handleAs: 'xml', load: bookCallback, content: {bookname: dojo.byId('param').value} });

(C)	`dojo.xhrGet({`
	`url: 'BookServlet',`
	`handleAs: 'xml',`
	`load: bookCallback,`
	`content: {param: dojo.byId('bookname').value}`
	`});`

(D)	`dojo.xhrGet({`
	`url: 'BookServlet.java',`
	`type: 'javascript',`
	`load: bookCallback,`
	`content: {param: dojo.byId('bookname').value}`
	`});`

Summary

In the module, **Using Dojo Toolkit**, you learnt about:

➔ **Dojo Toolkit**

Dojo is an open source JavaScript toolkit for developing AJAX-based applications. The benefits of using Dojo toolkit include code simplification, and reusability of code. The Dojo programming model follows object-oriented approach. Dojo toolkit includes the modules Dojo, Dijit, Dojox, and Util.

➔ **Dojo Widget Library**

Dojo Widget Library (Dijit) is a widget system that enables quick and easy development of Web pages. Dijit provides several layouts that determine the placement of widgets on a Web page. Some of the widgets available in Dijit are CheckBox, RadioButton, ComboBox, TextBox, DialogBox, and so on. Dojo supports asynchronous mode of communication by sending AJAX-based requests to a server.

Module - 3

JSON and DWR

Welcome to the module, **JSON and DWR**. This module introduces you to JavaScript Object Notation (JSON) which allows text-based data exchange between a client browser and a server. It is a subset of JavaScript language. Direct Web Remoting (DWR) allows a client browser to remotely invoke server functionalities.

- ➔ JSON
- ➔ DWR

3.1 Lesson Overview

In the first lesson, **JSON**, you will learn to:

- Explain data structure.
- Explain how to receive JSON data.
- Explain how to send JSON data.

3.1.1 What is JSON?

JavaScript Object Notation (JSON), a subset of JavaScript language, is a text-based data format that supports data exchange between a Web browser and a server. It is available as libraries in many languages, such as C#, Java, Python, and so on.

XML is also a text based format that supports data interchange. However, JSON is preferred over XML due to following reasons:

- JSON is lighter and faster as compared to XML.
- JSON objects are typed whereas XML objects are untyped. JSON supports data types such as string, number, array and boolean, whereas XML supports only the string data type.
- JSON code is native to JavaScript code. It is readily accessible to JavaScript code. However, XML code needs to be parsed and assigned to variables using tedious DOM operations.

3.1.2 JSON Data Types

JSON's text-based data format is based on JavaScript's object notation. The elements of the object notation supported by JSON are:

- **Object**

  An object is a collection of unordered name/value pairs. A JSON object is represented by `{}`.

- **Object Member**

  A JSON object member consists of a name/value pair, which is a combination of string and value. Members are separated by using commas. Object name and its value in a name/value pair are separated by a colon.

- **Array**

  A JSON array consists of elements or values that are separated by commas.

Array indexes are zero(0)-based. All the elements in an array are enclosed in []. Each individual element of the array is enclosed in { } if there are multiple name/value pairs.

➔ **Value**

A JSON value can be a string, a number, a boolean, an object, null or an array.

➔ **String**

A JSON string consists of Unicode characters except double quotes ("), backslash (\), or control characters. A JSON string is enclosed within double quotes.

The code snippet represents the personal details of Jack Daniels in JSON notation using JSON elements.

Code Snippet:

```
{
 "fullname": "Jack Daniels",
 "company": "JSON Consulting",
 "age" : 20,
 "email": [
  {"type": "work", "value": "jack.daniels@jsonconsult.com"},
  {"type": "home", "value": "jack@hotmail.com"}
 ],
 "contactno": [
  {"type": "work", "value": "123456"},
  {"type": "fax", "value": "345678"},
  {"type": "mobile", "value": "9987651111"}
 ],
 "addresses": [
  {"type": "work", "format": "us",
  "value": "1234 Manhutton"},
  {"type": "home", "format": "us",
  "value": "5678 SpringField"}
 ]
}
```

In the code snippet, "fullname" : "Jack Daniels" is an example of name/value pair or an object member.

The email, contactno and addresses are example of arrays that consists of name/value pairs separated by commas. Each array holds its elements as JSON objects in []. Each array element holds its name/value pairs within {}. In the email array, email[0] represents the first element: work, jackdaniels@jsonconsult.com. The code snippet uses two data types: string, number. For example, "Jack Daniels" is a string and 20 is a number.

3.1.3 JSON Objects

You can create a JSON object using curly braces. A JSON object can encapsulate other JSON objects that can be nested further. You can access an attribute of a JSON object by using the "dot" notation. For example, object_name.attribute_name. If an object is nested within another object, you can refer the attribute of the nested object as object1.object2.attribute_name.

The code snippet represents a JSON object, called employee:

```
var employee = {
        "name" : "Samson",
        "age" : 20
    };
```

The code snippet creates an object named employee. It contains two attributes or name/value pairs, name and age. The values are Samson and 20. Samson is a string but age is a number. You can access the attributes name and age by writing employee.name and employee.age.

Note - There is no limit to JSON object nesting.

3.1.4 Accessing JSON Objects

You can use JSON objects in JavaScript without any additional effort because JSON is a subset of JavaScript. You can declare a JavaScript variable and assign a JSON-formatted data structure to the variable. You can access and modify a particular element of a JSON-formatted object by using the "dot" notation from JavaScript code.

The different types of JSON objects that can be accessed using "dot" notations are:

➔ **JSON object elements**

You can access and modify JSON object elements using "dot" notation. For example, you can use Student.name to access the name attribute of a Student object.

➔ **Array elements**

You can access and modify an array element by using an index that is zero(0)- based. You can access attribute of an array element as array_name[index].attribute. If multiple name/value pairs are present instead of single attribute, these will be enclosed in {}.

➔ Nested JSON objects

You can access and modify a nested object using “dot” notation, like first_object.second_object. attribute.

The code snippet demonstrates use of “dot” notation to access different types of JSON objects.

Code Snippet:

```
var employee = {
 “fullname”: “Jack Daniels”,
 “company”: “JSON Consulting”,
 “age” : 20,
 “email”: [
  {“type”: “work”, “value”: “jack.daniels@jsonconsult.com”},
  {“type”: “home”, “value”: “jack@hotmail.com”}
 ],
 “contactno”: [
  {“type”: “work”, “value”: “123456”},
  {“type”: “fax”, “value”: “345678”},
  {“type”: “mobile”, “value”: “9987651111”}
 ],
 “addresses”: [
{“type”: “work”, “format”: “us”,
  “value”: “1234 Manhutton”},
  {“type”: “home”, “format”: “us”,
  “value”: “5678 SpringField”}
 ]
}
```

In the code snippet, the “fullname” : “Jack Daniels” is an example of name/value pair or an object member. The email, contactno and addresses are example of arrays that consist of name/value pairs separated by commas. Each array holds its elements as JSON objects in []. Each array element holds its name/value pairs within {}, email[0] represents first element, and so on. “Jack Daniels” is a String and 20 is a number.

3.1.5 Receiving JSON Data From Server

In an AJAX application, both client and server can receive and process JSON text.

When a client requests a server for some data, the server can send the requested data in JSON format, XML format, or as plain text. If the server returns the data in JSON format, the client receives the data in string format. The client uses the following steps to process the data:

1. The client converts the string data into a JavaScript object by using the statement, `eval("("+request.responseText+")")`.
2. The client can access and modify properties of the converted JavaScript object.

Figure 3.1 shows the client requesting for JSON data.

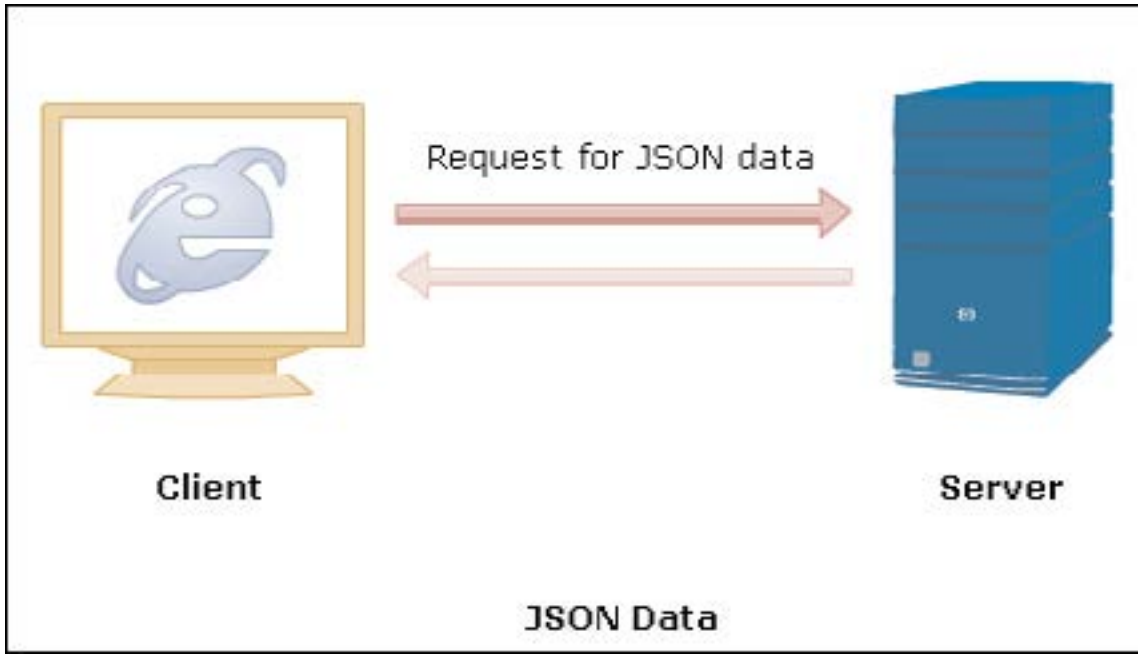


Figure 3.1: Client Requesting JSON Data

3.1.6 Receiving JSON Data From Client

When the client sends the processed data to the server, the client converts the JSON object to JSON text.

The server uses the following steps to processes the JSON text received from the client:

1. The server finds an appropriate parser depending on the programming language of the server-side program. For example, if the server application is written using JSP/servlets, the server uses a parser from the `org.json` package.
2. The parser interprets JSON text into a language that the server understands.

Figure 3.2 shows receiving of JSON data from server.

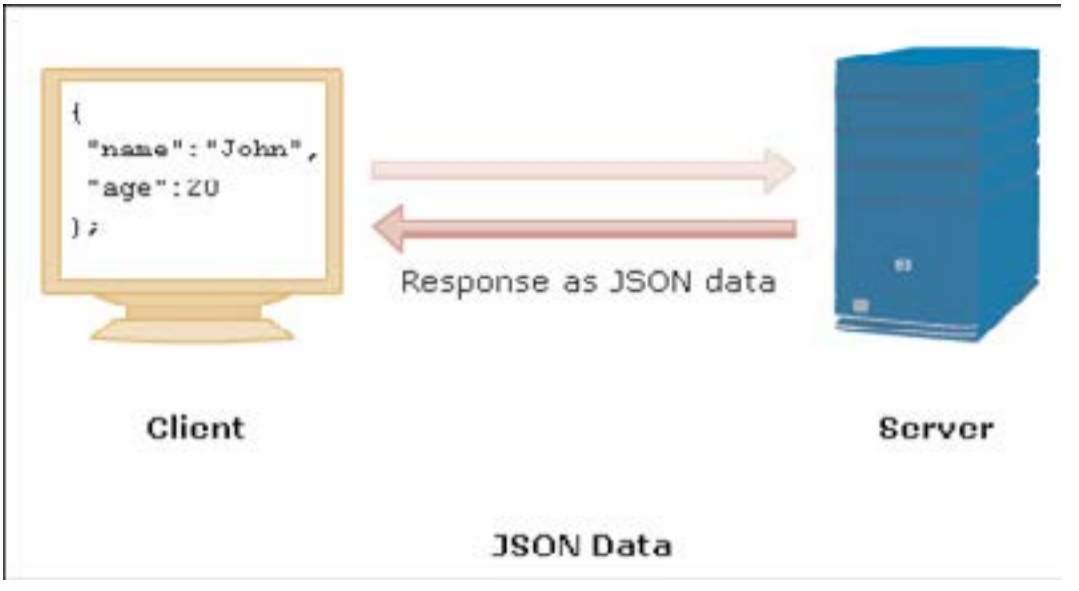


Figure 3.2: Client Receiving JSON Data

3.1.7 "Assignment" Technique

Three techniques of receiving JSON formatted text from server are: Assignment, Callback, and Parse.

The steps followed by the "Assignment" technique to receive JSON data are:

1. A JavaScript variable is assigned the JSON formatted text from server.
2. JavaScript's eval() method is invoked to convert the JSON text into a JSON object.
3. The "dot" notation is used to access properties of the JSON object.

   Figure 3.3 shows the "Assignment"technique to receive JSON data.

```
var JSONres = "week = { 'weekday' : 'Monday' }";
eval(JSONres);
document.writeln(week.weekday);
```

Figure 3.4: Assignment Technique

3.1.8 "Callback" Technique

In the "Callback" technique, a pre-defined function is called and the server response is passed, in JSON format, as the first argument to the function. The Callback technique is used to receive JSON data from Web sites of external domains.

Figure 3.4 shows the “Callback” technique to receive JSON data.

```
function processData(inputJSON) {
    document.writeln(inputJSON.weekday); // Outputs 'Monday'
}
var week = "processData( { 'weekday' : 'Monday' } )";
eval(week);
```

Figure 3.4: Callback Technique

The code defines a callback function with JSON object as argument that Outputs ‘Monday’. Then, passes JSON object as argument to processData() function and assigns it to JavaScript variable. Finally, executes JavaScript code.

3.1.9 “Parse” Technique

The “Parse” technique parses and executes only JSON text that comes from the server as part of response text. Therefore, this technique of receiving JSON data is the safest.

The JSON method `parseJSON()` is used to parse and execute JSON text that is received as a response from the server.

Figure 3.5 shows the “Parse” technique to receive JSON data.

```
JSONresponse = '{"weekday" : "Monday"}';
object= JSONresponse.parseJSON();
document.writeln(object.weekday);
```

Figure 3.5: Parse Technique

The code assigns JSON text to JSONresponse. Parses JSON text and displays the result.

Note - The method `parseJSON()` is part of JSON specification, but it is not yet part of JavaScript specification. It will be incorporated in the next version of JavaScript.

3.1.10 Sending JSON Data

To send a JSON object as part of a client request, convert the JSON object into a string data by using `toJSONString()`, and then use GET or POST method with `XMLHttpRequest` object.

GET request method allows transferring of JSON data from client to server, but it compromises security of data. Besides, it does not allow you to send large amount of data. If you want to send large amount of confidential data, you should use the POST request method.

Server can generate JSON data and send it to client as part of response text.

The steps for sending JSON data are:

1. Create `JSONObject` Java object
2. Add name/value pairs to the `JSONObject` using `put()`
3. Convert it to `String` type using `toString()`
4. Send it to the client with content-type as “text/plain” or “text/json”

Figure 3.6 shows the steps for sending JSON data.

```
JSONresponse = '{"weekday" : "Monday"}';
object= JSONresponse.parseJSON();
document.writeln(object.weekday);
```

Figure 3.6: Sending JSON Data

The code creates JSON object. Then, adds name/value pair to JSON object and converts JSON object to String.

Knowledge Check 1

1. Can you match the JSON elements with corresponding description?

	Description		Element
(A)	Each element is separated by comma	(1)	Object
(B)	Elements are enclosed in []	(2)	Object Member
(C)	Collection of unordered name/value pairs	(3)	Array
(D)	Can be string, number or Boolean	(4)	String
(E)	Consists of unicode characters	(5)	Value

(A)	(A)-(2), (B)-(3), (C)-(1), (D)-(5), (E)-(4)	(C)	(A)-(3), (B)-(2), (C)-(5), (D)-(1), (E)-(4)
(B)	(A)-(4), (B)-(3), (C)-(1), (D)-(5), (E)-(2)	(D)	(A)-(5), (B)-(4), (C)-(3), (D)-(2), (E)-(1)

2. Can you match the receiving methods for JSON data with descriptions?

	Description		Receiving Methods
(A)	Executes only JSON text	(1)	Assignment
(B)	Safest and parses only JSON text	(2)	Callback

(C)	Executes JSON as well as JavaScript	(3)	Parse
(D)	Assigns JSON text to JavaScript variable	(4)	parseJSON()
(E)	Calls user-defined callback function	(5)	eval()

(A)	(A)-(2), (B)-(3), (C)-(1), (D)-(5), (E)-(4)	(C)	(A)-(3), (B)-(2), (C)-(5), (D)-(1), (E)-(4)
(B)	(A)-(4), (B)-(3), (C)-(5), (D)-(1), (E)-(2)	(D)	(A)-(5), (B)-(4), (C)-(3), (D)-(2), (E)-(1)

3. Which of the following code snippet will send JSON data to client?

(A)	`JSONObject jsonObject = new JSONObject();`
	`jsonObject.put("JSON", "Hello from JSON!");`
	`jsonObject.toString();`

(B)	`JSON jsonObject = new JSONObject();`
	`jsonObject.put("JSON", "Hello from JSON!");`
	`jsonObject.toString();`

(C)	`JSONObject jsonObject = new JSONObject();` `jsonObject.toString();`
(D)	`JSONObject jsonObject = new JSONObject();` `jsonObject.put("JSON", "Hello from JSON!");`

3.2 Lesson Overview

In the second lesson, **DWR**, you will learn to:

- Explain the DWR architecture.
- Explain AJAX using DWR.
- Describe other technologies used with DWR.

Concepts

3.2.1 Direct Web Remoting

Direct Web Remoting (DWR) is an AJAX framework that allows clients to remotely access server functionalities. It is based on Java technology. It can generate JavaScript code from Java classes.

Client applications can not access the server-side Java classes directly. To allow the client applications to access the server-side Java classes, DWR converts the Java classes into JavaScript code. You can decide which Java classes will be accessible to the client browser by configuring DWR.

Figure 3.7 shows the DWR framework.

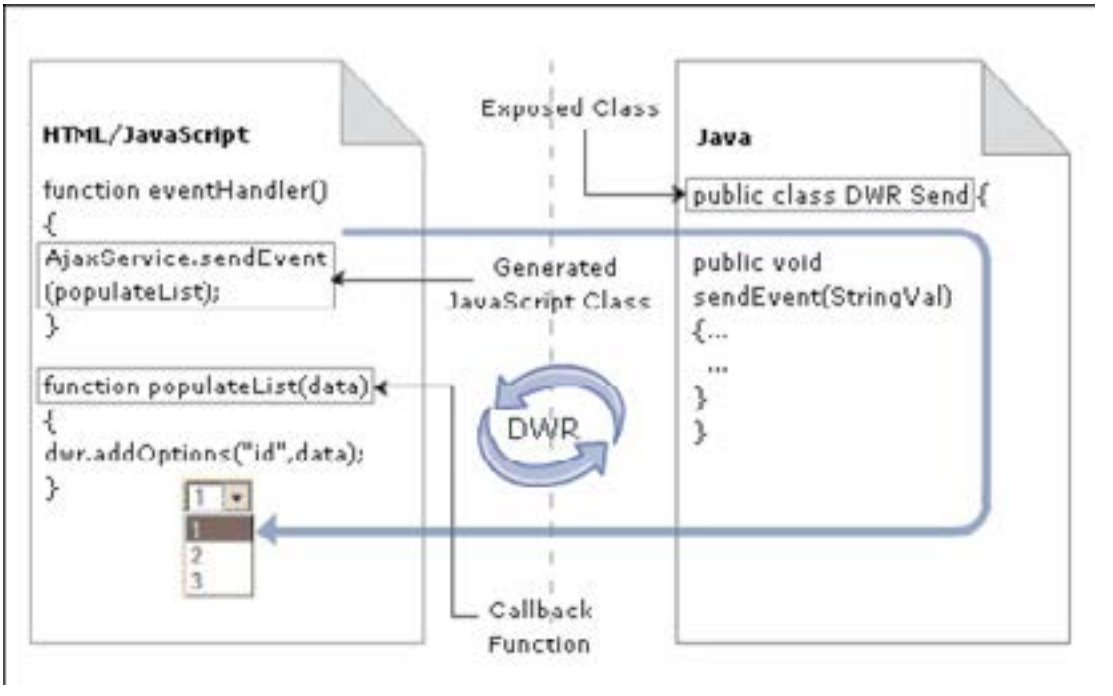


Figure 3.8: DWR Framework

3.2.2 Features of Direct Web Remoting

DWR hides the complexities of AJAX technology, and it relieves you from writing additional code for providing AJAX functionalities. For example, DWR automatically manages the XMLHttpRequest object, object serialization, and remote invocation of Java classes in Web server. DWR library is freely available.

Features that make DWR more useful compared to similar products are:

- Support for marshalling/unmarshalling of objects: DWR library can marshal/unmarshal any JavaScript object that is exchanged between a client and a server application. During the marshal/unmarshal process, DWR converts datatypes of Java to JavaScript and JavaScript to Java.

- Integration with popular frameworks: DWR library supports Spring, Hibernate, Struts and JSF frameworks.

- Documentation: DWR provides comprehensive documentation for its libraries.

- Reverse AJAX: DWR allows the server to connect with clients and send updated data to clients asynchronously.

3.2.3 Components of DWR

DWR exposes methods of server-side Java classes to client-side JavaScript code. DWR consists of two components:

- **Java Servlet**: Processes client requests and sends responses back to the client. DWR contains a runtime library that helps the servlet to process requests and responses.

- **JavaScript code**: Runs in the client application. It can dynamically update the Web page with the help of a JavaScript library that is part of the DWR architecture.

  Figure 3.8 shows the components of DWR.

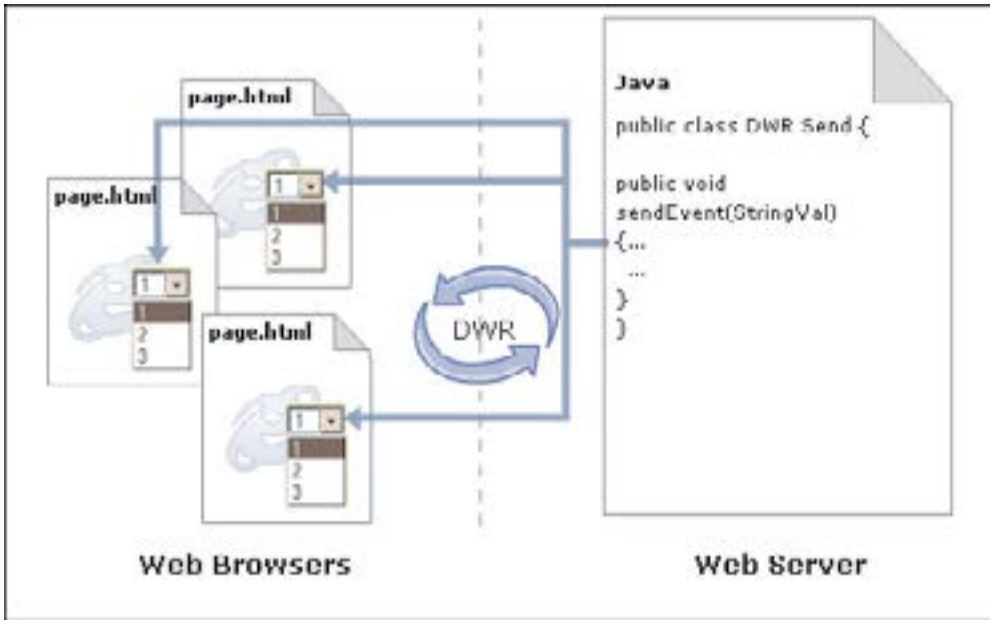


Figure 3.8: Components of DWR

3.2.4 Working of DWR

In DWR-based AJAX applications, the DWR servlet dynamically generates JavaScipt classes for each exposed server-side Java class.

The workflow of DWR can be summarized as:

1. DWR dynamically generates the client-side `JavaScript` code, called stub. The stub handles remote communication between browser and server.

2. The JavaScript code at the client-side routes calls to the server-side methods through the stub. The DWR servlet receives the client requests and calls the appropriate server-side methods.
3. DWR converts the data types of method parameters, received from the client application, to Java data types.
4. DWR converts the data types of return values, from the server-side methods, to `JavaScript` data types. Since data types of `Java` and `JavaScript` are different, conversion of the data types is required.
5. DWR asynchronously invokes the client-side callback methods by using the `XMLHttpRequest` object and sends the server response from the servlet to the callback function.

Note - Present version of DWR supports only limited data types for conversion between Java and JavaScript. It does not support overloaded Java methods.

Figure 3.9 shows the working of DWR.

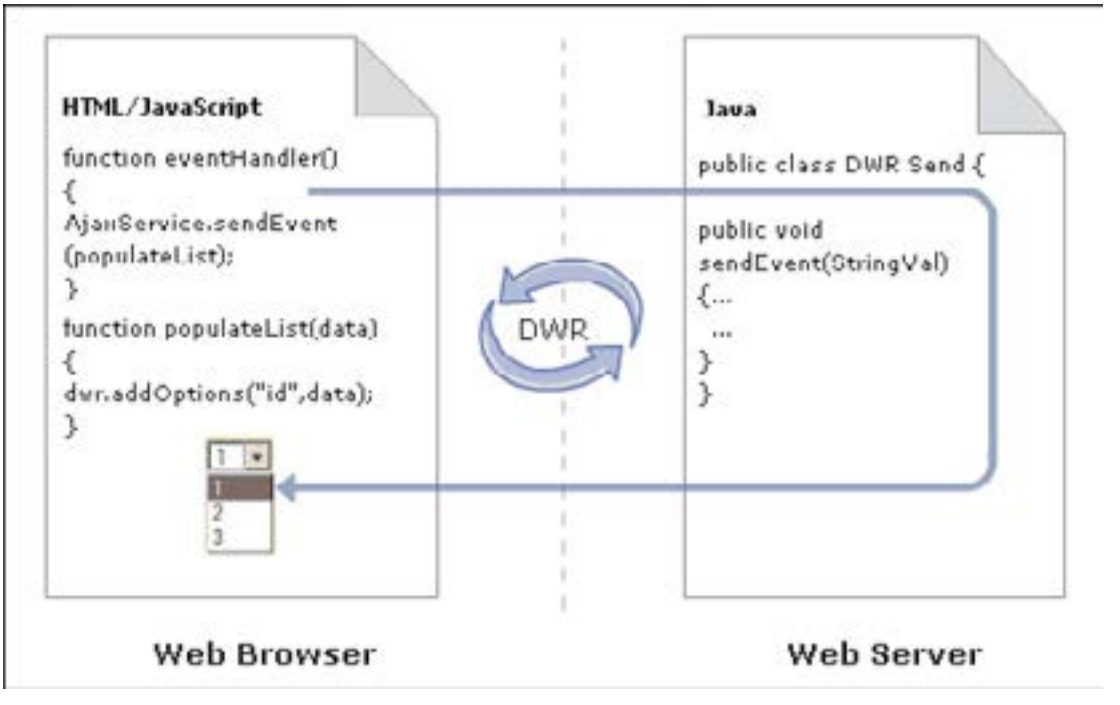


Figure: 3.9: Working of DWR

3.2.5 Handling Asynchronous AJAX Calls

In an AJAX application, a client sends a request asynchronously by using JavaScript code. The server that runs Java servlets processes the request synchronously, and sends back the response to the client. The processing at server is synchronous because Java technology is synchronous in nature.

To enable asynchronous communication between a client and a server, perform the following steps:

1. Declare a callback function in the client-side code. The callback function handles responses from the server.

2. Register the callback function with the server. To register the callback function, you need to pass an additional parameter when calling the remote methods. After making an asynchronous request, the client can carry out other tasks. When the server completes synchronous processing of the client request, the server invokes the callback function to pass the response data.

The code snippet shows how to declare a callback function in JavaScript- based client and register it with server.

Code Snippet:

```
//Declare callback function in client-side
function handleGetName(str){
alert(str);
}

//Server-side Java Class that will be remotely accessed by browser

public class Student{
public String getName(String name) {...}
}

// Invoke remotely getName() from client-side and
// register callback function getName()

StudentJavaScript.getName(42, handleGetName);
```

In code snippet, the `handleGetName()` is declared as callback function. The server-side Java class, `Student`, is exposed to client application. DWR generates JavaScript class, `StudentJavaScript`, from this class. Client calls the `getName()` method of the remote `Student` class by using the generated JavaScript class. It also registers callback method, `handleGetName()`, with the server by passing it as argument to `getName()`.

3.2.6 Steps of using DWR in AJAX

The steps demonstrate how an AJAX based application can be developed using DWR.

➔ **Server-side Java Class**

The code snippet declares a server-side Java class that will be exposed for generation of a JavaScript class.

Figure 3.10 shows the server-side Java class.

```
package com.course;

public class Student{
        public String getName(String
name){
                return "Hello" +
name;
        }
}
```

Figure 3.10: Server-side Java Class

➔ **Modify web.xml**

The DWR servlet in an AJAX application processes requests and responses. You must add mapping for DWR servlet in the web.xml file, located in /WEB-INF of the Web application.

The tag `<servlet-name>` indicates a DWR servlet named `dwr-invoker`. The tag `<servlet-class>` indicates actual class name of the DWR servlet. The DWR servlet is located in the package, `org.directwebremoting.servlet`. The initial parameters in tag `<init-param>` indicate that DWR servlet can be in debug mode. Finally, the `<servlet-mapping>` tag maps the servlet to the url pattern `/dwr/*`.

Figure 3.11 shows the web.xml file of the Web application.

```
<servlet>

<servlet-name>dwr-invoker</<servlet-name>
        <display-name>DWR
Servlet</display-name>

<servlet-class>org.directwebremoting.servlet
.DwrServlet
</servlet-class>
<init-param>
        <param-name>debug</param-name>
        <param-name>debug</param-name>
</init-param>
</servlet>
```

Figure 3.11: web.xml File

➔ **Configure DWR**

Add one more XML file, called `dwr.xml` in same folder with `WEB-INF/web.xml`. The file is DWR configuration file.

In the code snippet, `dwr.xml` determines which Java classes DWR can convert to JavaScript classes. A `dwr.dtd` is declared at the beginning. The tag `<create>` generates a JavaScript class `Student` from `Student` class defined in tag `<param>`.

Figure 3.12 shows the `dwr.xml` configuration file.

```
<!DOCTYPE dwr PUBLIC
 "-//GetAhead Limited//DTD DirectWebRemoting
2.0//EN"
 "http://getahead.org/dwr/dwr20.dtd">

<dwr>
  <allow>
    <create crator="new" javascript=Student">
    <param name="class"
value="com.student.Student">
    </create>
  </allow>
</dwr>
```

Figure: 3.12: Configuration File - dwr.xml

➔ Client-Side JavaScript

You must include the JavaScript files `engine.js` and `util.js` in the client-side application. These files are part of the DWR JavaScript library. The `engine.js` file provides core DWR functionalities. The `util.js` file provides DWR utilities. The file `Student.js` is the generated JavaScript file.

Figure 3.13 shows the inclusion of `engine.js` and `util.js` files.

```
<script type='text/javascript'
src='/dwr/engine.js'></script>

<script type='text/javascript'
src='/dwr/util.js'></script>

<script type='text/javascript'
src='/dwr/interface/Student.js'></script>
```

Figure 3.13: Inclusion of engine.js and util.js Files

3.2.7 Alternative Technologies

DWR is a remoting technology that is based on Java. It uses a custom protocol for remoting. Alternative remoting technologies available are: XML-RPC, JSON-RPC, and SOAP.

XML-RPC is a lightweight protocol that can exchange structured information in XML format between nodes of a distributed environment. It is programming language independent.

Simple Object Access Protocol (SOAP) is a protocol that exchanges XML-based messages in distributed environment using HTTP/HTTPS. SOAP provides a basic messaging framework for Web services.

JSON-RPC is a protocol that allows bidirectional communication between server and client, unlike XML-RPC or SOAP. JSON-RPC allows peer-to-peer communication between server and client. It contains only a few commands and data types.

Knowledge Check 2

1. Which of the following statements with respect to DWR architecture are true?

(A)	DWR is an MVC framework.
(B)	DWR is based on Microsoft technology.
(C)	DWR manipulates `XMLHttpRequest` object.
(D)	DWR can dynamically generate JavaScript code from Java code.
(E)	DWR converts Java data types to JavaScript data types.

(A)	A and B	(C)	B and D
(B)	D and E	(D)	C and E

2. Which of the following code snippet correctly handles asynchronous AJAX calls?

(A)

```
function handleGetName(str){
alert(str);
}

public class Student{
public String getName(String name) {...}
}

StudentJavaScript.getName(42, handleGetName);
```

(B)	`function handleGetName(str){`
	`alert(str);`
	`}`
	`JavaScript.getName(42, handleGetName);`

(C)	`function handleGetName(str){`
	`alert(str);`
	`}`
	`public class Student{`
	`public String getName(String name) {...}`
	`}`
	`Student.getName(42, handleGetName);`

(D)	`function handleGetName(str){`
	`alert(str);`
	`}`
	`public class Student{`
	`public String getName(String name) {...}`
	`}`

3. Can you match technology with description?

	Description		Technology
(A)	Allows bidirectional communication between client and server	(1)	DWR
(B)	Exchanges XML information in distributed system	(2)	XML-RPC
(C)	Exchanges XML messages using HTTP/HTTPS	(3)	JSON-RPC
(D)	Based on Java technology	(4)	SOAP
(E)	Based on peer-to-peer communication	(5)	JSON-RPC

(A)	(A)-(2), (B)-(3), (C)-(1), (D)-(5), (E)-(4)	(C)	(A)-(3), (B)-(2), (C)-(4), (D)-(1), (E)-(5)
(B)	(A)-(4), (B)-(3), (C)-(5), (D)-(1), (E)-(2)	(D)	(A)-(5), (B)-(4), (C)-(3), (D)-(2), (E)-(1)

Summary

In the module, **JSON and DWR**, you learnt about:

➔ JSON

JSON is a subset of JavaScript language that is used for interchanging of data between a browser and the server. It is easier to manipulate as compared to XML.

➔ DWR

DWR is an AJAX framework that can generate JavaScript code from Java classes. It is based on Java technology. A client browser can access remote Java classes that runs in the Web server by invoking the generated JavaScript functions. You can decide which Java classes will be accessible to the client browser by configuring DWR.

Module - 4

jMaki - I

Welcome to the module, **jMaki - I**. This module introduces you to jMaki and its features. jMaki is a light weight client-server framework that helps easy development of JavaScript-based AJAX Web applications.

In this module, you will learn about:

- ➔ jMaki Architecture
- ➔ jMaki Widgets

4.1 Lesson Overview

In the first lesson, **jMaki Architecture**, you will learn to:

- Explain the jMaki Features.
- List the advantages and disadvantages of jMaki.
- Describe the architecture.
- Explain the application structure of jMaki application.

4.1.1 Origin

jMaki is an open source, light weight client-server framework. jMaki originated in Kumamoto, Japan. The letter 'j' in jMaki represents JavaScript technology and Maki, which is a Japanese word, means 'to wrap'. In other words, jMaki means JavaScript wrappers.

jMaki is used for creating AJAX applications by integrating JavaScript technology into the applications. jMaki allows you to include styles and templates, widget model, and client services, such as event handling, in a client application. For server applications, jMaki provides server runtime component and a generic proxy, named `XMLHttpProxy`. `XMLHttpProxy` enables the server applications to interact with external Web services outside the application domain. JMaki provides access to widgets from various toolkits as a JSP taglib or as a JSF component.

Figure 4.1 shows the origin of jMaki.

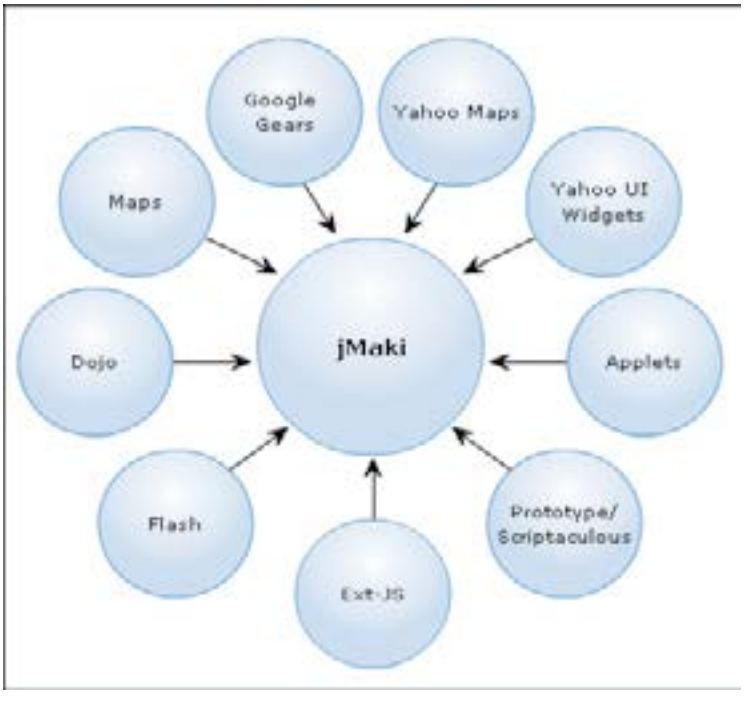


Figure 4.1: Origin of jMaki

Note - A lightweight client-server framework implements most of its functionality as independent modules. The advantages of this framework are that its sections are independently configurable, and it is easy to learn.

4.1.2 Features

The main aim of jMaki is to enable communication between client-side JavaScript and the multiple server technologies. Some of the features of jMaki are:

➔ **Wrapping of AJAX components in Tags**

jMaki uses JSP tags or JSF components to wrap the AJAX components. Thus, enabling easy development of applications. For example, to include the clock widget, present in Dojo library, in an application, the code to be written is: `<a:widget name="dojo.clock" />`.

➔ **Standardization of JavaScript Toolkit API**

jMaki provides a common framework for representing, documenting, and working with widgets. This is possible because jMaki wraps all the different APIs in a single widget style. You can reuse widgets from different libraries according to your requirements.

➔ **Support of Multiple Server Technologies**

jMaki supports multiple server technologies during runtime such as JSP, JSF, PHP, and JavaScript. jMaki applications does not limit itself to any specific server data model and these integrate easily with the existing technology in the server environment.

➔ **Preference for Convention over Configuration**

jMaki provides a considerable amount of default data and samples. Since, widgets and APIs are not mutually exclusive, jMaki follows 80/20 rule. Developers spend 20% of their time assembling, configuring the widgets visually and 80% of their time writing codes to implement the various APIs in AJAX application.

➔ **Provision of Standardized Event/ Data Model**

jMaki describes tree, table, and menu structures using JavaScript Object Notation (JSON) format. The consistent programming model helps in standardization of data and event model by jMaki. This enables the widgets from various toolkits to work with the same set of data.

4.1.3 Advantages of jMaki

jMaki provides a library of widgets. These widgets are present in various JavaScript technology libraries such as Dojo toolkit, and Yahoo toolkit. Some of the advantages of using jMaki to develop an AJAX based Web application are:

- It hides low level widget details by providing default values for widgets.
- It handles browser incompatibilities.
- It handles UI issues such as bookmarking.
- It minimizes the need to write JavaScript code for widgets.
- It references a widget in JSP page by adding appropriate tag library and including appropriate widget tag.
- It uses JSF architecture for handling inputs and validating user inputs by wrapping the widgets as JSF components.
- It requires no prior knowledge in DOM, CSS and JavaScript technology. However, you will need to use JavaScript code to implement widget functionalities.

Figure 4.2 shows the widgets provided by jMaki.

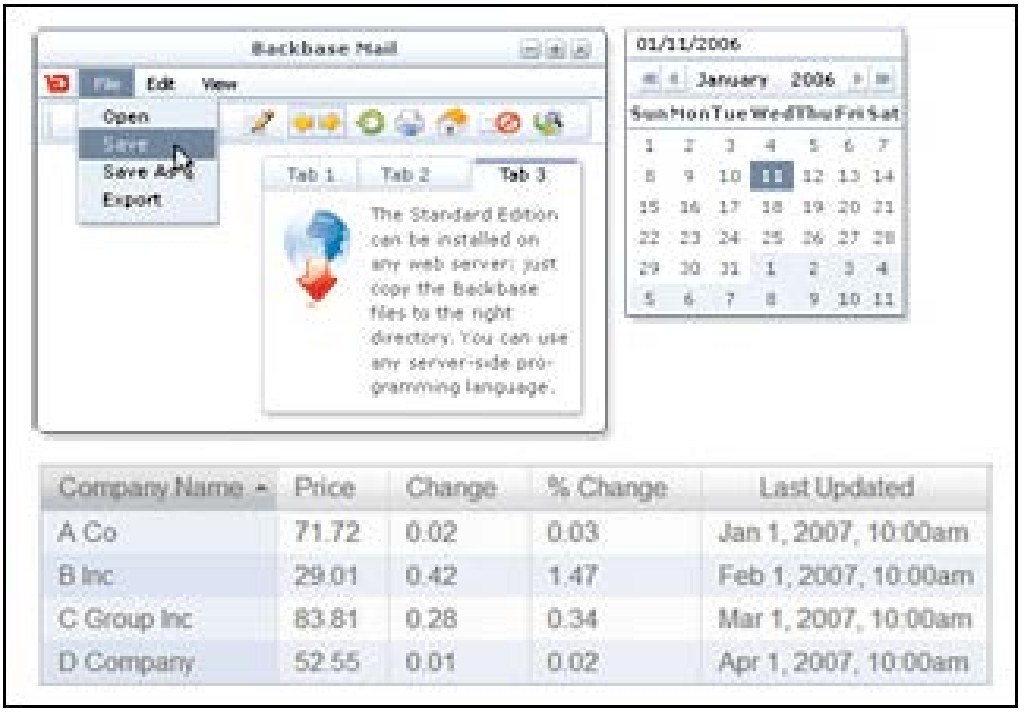


Figure 4.2: Widgets Provided by jMaki

4.1.4 Client–side Components

Architecture is a framework within which a system is built. It defines the components that constitute a system, and the information exchanged between the components. jMaki framework comprises of client components and server components. The client components that make up jMaki Architecture are:

➔ **jMaki Layouts**

jMaki provides different layouts to help reduce efforts and time required to create/design the layout of a Web page. jMaki uses HTML and CSS to create these layouts. You can easily customize these layouts.

➔ **jMaki Client Runtime**

jMaki client runtime uses JavaScript. It is responsible for bootstrapping the widgets and passing unique parameters provided by the server-side runtime to the widgets. JavaScript runtime ensures that the correct parameters are passed from the server-side runtime to initialize the widget. However, the runtime assigns default parameters to widgets if no specific parameters are provided.

➔ **jMaki Client Services**

jMaki client services provide APIs to use `XMLHttpRequest` object that allows data transfer between the client and the server. These services also provide publish/ subscribe event handling mechanism to enable communication between the widgets. jMaki Glue is built on top of publish/ subscribe mechanism that helps to define the application behavior. Widgets are tied together using the JavaScript actions. jMaki Timers invoke JavaScript action handlers or publishes events at regular intervals.

➔ **jMaki Widget Model**

jMaki widget model provides a component model for reusable JavaScript components. The structure of Widgets is based on HTML, CSS, and JavaScript. jMaki stores widget descriptions in `widget.json` format.

Figure 4.3 shows the client-side components of jMaki framework.

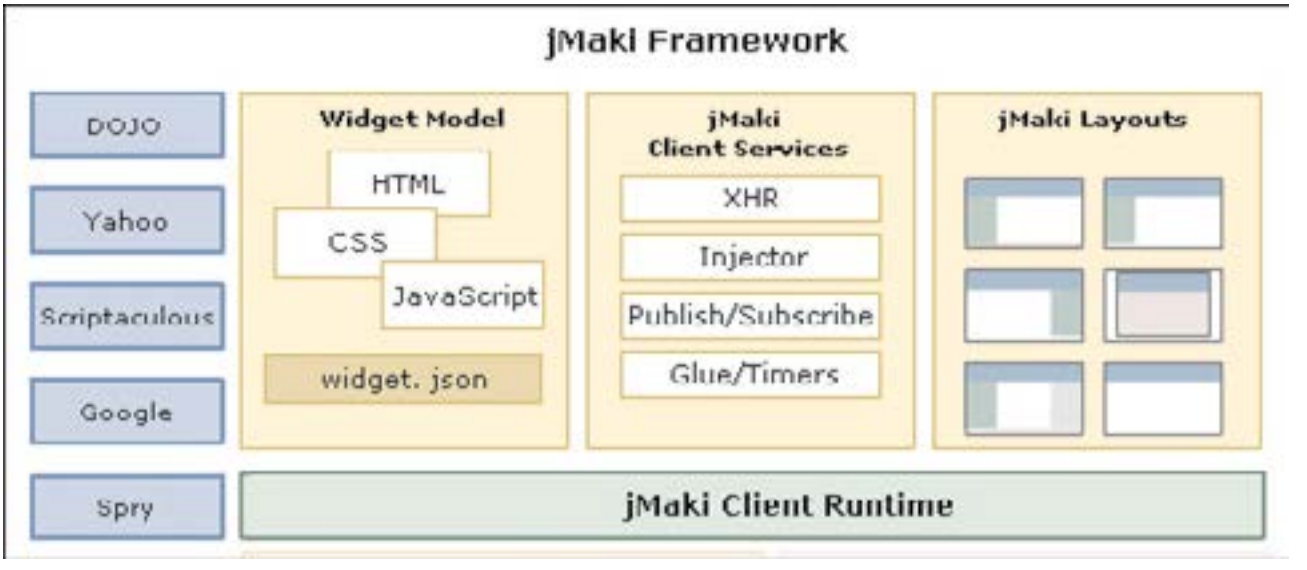


Figure 4.3: Client-side Components of jMaki Framework

4.1.5 Server–side Components

The two server components that make up jMaki framework are:

➔ **jMaki Server Runtime**

jMaki server runtime binds the server-side runtime with the jMaki client runtime. It is also responsible for tracking and delivering the correct JavaScript, CSS and HTML references based on the library being used so that these are not duplicated. For ensuring the availability of correct data to a widget instance, serialization of data in JavaScript is performed by server runtime.

➔ **XMLHttpProxy**

XMLHttpProxy allows widgets to access JSON or other external services, such as Flickr image searches. Direct Communication takes place between the widgets and the services.

Figure 4.4 shows the server-side components of jMaki framework.

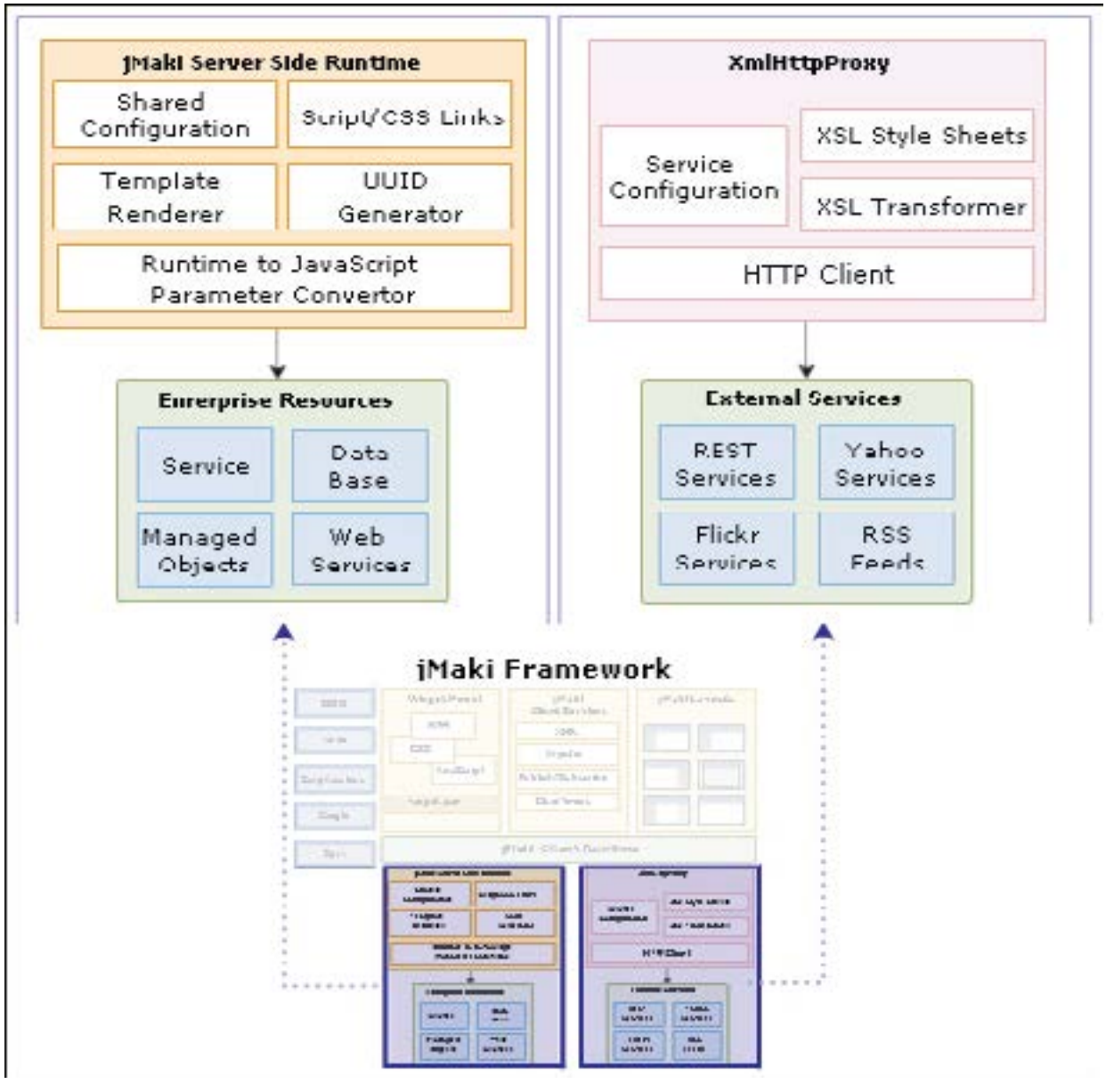


Figure 4.4: Server-side Components of jMaki Framework

4.1.6 Application Structure

jMaki applications can range from simple applications containing few widgets to complex applications containing multiple jMaki widgets.

Figure 4.5 displays the directory structure of an application.

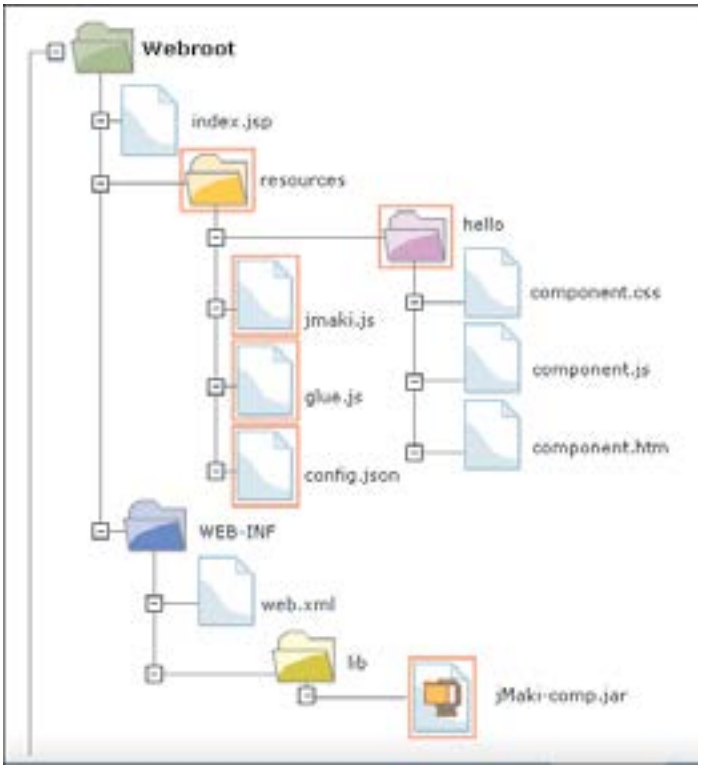


Figure 4.5: Directory Structure of an Application

➔ **resources**

The resources directory contains all the resources used by a jMaki application.

➔ **jmaki.js**

The jmaki.js is a JavaScript file that contains code for loading jMaki widgets and functions for widget communication. It is present in the resources directory.

➔ **config.json**

The config.json file contains theme information, extension mapping information and glue mapping information for wiring widgets.

➔ **jmaki-comp.jar**

The jmaki-comp.jar file that contains the server runtime code is present in the /WEB-INF/lib directory.

➔ **hello**

The hello is a widget and its resources are present in the /resources/hello directory. A widget is a directory comprising component.css, component.js, and component.htm file.

➔ **glue.js**

The glue.js file glues widgets together. It is used for registering and defining widget event listener, publishing events to a topic and subscribing to a topic.

Knowledge Check 1

1. Can you match the client and server side components with their descriptions?

	Description		Components
(A)	Uses HTML and CSS standards for creating Web applications	(1)	jMaki Widget Model
(B)	Provides APIs for performing XMLHttpRequest	(2)	jMaki Layouts
(C)	Provides a component model for widgets	(3)	jMaki Server Runtime
(D)	Binds the server-side runtime with the client runtime	(4)	XMLHttpProxy
(E)	Allows access to the external services	(5)	jMaki Client Services

(A)	(A)-(2), (B)-(5), (C)-(1), (D)-(3), (E)-(4)	(C)	(A)-(3), (B)-(2), (C)-(5), (D)-(1), (E)-(4)
(B)	(A)-(4), (B)-(3), (C)-(1), (D)-(5), (E)-(2)	(D)	(A)-(5), (B)-(4), (C)-(3), (D)-(2), (E)-(1)

2. Which of the following statements describing the features of jMaki are true?

(A)	AJAX components are wrapped as a JSP tags or JSF components.
(B)	Developers cannot reuse the widgets present in different libraries.
(C)	jMaki does not support the multiple server technologies.
(D)	jMaki standardizes the data and event model.
(E)	jMaki follows 80/ 20 rule.

(A)	A, B, and C	(C)	B, D, and E
(B)	A, D, and E	(D)	C, D, and E

3. Which of the following statements describing the advantages of using jMaki and application structure of jMaki are false?

(A)	The `config.json` file glues the widgets together.
(B)	jMaki maximizes the need to write JavaScript code for widgets.
(C)	jMaki handles browser incompatibilities.
(D)	The `jMaki.js` is a JavaScript file containing the code for loading jMaki widgets.
(E)	jMaki handles UI issues.

(A)	A and B	(C)	B and D
(B)	D and E	(D)	C and E

4.2 Lesson Overview

In the second lesson, **Widgets**, you will learn to:

- Explain the life cycle of widgets.
- Explain the style, layout, and theme of a widget.
- Explain how to add and load data into widgets.
- Explain the different types of widgets supported by jMaki.
- Describe event handling.

4.2.1 Widget Model

A jMaki widget is a reusable parameterized component. jMaki ensures that proper parameters are passed to a widget code to initialize the widget in a page.

The name of a widget maps to a directory. In other words, a jMaki widget is a directory or a package where the widget resides. The directories are separated using “dot” notation. The directory which makes a widget comprises of three core resource files. They are:

- **component.css**

  This file defines the CSS styles for a widget when it is displayed. It contains the code controlling the appearance of the widget. It is optional.

The code displays the content of a `component.css` file.

Code Snippet:

```
.header {
height:150px;
border: 1px solid #000000;
}
.main {
position: relative;
width: 100%;
height:auto;
}
.content {
margin: 0 0 0 250px;
height: auto;
border: 1px solid #000000;
} ...
```

The CSS file contains the code defining the appearance of the widget.

➔ **component.js**

This file defines the behavior of the widget. It contains code for wrapping of widgets, handling of widget events initiated by the user and interaction with AJAX. It is mandatory to have this file.

The code displays the content of a `component.js` file.

Code Snippet:

```
jmaki.namespace("jmaki.widgets.hello");
jmaki.widgets.hello.Widget = function(wargs) {
 //widget code
}
```

In the code snippet the widget is placed in a `jmaki.widgets.hello` namespace and is called a widget by appending the term `Widget` to it. The term `Widget` represents the constructor which is passed the widget argument. The jMaki server-side component will look in the same widgets directory for a directory named hello containing the subdirectory, foo. If the directory is found then it will look for `component.js` and `component.htm` file under jmaki/widgets/hello. If the directory is not found then the server side component will look under the resources directory for the widget and its resources.

➔ **component.htm**

This file defines the default HTML template that will be used by the rendering mechanism to display the widget in the page. In other words, it specifies the page layout for the widget. jMaki ensures that the HTML template is displayed with unique and instance specific parameters. It is mandatory to have this file.

The code displays the content of `component.htm` file.

Code Snippet:

```
<div id="${uuid}">

</div>
```

The markup that is included in the page is an instance of this widget. The code displays a template of a simple `<div>` element with a unique id. The `${uuid}` is replaced when jMaki processes the template before the page is displayed.

4.2.2 Widget Properties

The design pattern of jMaki helps to create widgets easily. Web applications can configure these widgets.

Each widget contains instance parameters. These instance parameters are passed by server-side runtime to JavaScript runtime using a function call. The JavaScript runtime passes the instance parameters, as object literal, to the widget as it is being created.

You can specify the property values of a widget by using tag attributes that matches the property name. JavaScript properties that are used with `args` and `value` are object literals. These should be enclosed in single quotes or escaped double quotes.

4.2.3 Life Cycle

Both client and server interactions are needed for displaying jMaki widgets. The sequence for the interactions are:

1. The jMaki widget defined in a JSP page along with the taglibs is interpreted.
2. The jMaki server-side components provide the correct HTML content along with their links to `component.css` file that is rendered to the page.
3. The jMaki server-side runtime component provides the content from the template file (`component.htm`) containing unique identifier of the widget to the page.
4. The jMaki bootstrapper script present in the `jmaki.js` file is rendered first to create a global object named jMaki. The object contains the properties and functions for registering, loading and supporting jMaki widgets.
5. Once the widget's template has been rendered, `addWidget()` function of jMaki object creates and registers the widget with the jMaki bootstrapper.

6. When the `onload` event of the page is fired, the registered widgets are initialized by the jmaki bootstrapper.

7. The rendered page is available for event processing.

Figure 4.6 shows the life cycle of a jMaki widget.

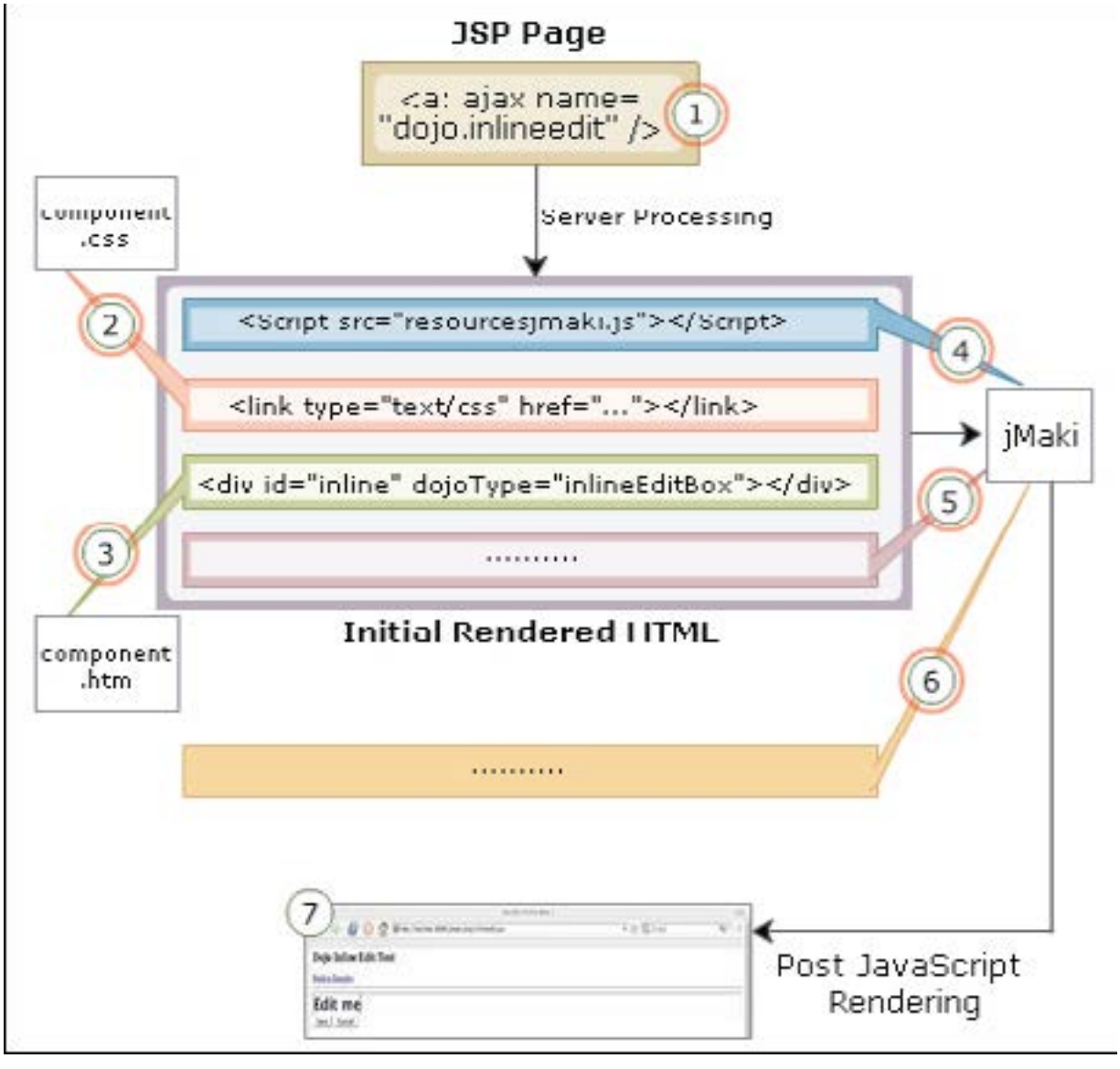


Figure 4.6: Life Cycle of a jMaki Widget

4.2.4 Layout

HTML pages are rendered by the browser based on the document type. The document type tells the browser to render the page strictly or transitionally in accordance with XHTML or CSS guidelines.
The CSS layout provided by jMaki uses XHTML transitional doctype as it follows XML syntax rules. jMaki uses CSS to define the layout of the page. To define the layout, you have to choose the layout and include the CSS file in the link tag.

The code snippet demonstrates how to use layout by specifying the `.css` filename in the link tag.

Code snippet:

```
<link rel="stylesheet" href="jmaki-standard.css"
type="text/css"></link>
```

The code demonstrates the use of standard layout.

Table 4.1 lists some of the layouts and templates provided by jMaki are explained in the table.

Layout	Description
Standard	Contains a left sidebar
Standard No Sidebars	Does not have any sidebars
Standard with Footer	Contains fixed sized left sidebar and footer
Centered	Contains a simple centered layout
Right Sidebar	Contains a right sidebar
Two Row Right Sidebar	Contains two rows on the left side and a right sidebar

Table 4.1: Layouts and Templates Provided by jMaki

Note - You can design your own layout by following the naming conventions in the CSS styles.

4.2.5 Style

A style sheet helps to improve the appearance of the Web pages in a Web application. The Library Level Styles and Widget Level Styles are two style types that can be applied to a Web application.

The Library Level Styles are applied to all the widgets in a given library. To define a library style for a given library, you have to use the `style` property when defining a widget type in the `widget.json` file. While defining the style you can specify the absolute or relative path of the style filename. The relative path is relative to the location of the widget where they are defined. The `resources` property specifies the directory containing the resources used in the Web application such as image files.

Widget Level Styles define the style and layout for a given widget. The `component.css` file contains the style definition for a given widget. The default colors and size for a specific widget are specified in the style definition. Widget Level styles override Library Level Styles. The style specified in `component.css` is applied after a library level style has been applied.

4.2.6 Theme

Themes are CSS styles and describe the color palette and typography used in a Web application. Themes do not change the layout or the structure of the documents.

The `config.json` file contains the definition of themes. Themes are applied in a Web application once all the widgets have been loaded and initialized. Themes override all the CSS properties defined for a page including the library level and widget level CSS files.

You can specify a theme by including the relative or absolute path of the `theme` file in the theme property of the `config.json` file.

The code snippet demonstrates how to declare theme in `config.json` file.

Code Snippet:

```
{
 "config": {
 "theme": "/resources/css/themes/orange/theme.css",
  "version": ".9"
 }
}
```

The code in code snippet specifies the theme file name, `theme.css`, in the theme property.

4.2.7 Common Theme Style

jMaki templates have some common themes styles that can be applied to pages/ widgets. jMaki themes include gradients, fonts and colors. While developing a widget, you can use these style names and provide a default color where theme is not used.

Table 4.2 lists the common style names along with their description.

Style Name	Description
jmakiTitle	Specifies the color used in titles such as a tables
jmakiBackground	Specifies the background color for a page or a widget
jmakiShadow	Specifies the shadow to be used for a widget or region
jmakiFont	Specifies the default font and color
jmakiFontHover	Specifies the font and color of the hover
body	Specifies the color applied to a document body
header	Specifies the background color or image

Table 4.2: Theme Styles

4.2.8 Precedence of Style Order

The order in which the cascade is applied to a page containing jMaki widgets are:

1. The structure of the document is described in the layout and is applied first to a page.
2. The Library Level Style containing the layout definition is applied and thus overrides the layout.
3. The Widget Level Style defining the style and layout for a given widget is applied. It overrides the library and layout styles.
4. The theme describing the typography and color used in an application overrides all the other styles. This style is applied after the page and widgets have been loaded.

4.2.9 Adding Widgets

A jMaki widget is added to a page only after the page with the required template has been created in the application. On adding a widget to a module, the three events that take place are:

- ➔ Widget resources such as `component.js` and `component.htm` files are added to the application under the resources directory.
- ➔ Definition of the jMaki tag library is added to the page.
- ➔ Custom jMaki `widget` tag is added to the page that refers the widgets and sets the widget attributes to default value. The tag represents a JSP handler. It also adds the tag library declaration.

For example, on adding a Dojo table widget, the `component.js` and `component.htm` files are added in the `resources/dojo directory`. Next, the Dojo widget code is added to the `resources/lib` directory of the application. Finally, the tag library is declared and `ajax` tag is used to add the widget. Once a widget has been dropped onto a page, the IDE uses the `name` and `value` attribute to initialize the widget.

The code demonstrates adding of the tag library declaration and ajax tag to the page.

Code Snippet:

```
<%@ taglib prefix="a" uri="http://jmaki/v1.0/jsp" %>

...

<a:widget name="dojo.table"

args="{columns: { 'title' : 'Title', 'author' : 'Author', 'bookId' : 'BookID',
'price' : 'Price' }}"
```

Concepts

```
value="{rows:[
['JavaScript by Dummies', 'Alex John','A101', '450'],
['Ajax with Java', 'Jean Thomas','A102', '650']
]}" />
```

The `name` attribute specifies the widget name. Dot notation specifies the directory structure containing the widget's resource files. The `args` attribute contains the column description whereas the `value` attribute contains the values for each column. The structure of the table has been separated from the data. The jMaki widgets accept data in JSON format and the data is provided using the `value` attribute.

4.2.10 Widget Attributes

You can determine the attribute of a widget by checking the `widget.json` file. It is mandatory to provide value for the `name` attribute of a widget.

Table 4.3 specifies the common attributes of a widget.

Attribute	Description
id	Identifies the widget for later reference
name	Specifies the name of the widget
style	Specifies the style for the widget. Default value is the style specified in component.css
service	Specifies the component name that provides service to the widget
value	Specifies the value of the widget
args	Defines the additional tag attributes

Table 4.3: Attributes of a Widget

4.2.11 Loading Data

jMaki widgets can be populated with data. There are three ways by which data can be loaded onto a widget. They are:

- Referring to a static file that contains JSON data
- Referring to the data in a bean by using an expression language (EL) expression in the tag's `value` attribute
- Referring to the data provided by a JSP page or servlet using the widget's `service` attribute

All the data needs to be passed to jMaki widget's in JSON format. In other words, data from a bean needs to be converted into JSON format. Data conversion is performed using JSON APIs.

Three steps to be followed for adding data into a widget using EL expression are: creation of a bean class that represent a single object, conversion of the data into JSON format, loading of the data from the bean into a widget.

➔ **Creation of a Bean Class**

Figure 4.7 demonstrates the creation of a Bean class.

```
public class Book {
    private int ID;
    private String bookName;
    private String author;
    private String price;

   public Book (int num, String bname, String auth, String price){
        this.ID = num;
        this. bookName = bname;
        this.author = author;
        this.price = price;
   }
   public int getBookID() {
        return ID;
   }

//getter methods for the other data members
}

public void setBookID(int no) {
     this.ID = no;
}
//setter methods for the other data members
}
```

Figure 4.7: Bean Class

The brief description of the code is as follows:

"public class Book" - Creates a Book class.

"public Book (int num, String bname, String auth, String price)" - Declares a parameterized constructor to initialize its data members.

"public int getBookID()" - Getter method that retrieves the id of the book.

"public void setBookID(int no)" - Setter method that assigns a value to its data member, ID.

➔ **Data Conversion into JSON Format**

Figure 4.8 demonstrates the code that converts data into JSON format.

```
public class BookApplicationBean {
public List addBooks() throws Exception {
        ArrayList bookList = new ArrayList();
        Book bookObj =
            new Book(201,
                "Who Moved My Cheese",
                "Alfred John",
                "450");
        bookList.add(bookObj);
        ....
        return bookList;
    }

...
    public JSONArray displayBookData() throws Exception {
            JSONArray booksArray = new JSONArray();
            JSONArray book = new JSONArray();
            ArrayList bookList =
                    (ArrayList)addBooks();
            Iterator itr = bookList.iterator();
            while(itr.hasNext()){
                    Book bookData = (Book)itr.next();
                    book.put(bookData.getBookID());
                    book.put(bookData.getName());
                    book.put(bookData.getAuthor());
                    booksArray.put(book);
                    book = new JSONArray();
            }
            return booksArray;
    }
```

Figure 4.8: JSON Format

The brief description of the code is as follows:

"**public List addBooks() throws Exception**" - Defines a method that will add object of the Book class to an `ArrayList` object.

"**ArrayList bookList = new ArrayList()**" - Creates an instance of `ArrayList` class named booklist.

"**Book bookObj = new Book(201, "Who Moved My Cheese", "Alfred John", "450")**" - Creates an instance of the Book class and initializes its data members (id, name, author, and price) by calling the parameterized constructor.

"**bookList.add(bookObj)**" - Invokes the `add()` method of the `ArrayList` class to store an instance of the Book class in the `ArrayList` object.

"**return booklist**" - Returns the `ArrayList` object to the calling method.

"**public JSONArray displayBookData() throws Exception**" - Defines a method that will convert the book data to JSON format.

"**JSONArray booksArray = new JSONArray()**" - Declares a JSON array.

"**ArrayList bookList = (ArrayList)addBooks()**" - Invokes the addbooks() method and stores the ArrayList object returned from the method in an ArrayList object.

"**while(i.hasNext())**" - Loops through the iterator using the hasNext() method.

"**Book bookData = (Book)itr.next()**" - Obtains the object stored in the ArrayList class. Converts the object to Book type by explicitly casting the record.

"**book.put(bookData.getBookID())**" - Obtains the ID of the book and stores it in a JSON array.

"**booksArray.put(book)**" - Converts the data into JSON format.

"**return booksArray**" - Returns the JSON array containing the book data in JSON format.

➔ **Populating the Widget**

Figure 4.9 demonstrates how to populate the widget.

```
<jsp:useBean id="bookBean" scope="session"
class="simpleBookjMaki.BookApplicationBean" />

<a:widget name="dojo.table"
   value="{columns: [ {label : 'ID', id : 'id'},
                      { label :'Name', id: 'name'},
                      { label :'Author', id: 'author'},
                      { label : 'Price', id : 'price'}
    ],
                      rows:${bookBean.bookData}}"
/>
```

Figure 4.9: Populate Widget

➔ **Populating the Widget**

```
<jsp:useBean id="bookBean" scope="session" class="simpleBookjMaki.Boo-
kApplicationBean" />
```

Uses the useBean tag to access the property of the bean, ApplicationBean.

```
<a:widget name="dojo.table"
```

Adds a widget, Dojo table, to the Web page. The widget name is specified using the name attribute of the widget tag.

```
"value="{columns:   [ label : 'ID', id : 'id'},
                    { label :'Name', id: 'name'},
                    { label :'Author', id: 'author'},
                    { label : 'Price', id : 'price'}
     ],
```

Creates the column data as the `displayBookData()` method does not create the column data. JSONObject API uses `HashMap` which inserts data in any order. To maintain insertion order, the column data is entered directly in the tag.

"rows:${bookBean.bookData}"

Obtains the row data by referencing the method from the `rows` attribute. The `displayBookData()` method returns the row data in JSON format.

4.2.12 Widget Libraries

In jMaki each widget type has the same data model irrespective of the toolkit, which provides the widget. It means that if you use a Dojo table widget and later you switch to Yahoo table widget, you can do it easily as you do not have to change the data format. Widgets that are included in jMaki Framework are from various JavaScript toolkits.

➔ **Flickr**

Flickr widget is used for creating captcha, word art and for searching images easily.

Table 4.4 list some of the tools created using Flicker widget

Widget Name	Topic Name	Type	Description
captcha	/flickr/captcha	publish	Returns a boolean value of true or false depending on whether there is a match
search	/flickr/search	subscribe	Subscribes to the flickr Search listener

Table 4.4: Tools from Flicker Widget

➔ **Google**

Google widgets are used for creating a map, mappopup and search.

Table 4.5 lists some of the tools created using Google widget.

Widget Name	args	Description
map	centerLat, centerLon	Sets the latitude and longitude of the map to a default value of 37.4041960114344 and -122.008194923401 respectively
mappopup	height, width	Sets the height and width of the map popup to a default value of 320 and 500 respectively
search	centerPoint	Sets the point to the center of the map at the default location of Santa Clara, CA

Table 4.5: Tools from Google Widget

➔ **Yahoo**

Yahoo widgets are used for creating a button, calendar, map, menu, rgbslider and so on.

Table 4.6 lists some of the tools created using Yahoo widgets.

Widget Name	Topic Name	Type	Description
calendar	/yahoo/calendar/onSelect	publish	On selection of a date, onSelect will publish to its topic name an object with id and value
button	/yahoo/button/onClick, /yahoo/button/onChange	publish	When the button is clicked or changed, both publish to their topic name an object with id and value. onChange is sent from checkbox buttons.
map	/yahoo/map/onClick, /yahoo/map/onChangeZoom	publish	When the map is zoomed or clicked both publishes to their topic name an object with id and value

Table 4.6: Tools from Yahoo Widgets

4.2.13 jMaki Widgets

Each jMaki widget present in different toolkits has the same data model. But this does not mean that the widgets present in different toolkits support the same functionality. Some toolkits support a more complicated structure than the other toolkit. For example, jMaki's menu widget cannot have child menus whereas Yahoo's menu widget can have child menus. If you pass the same data to jMaki's menu widget as you would pass to Yahoo's menu widget, it will ignore some of the data.

➔ **Data Model for Tables/Grid**

Separate arrays are created containing the row and column data. The data is presented in an object format. In the image the column `property` is used to identify the columns present in a table or a grid. The `title` property specifies the column heading, width property is used for setting the `width` of the column and `renderer` property sets the style of that column. The `row` property stores the details of each row.

Figure 4.10 shows the creation of a table.

```
{columns: [
        {'title' : 'Title', 'width': 100, 'locked': false},
        {'title':'Author', 'width': 50},
        {'title': 'Price', 'width': 75, 'renderer': 'usMoney'},
        {'title':'Description', 'width': 85, 'renderer': 'italic'}
],
rows:[
['JavaScript 101', 'Lu Sckrepter',441.20, 'Some long description'],
['Ajax with Java', 'Jean Bean','445.75', 'Excellent Book']
]
}
```

Figure 4.10: Table

➔ **Data Model for Trees**

In this model the expanded property for child trees have been by default set to false. In the image the `title` property specifies the label that will appear on each node. The `url` property specifies the page whose content will be displayed when the user clicks on that node.

Figure 4.11 shows the creation of a tree.

```
{ 'root' : {
            'title' : 'Aptech Tree Root Node',
            'expanded' : true,
            'children' : [
               { 'title' : 'Node 1.1'},
               { 'title' : 'Node 1.2',
               'children' : [
               { 'title' : 'Node 1.2.1','onclick' : {'url' :'http://foo'}}
               ]
            },
            { 'title' : 'Node 2.1'},
            { 'title' : 'Node 2.2'}
            ]
         }
   }
```

Figure 4.11: Tree

➔ Data Model for Menus

A menu's label property is followed by a `menu` or `url` property. All the menus do not support submenus. If you provide data for submenus, for those menus that do not support submenus, it will skip the entries for the sub menus in the data. In the image the url property specifies the `url` of the page that will be displayed when the user clicks on the menu.

Figure 4.12 shows the creation of menus.

```
menu: [
   { 'label': 'Animals',
      'menu' : [
         {label:'Birds', url: 'jsonibrowse.jsp'},
         {label:'Cats', url: 'ibrowse.jsp'}
      ]
   },
   {label: 'Bookmarks',
      'menu' : [
         {label:'Yahoo', url:'yahoo.jsp'},
         {label:'Delicious', url:'delicious.jsp'}
         ]
      }
   ]
}
```

Figure 4.12: Menus

➔ Data Model for Tabbed Views

In tabbed view data is presented in tabs. In figure 4.13 the `content` attribute is used to present static data that is presented as content for each of the tab. In the image the `label` specifies the

title that will appear on each tab and the `url` specifies the page that will be loaded in the area.

Figure 4.13 shows the creation of tabbed views.

```
{'tabs' :[
            {'label' :'Daniel Joe', 'content' : 'Daniel Joes Content'},
            {'label' :'Yahoo', 'url' : 'yahoo.jsp'}
          ]
}
```

Figure 4.13: Tabbed Views

4.2.14 Event Handling

Web sites can provide static information or can be interactive and provide dynamic information based on user inputs. Interactive Web sites allow a user to perform a number of actions on the Web site. Each user action fires a corresponding event. The event in turn calls an event handler to handle the event. For example, consider a ticket booking application. The user can select the travel date from a calendar on the Web site. When a user clicks a date on the calendar, the calendar disappears and the travel date is displayed on a text editor. Here, the click event is handled by an event handler. The event handler hides the calendar and displays the travel date on a text editor.

4.2.15 Glue

In jMaki, events are handled by Glue. Glue is a feature that allows JavaScript components to talk to each other by using the publish/subscribe mechanism. Components that generate events are called publishers and components that consume the generated events are called subscribers. The publish/subscribe mechanism helps in asynchronous communication.

When an event takes place on a publisher widget, the widget notifies the topic of the event to the consumers, also called listeners. If a widget is interested in the event, the widget can subscribe to the topic by registering itself with the topic. Topic of an event is a string that associates the publisher of the event with the subscriber of the event.

Figure 4.14 shows the publish/subscribe mechanism of Glue.

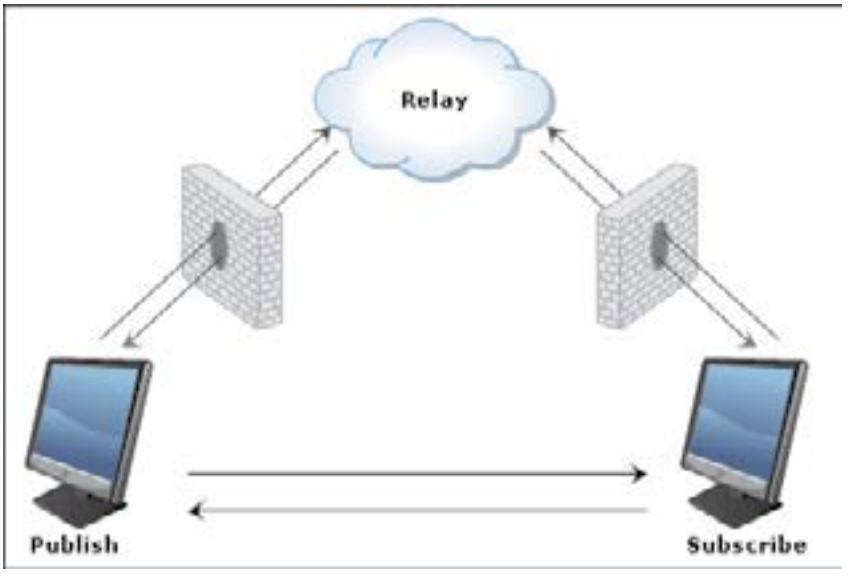


Figure 4.14: Publish/Subscribe Mechanism

4.2.16 Declarative Approach

The two ways in which publish/ subscribe mechanism of event handling can be used by jMaki are: actions and programmatic event.

Actions represent a declarative approach. It is used only when the actions are simple because this approach does not use JavaScript code to make changes to the other widgets. An action defines the publisher of events in a declarative form. There are some commonly used terms when working with actions. They are:

➔ **Topic Name**

A topic name is a string that starts with a forward slash ('/'), followed by jMaki library toolkit and by the widget name. For example, /jMaki/table is a topic name. The default topic name can be overridden.

➔ **Commands**

The consumer widgets contain event handlers. The event handlers perform a set of common operations known as commands, on the widgets. The command name is appended at the end of the topic name. For example, `/jMaki/table/onSelect`. Here, `onSelect` is the command name.

➔ **Payloads**

Payload represents the data that a widget publishes to the topic. When a widget publishes a topic, it specifies the topic name and the payload.

The action approach associates the action property of a widget with the event handler code within the widget. When a widget is initialized, the event topic is associated with the event handler.

The code demonstrates the use of action property of Yahoo button for publishing an event.

Code Snippet:

```
<a:widget name="yahoo.button"
  value="
   {label : 'Select Second Row',
   action : {topic : '/table/select', message: {targetId: 'add'}}}"/>
```

The `action` property of the button widget publishes the command /`select` to the topic, /`table`. The payload published to the topic is `targetId`: 'add'. The targetId attribute specifies the `id` of the element the action will affect.

The code demonstrates the use of the subscribe method for consuming the published event.

Code Snippet:

```
<a:widget name="yahoo.dataTable"
  subscribe="['/table', '/mytable']"
  value="{columns : [
   { label : 'Title', id : 'title'},
   { label :'Author', id : 'author'},
   { label :'ISBN', id : 'ISBN'},
   { label :'Description', id : 'description'},
   ],
   rows : [
   { title : 'Gone With the Wind',
    author : 'Scarlette O Hara',
    isbn : '103',
    description: 'A Must Read Book'
   },
   { id : 'add',
    title : 'Learn Java',
    author : 'John Lewis',
```

```
  isbn : '102',
  description: 'Good Book on Java'
 }
 …..
 ]
}" />
```

There are a set of properties that are common to all the widgets and one of them is `id`. The `id` property is used to identify an item such as a tab, a table row, or a tree node.

The `select` event of the Yahoo button widget has the payload published as `targetId: 'add'`. Thus, the second row of Yahoo datatable is targeted as it has the `id` value set to 'add'.

Figure 4.15 shows the output of the code.

Figure 4.15: Output - Consuming Published Event

4.2.17 Programmatic Approach

Sometimes, the action to be performed in response to an event is not simple or straight forward. For example, in response to an event, you may need to retrieve data from a database and then use the data to update another table. In such cases, the programmatic approach of Glue should be used, which is also based on publish/ subscribe mechanism.

To use the programmatic approach of Glue mechanism, you need to write code in the `glue.js` file. jMaki framework loads the `glue.js` file, other glue files included in the `config.json` file and makes the glue code available to the entire application.

The steps for implementing glue mechanism are:

➔ **Declare a topic for subscribers**

In the `glue.js` file, use the subscribe method to declare a topic that subscribers can listen to and the name of the listener that will handle the event notification. When an event publishes to the declared topic, the declared listener is called.

The code demonstrates how subscribers subscribe.

Code Snippet:

```
jmaki.subscribe("/jmaki/editor/onSave",jmaki.listeners.
editorListener");
```

In the code snippet, the glue listener is added to `jmaki.listeners object`. The code maps the `/jmaki/editor/onSave` topic to the event handler, `editorListener`.

➔ **Declare subscriber widgets**

Declare a subscriber widget in the Web page. While declaring the widget, you can either refer to the default topic name or use a different topic name. This indirectly maps the widget to the event handlers.

If you use the default topic name, you can add a listener by calling `jMaki.subscribe` in the `glue.js` file. If you want to use a different topic name, in the `args` parameter of the widget declaration, you can specify the new name. You will also need to change the topic name in the subscribe method to match the new name. The code demonstrates how to specify the topic name for the editor widget while adding the widget to a page.

```
<a:widget name=dojo.editor args="{topic : '/myeditor'}" />
```

The `args` parameter overwrites the default topic name by passing the value `myeditor`. The subscribe method in the `glue.js` file should match the new topic name, and hence should have the following code:

```
jmaki.subscribe("/myeditor", "jmaki.listener.editorListener");
```

➔ **Provide the code to handle notification**

In the `glue.js` file, add code to the listener. When a widget communicates with another widget, one widget acts as a publisher and the other widget acts as a subscriber. The publisher widget publishes its data, such as `id`, `value`, `args`, to a topic. The subscriber widget subscribes the topic and handles the event notification by using the listener/glue code. The listener code acts as the event handler.

The code demonstrates how to handle the notification.

Code Snippet:

```
jmaki.listener.editorListener = function (args) {
 var content = args.value;
 var contented = args.id;
 if (typeof content != 'undefined') {
  //send the data back to the server
jmaki.doAjax ({ method: "POST", url: "Service.jsp",
        content: {message: contentValue },
        callback: function(_req) {
         value = eval(req.responseText);
        //error handling code
          }
      });
  }
}
```

Every time the save operation is performed, the editor widget publishes to a topic. The handler receives the widget value through `args`. The listener processes the received data and after processing, returns the result to the caller.

Knowledge Check 2

1. Can you identify the code that demonstrates the use of action property for publishing an event?

(A)	`<a:name="yahoo.button"`
	`value="`
	`{label : 'Select Second Row',`
	`action : {topic : '/select', message: {targetId: 'add'}}}"/>`

(B)	`<a:widget name="button"`
	`value="`
	`{label : 'Select Second Row',`
	`action : {message: {targetId: 'add'}}}"/>`

Concepts

(C)	`<a:widget name="yahoo.button" value="{label : 'Select Second Row', action : {topic : '/table/select', message: {targetId: 'add'}}}"/>`
(D)	`<a:widget name="yahoo.button" column="{row : 'Select Second Row', message : {topic : '/table/select', action: {targetId: 'add'}}}"/>`

2. Which of the statements describing the characteristics of the widget model are true?

(A)	Widget name maps to a directory.
(B)	The component.css file defines the behavior of the widget.
(C)	Widget comprises of three resource files.
(D)	The component.htm file defines the default HTML template.
(E)	The component.js file defines the style for a widget.

(A)	A, C, and D	(C)	B, D, and E
(B)	B, C, and D	(D)	C, D, and E

3. Can you arrange the life cycle of widgets in a sequence?

(A)	Server side component runtime provides the template content to the page.
(B)	Widget is created and registered using the `addWidget()` method.
(C)	jMaki widget defined in a JSP page with taglibs are encountered.
(D)	jMaki object containing properties and functions registers, loads and supports jMaki widgets.
(E)	Registered widgets are initialized and the rendered page is available for event processing.

(A)	A, B, C, D, E	(C)	E, D, C, B, A
(B)	C, A, D, B, E	(D)	D, A, C, B, E

4. Which of the following statements about layout, style, and theme of jMaki widgets are true?

(A)	Left and Right Sidebars layout contains only fixed left and right sidebars.
(B)	Library Level Styles is applied to all the widgets in a given library.
(C)	Theme describes the color palette and typography used in a Web application.
(D)	Standard layout contains a right sidebar.
(E)	Widget Level Styles defines the style and layout for a given library.

(A)	A and C	(C)	B and C
(B)	C and D	(D)	D and E

Summary

In the module, **jMaki - I**, you learnt about:

➔ **jMaki Architecture**

jMaki is a lightweight client-server framework used for creating AJAX applications. It wraps the functionality of JavaScript technology. The main aim of jMaki is to use JavaScript on the client machine enabling it to communicate with different server technologies such as JSP, JSF and PHP. jMaki Architecture comprises of client and server components.

➔ **jMaki Widgets**

A widget is a reusable parameterized component. The three core resource files for a widget are `component.css`, `component.js` and `component.htm`, files that define the style of the widget, the behavior of the widget and the default HTML template for rendering the widget on the page.

Module - 5

jMaki - II

Welcome to the module, **jMaki - II**. This module introduces the concept of data models in jMaki mashups. The dData model pages include the formal specification of specifies the data expected by the widgets. The mashup is a wWeb site that combines content from various other wWeb sites into one convenient, easy-to-use portal.

In this module, you will learn about:

- ➔ Data Models
- ➔ Mashups

5.1 Lesson Overview

In the first lesson, **Data Models**, you will learn to:

- Describe the features of data models.
- Describe different types of GUI based data models.
- Explain the Drawer, and Multi View Container data model.

5.1.1 Data Models

Data models specify the type of data expected by various widgets. jMaki uses standardized data model to simplify communication between various widgets.

Following are some of the features of data models.

- Data models are standard for widgets, such as combo boxes, menus, trees, tables, and so on, across toolkits.
- Data model of a widget can be used in any toolkit without changing the format of the data. When you want to use a widget in another toolkit, the widget wrappers convert the jMaki data model as per the data requirement of the new toolkit.
- Data model of a widget includes publishers and subscribers. These allow and simplify dynamic update and communication between widgets.
- Data models across the various widgets have similar properties and events. If you learn one data model, most of the information of the data model will be applicable for other data models too.

Figure 5.1 shows the data model of jMaki.

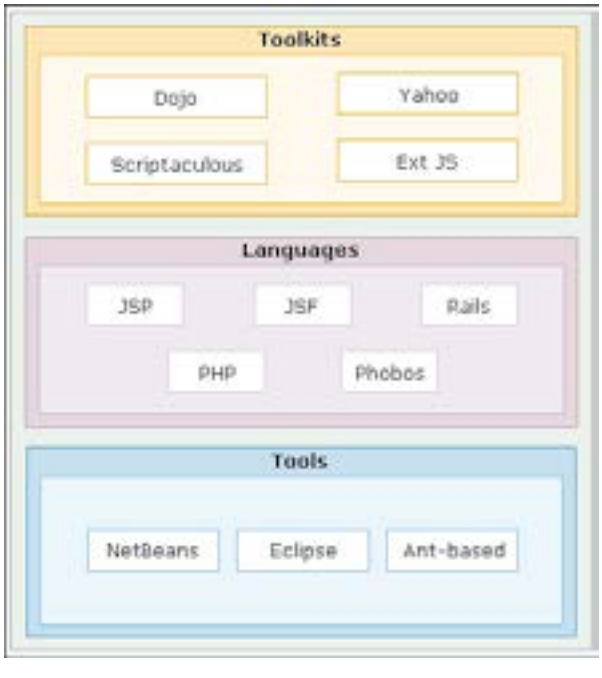


Figure 5.1: Data Model

Note - This standardization of data and event model helps in simplifying the programming model for the developer to build rich Web applications.

5.1.2 Types of Data Models

The data models supported by various widgets are listed in table 5.1.

Data Model	Widget supporting the data model
`jMaki Menu`	Yahoo Menu, jMaki Menu, jMaki Tab Menu, jMaki Accordion Menu
`jMaki Table`	Yahoo Datatable, Dojo Table
`jMaki Tree`	Yahoo Tree, Dojo Tree
`jMaki Combobox`	Dojo Combobox
`jMaki Multiview`	jMaki Dynamic Container, Dojo Accordion, Dojo Tabbedview, Yahoo Tabbedview
`jMaki Fisheye`	Dojo Fisheye
`jMaki Drawer`	Dojo Drawer
`jMaki Map`	Yahoo Map, Google Map

Table 5.1: Widgets Supporting Data Models

5.1.3 Common Properties

There are a set of properties which are common among the different data models. They are:

- `id` – Indicates the identifier of items such as table row, tree node, tab, and so on.
- `label` – Indicates the title of items such as table column, tree node, tab, and so on.
- `href` – Indicates that the string will act as a hyperlink. Clicking the link will navigate to the specified url.
- `include` – Indicates that the content from the specified url will be included in the page.
- `action` - Indicates that the object communicates an action to be performed by a widget.
- `targetId` – Indicates the `id` of an element on which a specific action is to be performed. The id of the target element and `targetId` of the data model should be the same.

Figure 5.2 shows the common properties of the data models.

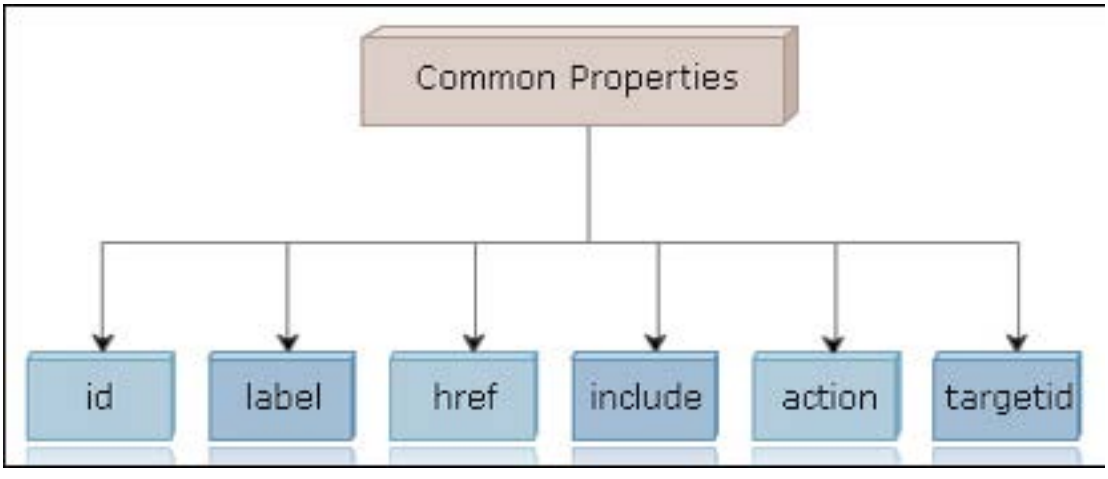


Figure 5.2: Common Properties of Data Models

5.1.4 Combobox Data Model

A Combobox is a GUI widget, which is a combination of a drop-down list and a single-line textbox. It allows the user to either type a value directly into the control or choose from the list of existing options.

➔ **Properties**

Figure 5.3 shows syntax.

```
combobox ::= "["  [<item>] "]"
item ::=  "{" "label:"  <string>, [<value>] [<selected> ] [<action> ]  "},"
selected ::=  "selected: false"
action ::= "action:" "{" [<topic>] <message>  "},"
topic ::=  "topic:" <string>,
message ::= "message:" <obj>
obj ::= <string> | <JavaScript object literal>
```

Figure 5.3: Combobox Data Model - Properties Tab

The `combobox` property represents an array of items. Each item in the Combobox contains a `label` property.

The `label` property specifies the text to be displayed for a specific item.

The `value` property stores the return value that will be published when an item in the Combobox is selected.

The `selected` property returns a boolean value of true or false. If an item in the Combobox is selected it returns `true`, else it returns `false`.

➔ **Subscribe/Publish**

There are two subscribe events:

➔ `select` – It selects a given item with the value provided in the payload.

➔ `setValues` – It specifies the payload value which is passed to the Combobox widget. This value is added to the Combobox list.

The publish event available is:

➔ `onSelect` – It is published when the user selects an item from the Combobox widget.

Figure 5.4 shows the `onSelect` event.

Event Type	Payload
onSelect	{WidgetId : uuid , topic : , type : 'onSelect' , targetId : selected_id, value : selected_value}

Figure 5.4: Combobox Data Model - onSelect Event

The code demonstrates use of the `setValues` subscribe event.

Code Snippet:

```
<a:widget name="yahoo.button"
value="{label : 'Set Values',
   action : {topic : '/mysettopic'}}"/>
```

The code demonstrates the glue.js file is:

Code Snippet:

```
jmaki.subscribe("/mysettopic", function (args) {
jmaki.log("in mytopic");
var list = [
    { label : 'SriLanka', value : 'SRI'},
    { label : 'India', value : 'IND'},
    { label : 'Canada', value : 'CAN', selected : true},
```

Concepts

```
{ label : 'England', value : 'ENG' }
  ]
 jMaki.publish("/dojo/combobox/setValues", {value: list}
  );
});
```

When a user clicks the Yahoo button, the button publishes a topic to add content in the Combobox. You have to add a listener to the topic, /`mysettopic`, in the glue.js file. The `jmaki.publish` API sends the hard coded data to the topic '`/dojo/combobox`'. The Dojo Combobox widget subscribes the topic and adds rows of data to the Combobox and displays it in the Web page.

5.1.5 Table Data Model

The Table data model represents a table with rows and columns.

➔ **Properties**

The rows in a table must be an array of objects which are mapped to the column names. You can provide the column names as a property of the `args` attribute. You can also pass to the widget as a service or value, the row and column values by combining it into a single object. The row id is implicitly assigned on a given row if not provided.

Figure 5.5 shows the syntax.

```
tabledata ::= "{" <columns> [<rows>] "}"
columns ::= "["  {<column>} "]"
column ::=  "{ label :"   <string>"  "id :" <columnid> "},"
columnid :: = <string>.
rows :: = "["  {<row>} "]"
row ::=  "{" [< rowId >] ,  <columnid>  ":  " [<string> | <object>]"}",
rowId :: = "rowId :"  "<string>"
object :: = "object :"  "<JavaScript Object Literal>"
```

Figure 5.5: Table Data Model - Properties Tab

➔ **Subscribe/Publish**

The different subscribe events are:

- `addRow` – This event appends the payload value passed to the widget at the end of the table.
- `addRows` – This event adds to the table the payload value passed to the widget and applies the filters to it.
- `clear` – This event clears all rows.

The publish event available is:

- `onSelect` – It is published when the user selects an item.

Figure 5.6 shows the `onSelect` event.

Event Type	Payload
onSelect	{WidgetId : uuid, topic : , type : 'onSelect' , targetId : selected_id, value : selected_value}

Figure 5.6: Table Data Model - onSelect Event

The code demonstrates use of the `addRows` subscribe event.

Code Snippet:

```
<a:widget name="yahoo.button"
value="{label : 'add rows',
   action : {topic : '/mytopic'}}"/>
```

The code demonstrates the glue.js file is:

Code Snippet:

```
jmaki.subscribe("/mytopic", function(args) {
  jmaki.log("in mytopic");

  jmaki.publish("/table/addRows",{value: [
    { title : 'The History of Montgomery Classis',
     author : 'William Nelson Potter',
     isbn : '4413',
     description: 'Average Book'
     },

    { title :'A Light Load' ,
     author : 'Radford',
```

```
      isbn : '4414',
      description: 'Good Book'
      }
    ]
});
 });
```

When a user clicks the Yahoo button, the button publishes a topic to add content rows. You have to add a listener to the topic, `/mytopic`, in the glue.js file. The `jmaki.publish` API sends the hard coded data to the topic '`/table`'. The Yahoo table widget subscribes the topic and adds two rows of data to the table and displays it in the Web page.

5.1.6 Menu Data Model

The Menu data model is used for all menu widgets supported in jMaki irrespective of the underlying toolkit. The data model enables the users to switch easily between toolkits and displays the data required by the widget in a standardized manner.

➔ **Properties**

The outer `menu` property identifies the labels of the menu bar.

The `publish` argument is overridden by the `topic` property for an item identified by a label.

The `message` property specifies the message that will displayed on the menu. The `disabled` property indicates if an item in the menu is disabled or not.

Figure 5.7 shows the syntax.

```
menuBar ::= "("  (<menu>) ")"
    menu ::= "menu:"  "["  {<label>}  "]"
    label ::=  "{" "label:"  <string>, [<menu> | <href>  [<action>] ]
[<disabled>] [<style> ]  "},"
    href ::= "href:" <string> ,
    action ::= "action:" "{" [<topic>] <message>  "},"
    topic ::=  "topic:" <string>,
    message ::= "message:" <obj>
    obj ::= <string> | <JavaScript object literal>
    style ::= "style:" "("  < CSS markup> ")"
    disabled ::=  "disabled: true"
```

Figure 5.7: Menu Data Model - Properties Tab

Concepts

➔ **Subscribe/Publish**

Only one event is available for the Menu data model that publishes the message.

- `onClick` – It is published when the user clicks an item.

Figure 5.8 shows the `onClick` event.

Event Type	Payload
onClick	{Widget id, topic name, href or message}

Figure 5.8: Menu Data Model - onClick Event

The code demonstrates the use of menu data model.

Code Snippet:

```
<a:widget name="jmaki.accordianMenu"
value="{menu : [
  {label: 'Links',
    menu: [
      { label : 'Aptech.com',
        href : 'http://www.aptechus.com'},
      { label : 'google.com',
        href : 'http://www.google.com'}
          ]
     },
  {label: 'Actions',
    menu: [
      { label : 'Select',
       action :{topic: '/ajax/select',
         message: { targetId : 'jmaki'}}
      },
```

```
      { label :'Set Content',
       action :{topic: '/ajax/setContent',
         message: { value : 'test.jsp'}}
      }
      ]}
      ]
  } />
```

The attributes, `value` and `service` pass data to the menu widget wrapper. You can use the `value` attribute when the data is located in the static page. The `value` attribute is assigned to the JSON representation of the data described in the menu data model. The menu creates links for two Web sites.

5.1.7 Tree data model

The tree data model represents tree widgets.

➔ **Properties**

The `tree` property specifies the node in the tree data structure. The outer tree property specifies the root of the `tree`.

The `expanded` property specifies if the children under this node will be initially visible or not.

The `children` property has an array of tree sub elements representing the child nodes of the current node. If a node exists, and a click action is performed on the node, the `action` property raises an event to be published. If the node has children, they are expanded.

The `action` and `children` properties are not permitted for the same node.

Figure 5.9 shows the syntax.

```
tree ::=    "{"    <label>    [ <expanded> ]  [ (    <action>    |    <children>    ) ]    "}"
label ::=    "'label'"    ":"    <string>
expanded ::=    "'expanded'"    ":"    ( true | false )
action ::= "action:" "{" [<topic>] [<message> ] "},"
topic ::=  "topic:" <string>,
message ::= "message:" <obj>
obj ::= <string> | <JavaScript object literal>
children ::= "'children'"    ":"    "["    <tree>    ...    "]"
```

Figure 5.9: Tree Data Model - Properties Tab

Concepts

➔ **Subscribe/Publish**

Some of the subscribe events are:

- `addNodes` – This event will add a sub tree of nodes under the node with the specified `targetId`.
- `expandNodes` – This event will expand the node with the specified targetId and its parent nodes.

The publish events available are:

- `onCollapse` – It is published when a user collapses a node.
- `onExpand` – It is published when a user expands a node.

Figure 5.10 shows the publish events.

Event Type	Payload
onClick	{widget id, topic name, href or message}
onCollapse	{widgetId : widget id, type :'onCollapse',targetId: <targetId>}
onExpand	{widgetId : widget id, type :'onExpand', targetId :<targetid>}

Figure 5.10: Tree Data Model - Publish Events

The code demonstrates the use of `addNodes` subscribe event.

Code Snippet:

```
<a:widget name="yahoo.button"
      value="{label : 'add nodes',
   action : {topic : '/mytopic'}}"/>
```

The code demonstrates the glue.js file is

Code Snippet:

```
jmaki.subscribe("/mytopic", function(args) {
  jmaki.log("in mytopic");
  var node = {
     label : 'Encyclopaedia' ,
     expanded : true,
     children : [
```

```
 {label : 'Culture'},
{label : 'Science'}
        ]
    };
jmaki.publish("/dojo/tree/addNodes", {value : node});
});
```

When a user clicks the Yahoo button, the button publishes a topic to add a tree node. You have to add a listener to the topic, `/mytopic`, in the glue.js file. The `jmaki.publish` API sends the hard coded data to the topic '`/dojo/tree`'. The Dojo tree widget subscribes the topic and add a node to the tree and displays it in the Web page. The code adds a node Encyclopaedia the Dojo tree widget and the child elements, Culture and Science, are then added to the Encyclopaedia node.

5.1.8 Drawer Data Model

A drawer is a container widget that expands or collapses the widget. It follows the general dynamic container model that allows setting the content or specifies the URL whose content will be displayed after loading the widget.

➔ **Properties**

The `content` property specifies the static content to be displayed in a row when the widget is rendered.

The `include` property specifies the page to be included in the pane.

The `action` property allows the users to specify the message to publish and the topic to publish to. If no topic is specified then the default topic is used that was specified by the `publish` property on the tag.

Figure 5.11 shows the syntax.

```
label ::=  "{" "label:"  <string>, [<content> | <include> |<action> ]
[<lazyload>]  <expanded>"},"
    expanded ::= "expanded:" "true" | "false"  (default is true)
    include ::= "include:" <string> ,
    lazyload ::= "lazyload:" "true" | "false",
    content ::= "content:" <string>,
    action ::= "action:" "{" [<topic>] <message> "},"
    topic ::=  "topic:" <string>,
    message ::= "message:" <obj>
    obj ::= <string> | <JavaScript object literal>
```

Figure 5.11: Drawer Data Model - Properties

➔ **Subscribe/Publish**

The different subscribe events are:

- `expand` – This event expands the pane.
- `collapse` – This event collapses the pane.
- `setContent` – This event sets the content of the pane to the value specified in the `value` property.
- `setInclude` – This event sets the URL of the page that will be included in the pane.

The publish event available is:

- `onSelect` – It is published when the user selects an accordion pane.

Figure 5.12 shows the `onSelect` event.

Event Type	Payload
onSelect	{widget id, topic name, targetId, mesage if it exists}

Figure 5.12: Drawer Data Model - onSelect Event

The code demonstrates the use of `setContent` subscribe event.

Code Snippet:

```
jmaki.publish("/dojo/drawer/setContent, {value : 'Welcome to Aptech's Ajax
Course'});
```

The code not only sets the content of the dojo drawer with the value 'Welcome to Aptechs Ajax Course' and displays the value when an event occurs such as clicking of a button.

5.1.9 Multi View data model

The Multi View data model is useful for all tabbed view widgets, accordion widgets, and jMaki dynamic container.

➔ **Properties**

An iframe is a frame inside a Web page that is not dependent on the sides of the Web page window.

The `iframe` property represents if the content is loaded into the iframes.

The `lazyLoad` property when set to `true` loads the content once the pane have been selected.

The `content` property specifies the static content to be displayed in a tab when the widget is rendered.

The `include` property specifies the page to be loaded in the tab. This is done using the container.

Figure 5.13 shows the syntax.

```
items ::= "{"  {<label> } "}"
label ::=  "{" "label:"  <string>, [<content> | <include> |<action> ] [<lazyload>]
[<id>] <selected> [<iframe>]  "},"
selected ::= "selected:" "true" | "false"  (default is false)
include ::= "include:" <string> ,
lazyload ::= "lazyLoad:" "true" | "false",
iframe ::= "iframe:" "true" | "false",
id ::= "id:" <string> ,
content ::= "content:" <string>,
action ::= "action:" "{" [<topic>] <message>  "},"
topic ::=  "topic:" <string>,
message ::= "message:" <obj>
obj ::= <string> | <JavaScript object literal>
```

Figure 5.13: Multi View Data Model - Properties

➔ **Subscribe/Publish**

The different subscribe events are:

- `select` – This event selects a tab whose id have been specified in the targetId property.
- `setContent` – This event sets the contents of a tab whose id have been specified in the `targetId` property.
- `setInclude` – This event sets the include URL of a tab whose id have been specified in the `targetId` property.

The publish event available is:

- `onSelect` – It is published when the user selects an item.

The code demonstrates the use of `select` subscribe event.

Code Snippet:

```
jmaki.publish("/yahoo/tabbedview/select", {targetId : 'Ajax'});
```

The code selects the tab with the targetId as 'Ajax'.

Knowledge Check 1

1. Can you match the description with the appropriate subscribe / publish events?

	Description		Subscribe/Publish Event
(A)	Adds a sub tree of nodes under the node with the targetId.	(1)	removeNode
(B)	Expands the node and its parent nodes.	(2)	removeChildren
(C)	Removes a node from the tree.	(3)	expandNodes
(D)	Removes all the child nodes of the nodeId and the nodeId itself.	(4)	clear
(E)	Clears all rows.	(5)	addNodes

(A)	(A)-(2), (B)-(5), (C)-(1), (D)-(3), (E)-(4)	(C)	(A)-(3), (B)-(2), (C)-(5), (D)-(1), (E)-(4)
(B)	(A)-(4), (B)-(3), (C)-(1), (D)-(5), (E)-(2)	(D)	(A)-(5), (B)-(3), (C)-(1), (D)-(2), (E)-(4)

2. Which of the following options describing the use and purpose of data models are true?

(A)	The data model does not maintain consistency across the different types of widgets.
(B)	The data model specifies the data passed to the widgets.
(C)	The data model describes the event payload that is published.
(D)	jMaki has a standard data model for only Google widgets.
(E)	The data models are described using BNF notation.

(A)	A, B, and C	(C)	B, C, and E
(B)	A, D, and E	(D)	C, D, and E

5.2 Lesson Overview

In the second lesson, **Mashups**, you will learn to:

- ➔ Explain the architecture of jMaki mashup.
- ➔ Describe different mashup styles.
- ➔ Explain server side mashups.
- ➔ Explain client side mashups.

5.2.1 Characteristics

Consider a Web site housingmaps.com that provides real estate information from the craglist.org Web site. The information is provided through clickable pushpins on interactive Google maps. When users click the pushpins, they can view the exact location of apartments that are available for sale or rent.

Users can view content of both Google maps and craiglist.org in housingmaps.com. Thus, users need not toggle back and forth between the sites, Google map and cariglist.org.

When services or content from various Web sites are combined in a single Web site, it is known as mashup. Unlike open source software, mashups typically function through APIs, which facilitate communication between technologies. Figure 5.14 information on Google map.

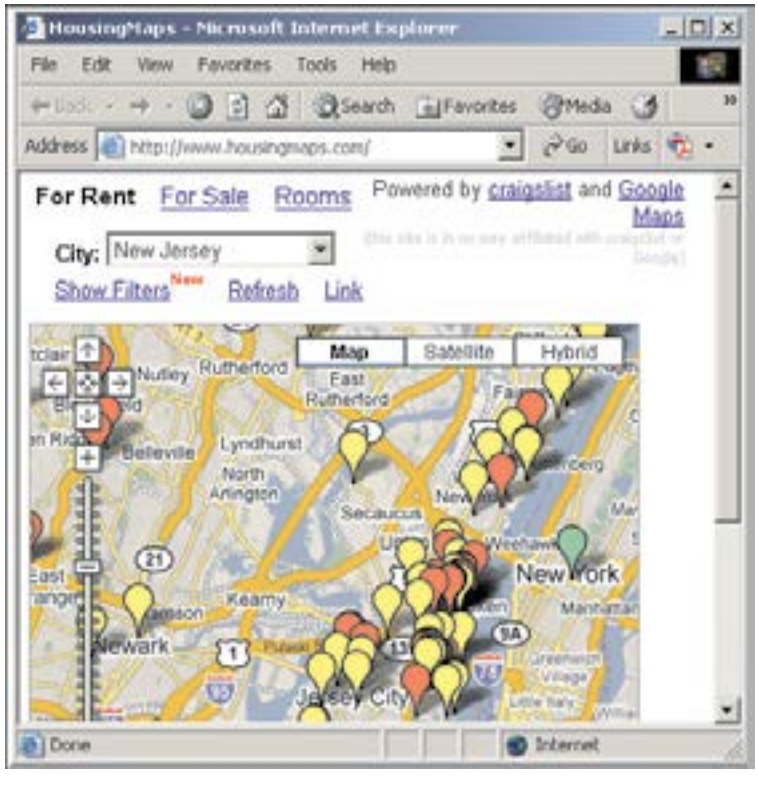


Figure 5.14: Google Maps

Concepts

5.2.2 Components

A mashup application consists of three different components. The three components are: API/content providers, the mashup site, and the client's Web browser.

The content provider is the data source. Data is retrieved by using APIs and different Web protocols. The Web protocols include Really Simple Syndication (RSS), Representational State Transfer (REST), and Web Service.

The mashup site uses data from different data sources.

The client's Web browser is the GUI of the mashup.

A mashup requires multiple input sources. The input sources will have an XML based output stream. The mashup uses these XML outputs as input, and consolidates them together. After combining the data streams, the mashup usually generates its own output, and displays a combination of the original inputs.

5.2.3 Types of Mashup Styles

Mashups can be implemented by using server-side content technologies such as Java servlets, CGI, PHP, and so on. Alternatively, they can also be implemented directly within the client's browser by using the client-side scripting technologies such as JavaScript, applets, and so on. Thus, mashups have two different styles – server-side mashups and client-side mashups. Generally, mashups use a combination of both server and client-side logic to achieve data aggregation.

Client side logic includes code embedded in the mashup Web pages, the scripting API libraries, or the applets referenced by the Web pages. Mashups created by using the client-side logic are termed as Rich Internet Applications (RIAs). An example of client-side technology is Google Maps API that can be accessed through client-side JavaScript. Figure 5.15 shows the implementation of Mashups.

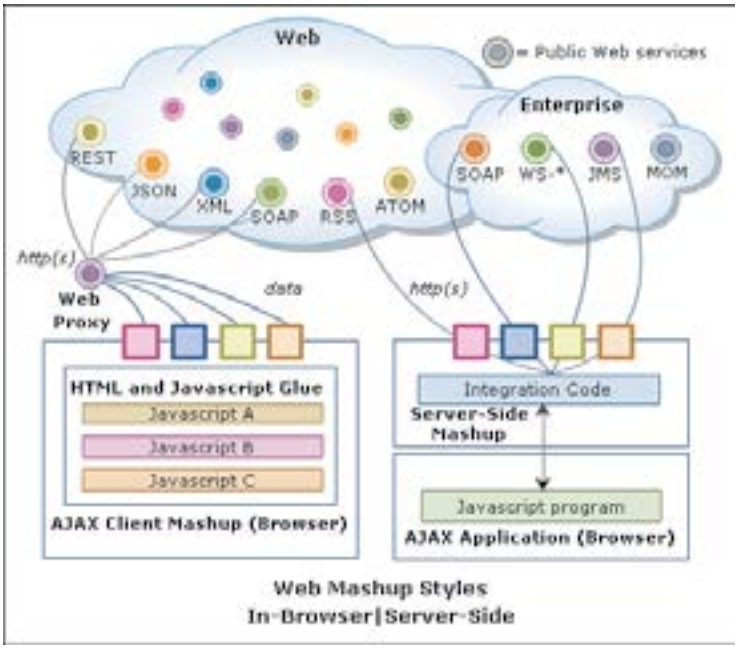


Figure 5.15: Implementation of Mashups

5.2.4 Working of Server-Side Mashup

The server is a proxy between Web applications on the client side, typically a browser, and the other Web sites used in the mashup.

Figure 5.16 shows the steps involved in the working of the server-side Mashup.

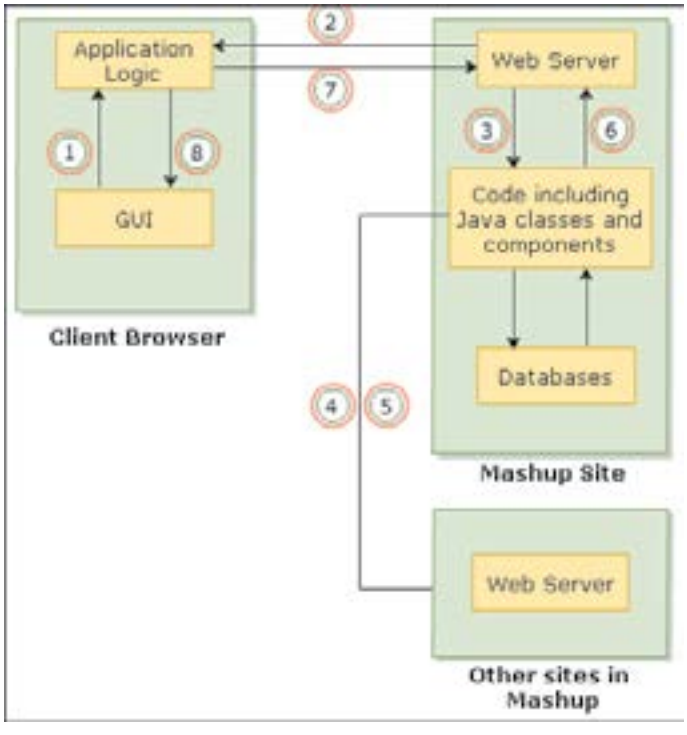


Figure 5.16: Server-side Mashup

1. The event invokes a JavaScript function.
2. The client sends a request in the form of `XmlHttpRequest` object to the server in the mashup site.
3. A Web component (servlets) receives a client request and invokes a method on the Java classes in mashup site. The Java classes contain the code that helps to connect and interact with the other Web sites in the mashup.
4. The server sends a request to the other Web sites in the mashup that provides the required service.
5. The other Web sites in the mashup site receives the request, processes the request, and returns the data to the server.
6. The server receives the response and transforms it to an appropriate data format which the client understands. It also caches the response which can be used for future request processing.
7. The Web component in the mashup site returns the response to the client.
8. A callback function in the `XmlHttpRequest` updates the page on the client side. This is achieved by manipulating the Document Object Model (DOM) that represents the page.

5.2.5 Working of Client-Side Mashup

The client-side mashups interact directly with the data and functionality of the other Web sites. It integrates the services and the content on the client side.

Figure 5.17 shows the steps involved in the working of the client-side Mashup.

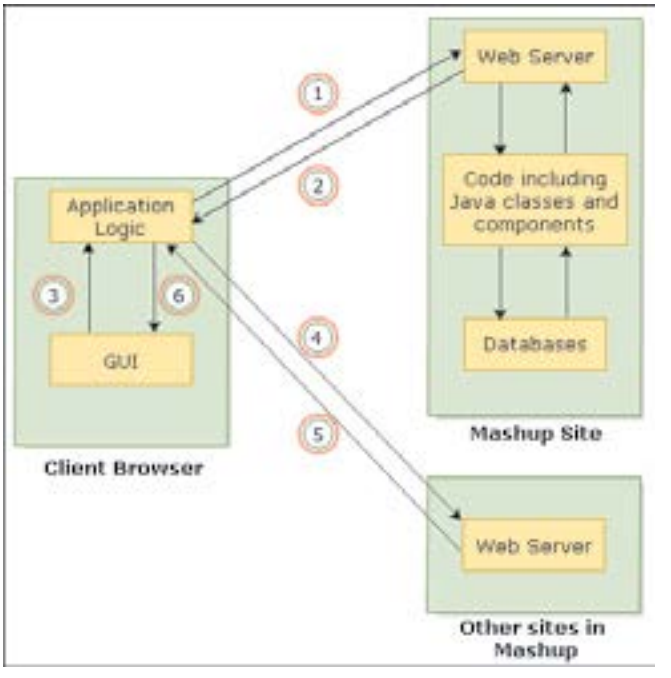


Figure 5.17: Client-side Mashup

1. The client browser requests a Web server in the mashup site for a Web page.
2. The server on the mashup site makes the JavaScript library available by loading the page into the client.
3. Actions in the Web page invoke a JavaScript function. The function generates a `<script>` element that links to the mashup site.
4. As per the request in the `<script>` element, appropriate requests are made to the Web sites in a mashup site to load the script.
5. A local callback in the browser executes with a JavaScript Object Notation (JSON) and loads the script.
6. A callback function in the `XmlHttpRequest` updates the page on the client side. This is achieved by manipulating the DOM that represents the page.

5.2.6 Advantages of Server-side Mashups

While deciding the mashup styles, the advantages of using server-side style are:

➔ It provides an easy access to the other mashup Web sites due to use of the Java EE and Java SE platforms libraries.

- The server-side mashups proxy acts as a buffer between the client and other Web sites to take care of the problems, and displays appropriate error messages.
- It manipulates the content by either sending data to the client in smaller chunks or sending only the portion of the data that the client needs.
- It supports caching and formatting of the data returned by the service.
- It handles the security requirements easily.
- It can make multiple concurrent and asynchronous calls.

5.2.7 Disadvantages of Server-side Mashups

Some of the disadvantages of server-side mashups are:

- It needs a server-side proxy.
- It may slow down the performance because of delay in receiving the response. This is so because a request and response move from the browser to the server side proxy, and then from the server side proxy back to the mashup server. Thus, before arriving at the client, the response makes same two network hops in reverse order.
- The server-side proxy needs protection from unauthorized access.

5.2.8 Design Considerations for Server-side Mashups

When developing a server-side mashup, some of the points to be considered in its design are:

- **Security**

  It is required to check whether the service requires a user login or some other authentication mechanism. Also check if the service hides the identity of the malicious user who uses the service illegally.

- **Response format**

  In a server side mashup, the response format that the service returns needs to be determined. The different response formats a service can return are XML, JSON, HTML, plain text, RSS/ATOM, and GData.

- **Results caching**

  It is required to check whether the data retrieved can be cached to be used by clients or is it required to be stored on the server.

- **Response modification**

  It is required to check whether there is a need to modify the response before it is sent to the client.

- **Exception and error handling**

  The exceptions that can occur when accessing the service are required to be handled. It is also required to validate the data input provided to a service.

5.2.9 Advantages of Client-side Mashups

While deciding mashup styles, the advantages of using client-side style are:

- It is easy to implement. It includes the JavaScript library in the Web page of the site that needs to be in a mashup. Then to use the functions in the library, provide the appropriate code in the Web page.
- It does not require a server-side component.
- It improves the performance because a request and response object moves directly from the browser to the mashup server and from server back to the browser.
- It reduces the processing load on the server by passing the service request and response object between client and mashup server without including the proxy server.
- It does not require installation of any custom plug in for its implementation.

5.2.10 Disadvantages of Client-side Mashups

Some of the disadvantages of client-side mashups are:

- It is difficult to handle the security requirements.
- It has no buffer like server-side styles has to protect the client from problems occurring in other Web sites.
- It needs to handle the data of any size and format returned by the Web sites, because the data returned is not formatted and manipulated.
- It has constraints on making many asynchronous calls to various data sources at the same time.

5.2.11 Design Considerations for Client-side Mashups

When developing a client-side mashup, some of the things to be considered in its design are:

➔ **Security**

It is required to check whether the client has access to the content and service for a mashup. Also, the unintended users should be restricted to access the site and data.

➔ **Performance**

It is required to check whether the other Web sites from mashup does not delay the response and frustrate the user.

➔ **Stability**

It is required to check whether the service's functions the same way as expected and remains the same way in future also.

➔ **Caching**

It is required to check whether the data retrieved can be cached to be used by multiple clients. Thus, it will reduce the number of calls to the mashup sites.

➔ **Format of returned data**

It is required to check whether the format of data returned by the Web sites in a mashup is compatible with the client browser. The responses returned from the mashup site can be either in XML, JSON, HTML, plain text or any other format. Browser can easily parse data in JSON format than in XML format.

Knowledge Check 2

1. Which of the following statements describing the mashup architecture are true?

(A)	The content provider acts as a source of data in a mashup.
(B)	A mashup is a Web site that combines content from various other Web sites.
(C)	Mashups does not facilitate communication between technologies.
(D)	Mashups can be implemented using only server side content technologies.
(E)	The client's Web browser is the GUI of the mashup.

(A)	A, B, and C	(C)	B, C, and E
(B)	A, B, and E	(D)	C, D, and E

2. Which of the following statements stating the advantages of client side mashups are false?

(A)	It reduces the processing load on the server.
(B)	It does not require any custom plug in for its implementation.
(C)	It can handle the security requirements easily.
(D)	It improves the performance because a request and response object communicates directly between the browser and the mashup server.
(E)	It requires a server side component.

(A)	B and C	(C)	A and D
(B)	C and E	(D)	D and E

Summary

In this module, **jMaki - II**, you learnt about:

➔ Data Models

The data model specifies the formal specification of the data expected by the widgets. The data models are defined for the various types of widgets such as menus, trees, tables, tabbed views and so on.

➔ Mashups

A mashup is a Web site that combines the content from various other Web sites into one convenient, easy-to-use portal. A mashup application consists of API/content providers, the mashup site, and the client's Web browser. The mashups have two different styles - server side mashups and client side mashups.

Module - 6

AJAX with Struts and JSF

Welcome to the module, **AJAX with Struts and JSF**. Two of the most widely used Web frameworks are Struts and JSF. However, both frameworks send and receive data synchronously. Therefore, while the server processes the user's request, the user cannot interact with the Web page. AJAX gives these frameworks the ability to send and receive data asynchronously. Thus, the user can interact with the Web page while the server is processing the request.

In the module, you will learn about:

- ➔ AJAX in Struts Applications
- ➔ AJAX in JSF Applications

6.1 Lesson Overview

In the first lesson, **AJAX in Struts Applications**, you will learn to:

- Explain AJAX and Web frameworks.
- Explain the steps to send an AJAX request using Struts.
- Explain the steps to process an AJAX request using Struts.

6.1.1 AJAX and Web Frameworks

To enable AJAX in a Web application, you write JavaScript code to send AJAX request. You write this code in a JSP page. A server-side component, such as a servlet processes the AJAX request.

Today developers prefer using Web frameworks like Struts and JSF to process AJAX requests. This is because these frameworks provide a higher-level of abstraction above the servlet API. Moreover, AJAX requests are similar to HTTP requests. The only difference is that an AJAX request is sent asynchronously. Therefore, using a Web framework with AJAX will allow you to leverage the benefits of both, the framework and AJAX. For example, AJAX will enable you to improve the response time and provide rich look and feel to Struts and JSF applications.

6.1.2 AJAX in Struts and JSF Application

Consider a Struts or a JSF Web application that allows you to download articles from a Web site only if you are a registered user. To become a registered user, you have to create an account. The Web application displays the account registration page as shown in figure 6.1.

The registration page contains textboxes to accept username, password, e-mail address and a Submit button. When a user enters his or her details and clicks the Submit button, the page waits for a response from the Web server. If the username already exists, the user will need to repeat the process of filling details in the registration page. This is because Struts and JSF follow the click-wait-refresh cycle.

If you use AJAX based applications, you can notify users about existence of the username as soon as the username textbox looses focus.

Figure 6.1 shows the registration page in a Web application with and without using AJAX.

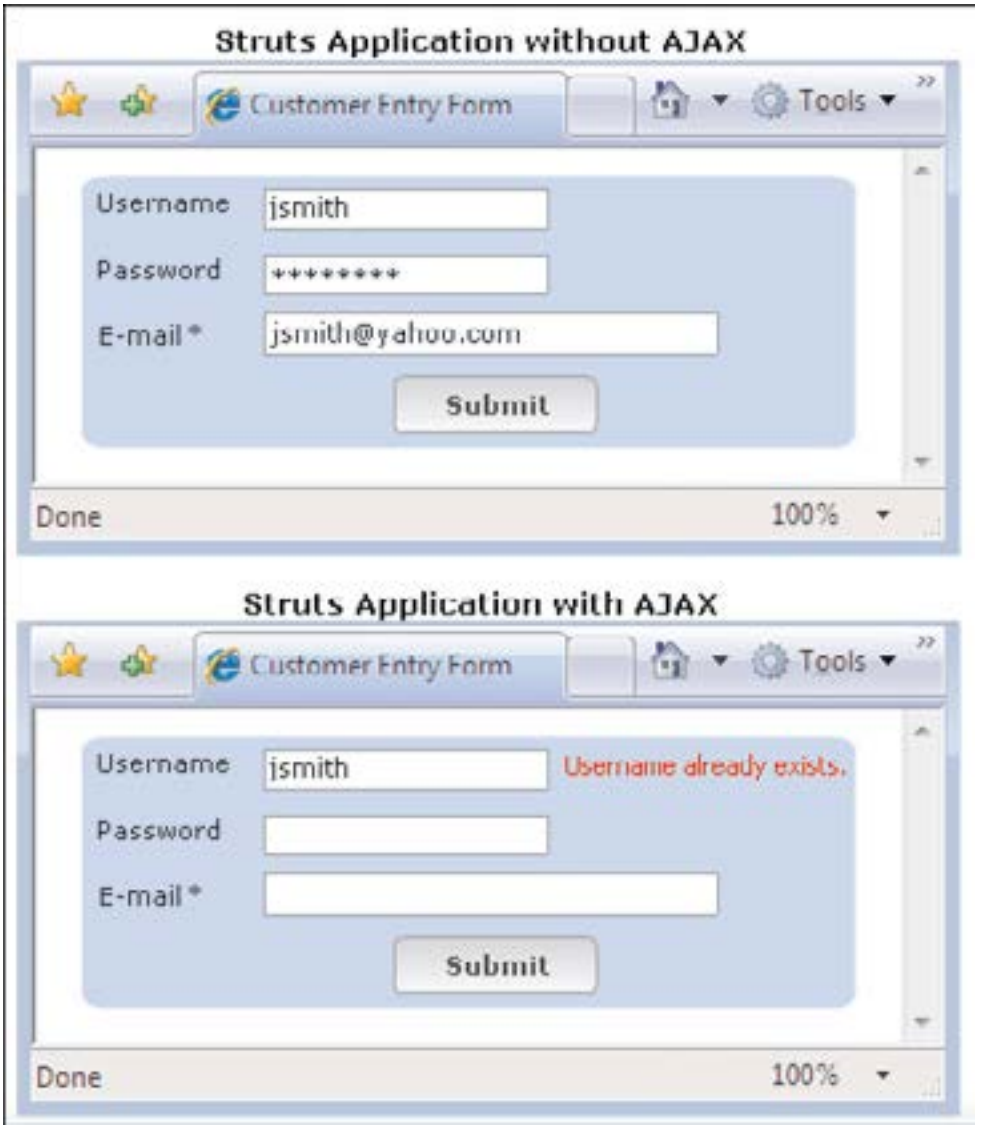


Figure 6.1: Registration Page

6.1.3 Steps to enable AJAX in Struts

To enable AJAX in Struts applications, you need to perform the following steps:

1. Bind a GUI component, such as a textbox in a JSP page, to an appropriate event handler.
2. Retrieve the data such as username from the GUI component.
3. Construct an AJAX request and send it to the Struts Action class.
4. Define a Struts Action class to accept data, forward the data to a model class and send the response back to the JSP page.
5. Define a model class that will process the data. For example, you can verify the existence of a

username in the database and then send corresponding information.

6. Update the JSP page such as account registration page to display information received.

6.1.4 Retrieving Username

For the application shown in figure 6.2, the username textbox is bound to the `onblur` event. The `onblur` event handler is bound to the JavaScript function `getUsername()`. The `getUsername()` function uses the `getElementById()` function to retrieve the textbox named `txtUsername`. This textbox is then passed to `sendAJAXRequest()` function to create an AJAX request.

The div element will display any response from the server.

Figure 6.2 shows the JavaScript code that retrieves the value of the textbox.

```
<script type="text/javascript">
  function getUsername() {
    // Retrieve the username entered by the user
    var txtName = document.getElementById(
    "txtUsername");

    sendAJAXRequest(txtName);
  }
</script>
...
...
<html:form action="UsernameAction">
  Username:<html:text property="username"
           styleId="txtUsername"
           onblur="getUsername();" />

  <div id="message"></div> <br>
  ...
</html:form>
```

Figure 6.2: JavaScript Code

6.1.5 Sending AJAX Request

Figure 6.3 shows the code of `sendAJAXRequest()` function. This function takes username as a parameter, constructs the AJAX request, and then sends the username to the Struts Action class.

```
...
// Step 1
var xmlRequest;
function sendAJAXRequest(username) {
  // Step 2
  if(window.XMLHttpRequest)
    xmlRequest = new XMLHttpRequest();
  else if(window.ActiveXObject)
    xmlRequest = new ActiveXObject("Microsoft.XMLHTTP");

  if(username != null) {
    // Step 3
    var context = "<%=request.getContextPath()%>";
    var actionURL = context +
      "/UsernameAction.do?username="
      + username.value;

    // Step 4
    xmlRequest.onreadystatechange = processResponse;

    // Step 5
    xmlRequest.open("GET", actionURL, true);
    xmlRequest.send(null);
  }
}
...
```

Figure 6.3: sendAJAXRequest() Function

➔ **Step 1**

The variable, `xmlRequest` holds a reference to the XML request object.

➔ **Step 2**

The if-else block creates the XML request object based on the browser type. If the Web page runs

on Internet Explorer, the `xmlRequest` variable is initialized to `Microsoft.XMLHTTP`. For all other browsers, the `xmlRequest` variable is initialized to `XMLHttpRequest`.

➔ **Step 3**

The URL is constructed. The URL comprises of following four parts:

- `context` – context path of the Web application
- `UsernameAction` – action path of the Struts Action class
- `username` – name of the request parameter
- `username.value` – username entered by the user

➔ **Step 4**

The `xmlRequest` variable's `onreadystatechange` property is set to the `processResponse()` function. Every time the state of the XML request object changes, the `processResponse()` function is executed.

➔ **Step 5**

Three arguments are passed to the `open()` method. The three arguments are:

- `GET` – indicates that request should be sent by using the HTTP GET method
- `actionURL` – specifies the action path of the Struts Action class
- `true` – indicates that the request will be sent asynchronously

Finally, the request is sent using the `send()` method.

6.1.6 Struts Action Class

To process the AJAX request in the Struts `Action` class, you write code in the `execute()` method.

Figure 6.4 demonstrates the code of the `execute()` method.

```
public ActionForward execute(ActionMapping mapping,
        ActionForm form,
        HttpServletRequest request,
        HttpServletResponse response)
        throws Exception {
  // Step 1
  String name = request.getParameter("username");

  // Step 2
  User objUser = new User();
  String xml = objUser.validateUsername(name);

  // Step 3
  response.setContentType("application/xml");
  response.setHeader("Cache-Control", "no-cache");
  PrintWriter writer = response.getWriter();
  writer.write(xml);
  return null;
}
```

Figure 6.4: execute() Method

Figure 6.4 shows the step-by-step code of the `execute()` method. The explanation of each step is as follows:

1. The `name` variable stores the value of the request parameter named `username`.
2. An instance of the model class, `User`, is created. The `validateUsername()` method checks for existence of the user's name and returns an XML fragment containing a true or a false value.
3. The response instance writes back the XML fragment on the client application. You must set the content-type of the response to `application/xml` to send an XML response.

6.1.7 Model Class

The Struts Action class is the controller of the application. The code that interacts with the database is placed in a different class. This class is also known as a model class or a data access object.

Figure 6.5 shows the code of a model class named `User`. To keep the program logic simple, a `HashMap` instance named `users` is initialized in the constructor and two usernames are added to it. This instance represents a dummy database.

The `validateUsername()` method checks if the user's name, passed as an argument, exists in the `users` instance. If the user's name exists, the string `true` is added to the `StringBuffer` instance, `xmlResponse`. Note that the element `exists` of user element encloses the string `true` or `false`. Finally, the XML fragment is returned as a `String` instance to the `execute()` method of the Struts `Action` class.

Figure 6.5 demonstrates the code for the `User` model class.

```
public class User {
  private HashMap<String, String> users;

  User() {
    users = new HashMap<String, String>();
    users.put("jsmith", "John Smith");
    users.put("nina.cortez", "Nina Cortez");
    ...
  }

  public String validateUsername(String name) {
    StringBuffer xmlResponse = new StringBuffer();

    xmlResponse.append("<user><exists>");
    if (users.containsKey(name))
      xmlResponse.append("true");
    else
        xmlResponse.append("false");
    xmlResponse.append("</exists></user>");

    return xmlResponse.toString();
  }
}
```

Figure 6.5: Model Class

6.1.8 Processing the Response

The Struts `Action` class sends the XML fragment to the JSP page for further processing. The function `processResponse()` in the JSP page processes the XML response.

Figure 6.6 demonstrates the code for the `processResponse()` function.

```
function processResponse() {
      // Step 1
      if(xmlRequest.readyState == 4) {
        if(xmlRequest.status == 200) {

      // Step 2
      var node =
xmlRequest.responseXML.getElementsByTagName("exists");
      var flag = node[0].childNodes[0].nodeValue;

      // Step 3
      var messageElement =
document.getElementById("message");
      messageElement.style.color = "red";

      // Step 4
      if( flag == "true") {
        messageElement.innerHTML = "Username already
exists.";
      } else {
        messageElement.innerHTML = "";
      }
    }
  }
}
```

Figure 6.6: processResponse() Function

➔ **Step 1**

The `processResponse()` function first checks the current values of the properties `readyState` and `status`. If the values of these properties are `4` and `200` respectively, it means that the function has received the response from the server successfully.

➔ **Step 2**

The function then retrieves all the elements having the name `exists` from the received XML response and stores the elements in a `node` variable. The text value of the first `exists` element is retrieved by accessing the first child node of the `node` variable. The value is stored in the `flag` variable. The `nodeValue` property is used to access the element value.

➔ **Step 3**

The `message` element is retrieved and the color of the element is changed to red. This element, in the JSP page, is a `div` element. The `div` element displays messages to the user.

➔ **Step 4**

If the `exists` element contains the text `true`, the `if` block is executed and message element's `innerHTML` property is set to display the message `Username already` exists. Otherwise, the `innerHTML` property is set to null.

Knowledge Check 1

1. Which of the following statements about AJAX and Web frameworks are true?

(A)	Struts and JSF use servlets to send HTTP requests.
(B)	AJAX-based applications can respond faster.
(C)	Web frameworks like Struts and JSF provide an abstraction layer above the servlet API.
(D)	Web applications write AJAX code in JSP pages.
(E)	Web frameworks like Struts and JSF improve the response time of Web applications.

(A)	A, C, and D	(C)	B, D, and E
(B)	B, C, and D	(D)	C, D, and E

2. Can you match the steps to enable AJAX in Struts applications with their corresponding descriptions?

	Description		Step
(A)	Sends AJAX request	(1)	Step 1
(B)	Updates the Web page	(2)	Step 4
(C)	Binds GUI component to event handler	(3)	Step 3
(D)	Processes AJAX request using Action and model class	(4)	Step 2
(E)	Retrieves data from GUI component.	(5)	Step 5

(A)	(A)-(2), (B)-(5), (C)-(1), (D)-(3), (E)-(4)	(C)	(A)-(3), (B)-(5), (C)-(1), (D)-(2), (E)-(4)
(B)	(A)-(4), (B)-(3), (C)-(1), (D)-(5), (E)-(2)	(D)	(A)-(5), (B)-(4), (C)-(3), (D)-(2), (E)-(1)

3. Consider a scenario where you have to use XMLHttpRequest object named xhrObject to send asynchronously the value of the textfield, orderID, to Action class named OrderAction. You process the response using orderResponse() function. Which one of the following code snippets will help you to achieve this?

(A)	`var context = "<%=request.getContextPath()%>";`
	`var actionURL = context +`
	`"orderId=" + orderID.value;`
	
	`xhrObject.onreadystatechange = orderResponse;`
	
	`xhrObject.open("GET", actionURL, true);`
	`xhrObject.send(null);`

(B)	`var context = "<%=request.getContextPath()%>";`
	`var actionURL = context +`
	`"/OrderAction.do?orderId="`
	`+ orderID.value;`
	
	`xhrObject.open("GET", actionURL, true);`
	`xhrObject.send(null);`

(C)	`var context = "<%=request.getContextPath()%>";`
	`var actionURL = context +`
	`"/OrderAction.do?orderId="`
	`+ orderID.value;`
	
	`xhrObject.onreadystatechange = orderResponse;`
	
	`xhrObject.open("GET", actionURL, true);`
	`xhrObject.send(null);`

Concepts

(D)	`var context = "<%=request.getContextPath()%>";`
	`var actionURL = context +`
	`"/OrderAction.do?orderId="`
	`+ orderID.value;`
	
	`xhrObject.onreadystatechange = orderResponse;`
	
	`xhrObject.open("GET", actionURL, false);`
	`xhrObject.send(null);`

6.2 Lesson Overview

In the second lesson, **AJAX in JSF Applications**, you will learn to:

- Describe custom component.
- Explain custom `UIOutput` component.
- Explain custom `UIInput` component.
- Describe the steps to add AJAX to custom `UIInput` component.

6.2.1 AJAX and Custom Component

JSF is a component UI framework. It provides a rich set of UI components, validators and converters to create Web applications. However, JSF also follows the click-wait-refresh cycle. In other words, JSF pages do not allow you to interact with the page until the response is received from the server.

You can use AJAX in JSF applications to send and receive data asynchronously. There are several approaches to include AJAX functionality in JSF applications. The easiest approach is to include the AJAX JavaScript code in JSF pages and use a servlet or a phase listener to process the AJAX request. However, this requires the page author to have sound knowledge of JavaScript. JSF provides the option of creating custom components. All the JavaScript code can be included in the custom component. The page author only needs to know how to use the custom component in the page.

6.2.2 Custom Component

The first step in AJAX enabling of a JSF application is to create a custom component or a custom tag. To create a JSF custom component, you create or use the following files:

- **UI Component Class** – contains the core logic of the component. This class is derived from either `UIInput` to create an input component, `UIOutput` to display static text, or `UICommand` class to create an event-based component.
- **Renderer Class** – contains the code to render the markup of the component. Usually, most simple custom components include the rendering code in the UI Component class itself.
- **UI Component Tag Class** – is the tag handler class and associates the custom tag with the component and renderer classes.
- **Tag Library Descriptor File** – describes the usage of custom component in JSP pages and associates the tag in JSP page with the UI Component tag class.
- **`faces-config.xml`** – specifies the name of custom component and the renderer.

Figure 6.7 shows the JSF components.

Figure 6.7: JSF Components

6.2.3 Custom "UIOutput" Component

To better understand the creation of a JSF custom component, the simplest example is to create a custom `UIOutput` component. Figure 6.8 shows the code of `CustomLabel` class representing a custom `UIOutput` component. This component will display the current date and time.

The class is derived from `UIOutput` class. This is because the component will display static text only. The code to write text or markup is included in the `encodeBegin()` method. In this case, you write the current date and time to the response. The class overrides the `getFamily()` method. This method returns a string identifying the family of a component. However, here the method returns `null`.

After the component class is defined, you configure it in the `faces-config.xml` file as shown in figure 6.8. The `component-type` element assigns a name to the component and `component-class` element specifies the fully qualified class name.

Figure 6.8 shows the definition of component class and its configuration in the `faces-config`.xml.

CustomLabel Class

```
public class CustomLabel extends UIOutput {
   public void encodeBegin(FacesContext context) {
    try {
      ResponseWriter writer =
context.getResponseWriter();
      writer.write(new Date().toString());
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  }

  public String getFamily() {
    return null;
  }
}
```

faces-config.xml

```
<component>
  <component-type>CustomLabel</component-type>
  <component-class>stock.CustomLabel</component-class>
</component>
```

Figure 6.8: Component Class and its Configuration

Note - The methods `encodeEnd()` and `encodeChildren()` are also used for rendering the component. The `encodeEnd()` method renders the ending tag. Some tags include child tags as well. The `encodeChildren()` method renders these child tags.

For simple components, all the rendering code is included in the `encodeBegin()` method. For complex components, you create a subclass of `Renderer` class and override the encode methods in the subclass.

6.2.4 Tag Handler Class

As the `CustomLabel` is a simple component, you need not create a separate renderer class. Hence, the next step is to create a tag handler class as shown in figure 6.9. Note that the class is derived from the `UIComponentELTag` class.

The `CustomLabelTag` class overrides the `getComponentType()` method. The return string of this method is the same as that of the `component-type` element in the faces-config.xml file. This is how the component and the tag handler are associated with each other.

The `getRendererType()` method returns the name of the renderer as specified in `renderer-type` element in `faces-config.xml` file. As the `CustomLabel` component does not provide a separate renderer class, this method returns `null`.

Figure 6.9 demonstrates the code of the `CustomLabel` tag handler class.

```
public class CustomLabelTag extends UIComponentELTag {

  public String getComponentType() {
    return "CustomLabel";
  }

  public String getRendererType() {
    return null;
  }
}
```

Figure 6.9: Tag Handler Class

Note - You may provide the rendering code in a separate class. This class must be derived from `Renderer` class and override one or more of the following methods: `encodeBegin()`, `encodeChildren()`, `encodeEnd()` and `decode()`.

6.2.5 Tag Library Descriptor

The next step in creating custom component is to create a tag library descriptor file as shown in figure 6.10. Some of the important XML elements of this file are:

- `uri` – specifies the path of tag library descriptor file; in this case, the file is located in the `tlds` folder of `WEB-INF` folder.
- `tag` – provides all the details about how the tag.
- `tag-class` – specifies the fully qualified class name of the tag handler class.

➔ `name` – `assigns` a short name to the tag handler class; in the JSP page, the tag is referred using the name `CustomLabelTag`.

Figure 6.10 shows the tag library descriptor file.

```
<?xml version="1.0" encoding="UTF-8"?>
<taglib version="2.0"
xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
web-jsptaglibrary_2_0.xsd">

  <tlib-version>1.1</tlib-version>
  <jsp-version>2.1</jsp-version>
  <short-name>CustomLabelTLD</short-name>
  <uri>/WEB-INF/tlds/CustomLabelTLD</uri>

  <tag>
    <name>CustomLabelTag</name>
    <tag-class>stock.CustomLabelTag</tag-class>
    <body-content>scriptless</body-content>
  </tag>
</taglib>
```

Figure 6.10: Tag Library Descriptor File

6.2.6 Using "CustomLabel" Component

After all the files related to custom component are created, you use it in the JSF page. The image shows a JSF page using the `CustomLabel` component.

Note that first you need to include a `taglib` directive for the tag library descriptor, `CustomLabelTLD`. The letters `si` are used as prefix for the custom tag.

In the body tag, the custom tag is referenced as `CustomLabelTag`. This same name was specified in the `name` element of tag library descriptor file.

Figure 6.11 demonstrates the code of the JSF page.

JSF Page

```
<%@taglib prefix="f" uri="http://java.sun.com/jsf/core"%>
<%@taglib prefix="ai" uri="/WEB-INF/tlds/CustomLabelTLD"%>

<html>
  <head>
    <title>Stock Quote</title>
  </head>
  <body>
    <f:view>
      <ai:CustomLabelTag />
    </f:view>
  </body>
</html>
```

Output

Tue Jul 22 09:24:38 IST 2008

Figure 6.11: JSF Page

6.2.7 Custom "UIInput" Component

A custom `UIInput` component renders a textbox to accept input from the user. Figure 6.12 shows a custom `UIInput` component that renders a textbox, a submit button and a message. This component accepts stock symbol as input and displays the price of the corresponding stock symbol.

To keep the application logic simple, this custom component will display the hard-coded price, `$1000.00`, only for the stock symbol `orcl`. For any other stock symbol, the component will display the stock price as `$0.00`. In a real-world scenario, you would use a Web service to get the stock price for any stock symbol entered.

The custom component is created in the JSF page using a custom tag, `stockItem`, as shown in the image. Note that `value` attribute is used to pass the stock symbol to the server.

Figure 6.12 shows the rendering of the `UIInput` component.

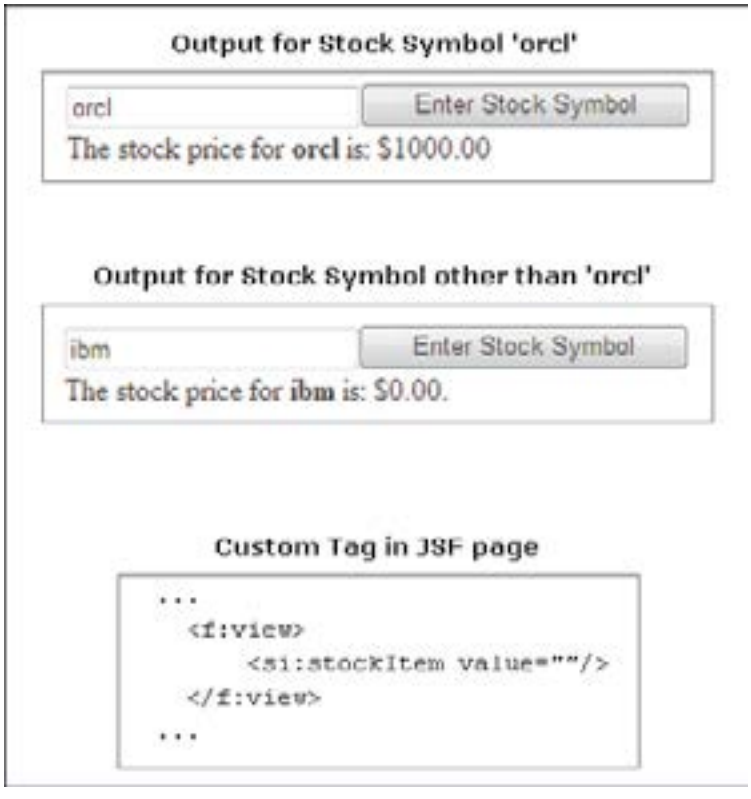


Figure 6.12: Rendering of UIInput Component

6.2.8 "StockUI" Component

The steps to create a custom `UIInput` component are almost similar to that of `UIOutput` component.

➔ **Step 1**

Figure 6.13 shows the code of a custom `UIInput` component named `StockUI` derived from `UI-Input`. The `StockUI` class overrides the `encodeBegin()` method to render the markup of the component.

First, you create an instance of `ResponseWriter` class. This instance is used to render the mark-up, of the component, on the client.

Figure 6.13 shows the code of the StockUI component.

```
public class StockUI extends UIInput {
  public void encodeBegin(FacesContext context) {
    try {
      ResponseWriter writer = context.getResponseWriter();

      writer.write("<input type='text' ");
      Object v = getValue();

      if (v != null)
        writer.write(" value='" + v.toString() + "'");

      writer.write(" name='stockSymbol'>");

      writer.write("<input type='submit'" +
                   " value='Enter Stock Symbol'>");

      String symbol = (String) getAttributes().get("value");
```

Figure 6.13: StockUI Component

➔ **Step 2**

Here, you create a textfield named `stockSymbol` to accept a symbol from the user. The custom tag has a `value` attribute to specify stock symbol. If this attribute is set, you retrieve its value using the `getValue()` method and assign it to the `value` attribute of the textfield.

Figure 6.14 shows the code to accept a symbol from the user.

```
      ResponseWriter writer = context.getResponseWriter();

      writer.write("<input type='text' ");
      Object v = getValue();

      if (v != null)
        writer.write(" value='" + v.toString() + "'");

      writer.write(" name='stockSymbol'>");

      writer.write("<input type='submit'" +
                   " value='Enter Stock Symbol'>");

      String symbol = (String) getAttributes().get("value");
      if (symbol != null) {
        if (symbol.equals("orcl")) {
          writer.write("<br>The stock price for <b>" +
```

Figure 6.14: Code - Accept Symbol

➔ **Step 3**

Here, you create a submit button with the caption `Enter Stock Symbol`. On clicking this button, the stock symbol entered in the `stockSymbol` textfield is sent as part of an HTTP POST request.

Figure 6.15 shows the code for the Submit button.

```
        writer.write(" name='stockSymbol'>");

        writer.write("<input type='submit'" +
                     " value='Enter Stock Symbol'>");

        String symbol = (String) getAttributes().get("value");
        if (symbol != null) {
          if (symbol.equals("orcl")) {
            writer.write("<br>The stock price for <b>" +
                         symbol + "</b> is: $1000.00");
          } else {
            writer.write("<br>The stock price for <b>" +
                         symbol + "</b> is: $0.00.");
          }
        }
      } catch (IOException ex) { ex.printStackTrace(); }
    }
```

Figure 6.15: Code - Submit Button

➔ Step 4

JSF maintains a request map that contains all the values sent in a HTTP POST request. In this step, you retrieve the value of the stock symbol from this map and compare it with `orcl`. If a match is found, the stock price `$1000.00` is displayed; else the stock price `$0.00` is displayed.

Figure 6.16 shows the code to compare the value of the stock symbol with `orcl`.

```
                     " value='Enter Stock Symbol'>");

        String symbol = (String) getAttributes().get("value");
        if (symbol != null) {
          if (symbol.equals("orcl")) {
            writer.write("<br>The stock price for <b>" +
                         symbol + "</b> is: $1000.00");
          } else {
            writer.write("<br>The stock price for <b>" +
                         symbol + "</b> is: $0.00.");
          }
        }
      } catch (IOException ex) { ex.printStackTrace(); }
    }
...
}
```

Figure 6.16: Code - Comparing Value of Stock Symbol

The `StockUI` class must override the `decode()` method as well. Figure 6.17 shows the code of decode() method.

The `decode()` method is used to process the request data. JSF maintains the request data in a `Map` instance. You retrieve this map using the `getRequestParameterMap()` method. Then, you use the `get()` method to retrieve the value of request parameter named `stockSymbol`. Then, you set the submitted value of the component to the value submitted by the user. This completes the creation of custom component, `StockUI`. Therefore, you configure it in `faces-config.xml` file as shown in the image.

Note that the `encodeBegin()` method rendered a textfield named `stockSymbol`. The user enters a stock symbol in this field and then clicks the submit button. This causes the textfield data to be sent as part of an HTTP POST request.

Figure 6.17 shows the code for the `decode()` method.

StockUI Component

```
public class StockUI extends UIInput {
  // code of encodeBegin() method is not included
  ...
  public void decode(FacesContext context) {
    ExternalContext eContext = context.getExternalContext();
    Map requestMap = eContext.getRequestParameterMap();

    try {
      String fieldValue = requestMap.get("stockSymbol").toString();
      setSubmittedValue(fieldValue);
      setValid(true);
    } catch (NumberFormatException ex) {
      // set the submitted value
      setSubmittedValue((String) requestMap.get("stockSymbol"));
    }
  }
}
```

faces-config.xml

```
...
<component>
  <component-type>StockUIComponent</component-type>
  <component-class>stock.StockUI</component-class>
</component>
...
```

Figure 6.17: decode() Method

Concepts

6.2.9 "StockUITag" Class

The next step in creating custom `UIInput` component is to create a tag handler class as shown in figure 6.18.

The `StockUITag` class is derived from `UIComponentELTag` class. The method `getComponentType()` returns the name of the component. The `getRendererType()` method returns null as the rendering was included in the `encodeBegin()` method of StockUI class.

At design time, you want to allow the user to specify the stock symbol using the value attribute of the custom tag. Hence, you declare an attribute named value of type ValueExpression and define a corresponding setter method.

Any attributes of a custom tag are set in the `setProperties()` method. Here, this method sets the `value` attribute.

Figure 6.18 shows the code of the `StockUITag` class.

```
public class StockUITag extends UIComponentELTag {
  public String getRendererType() {
    return null;
  }

  public String getComponentType() {
    return "StockUIComponent";
  }

  private ValueExpression value;
  public void setValue(ValueExpression symbol) {
    this.value = value;
  }

  protected void setProperties(UIComponent component) {
    super.setProperties(component);
    if (value != null) {
      if (!value.isLiteralText()) {
        component.setValueExpression("value", value);
      } else {
        component.getAttributes().put(
              "value", value.getExpressionString());
      }
    }
  }
}
```

Figure 6.18: StockUITag Class

Concepts

6.2.10 "StockUITag" Tag Library Descriptor

The next step in creating a custom `UIInput` component is to create a tag library descriptor file as shown in figure 6.19.

Most of the content of this file is similar to that of the tag library descriptor for `UIOutput` component. However, the `tag` element is different. The `name` element assigns the name, `stockItem` to the custom tag. The class, `StockUITag`, is specified as the tag handler class for the tag `stockItem`. The `attribute` element declares an attribute named `value` of type ValueExpression.

Figure 6.19 shows the `StockUITag` tag handler class.

```
<?xml version="1.0" encoding="UTF-8"?>
<taglib version="2.0"
xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
web-jsptaglibrary_2_0.xsd">
  <tlib-version>1.1</tlib-version>
  <jsp-version>2.1</jsp-version>
  <short-name>StockUITagTLD</short-name>
  <uri>/WEB-INF/tlds/StockUITagTLD</uri>

  <tag>
    <name>stockItem</name>
    <tag-class>stock.StockUITag</tag-class>
    <body-content>scriptless</body-content>
    <attribute>
      <name>value</name>
      <deferred-value>
        <type>javax.el.ValueExpression</type>
      </deferred-value>
    </attribute>
  </tag>
</taglib>
```

Figure 6.19: StockUITag Tag Handler Class

6.2.11 Adding AJAX to "UIInput" Component

To add AJAX functionality to a custom component you modify the custom component to include the AJAX script code and define a phase listener to process the AJAX request. Figure 6.20 shows the modifications made to `StockUI` component in dark blue color. Note that the `decode()` method is not overridden because the HTTP POST request will now be processed by a phase listener.

Figure 6.20 shows the code for the `StockUI` component to process the request by a phase listener.

```
public class StockUI extends UIInput {
  public void encodeBegin(FacesContext context) {
    try {
      // Step 1
      ResponseWriter writer = context.getResponseWriter();
      writer.write("<script type='text/javascript'"
              + " src='AJAXScript.js'> </script>");

      // Step 2
      writer.write("<input type='text' ");
      Object v = getValue();
      if (v != null) {
        writer.write(" value='" + v.toString() + "'");
      }
      writer.write(" name='stockSymbol'>");

      // Step 3
      writer.write("<input type='button'" +
              " value='Enter Stock Symbol' +
              " onclick='sendAJAXRequest();' >");

      // Step 4
      writer.write("<div id='message'></div>");
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  }
...
}
```

Figure 6.20: Phase Listener

➔ Step 1

In step 1, now you also render a script tag. This tag writes the code in `AJAXScript.js` to the client. The `AJAXScript.js` file contains code to send the stock symbol, asynchronously to the server, and display the stock price.

➔ Step 2

In step 2, you write the code to render a textfield named stockSymbol and set its value attribute to the `value` specified in the custom tag.

➔ Step 3

In step 3, now you render a simple button instead of a submit button. Additionally, you bind the

function, `sendAJAXRequest()` to the button's `onclick event`. Note that the `sendAJAXRequest()` is defined in `AJAXScript.js` file.

➔ **Step 4**

In step 4, now you just render a `div` tag. The `div` tag will be used to display the price of the stock symbol entered by the user.

6.2.12 AJAX Script Code

The `AJAXScript.js` file contains two functions. The `sendAJAXRequest()` function retrieves the stock symbol from the textfield `stockSymbol` and sends it to the phase listener. The `processResponse()` function displays the stock price received from the phase listener in the `div` element rendered on the client's Web page.

Figure 6.21 shows the code of the `sendAJAXRequest()` method.

```
var xhrObject;
var symbol;
function sendAJAXRequest() {
  // Create an XMLHttpRequest object
  if (window.XMLHttpRequest) {
    xhrObject = new XMLHttpRequest();
  } else if (window.ActiveXObject) {
    xhrObject = new ActiveXObject("Microsoft.XMLHTTP");
  }

   // Retrieve the textfield named stockSymbol
  symbol = document.getElementById("stockSymbol");

  // Create URL having stock symbol as a request parameter named value
  var url = "faces/Request-Ajax?value=" + escape(symbol.value);
  // Send the AJAX request and associate the function
  // processResponse()for processing the response
  xhrObject.open("GET", url, true);
  xhrObject.onreadystatechange = processResponse;
  xhrObject.send(null);
}
```

Figure 6.21: sendAJAXRequest() Method

Concepts

Figure 6.22 shows the code for the `processResponse()` method.

```
function processResponse( ) {
  // If the response is received successfuly,
  // display the stock price
  if (xhrObject.readyState == 4) {
    if (xhrObject.status == 200) {

      // Extract the XML response and then the node named stockprice
      var response = xhrObject.responseXML;
      var node = response.getElementsByTagName("stockprice")[0];

      // Extract div element named message and display the stock price
      // using the element's innerHTML property
      var message = document.getElementById("message");
      message.innerHTML = "Stock price of <b>" + symbol.value + "</b> is $"
        + node.childNodes[0].nodeValue;

    }
  }
}
```

Figure 6.22: processResponse() Method

6.2.13 Phase Listener

In JSF applications, an AJAX request from custom component is processed using a phase listener. Figure 6.23 shows the `StockPhaseListener` class implementing the `PhaseListener` interface. The class provides definitions for the interface methods `afterPhase()`, `beforePhase()`, and `getPhaseId()`.

Figure 6.23 shows the `StockPhaseListener` class

```
public class StockPhaseListener implements PhaseListener {
  public void afterPhase(PhaseEvent event) {
    String viewId =
event.getFacesContext().getViewRoot().getViewId();
    if (viewId.indexOf("Ajax") != -1) {
      try {
        processAjaxRequest(event);
      } catch (IOException ex) {
        ex.printStackTrace();
      }
    }
  }

  // Implement processAjaxRequest() method

  public void beforePhase(PhaseEvent event) {
  }
  public PhaseId getPhaseId() {
    return PhaseId.RESTORE_VIEW;
  }
}
```

Figure 6.23: StockPhaseListener Class

➔ **afterPhase**

The `afterPhase()` method is invoked after the processing of a particular phase is completed. In this method, you store the view identifier of the incoming request in variable `viewId`. If the variable `viewId` contains the text, `Ajax`, the processing of AJAX request is started by calling the `processAjaxRequest()` method. Note that this method is not defined yet.

➔ **beforePhase**

The `beforePhase()` method is invoked before the processing of a particular phase is started. In this case, the method body is empty because the phase listener processes the request in the `afterPhase()` method.

➔ **getPhaseId**

The `getPhaseId()` method identifies the phase for which the phase listener must process the request. In this case, the phase listener will process the AJAX request in the RESTORE_VIEW phase.

6.2.14 "processAjaxRequest()" Method

Figure 6.24 shows the code of `processAjaxRequest()` method. This method is responsible for processing the custom component, `StockUI's AJAX` request and sending it an XML response.

Figure 6.24 shows the code of the `processAjaxRequest()` method.

```
private void processAjaxRequest(PhaseEvent event)
        throws IOException {
  // Step 1
  FacesContext context = event.getFacesContext();
  Object object = context.getExternalContext().getRequest();
  HttpServletRequest request = (HttpServletRequest) object;
  String value = request.getParameter("value");

  // Step 2
  ExternalContext eContext = context.getExternalContext();
  HttpServletResponse response =
  (HttpServletResponse) eContext.getResponse();
  response.setContentType("application/xml");
  PrintWriter writer = response.getWriter();

  // Step 3
  if (value.equals("orcl")) {
    writer.write("<stockprice>1000.00</stockprice>");
  } else {
    writer.write("<stockprice>0.00</stockprice>");
  }
  event.getFacesContext().responseComplete();
}
```

Figure 6.24: processAjaxRequest() Method

➔ **Step 1**

In Step 1, you use an `ExternalContext` instance to store a reference to the current HTTP request in the variable, `request`. Next, the value of request parameter named `value` is stored in the variable `value`.

➔ **Step 2**

In Step 2, you use the `ExternalContext` instance to store a reference to the response. Next, you set the content type of the response to `application/xml` and create an instance of `PrintWriter` class.

➔ **Step 3**

In Step 3, you send the stock price to `StockUI` component by writing an XML response using the `writer` instance. Finally, you invoke the `responseComplete()` method to end the current request processing life cycle and return control to the `StockUI` component.

6.2.15 Configuring Phase Listener

After the phase listener class is defined, you must configure the phase listener in `faces-config.xml` file as shown in figure 6.25.

The `lifecycle` element is used to configure a phase listener. The fully qualified class name is specified in the `phase-listener` element. In this case, the phase listener is part of the package `stock` and hence the fully qualified class name becomes `stock.StockPhaseListener`.

Figure 6.25 shows the configuration of phase listener in `faces-config.xml` file.

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

<faces-config version="1.2"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee/
                      web-facesconfig_1_2.xsd">
  <lifecycle>
    <phase-listener>
      stock.StockPhaseListener
    </phase-listener>
  </lifecycle>
 ...
 ...
</faces-config>
```

Figure 6.25: Configuration of Phase Listener

Knowledge Check 2

1. Can you match the files and/or classes required to create custom component with their corresponding descriptions?

Description		File/Class	
(A)	Renders the component's markup.	(1)	UI Component
(B)	Describes the usage of the custom tag.	(2)	faces-config.xml
(C)	Contains core logic of the custom component.	(3)	Tag Class
(D)	Specifies the name of the custom component and renderer.	(4)	Renderer
(E)	Associates the custom tag, component and the renderer.	(5)	Tag Library Descriptor

(A)	(A)-(2), (B)-(5), (C)-(1), (D)-(3), (E)-(4)	(C)	(A)-(3), (B)-(5), (C)-(1), (D)-(2), (E)-(4)
(B)	(A)-(4), (B)-(5), (C)-(1), (D)-(2), (E)-(3)	(D)	(A)-(5), (B)-(4), (C)-(3), (D)-(2), (E)-(1)

2. Can you arrange the code to create a custom UIOutput component named Message to display a message Good Day!.

(A)	`ex.printStackTrace();` `}`
(B)	`try {` `ResponseWriter w = c.getResponseWriter();`
(C)	`}` `}`
(D)	`public class Message extends UIOutput{` `public void encodeBegin(FacesContext c){`
(E)	`w.write("Good Day!");` `} catch (IOException ex) {`

(A)	D, B, E, A, C	(C)	B, E, D, A, C
(B)	A, B, C, D, E	(D)	C, B, E, A, D

Concepts

3. Which one of the following `encodeBegin()` method can be used to create a custom `UIInput` component to render a textfield named `orderID`?

(A)	`public void encodeBegin(FacesContext context) {`
	`try {`
	`ResponseWriter writer = context.getResponseWriter();`
	`writer.write("<input type='text' ");`
	`writer.write(" name='orderID'>");`
	`} catch (IOException ex) {`
	`ex.printStackTrace();`
	`}`
	`}`

(B)	`public void encodeBegin(FacesContext context) {`
	`try {`
	`ResponseWriter writer = context.getResponseWriter();`
	`writer.write("<input type='button' ");`
	`writer.write(" name='orderID'>");`
	`} catch (IOException ex) {`
	`ex.printStackTrace();`
	`}`
	`}`

(C)	`public void encodeBegin() {`
	`try {`
	`ResponseWriter writer = getContext().getResponseWriter();`
	`writer.write("<input type='text' ");`
	`writer.write(" name='orderID'>");`
	`} catch (IOException ex) {`
	`ex.printStackTrace();`
	`}`
	`}`

Concepts

(D)	`public void encodeBegin(FacesContext context) {`
	`try {`
	`PrintStream writer = context.getResponseWriter();`
	`writer.write("<input type='text' ");`
	`writer.write(" name='orderID'>");`
	`} catch (IOException ex) {`
	`ex.printStackTrace();`
	`}`
	`}`

Summary

In the module, **AJAX with Struts and JSF**, you learnt about:

➔ AJAX in Struts Applications

The lesson, AJAX in Struts Application, explained that Struts suffers from click-wait-refresh cycle. Hence, you include AJAX code in JSP pages to send and receive data asynchronously. The AJAX request received from a JSP page is processed using the Struts `Action` class.

➔ AJAX in JSF Applications

The lesson, AJAX in JSF Applications, explained the basic concept of custom components. The lesson described the procedure to create custom `UIOutput` and `UIInput` components. At the end, the lesson explained how to add AJAX functionality to custom `UIInput` component and process the AJAX request using a phase listener.

Module 1

Knowledge Check 1

1. (A)
2. (B)

Knowledge Check 2

1. (C)

Module 2

Knowledge Check 1

1. (D)
2. (B)
3. (A)

Knowledge Check 2

1. (D)
2. (B)
3. (C)

Module 3

Knowledge Check 1

1. (A)
2. (B)
3. (A)

Knowledge Check 2

1. (B)
2. (A)
3. (C)

Module 4

Knowledge Check 1

1. (A)
2. (B)
3. (A)

Knowledge Check 2

1. **(C)**
2. (A)
3. (B)
4. (C)

Module 5

1. (D)
2. (C)

Knowledge Check 2

1. (B)
2. (B)

Module 6

Knowledge Check 1

1. (B)
2. (C)
3. (C)

Knowledge Check 2

1. (B)
2. (A)
3. (A)