Web Application Server: Evolution with Java Technologies

Phạm Huy Hoàng - 2015

hoangph@soict.hust.edu.vn

Mục lục
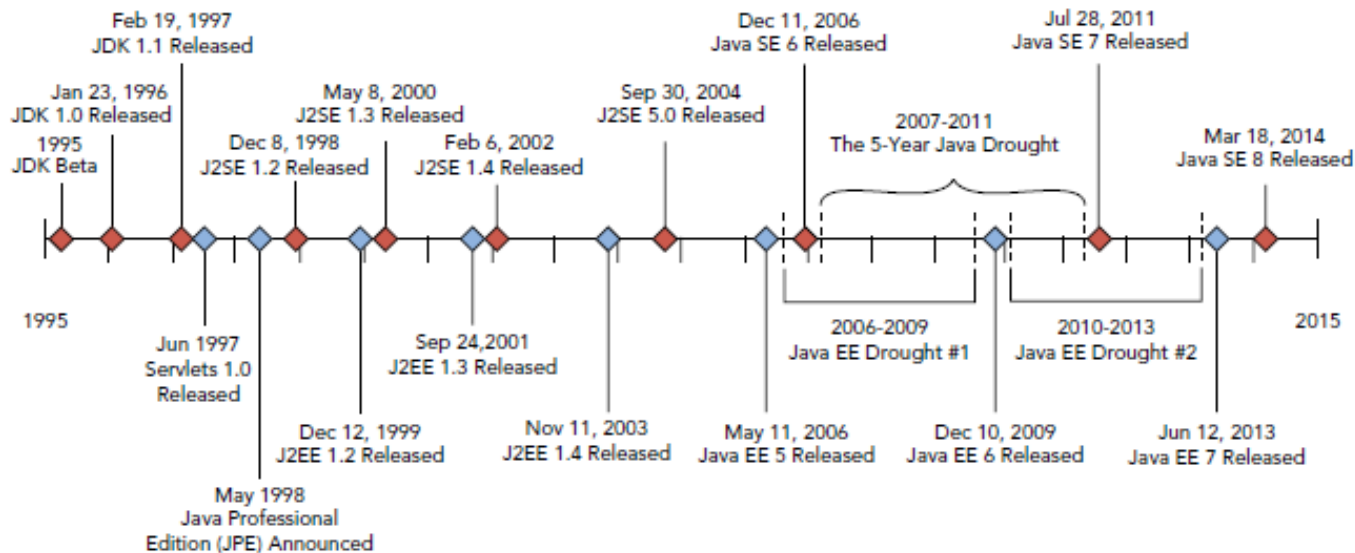
Chương 1. Tổng quan về Web tiến hóa với môi trường Java

1.1 A Timeline of Java Platforms

From (Williams, 2014) – trang 5

Hình vẽ 1: A timeline showing the correlation of the evolution of Java Platform

1.2 Java Servlet và JSP trong kiến trúc J2EE

(J2EE tutorial) trang 48.
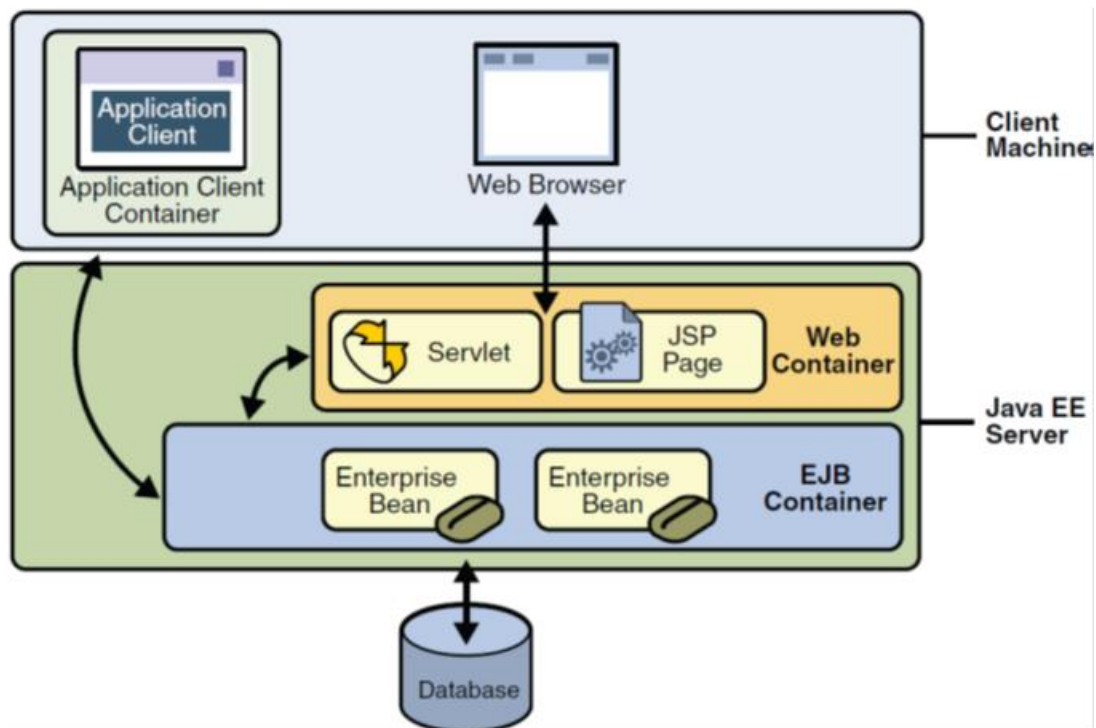
Hình vẽ 2: Vị trí của Java Servlet và JSP trong kiến trúc J2EE

1.3 J2EE API (phiên bản 5)

(J2EE tutorial) trang 53.
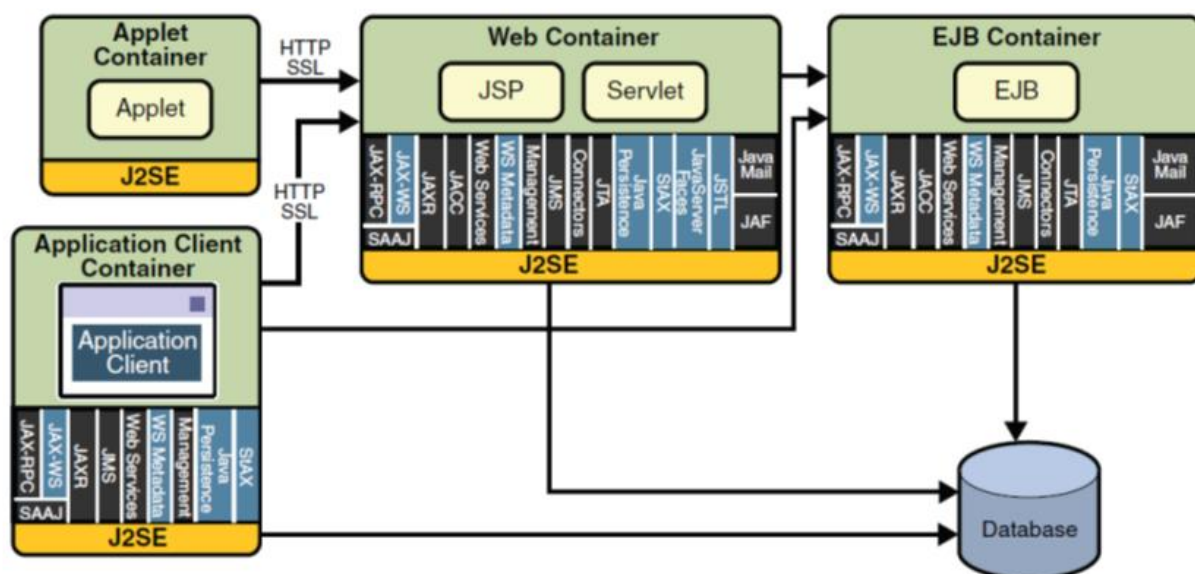
Hình vẽ 3: J2EE API (phiên bản 5)

Enterprise JavaBeansTechnology

An Enterprise JavaBeans (EJB) component, or enterprise bean, is a body of code having fields and methods to implement modules of business logic. You can think of an enterprise bean as a building block that can be used alone or with other enterprise beans to execute business logic on the Java EE server.

There are two kinds of enterprise beans: session beans and message-driven beans. A session bean represents a transient conversation with a client. When the client finishes executing, the session bean and its data are gone. A message-driven bean combines features of a session bean FIGURE 1–7 Java EE Platform APIs and a message listener, allowing a business component to receive messages asynchronously. Commonly, these are JavaMessage Service (JMS) messages.

In Java EE 5, entity beans have been replaced by Java persistence API entities. An entity represents persistent data stored in one row of a database table. If the client terminates, or if the server shuts down, the persistence manager ensures that the entity data is saved.

Java ServletTechnology

Java servlet technology lets you define HTTP-specific servlet classes. A servlet class extends the capabilities of servers that host applications that are accessed by way of a request-response programming model. Although servlets can respond to any type of request, they are commonly used to extend the applications hosted by web servers.

JavaServer PagesTechnology

JavaServer Pages (JSP) technology lets you put snippets of servlet code directly into a text-based document. A JSP page is a text-based document that contains two types of text:

static data (which can be expressed in any text-based format such as HTML, WML, and XML) and JSP elements, which determine how the page constructs dynamic content.

JavaServer Pages StandardTag Library

The JavaServer Pages Standard Tag Library (JSTL) encapsulates core functionality common to many JSP applications. Instead of mixing tags from numerous vendors in your JSP applications, you employ a single, standard set of tags. This standardization allows you to deploy your applications on any JSP container that supports JSTL and makes it more likely that the implementation of the tags is optimized.

JSTL has iterator and conditional tags for handling flow control, tags for manipulating XML documents, internationalization tags, tags for accessing databases using SQL, and commonly used functions.

JavaServer Faces

JavaServer Faces technology is a user interface framework for building web applications. The main components of JavaServer Faces technology are as follows:

- A GUI component framework.
- A flexible model for rendering components in different kinds of HTML or different markup languages and technologies. A Renderer object generates the markup to render the component and converts the data stored in a model object to types that can be represented in a view.
- A standard RenderKit for generating HTML/4.01 markup.

The following features support the GUI components:

- Input validation
- Event handling
- Data conversion between model objects and components
- Managed model object creation
- Page navigation configuration

All this functionality is available using standard Java APIs and XML-based configuration files.

Java Message Service API

The JavaMessage Service (JMS) API is a messaging standard that allows Java EE application components to create, send, receive, and read messages. It enables distributed communication that is loosely coupled, reliable, and asynchronous.

JavaTransaction API

The Java Transaction API (JTA) provides a standard interface for demarcating transactions. The Java EE architecture provides a default auto commit to handle transaction commits and rollbacks. An auto commit means that any other applications that are viewing data will see the updated data after each database read or write operation.However, if your application performs two separate database access operations that depend on each other, you

will want to use the JTA API to demarcate where the entire transaction, including both operations, begins, rolls back, and commits.

JavaMail API

Java EE applications use the JavaMail API to send email notifications. The JavaMail API has two parts: an application-level interface used by the application components to send mail, and a service provider interface. The Java EE platform includes JavaMail with a service provider that allows application components to send Internet mail.

JavaBeans Activation Framework

The JavaBeans Activation Framework (JAF) is included because JavaMail uses it. JAF provides standard services to determine the type of an arbitrary piece of data, encapsulate access to it, discover the operations available on it, and create the appropriate JavaBeans component to perform those operations.

Java API for XML Processing

The Java API for XML Processing (JAXP), part of the Java SE platform, supports the processing of XML documents usingDocument ObjectModel (DOM), Simple API for XML (SAX), and Extensible Stylesheet Language Transformations (XSLT). JAXP enables applications to parse and transform XML documents independent of a particular XML processing implementation.

JAXP also provides namespace support, which lets you work with schemas that might otherwise have naming conflicts.Designed to be flexible, JAXP lets you use any XML-compliant parser or XSL processor from within your application and supports the W3C schema. You can find information on the W3C schema at this URL: http://www.w3.org/XML/Schema.

Java API for XMLWeb Services (JAX-WS)

The JAX-WS specification provides support for web services that use the JAXB API for binding XML data to Java objects. The JAX-WS specification defines client APIs for accessing web services as well as techniques for implementing web service endpoints. The Web Services for J2EE specification describes the deployment of JAX-WS-based services and clients. The EJB and servlet specifications also describe aspects of such deployment. It must be possible to deploy JAX-WS-based applications using any of these deployment models.

The JAX-WS specification describes the support for message handlers that can process message requests and responses. In general, these message handlers execute in the same container and with the same privileges and execution context as the JAX-WS client or endpoint component with which they are associated. These message handlers have access to the same JNDI java:comp/env namespace as their associated component. Custom serializers and deserializers, if supported, are treated in the same way as message handlers.

Java Architecture for XML Binding (JAXB)

The Java Architecture for XML Binding (JAXB) provides a convenient way to bind an XML schema to a representation in Java language programs. JAXB can be used independently or in combination with JAX-WS, where it provides a standard data binding for web service messages.

All Java EE application client containers, web containers, and EJB containers support the JAXB API.

SOAP with Attachments API for Java

The SOAP with Attachments API for Java (SAAJ) is a low-level API on which JAX-WS and JAXR depend. SAAJ enables the production and consumption of messages that conform to the SOAP 1.1 specification and SOAP with Attachments note.Most developers do not use the SAAJ API, instead using the higher-level JAX-WS API.

Java API for XML Registries

The Java API for XML Registries (JAXR) lets you access business and general-purpose registries over the web. JAXR supports the ebXML Registry and Repository standards and the emerging UDDI specifications. By using JAXR, developers can learn a single API and gain access to both of these important registry technologies.

Additionally, businesses can submit material to be shared and search for material that others have submitted. Standards groups have developed schemas for particular kinds of XML documents; two businesses might, for example, agree to use the schema for their industry's standard purchase order form. Because the schema is stored in a standard business registry, both parties can use JAXR to access it.

J2EE Connector Architecture

The J2EE Connector architecture is used by tools vendors and system integrators to create resource adapters that support access to enterprise information systems that can be plugged in to any Java EE product. A resource adapter is a software component that allows Java EE application components to access and interact with the underlying resource manager of the EIS. Because a resource adapter is specific to its resource manager, typically there is a different resource adapter for each type of database or enterprise information system.

The J2EE Connector architecture also provides a performance-oriented, secure, scalable, and message-based transactional integration of Java EE-based web services with existing EISs that can be either synchronous or asynchronous. Existing applications and EISs integrated through the J2EE Connector architecture into the Java EE platform can be exposed as XML-based web services by using JAX-WS and Java EE component models. Thus JAX-WS and the J2EE Connector architecture are complementary technologies for enterprise application integration (EAI) and end-to-end business integration.

Java Database Connectivity API

The JavaDatabase Connectivity (JDBC) API lets you invoke SQL commands from Java programming language methods. You use the JDBC API in an enterprise bean when you

have a session bean access the database. You can also use the JDBC API from a servlet or a JSP page to access the database directly without going through an enterprise bean.

The JDBC API has two parts: an application-level interface used by the application components to access a database, and a service provider interface to attach a JDBC driver to the Java EE platform.

Java Persistence API

The Java Persistence API is a Java standards-based solution for persistence. Persistence uses an object-relational mapping approach to bridge the gap between an object oriented model and a relational database. Java Persistence consists of three areas:

- The Java Persistence API
- The query language
- Object/relational mapping metadata

Java Naming and Directory Interface

The JavaNaming and Directory Interface (JNDI) provides naming and directory functionality, enabling applications to access multiple naming and directory services, including existing naming and directory services such as LDAP, NDS,DNS, and NIS. It provides applications with methods for performing standard directory operations, such as associating attributes with objects and searching for objects using their attributes. Using JNDI, a Java EE application can store and retrieve any type of named Java object, allowing Java EE applications to coexist with many legacy applications and systems.

Java EE naming services provide application clients, enterprise beans, and web components with access to a JNDI naming environment. A naming environment allows a component to be customized without the need to access or change the component's source code. A container implements the component's environment and provides it to the component as a JNDI naming context.

A Java EE component can locate its environment naming context using JNDI interfaces. A component can create a javax.naming.InitialContext object and looks up the environment naming context in InitialContext under the name java:comp/env. A component's naming environment is stored directly in the environment naming context or in any of its direct or indirect subcontexts.

A Java EE component can access named system-provided and user-defined objects. The names of system-provided objects, such as JTA UserTransaction objects, are stored in the environment naming context, java:comp/env. The Java EE platform allows a component to name user-defined objects, such as enterprise beans, environment entries, JDBC DataSource objects, and message connections. An object should be named within a subcontext of the naming environment according to the type of the object. For example, enterprise beans are named within the subcontext java:comp/env/ejb, and JDBC DataSource references in the subcontext java:comp/env/jdbc.

Java Authentication and Authorization Service

The Java Authentication and Authorization Service (JAAS) provides a way for a Java EE application to authenticate and authorize a specific user or group of users to run it. JAAS is a Java programming language version of the standard Pluggable Authentication Module (PAM) framework, which extends the Java Platform security architecture to support user-based authorization.

Simplified Systems Integration

The Java EE platform is a platform-independent, full systems integration solution that creates an open marketplace in which every vendor can sell to every customer. Such a marketplace encourages vendors to compete, not by trying to lock customers into their technologies but instead by trying to outdo each other in providing products and services that benefit customers, such as better performance, better tools, or better customer support.

The Java EE 5 APIs enable systems and applications integration through the following:

- Unified application model across tiers with enterprise beans
- Simplified request-and-response mechanism with JSP pages and servlets
- Reliable security model with JAAS
- XML-based data interchange integration with JAXP, SAAJ, and JAX-WSSimplified interoperability with the J2EE Connector

1.4 Máy chủ ứng dụng Java (Java Application Server)

Thương mại:

- IBM WebSphere

Free:

- Tomcat

1.5 Cài đặt môi trường

1.5.1 JDK / JRE

- JRE 7: (https://java.com/en/download/manual_java7.jsp), cài đặt vào thư mục "C:\work\jre7"
- Tạo biến môi trường JRE_HOME = "C:\work\jre7"

1.5.2 Application Server Tomcat

- Tomcat 7: http://tomcat.apache.org/download-70.cgi (chỉ cần bản core), cài đặt vào thư mục "C:\work\tomcat7"
- Tạo biến môi trường CATALINA_HOME = "C:\work\tomcat7"
- Khởi động server Tomcat: C:\work\tomcat7\bin\startup.bat. Mặc định server nghe ở cổng TCP 8080. Dùng browser kết nối http://localhost:8080 để kiểm tra server đã chạy tốt. Để kết thúc server, chạy chương trình Shutdown: C:\work\tomcat7\bin\shutdown.bat hoặc đóng cửa sổ Tomcat server

- Kiểm tra server Tomcat bằng cách tạo file hello.jsp đặt trong thư mục "tomcat7\webapps\examples" với nội dung sau:

```html
<html>
    <head>
        <title>Hello World</title>
    </head>
    <body>
        Hello World!<br/>
        <% out.println("Your IP address is " + request.getRemoteAddr()); %>
    </body>
</html>
```

- Dùng browser truy nhập đến địa chỉ "http://localhost:8080/examples/hello.jsp", kết quả trang web là:

```
Hello World!
Your IP address is 127.0.0.1
```

Mặc định các file của mỗi ứng dụng web được để trong thư mục "tomcat7\webapps\xxx" trong đó "xxx" là tên ứng dụng và được truy nhập bằng browser tại địa chỉ "http://localhost:8080/xxx" như ví dụ với file hello.jsp. Có thể thiết lập để các ứng dụng này được đặt trong các thư mục khác bằng Context file. Ví dụ muốn tạo ứng dụng web "hello_jsp" và để các file tại thư mục "C:\work\JavaProg\hello_jsp", cần tạo file "hello_jsp.xml" trong thư mục "tomcat7\conf\Catalina\localhost" với nội dung như sau:

```xml
<Context path="/hello_jsp" docBase="C:/work/JavaProg/hello_jsp" />
```

Copy file hello.jsp bên trên vào thư mục "C:/work/JavaProg/hello_jsp", khi đó có thể truy nhập đến ứng dụng này theo địa chỉ "http://localhost:8080/hello_jsp/hello.jsp"

1.5.3 Eclipse

- Download Eclipse Luna for Java EE:
  http://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/luna/SR1a/eclipse-jee-luna-SR1a-win32-x86_64.zip
- Thiết lập server Tomcat trong Eclipse:

Khởi động Eclipse, hiển thị cửa sổ các server (menu Window > Show View > Server). Trong cửa sổ server, tạo server Tomcat (menu chuột phải > New > Server, trong mục "Select server type" chọn "Apache" và tìm đến phiên bản Tomcat đúng với bản đã cài đặt trong mục trên, đưa các thông số còn lại vào và click "Finish". Eclipse sẽ hoàn thành nốt các công việc kết nối môi trường với máy chủ Tomcat. Nếu thành công, trong cửa sổ Server sẽ nhìn thấy máy chủ Tomcats vừa tạo. Click chuột phải vào máy chủ này và chọn "Start". Kiểm tra thông tin log của máy chủ trong cửa sổ Console và dùng browser kết nối kiểm tra server Tomcat tại địa chỉ http://localhost:8080.

Chương 2. Java Servlet

API reference: https://docs.oracle.com/javaee/7/api/toc.htm

2.1 Mô hình ứng dụng Web với Java Servlet

In the Java 2 platform, web components provide the dynamic extension capabilities for a web server. Web components are either Java servlets, JSP pages, or web service endpoints. The interaction between a web client and a web application is illustrated in Figure 3–1. The client sends an HTTP request to the web server. A web server that implements Java Servlet and JavaServer Pages technology converts the request into an HTTPServletRequest object. This object is delivered to a web component, which can interact with JavaBeans components or a database to generate dynamic content. The web component can then generate an HTTPServletResponse or it can pass the request to another web component. Eventually a web component generates a HTTPServletResponse object. The web server converts this object to an HTTP response and returns it to the client.

(J2EE tutorial) trang 78
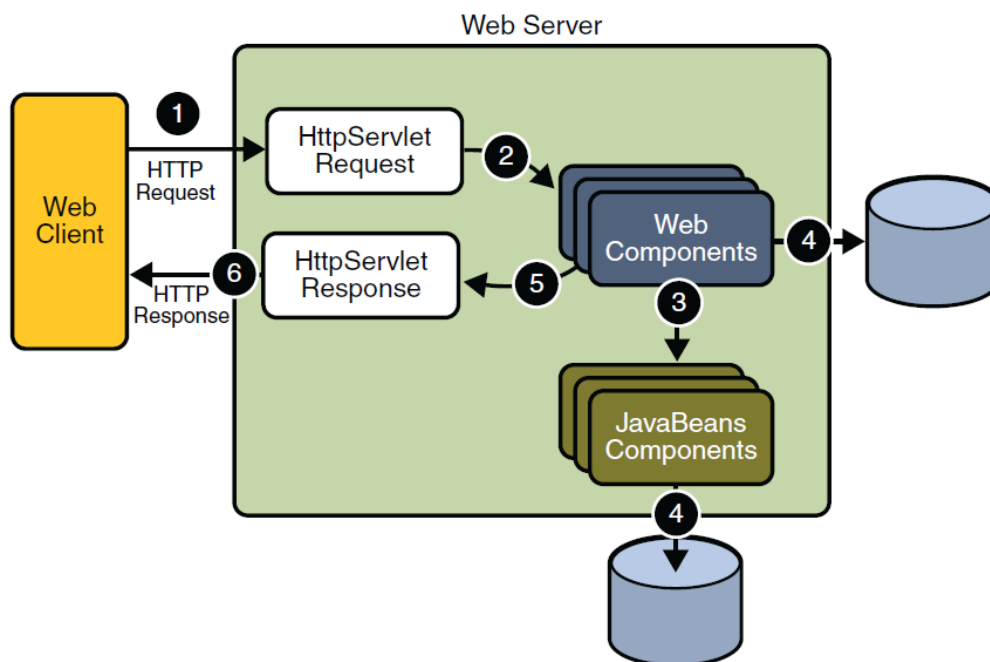
Hình vẽ 4: Java WebApplication RequestHandling

Servlets are Java programming language classes that dynamically process requests and construct responses. JSP pages are text-based documents that execute as servlets but allow a more natural approach to creating static content. Although servlets and JSP pages can be used interchangeably, each has its own strengths. Servlets are best suited for service-oriented applications (web service endpoints are implemented as servlets) and the control functions of a presentation-oriented application, such as dispatching requests and handling nontextual data. JSP pages are more appropriate for generating text-based markup such as HTML, Scalable Vector Graphics (SVG), WirelessMarkup Language (WML), and XML.

Since the introduction of Java Servlet and JSP technology, additional Java technologies and frameworks for building interactive web applications have been developed. Figure 3–2 illustrates these technologies and their relationships.
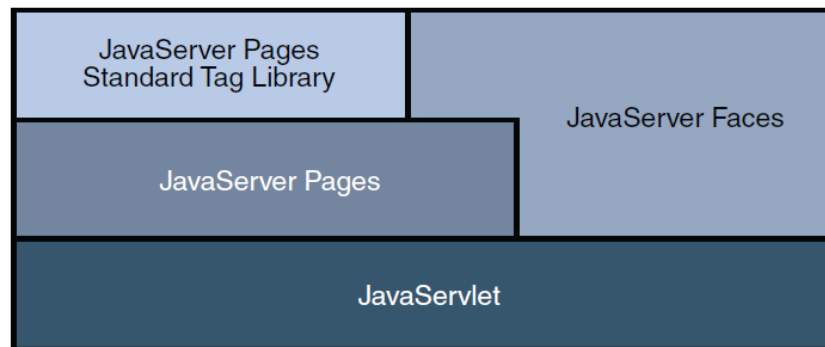
Hình vẽ 5: JavaWebApplication Technologies

Notice that Java Servlet technology is the foundation of all the web application technologies, so you should familiarize yourself with the material in Chapter 4, "Java Servlet Technology," even if you do not intend to write servlets. Each technology adds a level of abstraction that makes web application prototyping and development faster and the web applications themselves more maintainable, scalable, and robust.

Web components are supported by the services of a runtime platform called a web container. A web container provides services such as request dispatching, security, concurrency, and life-cycle management. It also gives web components access to APIs such as naming, transactions, and email.

Certain aspects of web application behavior can be configured when the application is installed, or deployed, to the web container. The configuration information is maintained in a text file in XML format called a web application deployment descriptor (DD). ADDmust conform to the schema described in the Java Servlet Specification.

This chapter gives a brief overview of the activities involved in developing web applications. First it summarizes the web application life cycle. Then it describes how to package and deploy very simple web applications on the Application Server. It moves on to configuring web applications and discusses how to specify the most commonly used configuration parameters. It then introduces an example, Duke's Bookstore, which illustrates all the Java EE web-tier technologies, and describes how to set up the shared components of this example. Finally it discusses how to access databases from web applications and set up the database resources needed to run Duke's Bookstore.

2.2 Hello World Servlet Application

2.2.1 Tạo project my_servet với Eclipse

- Eclipse: new Dynamic Web Project, đặt tên là my_servlet
- Thêm library Servlet API "servlet-api" trong thư mục "tomcat7\lib"
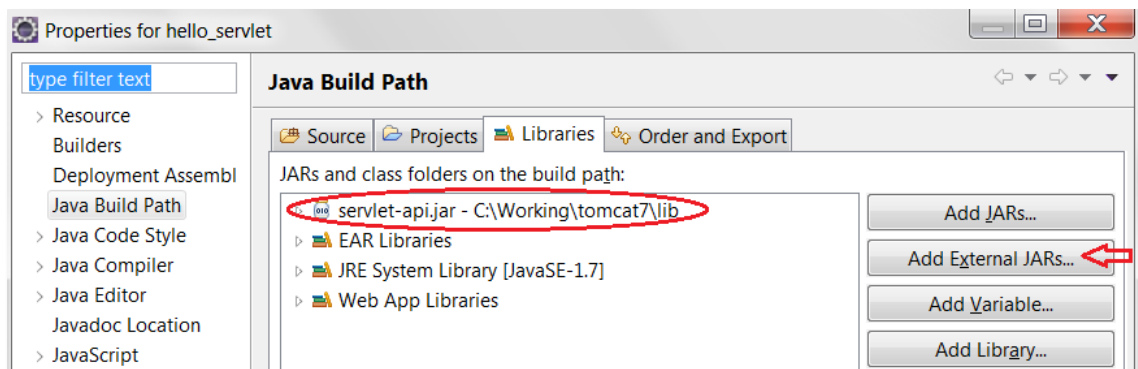
Hình vẽ 6: Bổ sung thư viện Servlet API vào project trong Eclipse

- New Servlet: GreetingServlet

```java
package servlets;

import java.io.*;
import java.util.*;
import java.sql.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class GreetingServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
        response.setContentType("text/html");
        response.setBufferSize(8192);

        PrintWriter out = response.getWriter();

        // then write the data of the response
        out.println("<html>" + "<head><title>Hello</title></head>");

        // then write the data of the response
        out.println(
            "<body  bgcolor=\"#ffffff\">"
            + "<h2>Hello, my name is Duke. What's yours?</h2>"
            + "<form method=\"get\" >"
            //+ "<form method=\"get\" action=\"/hello_servlet/response\">"
            + "<input type=\"text\" name=\"username\" size=\"25\">"
            + "<p></p>" + "<input type=\"submit\" value=\"Submit\">"
            + "<input type=\"reset\" value=\"Reset\">" + "</form>");

        String username = request.getParameter("username");

        if ((username != null) && (username.length() > 0)) {
            RequestDispatcher dispatcher = getServletContext()
                            .getRequestDispatcher(
                "/response");

            if (dispatcher != null) {
                dispatcher.include(request, response);
            }
        }

        out.println("</body></html>");
```

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

- New Servlet: ResponseServlet

```java
package servlets;

import java.io.*;
import java.util.*;
import java.sql.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ResponseServlet extends HttpServlet {
    public void doGet(
        HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException {
        PrintWriter out = response.getWriter();

        // then write the data of the response
        String username = request.getParameter("username");

        if ((username != null) && (username.length() > 0)) {
            out.println("<h2>Hello, " + username + "!</h2>");
        }
    }
}
```

- Tạo file web.xml trong thư mục "my_servlet\WebContent\WEB-INF"

```xml
<web-app >
 <display-name>My Servlet Application</display-name>

 <servlet>
  <display-name>GreetingServlet</display-name>
  <servlet-name>GreetingServlet</servlet-name>
  <servlet-class>servlets.GreetingServlet</servlet-class>
 </servlet>

 <servlet>
  <display-name>ResponseServlet</display-name>
  <servlet-name>ResponseServlet</servlet-name>
  <servlet-class>servlets.ResponseServlet</servlet-class>
 </servlet>

 <servlet-mapping>
  <servlet-name>GreetingServlet</servlet-name>
  <url-pattern>/greeting</url-pattern>
 </servlet-mapping>

 <servlet-mapping>
  <servlet-name>ResponseServlet</servlet-name>
  <url-pattern>/response</url-pattern>
 </servlet-mapping>

</web-app>
```

- Thiết lập sử dụng máy chủ Tomcat để chạy ứng dụng servlet (Run Configurations): Menu Run > Run Configurations, chọn Tomcat server.
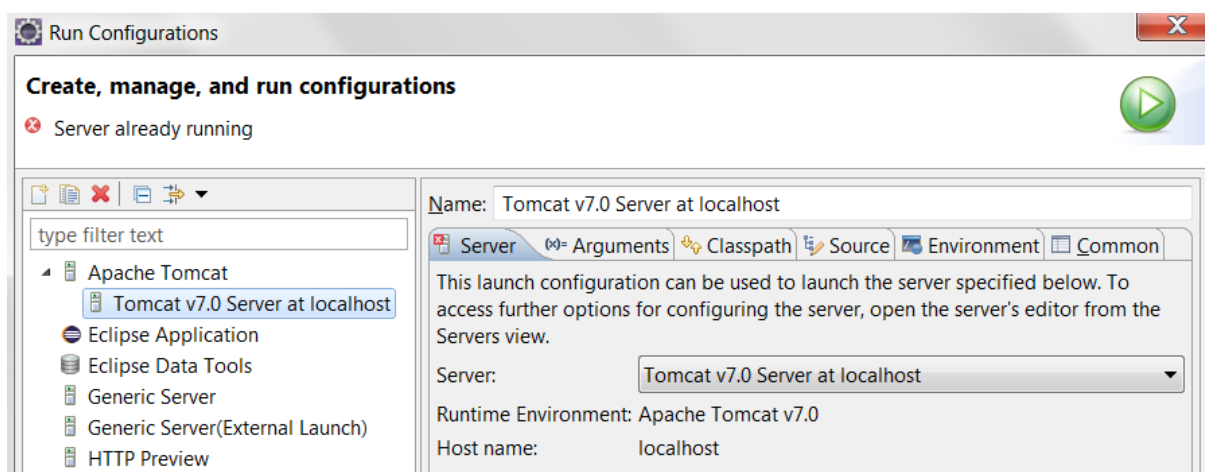
Hình vẽ 7: Kiểm tra cấu hình chạy máy chủ ứng dụng J2EE trong Eclipse

- Chạy ứng dụng my_servlet với Tomcat server (menu Run > Run As > Run on Server). Máy chủ Tomcat sẽ tự động được chạy và thông tin log sẽ hiển thị trong cửa sổ Console.
- Dùng web browser truy nhập đến địa chỉ http://localhost:8080/my_servlet/greeting. Giao diện chương trình web nhận được như bên dưới:

Hình vẽ 8: Giao diện ứng dụng web my_servlet

Bài tập: hãy xử lý code để bắt tình huống người sử dụng click "Submit" mà không nhập tên vào text box. Yêu cầu hiện câu nhắc "Tên đang để trống. Hãy nhập vào tên của bạn!!!" vào ngay bên dưới text box. *Chú ý: lần đầu khi truy nhập đến trang web thì câu nhắc này không được hiển thị.*

2.2.2   Đóng gói my_servlet

Sau khi đã chạy thử thành công project my_servlet, có thể đóng gói project này để cài đặt (deploy) lên các máy chủ ứng dụng. Sử dụng chức năng export project trong Eclipse và tạo ra file WAR. Cấu trúc file WAR được mô tả như trong hình vẽ bên dưới ((J2EE tutorial) trang 81).
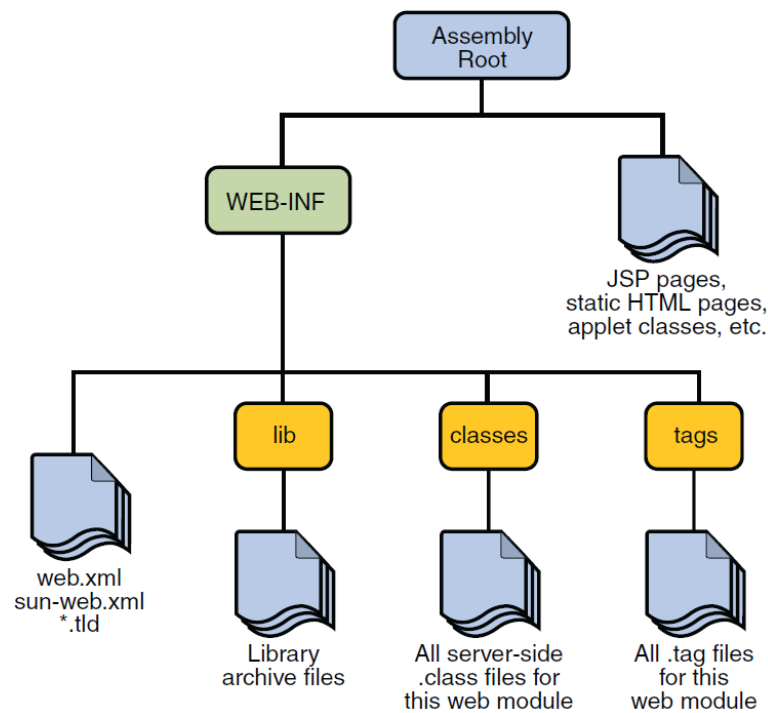
Hình vẽ 9: WebModule Structure

In the Java EE architecture, web components and static web content files such as images are called web resources. A web module is the smallest deployable and usable unit of web resources. A Java EE web module corresponds to a web application as defined in the Java Servlet specification.

In addition to web components and web resources, a web module can contain other files:

- Server-side utility classes (database beans, shopping carts, and so on). Often these classes conform to the JavaBeans component architecture.
- Client-side classes (applets and utility classes).

A web module has a specific structure. The top-level directory of a web module is the document root of the application. The document root is where JSP pages, client-side classes and archives, and static web resources, such as images, are stored.

The document root contains a subdirectory named WEB-INF, which contains the following files and directories:

- web.xml: The web application deployment descriptor
- Tag library descriptor files (see "Tag LibraryDescriptors" on page 245)
- classes: A directory that contains server-side classes: servlets, utility classes, and JavaBeans components
- tags: A directory that contains tag files, which are implementations of tag libraries (see "Tag File Location" on page 233)
- lib: A directory that contains JAR archives of libraries called by server-side classes

If your web module does not contain any servlets, filter, or listener components then it does not need a web application deployment descriptor. In other words, if your web module

only contains JSP pages and static files then you are not required to include a web.xml file. The hello1 example, first discussed in "Packaging WebModules" on page 83, contains only JSP pages and images and therefore does not include a deployment descriptor.

You can also create application-specific subdirectories (that is, package directories) in either the document root or the WEB-INF/classes/ directory. A web module can be deployed as an unpacked file structure or can be packaged in a JAR file known as a web archive (WAR) file. Because the contents and use ofWARfiles differ from those of JAR files,WARfile names use a .war extension. The web module just described is portable; you can deploy it into any web container that conforms to the Java Servlet Specification.

To deploy aWARon the Application Server, the file must also contain a runtime deployment descriptor. The runtime deployment descriptor is an XML file that contains information such as the context root of the web application and the mapping of the portable names of an application's resources to the Application Server's resources. The Application Server web application runtimeDDis named sun-web.xml and is located in the WEB-INF directory along

with the web applicationDD. The structure of a web module that can be deployed on the Application Server is shown in Figure 3–5.

2.2.3   Deploy my_servlet lên Tomcat server

Tomcat mặc định sử dụng thư mục "tomcat7/webapps" để chứa các ứng dụng web. Cách đơn giản nhất để deploy my_servlet lên Tomcat là copy file này vào thư mục wabapp. Tomcat sẽ tự động phát hiện file đóng gói ứng dụng này và deploy. Sau khi Tomcat deploy thành công, dùng web browser truy nhập vào địa chỉ http://localhost:8080/my_servlet/greeting để sử dụng dịch vụ.

Phương pháp copy file WAR vào thư mục webapp ẩn chứa nhiều yếu tố an ninh (vì đòi hỏi có toàn quyền truy nhập đến thư mục webapp của Tomcat). Một phương pháp khác khá phổ biến là sử dụng Context.

<copy from Tomcat help: http://localhost:8080/docs/deployer-howto.html >

In talking about deployment of web applications, the concept of a Context is required to be understood. A Context is what Tomcat calls a web application.

In order to configure a Context within Tomcat a Context Descriptor is required. A Context Descriptor is simply an XML file that contains Tomcat related configuration for a Context, e.g naming resources or session manager configuration. In earlier versions of Tomcat the content of a Context Descriptor configuration was often stored within Tomcat's primary configuration file server.xml but this is now discouraged (although it currently still works).

Context Descriptors not only help Tomcat to know how to configure Contexts but other tools such as the Tomcat Manager and TCD often use these Context Descriptors to perform their roles properly.

The locations for Context Descriptors are:

[1] $CATALINA_BASE/conf/[enginename]/[hostname]/[webappname].xml
[2] $CATALINA_BASE/webapps/[webappname]/META-INF/context.xml

Files in (1) are named [webappname].xml but files in (2) are named context.xml. If a Context Descriptor is not provided for a Context, Tomcat configures the Context using default values.

Sử dụng phương pháp số (1), copy các file class (dịch tử 2 file servlet bên trên) vào thư mục "C:/Working/workspace/java/my_servlet/WebContent\WEB-INF" và tạo file context my_servlet.xml trong thư mục "tomcat7\conf\Catalina\localhost" với nội dung như sau:

```
<Context docBase="C:/Working/workspace/java/my_servlet/WebContent" reloadable="true">
</Context>
```

Khi đó, Tomcat tự đông tạo ra một ứng dụng web có địa chỉ truy nhập trùng với tên của file xml (là my_servlet) và nội dung là các file nằm trong thư mục docBase. Tham số reloadable = "true" dùng để thông báo cho Tomcat biết context này yêu cầu mỗi khi có sự thay đổi trong thư mục WEB-INF (lib, classes, v.v..) thì tự động load lại ứng dụng my_servlet mà không phải khởi động lại Tomcat. Phương pháp này rất có ích khi ứng dụng đang trong quá trình xây dựng hoặc update.

Sau khi Tomcat deploy thành công, dùng web browser truy nhập vào địa chỉ http://localhost:8080/my_servlet/greeting để sử dụng dịch vụ.

2.2.4 Phân tích kết quả chạy chương trình my_servlet

- Mỗi ứng dụng servlet được đặt trong một thư mục hoặc một file WAR, tương ứng với một contex. Tên của context chính là đường dẫn gốc để để truy nhập đến ứng dụng servlet này từ client browser. Trong ví dụ trên, ứng dụng "my_servlet" có thể được truy nhập đến bằng đường dẫn URL http://localhost:8080/my_servlet

- Một servlet có thể được triệu gọi xuất phát từ một request của client browser hoặc cũng có thể từ một servlet khác thông qua RequestDispatcher. Ví dụ servlet ResponseServlet được triệu gọi khi servlet GreetingServlet xử lý request và dispatch đến đường dẫn "/response".

- Mỗi servlet tương ứng với một URL "path". Ví dụ "/greeting" được khai báo ứng với servlet GreetingServlet. Khi client browser truy nhập đến đường dẫn này (sau đường dẫn gốc, ví dụ http://localhost:8080/my_servlet/greeting), servlet tương ứng sẽ được triệu gọi.

- Nội dung trang web trả về cho client browser được tạo ra bằng cách xử lý phương thức *doGet()* của servlet. Về lý thuyết, có thể tạo ra trang web tùy ý bằng cách "in code HTML ra browser" thông qua object *PrintWriter* trong object *response*, tuy nhiên với các trang web phức tạp, chưa các code JavaScript thì không dễ thực hiện.

2.3 Các kỹ thuật xử lý Servlet

2.3.1 Servlet làm việc như thế nào

A servlet life cycle can be defined as the entire process from its creation till the destruction. The following are the paths followed by a servlet

- The servlet is initialized by calling the **init()** method.
- The servlet calls **service()** method to process a client's request.
- The servlet is terminated by calling the **destroy()** method.
- Finally, servlet is garbage collected by the garbage collector of the JVM.

Now let us discuss the life cycle methods in details.

2.3.1.1 The init() method

The init method is designed to be called only once. It is called when the servlet is first created, and not called again for each user request. So, it is used for one-time initializations, just as with the init method of applets.

The servlet is normally created when a user first invokes a URL corresponding to the servlet, but you can also specify that the servlet be loaded when the server is first started.

When a user invokes a servlet, a single instance of each servlet gets created, with each user request resulting in a new thread that is handed off to doGet or doPost as appropriate. The init() method simply creates or loads some data that will be used throughout the life of the servlet.

The init method definition looks like this:

```
public void init() throws ServletException {
  // Initialization code…
}
```

2.3.1.2 The service() method

The service() method is the main method to perform the actual task. The servlet container (i.e. web server) calls the service() method to handle requests coming from the client(browsers) and to write the formatted response back to the client.

Each time the server receives a request for a servlet, the server spawns a new thread and calls service. The service() method checks the HTTP request type (GET, POST, PUT, DELETE, etc.) and calls doGet, doPost, doPut, doDelete, etc. methods as appropriate.

The service() method is called by the container and service method invokes doGet, doPost, doPut, doDelete, etc. methods as appropriate. So you have nothing to do with service() method but you override either doGet() or doPost() depending on what type of request you receive from the client.

The doGet() and doPost() are most frequently used methods with in each service request. Here is the signature of these two methods.

2.3.1.3 The doGet(), doPost() method

A GET, POST request results from a normal request for a URL or from an HTML form that has no METHOD specified and it should be handled by doGet() method.

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
   throws ServletException, IOException {
   // Servlet code
}

public void doPost(HttpServletRequest request, HttpServletResponse response)
   throws ServletException, IOException {
   // Servlet code
}
```

2.3.1.4 The destroy() method

The destroy() method is called only once at the end of the life cycle of a servlet. This method gives your servlet a chance to close database connections, halt background threads, write cookie lists or hit counts to disk, and perform other such cleanup activities.

After the destroy() method is called, the servlet object is marked for garbage collection. The destroy method definition looks like this:

```
public void destroy() {
  // Finalization code...
}
```

2.3.2   Servlet Container

2.3.2.1 Container là gì

(Bryan Basham, 2008) – trang 39

Servlets don't have a main() method. They're under the control of another Java application called a Container. Tomcat is an example of a Container. When your web server application (like Apache) gets a request for a servlet (as opposed to, say, a plain old static HTML page), the server hands the request not to the servlet itself, but to the Container in which the servlet is deployed. It's the Container that gives the servlet the HTTP request and response, and it's the Container that calls the servlet's methods (like doPost() or doGet()).

What does the Container give you?

- **Communications support** The container provides an easy way for your servlets to talk to your web server. You don't have to build a ServerSocket, listen on a port, create streams, etc. The Container knows the protocol between the web server and itself, so that your servlet doesn't have to worry about an API between, say, the Apache web server and your own web application code. All you have to worry about is your own business logic that goes in your Servlet (like accepting an order from your online store).

- **Lifecycle Management** The Container controls the life and death of your servlets. It takes care of loading the classes, instantiating and initializing the servlets, invoking the servlet methods, and making servlet instances eligible for garbage collection. With the Container in control, *you* don't have to worry as much about resource management.
- **Multithreading Support** The Container automatically creates a new Java thread for every servlet request it receives. When the servlet's done running the HTTP service method for that client's request, the thread completes (i.e. dies). This doesn't mean you're off the hook for thread safety—you can still run into synchronization issues. But having the server create and manage threads for multiple requests still saves you a lot of work.
- **Declarative Security** With a Container, you get to use an XML deployment descriptor to configure (and modify) security without having to hard-code it into your servlet (or any other) class code. Think about that! You can manage and change your security without
- **JSP Support** You already know how cool JSPs are. Well, who do you think takes care of translating that JSP code into real Java? Of course. The *Container*.

How the Container handles a request

① User clicks a link that has a URL to a servlet instead of a static page.

② The container "sees" that the request is for a servlet, so the container creates two objects:
1) HttpServletResponse
2) HttpServletRequest

③ The container finds the correct servlet based on the URL in the request, creates or allocates a thread for that request, and passes the request and response objects to the servlet thread.

④ The container calls the servlet's service() method. Depending on the type of request, the service() method calls either the doGet() or doPost() method.

For this example, we'll assume the request was an HTTP GET.

⑤ The doGet() method generates the dynamic page and stuffs the page into the response object. Remember, the container still has a reference to the response object!

⑥ The thread completes, the container converts the response object into an HTTP response, sends it back to the client, then deletes the request and response objects.

2.3.2.2 Kết nối các phương thức Servlet và vai trò của Container

The following figure depicts a typical servlet life-cycle scenario.

- First the HTTP requests coming to the server are delegated to the servlet container.
- The servlet container loads the servlet before invoking the service() method.
- Then the servlet container handles multiple requests by spawning multiple threads, each thread executing the service() method of a single instance of the servlet.
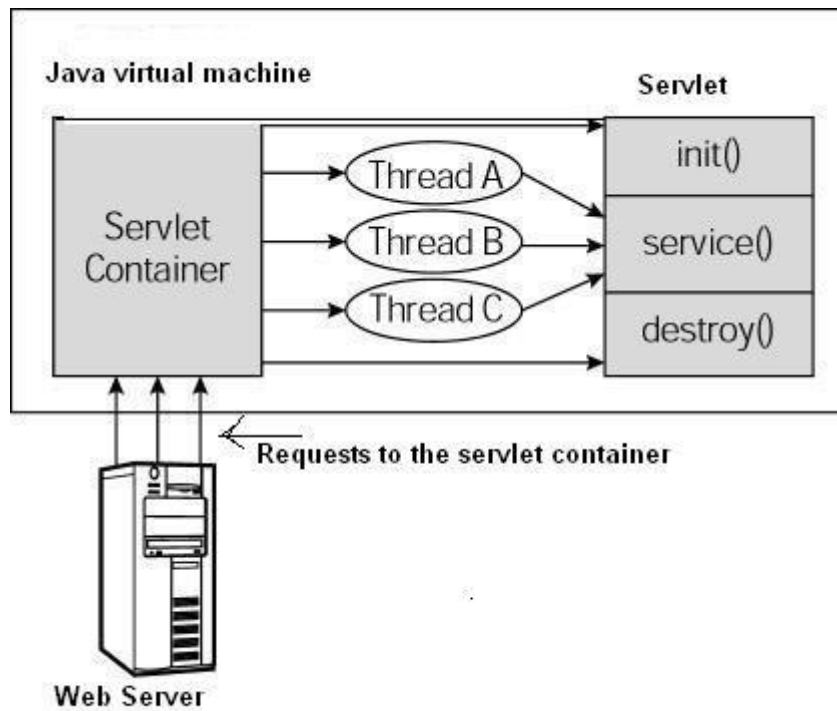
Hình vẽ 10: 3.1.5   Vòng đời Servlet

Thêm các phương thức init(), destroy() hiển thị thông tin log như bên dưới đồng thời cũng bổ sung hiển thị thông tin log trong doGet():

```java
public void init() throws ServletException {
        System.out.println("--> GreetingServlet.init()...");
}

public void destroy() {
        System.out.println("--> GreetingServlet.destroy()...");
}
```

Để ý thất kết quả thông tin log trên Tomcat server khi nhiều lần truy nhập đến /greeting từ một browser hoặc cùng lúc từ nhiều browser như sau:

```
--> GreetingServlet.init()...
--> GreetingServlet.doGet()...
--> GreetingServlet.doGet()...
--> ResponseServlet.init()...
--> ResponseServlet.doGet()...
--> GreetingServlet.doGet()...
--> ResponseServlet.doGet()...
--> GreetingServlet.doGet()...
--> ResponseServlet.doGet()...
--> GreetingServlet.doGet()...
--> GreetingServlet.doGet()...
```

Có nghĩa là dù được truy nhập từ nhiều browser khác nhau, mỗi object servlet được tạo ra khi browser đầu tiên truy nhập đến và object này sẽ tồn tại mãi, kể cả khi một browser

khác truy nhập đến thì object servlet cũ lại được sử dụng. Việc quản lý servlet như vậy dường như đi ngược lại cấu trúc quản lý phiên làm việc của user. Thông thường, các user khác nhau truy nhập đến cùng một ứng dụng web từ các browser khác nhau sẽ phải được quản lý phiên làm việc riêng rẽ. Vấn đề này sẽ được bàn kỹ thêm trong phần sau (quản lý phiên làm việc).

2.3.3 Trao đổi dữ liệu với client

2.3.3.1 Form Data

Trong ví dụ GreetingServlet, dữ liệu nhập vào ở text box được gửi từ client lên server và được lấy ra trong phương thức **doGet()**. Các dữ liệu client nhập vào thông qua các input box dạng khác như check box, radio button, v.v.. cũng có thể được lấy ra tương tự như bằng phương thưc **request.getParameter()** trong phương thức **doGet()**.

Ví dụ sau đây kết hợp giữa form data trong một file HTML và gọi đến servlet để xử lý dữ liệu client gửi lên. Tạo file CheckBox.html trong thư mục WebContent với nội dung sau:

```html
<html>
<body>
<form action="CheckBox" method="POST" target="_blank">
<input type="checkbox" name="maths" checked="checked" /> Maths
<input type="checkbox" name="physics"  /> Physics
<input type="checkbox" name="chemistry" checked="checked" />
                        Chemistry
<input type="submit" value="Select Subject" />
</form>
</body>
</html>
```

Tạo servlet CheckBoxServlet với phương thức doGet() như sau:

```java
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class CheckBox extends HttpServlet {

        public void doGet(HttpServletRequest request, HttpServletResponse response)
                                        throws ServletException, IOException {
                response.setContentType("text/html");

                PrintWriter out = response.getWriter();
                String title = "Reading Checkbox Data";
                String docType = "<!doctype html public \"-//w3c//dtd html 4.0 " +
                                        "transitional//en\">\n";
                out.println(docType +
                                "<html>\n" +"<head><title>" + title + "</title></head>\n" +
                                "<body bgcolor=\"#f0f0f0\">\n" +
                                "<h1 align=\"center\">" + title + "</h1>\n" +"<ul>\n" +
                                " <li><b>Maths Flag : </b>: "
                                + request.getParameter("maths") + "\n" +
                                " <li><b>Physics Flag: </b>: "
```

```
                              + request.getParameter("physics") + "\n" +
                           "  <li><b>Chemistry Flag: </b>: "
                              + request.getParameter("chemistry") + "\n" +
                           "</ul>\n" +
                           "</body></html>");
          }
}
```

Cuối cùng là khai báo thêm servlet này trong file web.xml:

```xml
<servlet>
  <display-name>CheckBox</display-name>
  <servlet-name>CheckBox</servlet-name>
  <servlet-class>CheckBox</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>CheckBox</servlet-name>
  <url-pattern>/CheckBox</url-pattern>
</servlet-mapping>
```

Chạy server truy nhập địa chỉ http://localhost:8080/my_servlet/CheckBox.html để kiểm tra chương trình.

2.3.3.2 Đọc thông tin trong client request

When a browser requests for a web page, it sends lot of information to the web server which can not be read directly because this information travel as a part of header of HTTP request. You can check HTTP Protocol for more information on this.

Following is the important header information which comes from browser side and you would use very frequently in web programming:

Header	Description
Accept	This header specifies the MIME types that the browser or other clients can handle. Values of image/png or image/jpeg are the two most common possibilities.
Accept-Charset	This header specifies the character sets the browser can use to display the information. For example ISO-8859-1.
Accept-Encoding	This header specifies the types of encodings that the browser knows how to handle. Values of gzip or compress are the two most common possibilities.
Accept-Language	This header specifies the client's preferred languages in case the servlet can produce results in more than one language. For example en, en-us, ru, etc.
Authorization	This header is used by clients to identify themselves when accessing password-protected Web pages.
Connection	This header indicates whether the client can handle persistent HTTP connections. Persistent connections permit the client or

	other browser to retrieve multiple files with a single request. A value of Keep-Alive means that persistent connections should be used
Content-Length	This header is applicable only to POST requests and gives the size of the POST data in bytes.
Cookie	This header returns cookies to servers that previously sent them to the browser.
Host	This header specifies the host and port as given in the original URL.
If-Modified-Since	This header indicates that the client wants the page only if it has been changed after the specified date. The server sends a code, 304 which means Not Modified header if no newer result is available.
If-Unmodified-Since	This header is the reverse of If-Modified-Since; it specifies that the operation should succeed only if the document is older than the specified date.
Referer	This header indicates the URL of the referring Web page. For example, if you are at Web page 1 and click on a link to Web page 2, the URL of Web page 1 is included in the Referer header when the browser requests Web page 2.
User-Agent	This header identifies the browser or other client making the request and can be used to return different content to different types of browsers.

Để đọc các thông tin này, HttpServletRequest cung cấp một số phương thức:

S.N.	Method & Description
1	Cookie[] getCookies() Returns an array containing all of the Cookie objects the client sent with this request.
2	Enumeration getAttributeNames() Returns an Enumeration containing the names of the attributes available to this request.
3	Enumeration getHeaderNames() Returns an enumeration of all the header names this request contains.
4	Enumeration getParameterNames() Returns an Enumeration of String objects containing the names of the parameters contained in this request.
5	HttpSession getSession() Returns the current session associated with this request, or if the request does not have a session, creates one.
6	HttpSession getSession(boolean create) Returns the current HttpSession associated with this request or, if if there is no current

	session and create is true, returns a new session.
7	Locale getLocale() Returns the preferred Locale that the client will accept content in, based on the Accept-Language header
8	Object getAttribute(String name) Returns the value of the named attribute as an Object, or null if no attribute of the given name exists.
9	ServletInputStream getInputStream() Retrieves the body of the request as binary data using a ServletInputStream.
10	String getAuthType() Returns the name of the authentication scheme used to protect the servlet, for example, "BASIC" or "SSL," or null if the JSP was not protected
11	String getCharacterEncoding() Returns the name of the character encoding used in the body of this request.
12	String getContentType() Returns the MIME type of the body of the request, or null if the type is not known.
13	String getContextPath() Returns the portion of the request URI that indicates the context of the request.
14	String getHeader(String name) Returns the value of the specified request header as a String.
15	String getMethod() Returns the name of the HTTP method with which this request was made, for example, GET, POST, or PUT.
16	String getParameter(String name) Returns the value of a request parameter as a String, or null if the parameter does not exist.
17	String getPathInfo() Returns any extra path information associated with the URL the client sent when it made this request.
18	String getProtocol() Returns the name and version of the protocol the request.
19	String getQueryString() Returns the query string that is contained in the request URL after the path.
20	String getRemoteAddr() Returns the Internet Protocol (IP) address of the client that sent the request.
21	String getRemoteHost() Returns the fully qualified name of the client that sent the request.
22	String getRemoteUser() Returns the login of the user making this request, if the user has been authenticated, or null if the user has not been authenticated.

23	String getRequestURI() Returns the part of this request's URL from the protocol name up to the query string in the first line of the HTTP request.
24	String getRequestedSessionId() Returns the session ID specified by the client.
25	String getServletPath() Returns the part of this request's URL that calls the JSP.
26	String[] getParameterValues(String name) Returns an array of String objects containing all of the values the given request parameter has, or null if the parameter does not exist.
27	boolean isSecure() Returns a boolean indicating whether this request was made using a secure channel, such as HTTPS.
28	int getContentLength() Returns the length, in bytes, of the request body and made available by the input stream, or -1 if the length is not known.
29	int getIntHeader(String name) Returns the value of the specified request header as an int.
30	int getServerPort() Returns the port number on which this request was received.

Đoạn chương trình sau đọc và hiển thị trên trang web tất cả các thông tin trong client request:

```
Enumeration headerNames = request.getHeaderNames();
out.print("<table width=\"100%\" border=\"1\" align=\"center\">\n" +
    "<tr bgcolor=\"#949494\">\n" +
    "<th>Header Name</th><th>Header Value(s)</th>\n"+
    "</tr>\n");

while(headerNames.hasMoreElements()) {
    String paramName = (String)headerNames.nextElement();
    out.print("<tr><td>" + paramName + "</td>\n");
    String paramValue = request.getHeader(paramName);
    out.println("<td> " + paramValue + "</td></tr>\n");
}
out.println("</table>\n");
```

2.3.3.3 Trả thông tin từ server về client

Ưhen a Web server responds to a HTTP request to the browser, the response typically consists of a status line, some response headers, a blank line, and the document. A typical response looks like this:

```
HTTP/1.1 200 OK
```

```
Content-Type: text/html
Header2: ...
...
HeaderN: ...
  (Blank Line)
<!doctype ...>
<html>
<head>...</head>
<body>
...
</body>
</html>
```

The status line consists of the HTTP version (HTTP/1.1 in the example), a status code (200 in the example), and a very short message corresponding to the status code (OK in the example).

Following is a summary of the most useful HTTP 1.1 response headers which go back to the browser from web server side and you would use them very frequently in web programming:

Header	Description
Allow	This header specifies the request methods (GET, POST, etc.) that the server supports.
Cache-Control	This header specifies the circumstances in which the response document can safely be cached. It can have values public, private or no-cache etc. Public means document is cacheable, Private means document is for a single user and can only be stored in private (nonshared) caches and no-cache means document should never be cached.
Connection	This header instructs the browser whether to use persistent in HTTP connections or not. A value of close instructs the browser not to use persistent HTTP connections and keep-alive means using persistent connections.
Content-Disposition	This header lets you request that the browser ask the user to save the response to disk in a file of the given name.
Content-Encoding	This header specifies the way in which the page was encoded during transmission.
Content-Language	This header signifies the language in which the document is written. For example en, en-us, ru, etc.
Content-Length	This header indicates the number of bytes in the response. This information is needed only if the browser is using a persistent (keep-alive) HTTP connection.
Content-Type	This header gives the MIME (Multipurpose Internet Mail Extension) type of the response document.

Expires	This header specifies the time at which the content should be considered out-of-date and thus no longer be cached.
Last-Modified	This header indicates when the document was last changed. The client can then cache the document and supply a date by an If-Modified-Since request header in later requests.
Location	This header should be included with all responses that have a status code in the 300s. This notifies the browser of the document address. The browser automatically reconnects to this location and retrieves the new document.
Refresh	This header specifies how soon the browser should ask for an updated page. You can specify time in number of seconds after which a page would be refreshed.
Retry-After	This header can be used in conjunction with a 503 (Service Unavailable) response to tell the client how soon it can repeat its request.
Set-Cookie	This header specifies a cookie associated with the page.

Để đọc các thông tin này, HttpServletResponse cung cấp một số phương thức:

S.N.	Method & Description
1	String encodeRedirectURL(String url) Encodes the specified URL for use in the sendRedirect method or, if encoding is not needed, returns the URL unchanged.
2	String encodeURL(String url) Encodes the specified URL by including the session ID in it, or, if encoding is not needed, returns the URL unchanged.
3	boolean containsHeader(String name) Returns a boolean indicating whether the named response header has already been set.
4	boolean isCommitted() Returns a boolean indicating if the response has been committed.
5	void addCookie(Cookie cookie) Adds the specified cookie to the response.
6	void addDateHeader(String name, long date) Adds a response header with the given name and date-value.
7	void addHeader(String name, String value) Adds a response header with the given name and value.
8	void addIntHeader(String name, int value) Adds a response header with the given name and integer value.
9	void flushBuffer()

	Forces any content in the buffer to be written to the client.
10	void reset() Clears any data that exists in the buffer as well as the status code and headers.
11	void resetBuffer() Clears the content of the underlying buffer in the response without clearing headers or status code.
12	void sendError(int sc) Sends an error response to the client using the specified status code and clearing the buffer.
13	void sendError(int sc, String msg) Sends an error response to the client using the specified status.
14	void sendRedirect(String location) Sends a temporary redirect response to the client using the specified redirect location URL.
15	void setBufferSize(int size) Sets the preferred buffer size for the body of the response.
16	void setCharacterEncoding(String charset) Sets the character encoding (MIME charset) of the response being sent to the client, for example, to UTF-8.
17	void setContentLength(int len) Sets the length of the content body in the response In HTTP servlets, this method sets the HTTP Content-Length header.
18	void setContentType(String type) Sets the content type of the response being sent to the client, if the response has not been committed yet.
19	void setDateHeader(String name, long date) Sets a response header with the given name and date-value.
20	void setHeader(String name, String value) Sets a response header with the given name and value.
21	void setIntHeader(String name, int value) Sets a response header with the given name and integer value.
22	void setLocale(Locale loc) Sets the locale of the response, if the response has not been committed yet.
23	void setStatus(int sc) Sets the status code for this response.

Ví dụ sau đây thiết lập trang web mà phía client tự động refresh lại sau mỗi 5 giây:

```java
public class Refresh extends HttpServlet {

  public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
```

```
{
    // Set refresh, autoload time as 5 seconds
    response.setIntHeader("Refresh", 5);

    // Set response content type
    response.setContentType("text/html");

    // Get current time
    Calendar calendar = new GregorianCalendar();
    String am_pm;
    int hour = calendar.get(Calendar.HOUR);
    int minute = calendar.get(Calendar.MINUTE);
    int second = calendar.get(Calendar.SECOND);
    if(calendar.get(Calendar.AM_PM) == 0)
        am_pm = "AM";
    else
        am_pm = "PM";

    String CT = hour+":"+ minute +":"+ second +" "+ am_pm;

    PrintWriter out = response.getWriter();
    String title = "Auto Refresh Header Setting";
    String docType =
    "<!doctype html public \"-//w3c//dtd html 4.0 " +
    "transitional//en\">\n";
    out.println(docType +
     "<html>\n" +
     "<head><title>" + title + "</title></head>\n"+
     "<body bgcolor=\"#f0f0f0\">\n" +
     "<h1 align=\"center\">" + title + "</h1>\n" +
     "<p>Current Time is: " + CT + "</p>\n");
}
}
```

2.3.3.4 Xử lý Cookie

Cookies are text files stored on the client computer and they are kept for various information tracking purpose. Java Servlets transparently supports HTTP cookies. There are three steps involved in identifying returning users:

- Server script sends a set of cookies to the browser. For example name, age, or identification number etc.
- Browser stores this information on local machine for future use.
- When next time browser sends any request to web server then it sends those cookies information to the server and server uses that information to identify the user.

This chapter will teach you how to set or reset cookies, how to access them and how to delete them.

Tìm hiểu về Cookie:

Cookies are usually set in an HTTP header (although JavaScript can also set a cookie directly on a browser). A servlet that sets a cookie might send headers that look something like this:

```
HTTP/1.1 200 OK
Date: Fri, 04 Feb 2000 21:03:38 GMT
Server: Apache/1.3.9 (UNIX) PHP/4.0b3
Set-Cookie: name=xyz; expires=Friday, 04-Feb-07 22:03:38 GMT;
         path=/; domain=tutorialspoint.com
Connection: close
Content-Type: text/html
```

As you can see, the Set-Cookie header contains a name value pair, a GMT date, a path and a domain. The name and value will be URL encoded. The expires field is an instruction to the browser to "forget" the cookie after the given time and date.

If the browser is configured to store cookies, it will then keep this information until the expiry date. If the user points the browser at any page that matches the path and domain of the cookie, it will resend the cookie to the server. The browser's headers might look something like this:

```
GET / HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/4.6 (X11; I; Linux 2.2.6-15apmac ppc)
Host: zink.demon.co.uk:1126
Accept: image/gif, */*
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
Cookie: name=xyz
```

A servlet will then have access to the cookie through the request method request.getCookies() which returns an array of *Cookie* objects.

Thiết lập Cookies với Servlet:

Setting cookies with servlet involves three steps:

(1) Creating a Cookie object: You call the Cookie constructor with a cookie name and a cookie value, both of which are strings.

```
Cookie cookie = new Cookie("key","value");
```

Keep in mind, neither the name nor the value should contain white space or any of the following characters:

```
[ ] ( ) = , " / ? @ : ;
```

(2) Setting the maximum age: You use setMaxAge to specify how long (in seconds) the cookie should be valid. Following would set up a cookie for 24 hours.

```
cookie.setMaxAge(60*60*24);
```

(3) Sending the Cookie into the HTTP response headers: You use **response.addCookie** to add cookies in the HTTP response header as follows:

```
response.addCookie(cookie);
```

Ví dụ, tạo servlet CookieServlet như bên dưới:

```java
package servlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class CookieServlet extends HttpServlet {

  public void doGet(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException  {
    // Create cookies for first and last names.
    Cookie firstName = new Cookie("first_name", request.getParameter("first_name"));
    Cookie lastName = new Cookie("last_name", request.getParameter("last_name"));

    // Set expiry date after 24 Hrs for both the cookies.
    firstName.setMaxAge(60*60*24);
    lastName.setMaxAge(60*60*24);

    // Add both the cookies in the response header.
    response.addCookie( firstName );
    response.addCookie( lastName );

    // Set response content type
    response.setContentType("text/html");

    PrintWriter out = response.getWriter();
    String title = "Setting Cookies Example";
    String docType = "<!doctype html public \"-//w3c//dtd html 4.0 " + "transitional//en\">\n";
    out.println(docType + "<html>\n" + "<head><title>" + title + "</title></head>\n" +
        "<body bgcolor=\"#f0f0f0\">\n" + "<h1 align=\"center\">" + title + "</h1>\n" + "<ul>\n" +
        "  <li><b>First Name</b>: " + request.getParameter("first_name") + "\n" +
        "  <li><b>Last Name</b>: " + request.getParameter("last_name") + "\n" +"</ul>\n" +
        "</body></html>");
  }
}
```

Thiết lập các thông số servlet mapping trong web.xml:

```xml
<servlet>
 <display-name>CookieServlet</display-name>
 <servlet-name>CookieServlet</servlet-name>
 <servlet-class>servlets.CookieServlet</servlet-class>
</servlet>
<servlet-mapping>
 <servlet-name>CookieServlet</servlet-name>
 <url-pattern>/cookie</url-pattern>
</servlet-mapping>
```

Tạo file cookie.html có nội dung như bên dưới và để vào thư mục WebContent.

```html
<html>
<body>
<form action="cookie" method="GET">
First Name: <input type="text" name="first_name">
<br />
Last Name: <input type="text" name="last_name" />
```

```
<input type="submit" value="Submit" />
</form>
</body>
</html>
```

Dùng browser truy nhập tới địa chỉ http://localhost:8080/my_servlet/cookie.html, nội dung trang web hiển thị trên browser có dạng như bên dưới:

Đến đây ta sẽ dùng Firefox developer tools để khám phá chương trình ví dụ cookie này. Chọn hiển thị Developer tools trong Firefox, và chọn view Network như bên dưới để xem hiển thị các thông tin trong phần header của HTTP request và response

Nhập thông tin vào các text box First Name, Last Name và submit, dùng Network view để phân tích các thông tin trong HTTP request & response header:

- Ban đầu, browser Firefox chưa được thiết lập cookie với web server locahost. HTTP request header chưa có thông tin cookie.
- Sau khi submit thông tin lên web server với tham số First Name và Last Name, 2 cookie cùng tên được tạo ra trên server và được gửi về browser trong HTTP response header.
- Hai cookie này được lưu lại trong browser (trong Firefox, chọn Vew Option > Privacy xem phần Individual Cookies và tìm đến server local để xem thông tin 2 cookie First Name và Last Name được lưu trong browser).
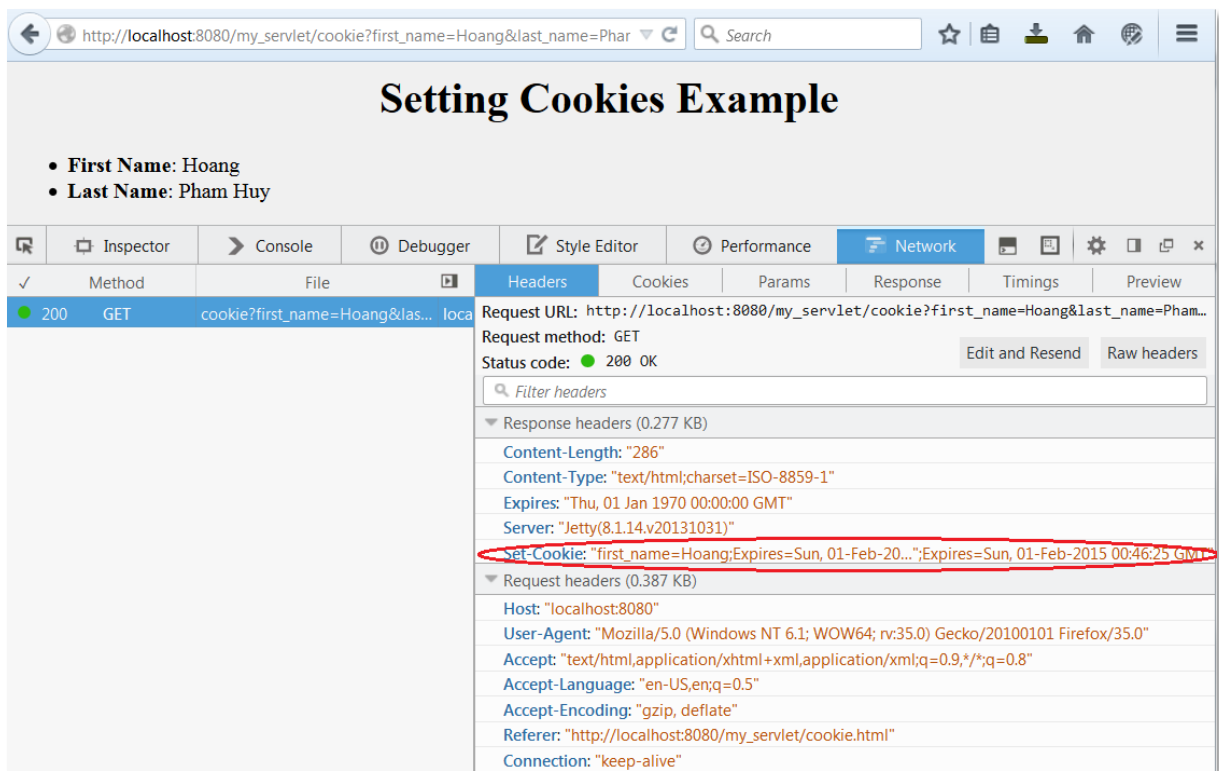
Hình vẽ 13: Cookie được tạo ra trên server và gửi về client trong HTTP response header

Dùng browser kết nối lại đến bất kỳ trang web nào thuộc đường dẫn /my_servlet (ví dụ http://localhost:8080/my_servlet/greeting), phân tích thông tin HTTP request header sẽ thấy thông tin 2 cookie First Name và Last Name được tự động gửi từ browser lên server. Các thông tin cookie này sẽ được tự động hết hạn sau 1 ngày (theo thiết lập MaxAge khi tạo cookie) hoặc được thay đổi khi vào lại trang http://localhost:8080/my_servlet/cookie.html.
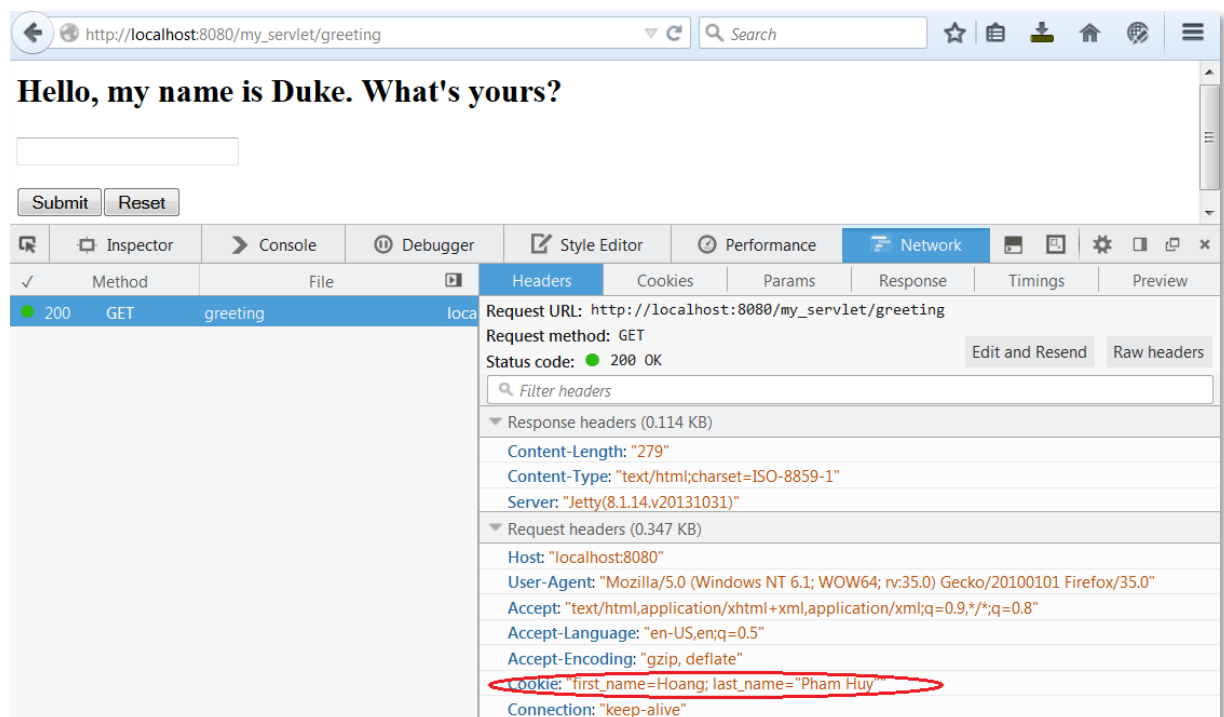
Đọc Cookies với Servlet:

Sau khi đã thiết lập được cookie phía browser của client, việc cuối cùng là đọc thông tin cookie trong mỗi HTTP Request mà client gửi lên server để xử lý.

To read cookies, you need to create an array of *javax.servlet.http.Cookie* objects by calling the **getCookies()** method of *HttpServletRequest*. Then cycle through the array, and use getName() and getValue() methods to access each cookie and associated value.

Example:

```java
package servlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ReadCookies extends HttpServlet {

 public void doGet(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
   Cookie cookie = null;
   Cookie[] cookies = null;
   // Get an array of Cookies associated with this domain
   cookies = request.getCookies();

  // Set response content type
   response.setContentType("text/html");

   PrintWriter out = response.getWriter();
   String title = "Reading Cookies Example";
   String docType = "<!doctype html public \"-//w3c//dtd html 4.0 " + "transitional//en\">\n";
   out.println(docType + "<html>\n" + "<head><title>" + title + "</title></head>\n" +
        "<body bgcolor=\"#f0f0f0\">\n" );
   if( cookies != null ){
     out.println("<h2> Found Cookies Name and Value</h2>");
     for (int i = 0; i < cookies.length; i++){
       cookie = cookies[i];
       out.print("Name : " + cookie.getName( ) + ",  ");
       out.print("Value: " + cookie.getValue( )+" <br/>");
     }
   }else{
     out.println(
     "<h2>No cookies founds</h2>");
   }
   out.println("</body>");
   out.println("</html>");
 }
}
```

Delete Cookies with Servlet:

To delete cookies is very simple. If you want to delete a cookie then you simply need to follow up following three steps:

- Read an already exsiting cookie and store it in Cookie object.

- Set cookie age as zero using **setMaxAge()** method to delete an existing cookie.
- Add this cookie back into response header.

Example:

```java
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class DeleteCookies extends HttpServlet {

 public void doGet(HttpServletRequest request,  HttpServletResponse response)
        throws ServletException, IOException {
    Cookie cookie = null;
    Cookie[] cookies = null;
    // Get an array of Cookies associated with this domain
    cookies = request.getCookies();

   // Set response content type
    response.setContentType("text/html");

    PrintWriter out = response.getWriter();
    String title = "Delete Cookies Example";
    String docType = "<!doctype html public \"-//w3c//dtd html 4.0 " + "transitional//en\">\n";
    out.println(docType + "<html>\n" + "<head><title>" + title + "</title></head>\n" +
         "<body bgcolor=\"#f0f0f0\">\n" );
    if( cookies != null ){
      out.println("<h2> Cookies Name and Value</h2>");
      for (int i = 0; i < cookies.length; i++){
        cookie = cookies[i];
        if((cookie.getName( )).compareTo("first_name") == 0 ){
           cookie.setMaxAge(0);
           response.addCookie(cookie);
           out.print("Deleted cookie : " +
                   cookie.getName( ) + "<br/>");
        }
        out.print("Name : " + cookie.getName( ) + ",  ");
        out.print("Value: " + cookie.getValue( )+" <br/>");
      }
    }else{
      out.println(
        "<h2>No cookies founds</h2>");
    }
    out.println("</body>");
    out.println("</html>");
  }
}
```

You can also delete your cookies in web browser manually.

Các phương thức xử lý Cookies của Servlet:

Following is the list of useful methods which you can use while manipulating cookies in servlet.

S.N. Method & Description

1	public void setDomain(String pattern) This method sets the domain to which cookie applies, for example tutorialspoint.com.
2	public String getDomain() This method gets the domain to which cookie applies, for example tutorialspoint.com.
3	public void setMaxAge(int expiry) This method sets how much time (in seconds) should elapse before the cookie expires. If you don't set this, the cookie will last only for the current session.
4	public int getMaxAge() This method returns the maximum age of the cookie, specified in seconds, By default, -1 indicating the cookie will persist until browser shutdown.
5	public String getName() This method returns the name of the cookie. The name cannot be changed after creation.
6	public void setValue(String newValue) This method sets the value associated with the cookie.
7	public String getValue() This method gets the value associated with the cookie.
8	public void setPath(String uri) This method sets the path to which this cookie applies. If you don't specify a path, the cookie is returned for all URLs in the same directory as the current page as well as all subdirectories.
9	public String getPath() This method gets the path to which this cookie applies.
10	public void setSecure(boolean flag) This method sets the boolean value indicating whether the cookie should only be sent over encrypted (i.e. SSL) connections.
11	public void setComment(String purpose) This method specifies a comment that describes a cookie's purpose. The comment is useful if the browser presents the cookie to the user.
12	public String getComment() This method returns the comment describing the purpose of this cookie, or null if the cookie has no comment.

2.3.4 *Quản lý phiên làm việc (session)*

HTTP is a "stateless" protocol which means each time a client retrieves a Web page, the client opens a separate connection to the Web server and the server automatically does not keep any record of previous client request. Still there are following three ways to maintain session between web client and web server:

Cookies:

A webserver can assign a unique session ID as a cookie to each web client and for subsequent requests from the client they can be recognized using the recieved cookie. This may not be an effective way because many time browser does not support a cookie, so I would not recommend to use this procedure to maintain the sessions.

Hidden Form Fields:

A web server can send a hidden HTML form field along with a unique session ID as follows:

```
<input type="hidden" name="sessionid" value="12345">
```

This entry means that, when the form is submitted, the specified name and value are automatically included in the GET or POST data. Each time when web browser sends request back, then session_id value can be used to keep the track of different web browsers.

This could be an effective way of keeping track of the session but clicking on a regular (<A HREF...>) hypertext link does not result in a form submission, so hidden form fields also cannot support general session tracking.

URL Rewriting:

You can append some extra data on the end of each URL that identifies the session, and the server can associate that session identifier with data it has stored about that session. For example, with http://tutorialspoint.com/file.htm;sessionid=12345, the session identifier is attached as sessionid=12345 which can be accessed at the web server to identify the client.

URL rewriting is a better way to maintain sessions and works for the browsers when they don't support cookies but here drawback is that you would have generate every URL dynamically to assign a session ID though page is simple static HTML page.

The HttpSession Object:

Apart from the above mentioned three ways, servlet provides HttpSession Interface which provides a way to identify a user across more than one page request or visit to a Web site and to store information about that user.

The servlet container uses this interface to create a session between an HTTP client and an HTTP server. The session persists for a specified time period, across more than one connection or page request from the user. You would get HttpSession object by calling the public method **getSession()** of HttpServletRequest, as below:

```
HttpSession session = request.getSession();
```

You need to call *request.getSession()* before you send any document content to the client. Here is a summary of the important methods available through HttpSession object:

S.N.	Method & Description
1	public Object getAttribute(String name)

	This method returns the object bound with the specified name in this session, or null if no object is bound under the name.
2	public Enumeration getAttributeNames() This method returns an Enumeration of String objects containing the names of all the objects bound to this session.
3	public long getCreationTime() This method returns the time when this session was created, measured in milliseconds since midnight January 1, 1970 GMT.
4	public String getId() This method returns a string containing the unique identifier assigned to this session.
5	public long getLastAccessedTime() This method returns the last time the client sent a request associated with this session, as the number of milliseconds since midnight January 1, 1970 GMT.
6	public int getMaxInactiveInterval() This method returns the maximum time interval, in seconds, that the servlet container will keep this session open between client accesses.
7	public void invalidate() This method invalidates this session and unbinds any objects bound to it.
8	public boolean isNew(This method returns true if the client does not yet know about the session or if the client chooses not to join the session.
9	public void removeAttribute(String name) This method removes the object bound with the specified name from this session.
10	public void setAttribute(String name, Object value) This method binds an object to this session, using the name specified.
11	public void setMaxInactiveInterval(int interval) This method specifies the time, in seconds, between client requests before the servlet container will invalidate this session.

Session Tracking Example:

Ví dụ sau đây cho phép người sử dụng nhập vào tên và phía server sẽ lưu dữ liệu này vào session data. Khi user lần thứ 2 truy nhập vào ứng dụng, tên của user sẽ được lấy ra từ dữ liệu của session và hiển thị lại. Ứng dụng cũng cho phép user click button để reset toàn bộ session đã được tạo:

```java
package servlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

public class SessionTrack extends HttpServlet {
```

```java
public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

    HttpSession session = request.getSession(true);
    if (request.getParameter("reset") != null) {
            session.invalidate();
            session = request.getSession(true);
    }
// Get session creation time.
Date createTime = new Date(session.getCreationTime());
// Get last access time of this web page.
Date lastAccessTime =  new Date(session.getLastAccessedTime());

String uName = request.getParameter("username");
String userName = "";
String userNameKey = "userNameKey";

    if ((uName != null) && (uName.length() > 0)) {
            System.out.println("Putting user name: " + uName + " to session data...");
    session.setAttribute(userNameKey, uName);
}


// Check if this is new comer on your web page.
String title = "Welcome Back to my website";
if (session.isNew()){
    title = "Welcome to my website";
} else {
    userName = (String)session.getAttribute(userNameKey);
    System.out.println("Got user name: " + userName + " from session data...");
    title = "<b>Welcome Back to my website</b><br>" +
                    "Session create time: " + createTime.toString() + "<br>" +
                    "Session last access: " + lastAccessTime.toString() + "<br>" +
                    "Stored user in session data: " + userName + "<p>-----<br>" +
                    "Please enter new user:";
}

response.setContentType("text/html");
response.setBufferSize(8192);

PrintWriter out = response.getWriter();

// then write the data of the response
out.println("<html>" + "<head><title>Hello</title></head>");

// then write the data of the response
out.println(
    "<body  bgcolor=\"#ffffff\">" + title
    + "<form method=\"get\" >"
    //+ "<form method=\"get\" action=\"/hello_servlet/response\">"
    + "<input type=\"text\" name=\"username\" size=\"25\" value=\"" + userName +
"\">"

    + "<p></p>" + "<input type=\"submit\" value=\"Submit\">"
    + "</form>");

out.println(
    "<form method=\"get\" >"
            + "<input type=\"hidden\" + name=\"reset\" + value=\"true\" >"
    //+ "<form method=\"get\" action=\"/hello_servlet/response\">"
```
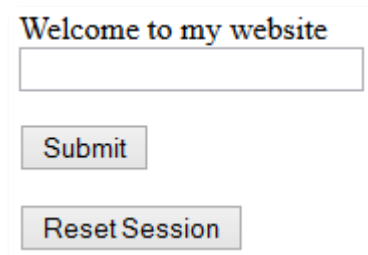
```
                    + "<input type=\"submit\" value=\"Reset Session\">" + "</form>");

            out.println("</body></html>");
            out.close();
        }
}
```

Code 1: Quản lý session với Servlet

Dịch và chạy ứng dụng. Lần đầu truy nhập đến ứng dụng, trang web sẽ hiện ra như sau:
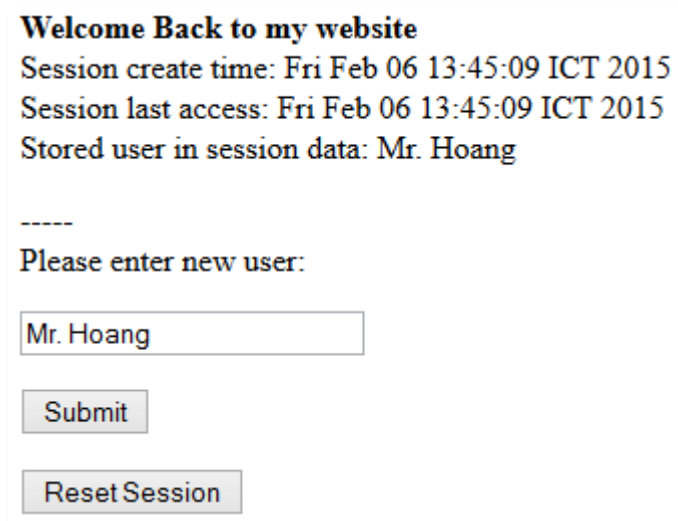
Nhập vào tên user và submit. Từ các lần truy nhập vào ứng dụng sau đó, trang web sẽ hiện ra các thông tin về session cùng với giá trị user name đã nhập vào từ trước:

User có thể click Reser Session để xóa toàn bộ dữ liệu thuộc session hiện tại.

Deleting Session Data:

When you are done with a user's session data, you have several options:

- **Remove a particular attribute:** You can call *public void removeAttribute(String name)* method to delete the value associated with a particular key.
- **Delete the whole session:** You can call *public void invalidate()* method to discard an entire session.
- **Setting Session timeout:** You can call *public void setMaxInactiveInterval(int interval)* method to set the timeout for a session individually.

- **Log the user out:** The servers that support servlets 2.4, you can call **logout** to log the client out of the Web server and invalidate all sessions belonging to all the users.
- **web.xml Configuration:** If you are using Tomcat, apart from the above mentioned methods, you can configure session time out in web.xml file as follows.

```
<session-config>
  <session-timeout>15</session-timeout>
</session-config>
```

The timeout is expressed as minutes, and overrides the default timeout which is 30 minutes in Tomcat. The getMaxInactiveInterval() method in a servlet returns the timeout period for that session in seconds. So if your session is configured in web.xml for 15 minutes, getMaxInactiveInterval() returns 900.

2.3.5 Điều khiển luồng xử lý request

2.3.5.1 Sử dụng RequestDispatcher

Một ứng dụng web thường gồm nhiều trang web tương ứng với nhiều servlet. Thiết kế đơn giản là mỗi request gửi từ client browser sẽ được xử lý bằng một servlet trên server và kết quả trả về client browser là một trang HTML. Tuy nhiên, thiết kế ứng dụng web có thể tích hợp nhiều servlet để xử lý một request gửi lên từ client browser. Trong ví dụ ứng dụng my_servlet, request từ client browser được xử lý đầu tiên tại servlet GreetingServlet và sau đó được xử lý tiếp tại ResponseServlet.
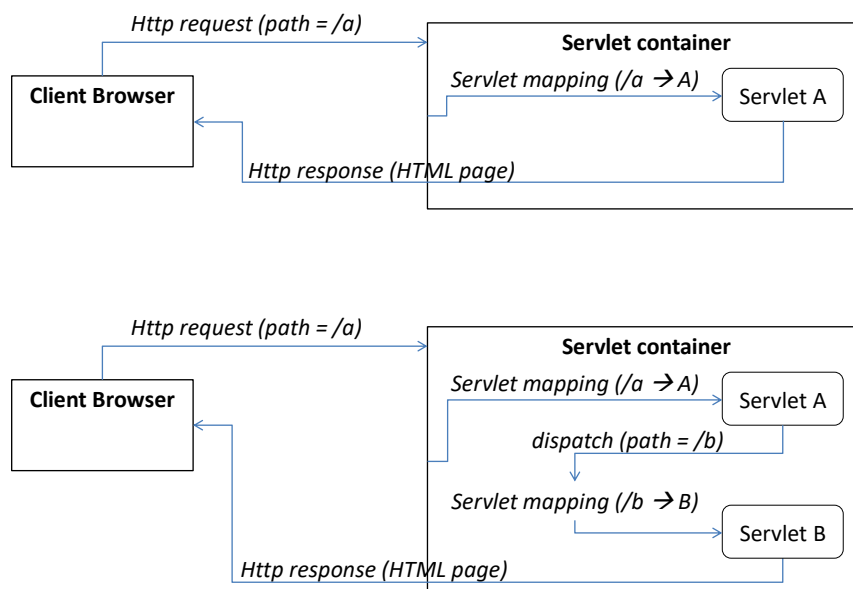
Hình vẽ 14: Xử lý luồng Http request với một servlet và nhiều servlet bằng RequestDispatcher

Ngoài cách xử lý luồng Http request bằng cách sử dụng đối tượng RequestDispatcher, Java servlet cho phép tham gia điều khiển các luồng này ở mức hệ thống (mà không cần xử lý ở từng servlet với dĩ cần tập trung cho việc xử lý nghiệp vụ của ứng dụng).

2.3.5.2 Định hướng luồng tự động khi có lỗi

When a servlet throws an exception, the web container searches the configurations in web.xml that use the exception-type element for a match with the thrown exception type. You would have to use the error-page element in web.xml to specify the invocation of servlets in response to certain exceptions or HTTP status codes.

Consider, you have an ErrorHandler servlet which would be called whenever there is any defined exception or error. Following would be the entry created in web.xml.

```xml
<servlet>
 <display-name>ErrorHandler</display-name>
 <servlet-name>ErrorHandler</servlet-name>
 <servlet-class>servlets.ErrorHandler</servlet-class>
</servlet>
<servlet-mapping>
 <servlet-name>ErrorHandler</servlet-name>
 <url-pattern>/error</url-pattern>
</servlet-mapping>

<!-- error-code related error pages -->
<error-page>
 <error-code>404</error-code>
 <location>/error</location>
</error-page>
<error-page>
 <error-code>403</error-code>
 <location>/error</location>
</error-page>

<!-- exception-type related error pages -->
<error-page>
 <exception-type>
     javax.servlet.ServletException
 </exception-type >
 <location>/error</location>
</error-page>

<error-page>
 <exception-type>java.lang.Exception</exception-type >
 <location>/error</location>
</error-page>
```

Following are the points to be noted about above web.xml for Exception Handling:

▪ The servlet ErrorHandler is defined in usual way as any other servlet and configured in web.xml.

▪ If there is any error with status code either 404 (Not Found) or 403 (Forbidden), then ErrorHandler servlet would be called.

- If the web application throws either ServletException or IOException, then the web container invokes the /ErrorHandler servlet.
- You can define different Error Handlers to handle different type of errors or exceptions. Above example is very much generic and hope it serve the purpose to explain you the basic concept.

Following is the Servlet Example that would be used as Error Handler in case of any error or exception occurs with your any of the servlet defined. This example would give you basic understanding of Exception Handling in Servlet, but you can write more sophisticated filter applications using the same concept:

```java
package servlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

public class ErrorHandler extends HttpServlet {

 public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException  {
    // Analyze the servlet exception
    Throwable throwable = (Throwable) request.getAttribute("javax.servlet.error.exception");
    Integer statusCode = (Integer) request.getAttribute("javax.servlet.error.status_code");
    String servletName = (String) request.getAttribute("javax.servlet.error.servlet_name");

    if (servletName == null){
      servletName = "Unknown";
    }
    String requestUri = (String) request.getAttribute("javax.servlet.error.request_uri");
    if (requestUri == null){
      requestUri = "Unknown";
    }

    // Set response content type
    response.setContentType("text/html");

    PrintWriter out = response.getWriter();
    String title = "Error/Exception Information";
    String docType = "<!doctype html public \"-//w3c//dtd html 4.0 " + "transitional//en\">\n";
    out.println(docType + "<html>\n" + "<head><title>" + title + "</title></head>\n" +
      "<body bgcolor=\"#f0f0f0\">\n");

    if (throwable == null && statusCode == null){
      out.println("<h2>Error information is missing</h2>");
      out.println("Please return to the <a href=\"" + response.encodeURL("http://localhost:8080/") +
        "\">Home Page</a>.");
     }else if (throwable == null){
       out.println("The status code : " + statusCode);
     }else{
       out.println("<h2>Error information</h2>");
       out.println("The status code : " + statusCode + "</br></br>");
       out.println("Servlet Name : " + servletName + "</br></br>");
       out.println("Exception Type : " + throwable.getClass( ).getName( ) + "</br></br>");
       out.println("The request URI: " + requestUri + "<br><br>");
       out.println("The exception message: " + throwable.getMessage( ));
```

```
        }
        out.println("</body>");
        out.println("</html>");
    }
}
```

2.3.5.3 Sử dụng bộ lọc (filter)

Servlet Filters are Java classes that can be used in Servlet Programming for the following purposes:

- To intercept requests from a client before they access a resource at back end.
- To manipulate responses from server before they are sent back to the client.

There are are various types of filters suggested by the specifications:

- Authentication Filters.
- Data compression Filters.
- Encryption Filters.
- Filters that trigger resource access events.
- Image Conversion Filters.
- Logging and Auditing Filters.
- MIME-TYPE Chain Filters.
- Tokenizing Filters .
- XSL/T Filters That Transform XML Content.

Filters are deployed in the deployment descriptor file web.xml and then map to either servlet names or URL patterns in your application's deployment descriptor.

When the web container starts up your web application, it creates an instance of each filter that you have declared in the deployment descriptor. The filters execute in the order that they are declared in the deployment descriptor.

Following is the Servlet Filter Example that would print the clients IP address and current date time. This example would give you basic understanding of Servlet Filter, but you can write more sophisticated filter applications using the same concept:

```
package servlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

public class LogFilter implements Filter  {
  public void  init(FilterConfig config) throws ServletException {
    // Get init parameter
    String testParam = config.getInitParameter("test-param");

    //Print the init parameter
    System.out.println("Test Param: " + testParam);
  }
  public void doFilter(ServletRequest request, ServletResponse response,  FilterChain chain)
```

```
            throws java.io.IOException, ServletException {

   // Get the IP address of client machine.
   String ipAddress = request.getRemoteAddr();

   // Log the IP address and current timestamp.
   System.out.println("IP "+ ipAddress + ", Time " + new Date().toString());

   // Pass request back down the filter chain
   chain.doFilter(request,response);
  }

  public void destroy() {
  }
}
```

Filters are defined and then mapped to a URL or Servlet, in much the same way as Servlet is defined and then mapped to a URL pattern. Create the following entry for filter tag in the deployment descriptor file **web.xml**

```xml
<filter>
  <filter-name>LogFilter</filter-name>
  <filter-class>LogFilter</filter-class>
  <init-param>
        <param-name>test-param</param-name>
        <param-value>Initialization Paramter</param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>LogFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

The above filter would apply to all the servlets because we specified **/*** in our configuration. You can specicy a particular servlet path if you want to apply filter on few servlets only. Now try to call any servlet in usual way and you would see generated log in your web server log. You can use Log4J logger to log above log in a separate file.

Your web application may define several different filters with a specific purpose. Consider, you define two filters *AuthenFilter* and *LogFilter*. Rest of the process would remain as explained above except you need to create a different mapping as mentioned below:

```xml
<filter>
  <filter-name>LogFilter</filter-name>
  <filter-class>LogFilter</filter-class>
  <init-param>
        <param-name>test-param</param-name>
        <param-value>Initialization Paramter</param-value>
  </init-param>
</filter>

<filter>
  <filter-name>AuthenFilter</filter-name>
  <filter-class>AuthenFilter</filter-class>
```

```
    <init-param>
            <param-name>test-param</param-name>
            <param-value>Initialization Paramter</param-value>
    </init-param>
</filter>

<filter-mapping>
  <filter-name>LogFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>

<filter-mapping>
  <filter-name>AuthenFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

The order of filter-mapping elements in web.xml determines the order in which the web container applies the filter to the servlet. To reverse the order of the filter, you just need to reverse the filter-mapping elements in the web.xml file. For example, above example would apply LogFilter first and then it would apply AuthenFilter to any servlet but the following example would reverse the order:

```
<filter-mapping>
  <filter-name>AuthenFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>

<filter-mapping>
  <filter-name>LogFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

2.3.6 Truy nhập cơ sở dữ liệu

2.3.6.1 Chuẩn bị môi trường

Sử dụng database MySQL:

- ▪ Cài đặt cơ sở dữ liệu MySQL, tạo cơ sở dữ liệu emp, tạo bảng dữ liệu employees, tạo user emp (password emp) và cấp quyền truy nhập đến cơ sở dữ liệu emp. Bảng dữ liệu employees có thể được tạo ra bằng các câu lệnh SQL như bên dưới

```
mysql> use emp;
mysql> create table employees
   -> (
   -> id int not null,
   -> age int not null,
   -> first varchar (255),
   -> last varchar (255)
   -> );
Query OK, 0 rows affected (0.08 sec)
mysql>
```

- Tạo các dữ liệu trong bảng employees:

```
mysql> INSERT INTO employees VALUES (100, 18, 'Zara', 'Ali');
Query OK, 1 row affected (0.05 sec)

mysql> INSERT INTO employees VALUES (101, 25, 'Mahnaz', 'Fatma');
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO employees VALUES (102, 30, 'Zaid', 'Khan');
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO employees VALUES (103, 28, 'Sumit', 'Mittal');
Query OK, 1 row affected (0.00 sec)

mysql>
```

Cấu hình thư viện JDBC cho Tomcat:

Mỗi cơ sở dữ liệu hỗ trợ JDBC đều có kèm theo thư viện Java. Với phiên bản database MySQL 5.6, file thư viện JDBC nằm trong bộ Connector.J 5.1 có tên là mysql-connector-java-5.1.34-bin.jar. Để Tomcat tìm được thư viện này, chỉ cần copy file jar vào thư mục lib của Tomcat.

2.3.6.2 Servlet DatabaseAccess

```java
package servlets;

import java.io.*;
import java.util.*;

import javax.servlet.*;
import javax.servlet.http.*;

import java.sql.*;

public class DatabaseAccess extends HttpServlet{

 public void doGet(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException  {
   // JDBC driver name and database URL
   String JDBC_DRIVER="com.mysql.jdbc.Driver";
   String DB_URL="jdbc:mysql://localhost/emp";

   // Database credentials
   String USER = "emp";
   String PASS = "emp";

   Connection conn = null;
   Statement stmt = null;

   // Set response content type
   response.setContentType("text/html");
   PrintWriter out = response.getWriter();
   String title = "Database Result";
   String docType =
    "<!doctype html public \"-//w3c//dtd html 4.0 " +
     "transitional//en\">\n";
```

```java
      out.println(docType +
        "<html>\n" +
        "<head><title>" + title + "</title></head>\n" +
        "<body bgcolor=\"#f0f0f0\">\n" +
        "<h1 align=\"center\">" + title + "</h1>\n");
      try{
        // Register JDBC driver
        Class.forName("com.mysql.jdbc.Driver");

        // Open a connection
        conn = DriverManager.getConnection(DB_URL,USER,PASS);

        // Execute SQL query
        stmt = conn.createStatement();
        String sql;
        sql = "SELECT id, first, last, age FROM Employees";
        ResultSet rs = stmt.executeQuery(sql);

        // Extract data from result set
        while(rs.next()){
          //Retrieve by column name
          int id  = rs.getInt("id");
          int age = rs.getInt("age");
          String first = rs.getString("first");
          String last = rs.getString("last");

          //Display values
          out.println("ID: " + id + "<br>");
          out.println(", Age: " + age + "<br>");
          out.println(", First: " + first + "<br>");
          out.println(", Last: " + last + "<br>");
        }
        out.println("</body></html>");

        // Clean-up environment
        rs.close();
        stmt.close();
        conn.close();
      }catch(SQLException se){
        //Handle errors for JDBC
        se.printStackTrace();
      }catch(Exception e){
        //Handle errors for Class.forName
        e.printStackTrace();
      }finally{
        //finally block used to close resources
        try{
          if(stmt!=null)
            stmt.close();
        }catch(SQLException se2){
        }// nothing we can do
        try{
          if(conn!=null)
            conn.close();
        }catch(SQLException se){
          se.printStackTrace();
        }//end finally try
      } //end try
    }
```

```
}
```

Cập nhật file web.xml cho servlet này:

```xml
<servlet>
  <servlet-name>DatabaseAccess</servlet-name>
  <servlet-class>servlets.DatabaseAccess</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>DatabaseAccess</servlet-name>
  <url-pattern>/database</url-pattern>
</servlet-mapping>
```

Chạy ứng dụng và truy nhập vào địa chỉ http://localhost:8080/my_servlet/database, kết quả trên trang web hiện thị danh sách các dữ liệu đã được đưa vào bảng employees.

Chương 3. JSP

3.1 Từ Servlet đến JSP

Servlet cung cấp đủ khả năng xây dựng các ứng dụng web động phía server (những gì Java làm được thì Servlet cũng làm được). Tuy nhiên việc tạo giao diện tĩnh (trang HTML) bằng cách xây dựng toàn bộ nội dung code HTML trong chương trình là rất phức tạp. Khi cần thay đổi một số điểm nhỏ trong giao diện web vốn dĩ không liên quan đến xử lý logic của ứng dụng nhưng vẫn cần thay đổi code và biên dịch lại chương trình. Đặc biệt, đối với các ứng dụng web RIA (web động cả phía client dựa trên các script chạy trong browser) thì việc tạo code script chứa trong code HTML từ các servlet là cực kỳ phức tạp. Đây là lý do cần thiết phải tách phần xử lý giao diện ra khỏi phần xử lý logic của ứng dụng và JSP là một giải pháp.

3.1.1 Project my_jsp với Eclipse và Tomcat

Sử dụng Eclipse, tạo project kiểu Dynamic Web Project đặt tên là my_jsp và tạo file JSP với nội dung như bên dưới:

```html
<html>
    <head>
            <title>Hello World</title>
    </head>
    <body>
            <h1>Hello World!</h1><br>
            <% out.println("Your IP address is " + request.getRemoteAddr()); %>
    </body>
</html>
```

Chạy server Tomcat và dùng web browser truy nhập vào trang jsp này tại địa chỉ http://localhost:8080/my_jsp/hello.jsp, kết quả nhận được là

Hello World!

Your IP address is 127.0.0.1

3.1.2 Servlet Hello_JSP

Kết quả chương trình khi chạy với hello.jsp hoàn toàn có thể được thực hiện bằng một servlet với nội dung như bên dưới:

```java
public class Hello_JSP extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
        response.setContentType("text/html");
        response.setBufferSize(8192);

        PrintWriter out = response.getWriter();

        out.println("<html>" + "<head><title>Hello World</title></head> +
            "<body><h1>Hello World</h2><br>");

        out.println("Your IP address is " + request.getRemoteAddr());
```

```
    out.println("</body></html>");
    out.close();
  }
}
```

Code 2: Java Servlet được tạo ra tự động từ trang JSP

Hơn thế nữa, để ý thấy rằng việc chuyển đổi từ nội dung file JSP hello.jsp thành Servlet Hello_JSP.java có thể được thực hiện dựa trên một số qui tắc chuẩn, không cần có sự tham gia soạn lại chương trình của con người.

Thực tế xảy ra là you *write* a JSP, but it *becomes* a servlet. The only way to really tell what's happening is to look at what the Container does to your JSP code. In other words, how does the Container *translate* your JSP intoa servlet? Once you know where different JSP elements land in the servlet's class file, you'll find it much easier to know how to structure your JSP.

The servlet code above is *not* the real code generated by the Container—we simplified it down to the essential parts. The Container-generated servlet file is, well, *uglier*. The real generated servlet source code is slightly harder to read, but we will look at the real thing in a few pages. For now, though, all we care about is *where* in the servlet class our JSP code actually ends up.

Time to see the REAL generated servlet

We've been looking at a super-simplified version of the servlet the Container actually creates from your JSP. There's no need to look at the Container-generated code during development, but you can use it to help learn. Once you've seen what the Container does with the different elements of a JSP, you shouldn't need to ever look at the Container-generated .java source files. Some vendors won't let you see the generated Java source, and keep only the compiled .class files. Don't be intimidated when you see parts of the API that you don't recognize. Most of the class and interface types are vendor-specific implementations you shouldn't care about.

Để biết các file servlet mà Tomcat tự động tạo ra để ở đâu, hiển thị mục Servers trong Eclipse, tìm đến Tomcat server và click chuột đúp vào đó để hiển thị thông tin chi tiết cấu hình server. Kiểm tra mục Server path trong phần Server Locations.
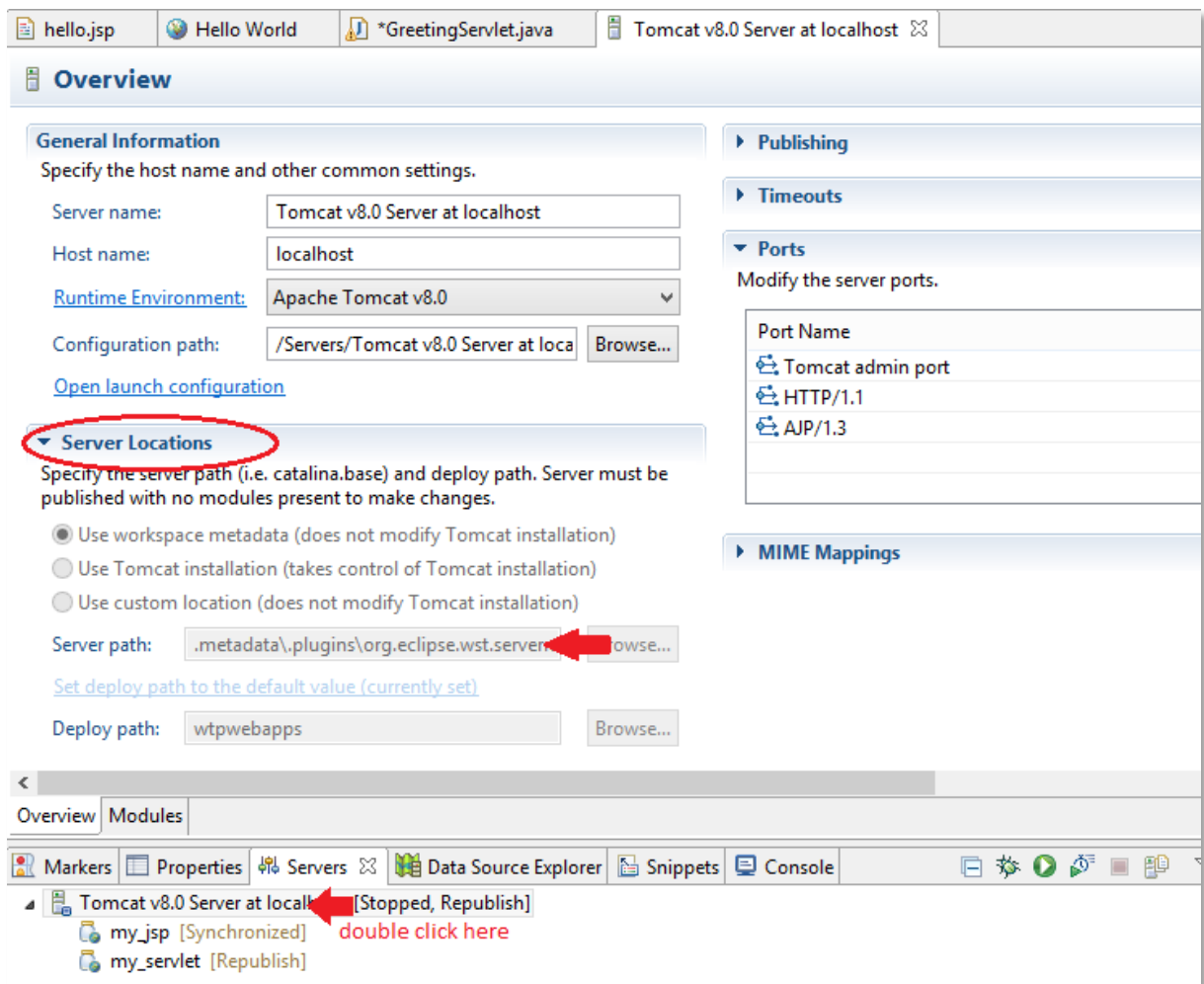
Servlet được Tomcat tự động tạo ra cho JSP hello.jsp có nội dung như bên dưới:

```
/*
 * Generated by the Jasper component of Apache Tomcat
 * Version: Apache Tomcat/8.0.18
 * Generated at: 2015-02-03 07:15:58 UTC
 * Note: The last modified time of this file was set to
 *       the last modified time of the source file after
 *       generation to assist with modification tracking.
 */
package org.apache.jsp;

import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;

public final class hello_jsp extends org.apache.jasper.runtime.HttpJspBase
    implements org.apache.jasper.runtime.JspSourceDependent,
            org.apache.jasper.runtime.JspSourceImports {

  private static final javax.servlet.jsp.JspFactory _jspxFactory =
        javax.servlet.jsp.JspFactory.getDefaultFactory();

  private static java.util.Map<java.lang.String,java.lang.Long> _jspx_dependants;

  private static final java.util.Set<java.lang.String> _jspx_imports_packages;
```

```java
  private static final java.util.Set<java.lang.String> _jspx_imports_classes;

  static {
    _jspx_imports_packages = new java.util.HashSet<>();
    _jspx_imports_packages.add("javax.servlet");
    _jspx_imports_packages.add("javax.servlet.http");
    _jspx_imports_packages.add("javax.servlet.jsp");
    _jspx_imports_classes = null;
  }

  private javax.el.ExpressionFactory _el_expressionfactory;
  private org.apache.tomcat.InstanceManager _jsp_instancemanager;

  public java.util.Map<java.lang.String,java.lang.Long> getDependants() {
    return _jspx_dependants;
  }

  public java.util.Set<java.lang.String> getPackageImports() {
    return _jspx_imports_packages;
  }

  public java.util.Set<java.lang.String> getClassImports() {
    return _jspx_imports_classes;
  }

  public void _jspInit() {
    _el_expressionfactory =
_jspxFactory.getJspApplicationContext(getServletConfig().getServletContext()).getExpressionFactory();
    _jsp_instancemanager =
org.apache.jasper.runtime.InstanceManagerFactory.getInstanceManager(getServletConfig());
  }

  public void _jspDestroy() {
  }

  public void _jspService(final javax.servlet.http.HttpServletRequest request, final
javax.servlet.http.HttpServletResponse response)
      throws java.io.IOException, javax.servlet.ServletException {

final java.lang.String _jspx_method = request.getMethod();
if (!"GET".equals(_jspx_method) && !"POST".equals(_jspx_method) &&
!"HEAD".equals(_jspx_method) &&
!javax.servlet.DispatcherType.ERROR.equals(request.getDispatcherType())) {
response.sendError(HttpServletResponse.SC_METHOD_NOT_ALLOWED, "JSPs only permit GET POST
or HEAD");
return;
}

    final javax.servlet.jsp.PageContext pageContext;
    javax.servlet.http.HttpSession session = null;
    final javax.servlet.ServletContext application;
    final javax.servlet.ServletConfig config;
    javax.servlet.jsp.JspWriter out = null;
    final java.lang.Object page = this;
    javax.servlet.jsp.JspWriter _jspx_out = null;
    javax.servlet.jsp.PageContext _jspx_page_context = null;


    try {
```

```java
        response.setContentType("text/html");
        pageContext = _jspxFactory.getPageContext(this, request, response,
                              null, true, 8192, true);
        _jspx_page_context = pageContext;
        application = pageContext.getServletContext();
        config = pageContext.getServletConfig();
        session = pageContext.getSession();
        out = pageContext.getOut();
        _jspx_out = out;

        out.write("<html>\r\n");
        out.write("\t<head>\r\n");
        out.write("\t\t<title>Hello World</title>\r\n");
        out.write("\t</head>\r\n");
        out.write("\t<body>\r\n");
        out.write("\t\t<h1>Hello World!</h1><br>\r\n");
        out.write("\t\t");
  out.println("Your IP address is " + request.getRemoteAddr());
        out.write("\t\t\r\n");
        out.write("\t</body>\r\n");
        out.write(" </html>");
      } catch (java.lang.Throwable t) {
        if (!(t instanceof javax.servlet.jsp.SkipPageException)){
          out = _jspx_out;
          if (out != null && out.getBufferSize() != 0)
            try {
              if (response.isCommitted()) {
                out.flush();
              } else {
                out.clearBuffer();
              }
            } catch (java.io.IOException e) {}
          if (_jspx_page_context != null) _jspx_page_context.handlePageException(t);
          else throw new ServletException(t);
        }
      } finally {
        _jspxFactory.releasePageContext(_jspx_page_context);
      }
    }
}
```

3.1.3 Vòng đời hoạt động của JSP

(Bryan Basham, 2008) – trang 306:

① Kim writes a .jsp file, and deploys it as part of a web app.

The Container "reads" the web.xml (DD) for this app, but doesn't do anything else with the .jsp file (until the first time it's requested).

It's just sitting here on the server...waiting for a client to request it

Web Container   web.xml   MyJSP.jsp
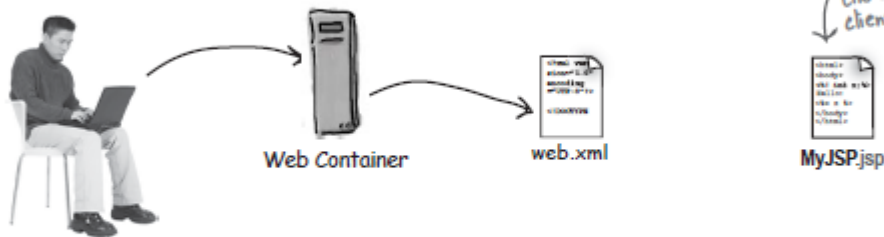
② The client hits a link that asks for the .jsp.

The Container tries to TRANSLATE the .jsp into .java source code for a servlet class.

JSP syntax errors are caught in this phase.

request → translate → MyJSP.jsp → generate → MyJSP_jsp.java
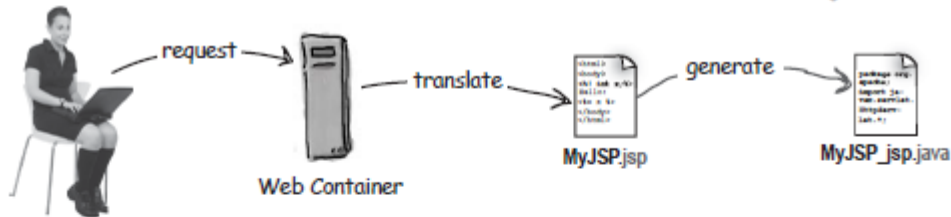
Web Container

③ The Container tries to COMPILE the servlet .java source into a .class file.

Java language/syntax errors are caught here.

compile → MyJSP_jsp.java → generate → MyJSP_jsp.class
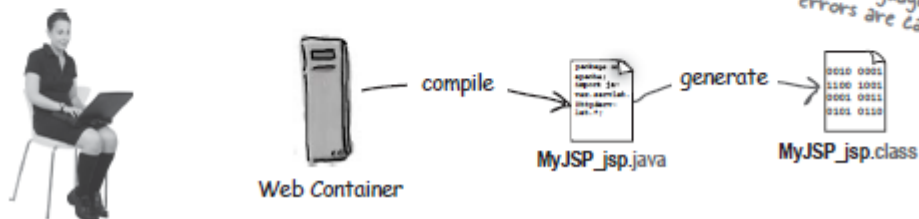
Web Container

④ The Container LOADS the newly-generated servlet class.

load → MyJSP_jsp.class
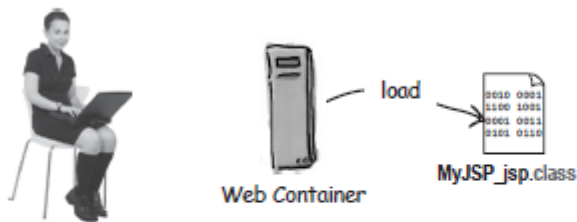
Web Container

⑤ The Container instantiates the servlet and causes the servlet's jspInit() method to run.

The object is now a full-fledged servlet, ready to accept client requests.

jspInit() → MyJSP_jsp becomes → MyJSP_jsp
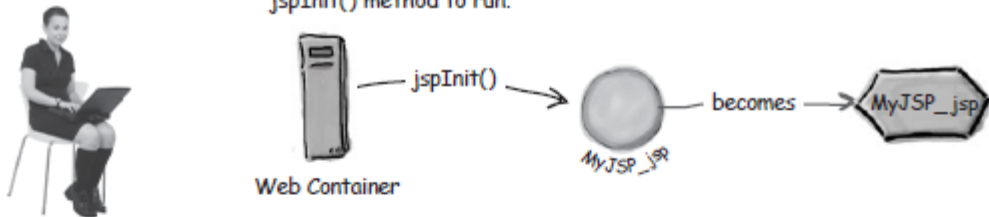
Web Container

⑥ The Container creates a new thread to handle this client's request, and the servlet's _jspService() method runs.

Everything that happens after this is just plain old servlet request-handling.

Eventually the servlet sends a response back to the client (or forwards the request to another web app component).

_jspService() → MyJSP_jsp / Thread A

Web Container
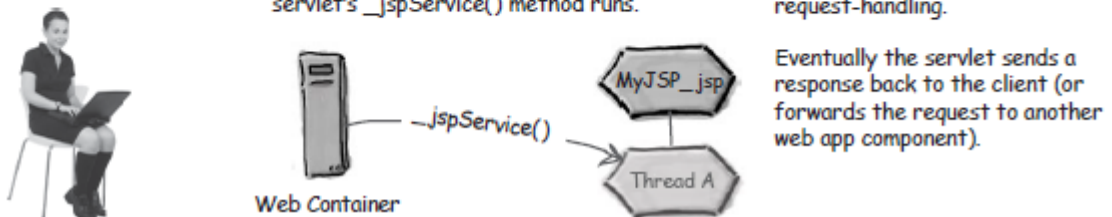
3.2 Xử giao diện với JSP

3.2.1 JSP = HTML++

Điểm mạnh nhất của JSP chính là khả năng tích hợp toàn bộ mã HTML. Một trang HTML bất kỳ đều có thể được khai báo là một JSP. Nói một cách khác, những gì có thể làm giao diện web với mã HTML thì cũng có thể làm như vậy với JSP.

3.2.2 Xử dụng Directives, Declarations, Scriptlets, and Expressions

In addition to the various HTML tags you can use within a JSP, there are several unique structures that define a sort of JSP language. They are directives, declarations, scriptlets, and expressions. In the simplest terms, they look like this:

```
<%@ this is a directive %>
<%! This is a declaration %>
<% this is a scriptlet %>
<%= this is an expression %>
```

Directive:

Trong 3 loại directive (Page, Include và Taglib), Include có thể được sử dụng để ghép nối các phần giao diện trong một trang web. The following example demonstrates using the include directive to include a standard JSP header and footer in the main.jsp:

```
<%@ include file="header.jsp" %>
<p>content</p>
<%@ include file="footer.jsp" %>
```

Trong đó header.jsp có nội dung sau:

```
<html>
<head></head>
<body>
  <%out.print("header"); %>
```

Và footer.jsp có nội dung sau:

```
1.<%out.print("footer"); %>
2.</body>
3.</html>
```

Kết quả chạy chương trình:
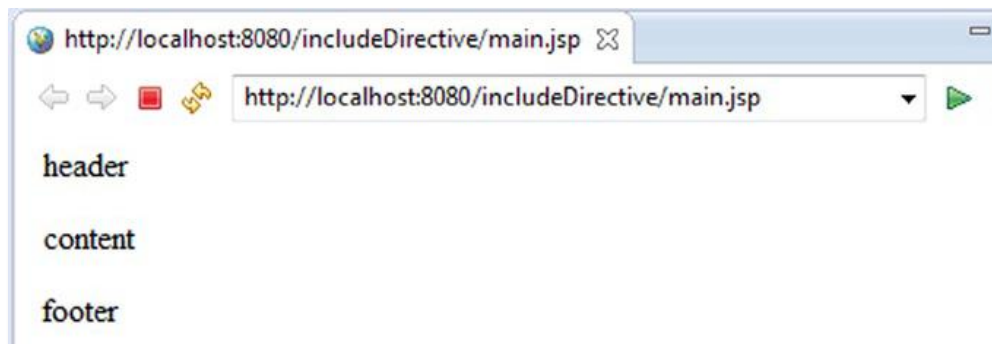
Declarations & Scriptlets

Using declarations, JSP allows you to declare methods and variables in JSP pages. Once they are in a JSP page, they are available to scriptlets and expressions throughout the page. JSP declarations are placed between <%! and %> declaration delimiters. Since declarations are used with expressions and scriptlets, I will introduce expressions and scriptlets in the following sections, and then I will show you how declarations, scriptlets, and expressions are used in a JSP page.

Scriptlets are blocks of Java code surrounded within the <% and %> delimiters to create dynamic content. Listing 2-19 illustrates the usage of a declaration with a scriptlet and expression. Listing 2-19. Usage of Declaration, Scriptlet, and Expression

```
.<%!
  public String hello() {
    String msg = "Hello World";
    return msg;
  }
```

```
%>
Message from <b>Scriptlet</b>: <%hello();%><br/>
Message from <b>Expression</b>: <%=hello() %>
```

Expressions

Expressions are similar to scriptlets, but they evaluate a regular Java expression and return a result, which is a String or something convertible to a String, to the client as part of the response. The general syntax is as follows:

```
<%= expression %>
```

3.3   Truy nhập đến các đối tượng có sẵn (implicit objects)

Các trang JSP được tự động chuyển đổi thành các servlet với qui tắc giữ nguyên phần mã Java (Declaration, Scriplet và Expression) và chuyển phần code HTML vào câu lệnh *out.write()* (xem Code 2: Java Servlet được tạo ra tự động từ trang JSP). Phần mã Java có thể truy nhập đến tất cả các đối tượng được Container tạo ra để phục vụ servlet. Ví dụ các đối tượng *ServletRequest*, *ServletResponse*, v.v.. Đây là cách thức trao đổi thông tin giữa phần xử lý giao diện (bằng JSP) với phần xử lý logic ứng dụng (bằng servlet).

A JSP page can access some specific objects through scripting variables. These objects are provided by the JSP container and are called implicit objects. These implicit objects can be accessed in scriptlets, in expressions, or as part of the EL expressions. (The EL expressions are introduced in Chapter 3.) Table 2-5 lists the nine implicit objects with the corresponding API.

Implicit Object	Usage	API
application	Accesses application-level objects	ServletContext
config	Provides configuration information	ServletConfig
exception	Accesses error status	JSPException
out	Accesses the JSP output stream	JSPWriter
page	Provides a reference to the current JSP	Object
pageContext	Accesses the JSP container	PageContext
request	Provides access to the client request	ServletRequest
response	Provides access to the JSP response	ServletResponse
session	Shares information across client requests	HttpSession

Tham khảo phương thức *_jspService()* của servlet được tạo ra tự động từ trang JSP (Code 2: Java Servlet được tạo ra tự động từ trang JSP) có thể thấy sự xuất hiện của các đối tượng có sẵn này.

request and response

The *request* variable is an instance of `HttpServletRequest` and the *response* variable is an instance of `HttpServletResponse`, both of which you learned about in detail in Chapter 3. Anything you can do with a request in a Servlet you can also do in a JSP, including getting request parameters, getting and setting attributes, and even reading from the response body.

The same rules you learned about in the last chapter apply here. However, there are some restrictions on what you can do with the response object in a JSP. These restrictions are not contract restrictions, so they are not enforced at compile time. Instead, they are enforced at run time because violating them could cause unexpected behavior or even errors. For example, you should not call `getWriter` or `getOutputStream` because the JSP is already writing to the response output. You also should not set the content type or character encoding, flush or reset the buffer, or change the buffer size. These are all things that the JSP does, and if your code does them, too, it can cause problems.

session

This variable is an instance of `HttpSession`. You learn more about sessions in the next chapter. Remember from the previous section that the `page` directive has a `session` attribute that defaults to `true`. This is why the *session* variable is available in the previous code example and will be available by default in all of your JSPs. If you set the `page` directive's `session` attribute to `false`, the *session* variable in the JSP is not defined and cannot be used.

out

The `JspWriter` instance *out* is available for you to use in all your JSPs. It is a `Writer`, just like what you get from calling the `getWriter` method on `HttpServletResponse`. If for some reason you need to write directly to the response, you should use the *out* variable. However, in most cases you can simply use an expression or write text or HTML content in the JSP.

application

This is an instance of the `ServletContext` interface. Recall from Chapter 3 that this interface gives you access to the configuration of the web application as a whole, including all the context init parameters. Why this variable was named *application* instead of *context* or *servletContext* is a mystery.

config

The *config* variable is an instance of the `ServletConfig` interface. Unlike the *application* variable, its name actually reflects what it is. As you learned in Chapter 3, you can use this object to access the configuration of the JSP Servlet, such as the Servlet init parameters.

pageContext

This object, an instance of the `PageContext` class, provides several convenience methods for getting request attributes and session attributes, accessing the request and response, including other files, and forwarding the request. You will probably never need to use this class within a JSP. It will, however, come in handy when you write custom JSP tags in Chapter 8.

page

The *page* variable is an interesting object to examine. It is an instance of `java.lang.Object`, which initially makes it seem unuseful. However, it essentially is the *this*

variable from the JSP Servlet object. So, you could cast it to `Servlet` and use methods defined on the `Servlet` interface. It is also a `javax.servlet.jsp.JspPage` (which extends `Servlet`) and a `javax.servlet.jsp.HttpJspPage` (which extends `JspPage`), so you could cast it to either of those and use methods defined on those interfaces. In reality, you will probably never have a reason to use this variable. It may be useful if other JSP scripting languages are ever supported. However, the JSP 2.3 specification, section 1.8.3 note "a," says that *page* is always a synonym for *this* when the scripting language is Java. Thus, anything you can do with *page* (such as get the Servlet name or access methods or instance variables you defined in a JSP declaration) you can also do with *this*.

exception

This is the variable that was missing from the previous code example. Recall from the previous section that you can specify as `true` the `isErrorPage` attribute on the `page` directive to indicate that the JSP's purpose is to handle errors. Doing so makes the *exception* variable available for use within the JSP. Because the default value for `isErrorPage` is `false` and you have not used it anywhere, the *exception* variable has not been defined in any JSPs you created. If you create a JSP with `isErrorPage` set to `true`, the implicit *exception* variable, a `Throwable`, is defined automatically.

Chương trình ví dụ

Now that you understand the available implicit variables and their purposes, you should explore this more by writing some JSP code that uses the implicit variables. In your project, create a `greeting.jsp` file in the web root, and place the following code in it

```jsp
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<%!
        private static final String DEFAULT_USER = "Guest";
%>
<%

        String user = request.getParameter("user");
        if(user == null) user = DEFAULT_USER;
%>
<!DOCTYPE html>
<html>
        <head>
                <title>Hello User Application</title>
        </head>
        <body>
                Hello, <%= user %>!<br /><br />
                <form action="greeting.jsp" method="POST">
                Enter your name:<br />
                <input type="text" name="user" /><br />
                <input type="submit" value="Submit" />
                </form>
        </body>
</html>
```

Compare this to the code you wrote in `HelloServlet.java` for the Hello-User project in the previous chapter. There's much less to it, but it accomplishes the same thing. Notice the use of a declaration to define the *DEFAULT_USER* variable, a scriptlet to look for the *user* request

parameter and default it if it is not set, and an *expression* to output the value of the user variable. Now compile and debug this code and go to `http://localhost:8080/hello-world/greeting.jsp` in your browser. Try entering a name in the input field and clicking the Submit button — the post variable is detected and used. Now try going to `http://localhost:8080/hello-world/greeting.jsp?user=Allison`, and you should see that the query parameter is also detected and used. You are encouraged to explore the Java code that Tomcat translated your JSP into.

Another thing you did in the Hello-User project was create a Servlet to demonstrate using multiplevalue parameters. This, too, can be replicated using JSPs. Create a file in your project web root named `checkboxes.jsp`

```jsp
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<!DOCTYPE html>
<html>
        <head>
                <title>Hello User Application</title>
        </head>
        <body>
                <form action="checkboxesSubmit.jsp" method="POST">
                        Select the fruits you like to eat:<br />
                        <input type="checkbox" name="fruit" value="Banana" /> Banana<br />
                        <input type="checkbox" name="fruit" value="Apple" /> Apple<br />
                        <input type="checkbox" name="fruit" value="Orange" /> Orange<br />
                        <input type="checkbox" name="fruit" value="Guava" /> Guava<br />
                        <input type="checkbox" name="fruit" value="Kiwi" /> Kiwi<br />
                        <input type="submit" value="Submit" />
                </form>
        </body>
</html>
```

This file replicates the output of the `doGet` method in the `MultiValueParameterServlet.java` file from the Hello-User project. Next, create `checkboxesSubmit.jsp`

```jsp
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<%
        String[] fruits = request.getParameterValues("fruit");
%>
<!DOCTYPE html>
<html>
        <head>
                <title>Hello User Application</title>
        </head>
        <body>
                <h2>Your Selections</h2>
                <%
                        if(fruits == null) {
                %>You did not select any fruits.
                <% } else {
                %><ul><%
                        for(String fruit : fruits) {
                                out.println("<li>" + fruit + "</li>");
                        }
```

```
            %></ul>
            <% }
            %>
        </body>
</html>
```

This file replicates the logic and output of the `doPost` method from the `MultiValueParameterServlet` class. Notice how the bold code jumps in and out of scriptlets, using Java only where the logic requirements demand and leaving the scriptlets to use straight output instead of writing with the implicit *out* variable. The exception is inside the `for` loop, which demonstrates one use case for the *out* variable. This could have just as easily been replaced with `%><li><%= fruit %></li><%` to accomplish the same thing. Now compile and debug the project and go to `http://localhost:8080/hello-world/checkboxes.jsp` in your browser. You should see a page like that in Figure 4-1. Experiment with different combinations of the check boxes, and verify that it behaves identically to the Hello-User project in Chapter 3. Try replacing the use of *out* in the `for` loop with `%><li><%= fruit %></li><%`. When you recompile and run the project again, the output should not change.

Finally, create a file named `contextParameters.jsp` to explore the use of the *application* implicit variable and the retrieval of context init parameters. Alternatively, use the file already in the Hello-User-JSP project.

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<!DOCTYPE html>
<html>
<head>
<title>Hello User Application</title>
</head>
<body>
settingOne: <%= application.getInitParameter("settingOne") %>,
settingTwo: <%= application.getInitParameter("settingTwo") %>
</body>
</html>
```

Also, you need to have some context init parameters defined in your deployment descriptor, just like in Chapter 3:

```
<context-param>
<param-name>settingOne</param-name>
<param-value>foo</param-value>
</context-param>
<context-param>
<param-name>settingTwo</param-name>
<param-value>bar</param-value>
</context-param>
```

Now compile, debug, and navigate to `http://localhost:8080/hello-world/contextParameters.jsp`. As with the Servlet-based Hello-User project, you should see the values of the context init parameters
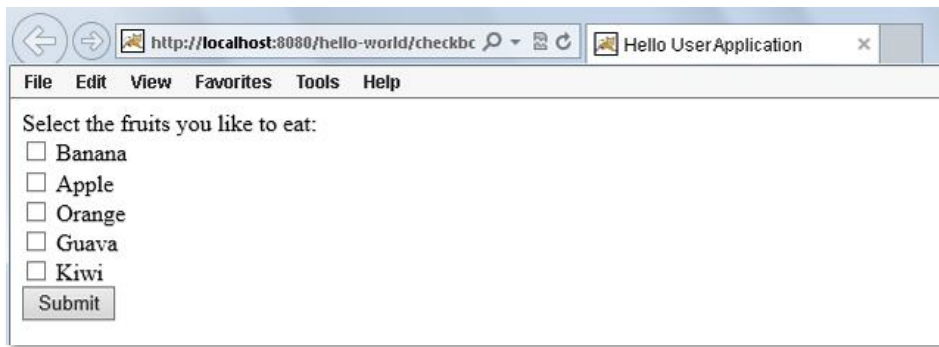
Chương trình quản lý session bằng servlet như trong Code 1: Quản lý session với Servlet khi chuyển sang JSP sẽ như sau:

```html
<html>
  <head>
    <title>Session Tracking by JSP</title>
  </head>
  <body>

<%@page import="java.util.*" %>

<%
boolean firstVisit = false;
String uName = request.getParameter("username");

if ((uName != null) && (uName.length() > 0)) {
        System.out.println("Putting user name: " + uName + " to session data...");
        session.setAttribute("userNameKey", uName);
}

session = request.getSession(true);
if (session.isNew()) firstVisit = true;

if (request.getParameter("reset") != null) {
        session.invalidate();
        session = request.getSession(true);
        firstVisit = true;
}

if (firstVisit) {
%>
        <b>Welcome to my website</b>
        <form method="get">
                <input type="text" name="username" size="25" ><p>
                <input type="submit" value="Submit">
        </form>
<% } else { %>
        <b>Welcome Back to my website</b><br>
        Session create time: <%= new Date(session.getCreationTime()) %><br>
        Session last access: <%= new Date(session.getLastAccessedTime()) %><br>
        Stored user in session data: <%= (String)session.getAttribute("userNameKey")%><p>-----<br>
        Please enter new user:
        <form method="get">
                <input type="text" name="username" size="25" value="<%= uName %>"><p>
                <input type="submit" value="Submit">
        </form>
        <form method="get" >
```

```
                <input type="hidden" name="reset" value="true" >
                <input type="submit" value="Reset Session">
        </form>
<% } %>
  </body>
</html>
```

Code 3: Quản lý Session bằng đối tượng có sẵn session trong JSP

3.4 Custom Tag Library

Nhược điểm của JSP là trộn lẫn các tag xử lý format hiển thị cùng với các tag chương trình. Điều này dẫn đến người xử lý giao diện có thể gây ra các lỗi chương trình và ngược lại, người viết code chương trình lại làm hỏng giao diện. Đối với các hệ thống ứng dụng web lớn, công việc xây dựng giao diện web và xử lý chương trình Java thường được tách riêng cho nhưng nhân viên có chuyên môn khác nhau. Giải pháp mà JSP cung cấp là sử dụng các tag tương ứng với các mã lệnh Java.

Chúng ta đã làm quen với một số tag chuẩn của JSP như <jsp:include>, <jsp:forward>, v.v.. Thực tế là đằng sau các tag này chính là các dòng lệnh Java đã được xây dựng trước. Và khi người xây dựng giao diện web chèn các tag này vào trang JSP thì tương đương với việc chèn các mã lệnh Java tương ứng vào trang JSP.

Standard Tag Library (JSTL)

The JavaServer Pages Standard Tag Library (JSTL) is a collection of useful JSP tags which encapsulates core functionality common to many JSP applications. JSTL has support for common, structural tasks such as iteration and conditionals, tags for manipulating XML documents, internationalization tags, and SQL tags. It also provides a framework for integrating existing custom tags with JSTL tags.

The JSTL tags can be classified, according to their functions, into following JSTL tag library groups that can be used when creating a JSP page:

- Core Tags
- Formatting tags
- SQL tags
- XML tags
- JSTL Functions

<mark>Chỗ này cần mô tả vắn tắt cách sử dụng tag lib thay cho code Java cùng một vài ví dụ</mark>

Xây dựng Tag Library

Phát triển ý tưởng sử dụng tag thay cho mã lệnh Java, JSP cung cấp giải pháp cho phép tự định nghĩa các tag để mã lệnh Java không xuất hiện trong các trang JSP phức tạp.

Gói API *javax.servlet.jsp.tagext.** cung cấp thư viện hàm cho phép xây dựng các Java class mà có thể thao tác trên các tag của trang JSP. Dựa trên thư viện này, người dùng có thể tự tạo ra các tag xử lý trang JSP theo cách riêng của mình.

Hello tag

Consider you want to define a custom tag named <ex:Hello> and you want to use it in the following fashion without a body:

```
<ex:Hello />
```

To create a custom JSP tag, you must first create a Java class that acts as a tag handler. So let us create HelloTag class as follows:

```java
package my_jsp;

import javax.servlet.jsp.tagext.*;
import javax.servlet.jsp.*;
import java.io.*;

public class HelloTag extends SimpleTagSupport {

  public void doTag() throws JspException, IOException {
    JspWriter out = getJspContext().getOut();
    out.println("Hello Custom Tag!");
  }
}
```

Above code has simple coding where doTag() method takes the current JspContext object using getJspContext() method and uses it to send "Hello Custom Tag!" to the current JspWriter object.

Let us compile above class and copy it in a directory available in environment variable CLASSPATH. Finally create following tag library file: <Tomcat-Installation-Directory>webapps\ROOT\WEB-INF\custom.tld.

```xml
<taglib>
 <tlib-version>1.0</tlib-version>
 <jsp-version>2.0</jsp-version>
 <short-name>Example TLD</short-name>
 <tag>
  <name>Hello</name>
  <tag-class>my_jsp.HelloTag</tag-class>
  <body-content>empty</body-content>
 </tag>
</taglib>
```

Try to call above JSP and this should produce following result:

```
Hello Custom Tag!
```

Accessing the Tag Body:

You can include a message in the body of the tag as you have seen with standard tags. Consider you want to define a custom tag named <ex:Hello> and you want to use it in the following fashion with a body:

```
<ex:Hello>
   This is message body
</ex:Hello>
```

Let us make following changes in above our tag code to process the body of the tag:

```
package com.tutorialspoint;

import javax.servlet.jsp.tagext.*;
import javax.servlet.jsp.*;
import java.io.*;

public class HelloTag extends SimpleTagSupport {

  StringWriter sw = new StringWriter();
  public void doTag()
    throws JspException, IOException
  {
    getJspBody().invoke(sw);
    getJspContext().getOut().println(sw.toString());
  }

}
```

In this case, the output resulting from the invocation is first captured into a StringWriter before being written to the JspWriter associated with the tag. Now accordingly we need to change TLD file as follows:

```
<taglib>
 <tlib-version>1.0</tlib-version>
 <jsp-version>2.0</jsp-version>
 <short-name>Example TLD with Body</short-name>
 <tag>
  <name>Hello</name>
  <tag-class>com.tutorialspoint.HelloTag</tag-class>
  <body-content>scriptless</body-content>
 </tag>
</taglib>
```

Now let us call above tag with proper body as follows:

```
<%@ taglib prefix="ex" uri="WEB-INF/custom.tld"%>
<html>
 <head>
  <title>A sample custom tag</title>
 </head>
 <body>
  <ex:Hello>
    This is message body
  </ex:Hello>
 </body>
</html>
```

This will produce following result:

```
This is message body
```

Custom Tag Attributes:

You can use various attributes along with your custom tags. To accept an attribute value, a custom tag class needs to implement setter methods, identical to JavaBean setter methods as shown below:

```
package com.tutorialspoint;

import javax.servlet.jsp.tagext.*;
import javax.servlet.jsp.*;
import java.io.*;

public class HelloTag extends SimpleTagSupport {

  private String message;

  public void setMessage(String msg) {
    this.message = msg;
  }

  StringWriter sw = new StringWriter();

  public void doTag()
    throws JspException, IOException
  {
    if (message != null) {
      /* Use message from attribute */
      JspWriter out = getJspContext().getOut();
      out.println( message );
    }
    else {
      /* use message from the body */
      getJspBody().invoke(sw);
      getJspContext().getOut().println(sw.toString());
    }
  }

}
```

The attribute's name is "message", so the setter method is setMessage(). Now let us add this attribute in TLD file using <attribute> element as follows:

```
<taglib>
 <tlib-version>1.0</tlib-version>
 <jsp-version>2.0</jsp-version>
 <short-name>Example TLD with Body</short-name>
 <tag>
  <name>Hello</name>
  <tag-class>com.tutorialspoint.HelloTag</tag-class>
  <body-content>scriptless</body-content>
  <attribute>
    <name>message</name>
  </attribute>
 </tag>
</taglib>
```

Now let us try following JSP with message attribute as follows:

```
<%@ taglib prefix="ex" uri="WEB-INF/custom.tld"%>
<html>
 <head>
  <title>A sample custom tag</title>
 </head>
 <body>
  <ex:Hello message="This is custom tag" />
 </body>
</html>
```

This will produce following result:

```
This is custom tag
```

Hope above example makes sense for you. It would be worth to note that you can include following properties for an attribute:

Property	Purpose
name	The name element defines the name of an attribute. Each attribute name must be unique for a particular tag.
required	This specifies if this attribute is required or optional. It would be false for optional.
rtexprvalue	Declares if a runtime expression value for a tag attribute is valid
type	Defines the Java class-type of this attribute. By default it is assumed as String
description	Informational description can be provided.
fragment	Declares if this attribute value should be treated as a JspFragment.

Following is the example to specify properties related to an attribute:

```
.....
  <attribute>
   <name>attribute_name</name>
   <required>false</required>
   <type>java.util.Date</type>
   <fragment>false</fragment>
  </attribute>
.....
```

If you are using two attributes then you can modify your TLD as follows:

```
.....
  <attribute>
   <name>attribute_name1</name>
   <required>false</required>
   <type>java.util.Boolean</type>
   <fragment>false</fragment>
```

```
    </attribute>
   <attribute>
    <name>attribute_name2</name>
    <required>true</required>
    <type>java.util.Date</type>
   </attribute>
.....
```

Chương 4. Struts - Java Web Framework

4.1 Đặt vấn đề

Cần có một đoạn phân tích nhược điểm của Servlet & JSP khi xây dựng các ứng dụng Web để dẫn dắt đến nhu cầu cần có các framework chuẩn.

(Layka, 2014) trang 159:

Why Use a Framework?

While Java EE does a great job of standardizing the enterprise infrastructure and providing an application model, there are few major problems associated with it.

- Interacting directly with the Java EE components often results in massive boilerplate code and even code redundancy.
- You have to write code for dealing with common business domain problems.
- You have to write code for solving architectural domain problems.

You could roll your own framework to address the problems associated with building Java EE based web applications using OO patterns and Java EE patterns. But writing an in-house framework entails efforts that are orthogonal to the business goals of the application; in addition, the in-house framework is unlikely to be upgraded, and the new versions of the in-house framework will never see the sun, unlike mainstream frameworks that continuously evolve instead of falling into architectural entropy. With that in mind, it's time to look at some of the available JVM-based web frameworks (see Table 3-25). This table is far from exhaustive; a myriad of frameworks are available, but this book will cover the most successful JVM-based web frameworks listed in the table.

(Layka, 2014) trang 158:

As developers gather more experience, they start discovering generic objects that can be used over and over, and patterns begin to emerge. Once you have a collection of such generic objects, a framework begins to emerge. A framework is a collection of generic objects and other supporting classes that provide the infrastructure for application development. Frameworks are, essentially, a collection of design patterns guarded by the basic framework principles discussed next. A Java framework uses two types of patterns.

- OO patterns
- Java EE patterns

A framework uses OO patterns for its own construction to address the architectural problem domain, such as extensibility, maintainability, reusability, performance, and scalability. Both OO patterns and Java EE patterns address the business problem domain areas, such as processing requests, authentication, validation, session management, and view management to name a few. .Frameworks address these two major architectural and business

problem areas by providing patterns based generic objects and supporting classes, closely guarded by the following key principles:

- *Configurability:* Configurability is the ability of the framework to be able to use the metadata to alter the behavior of the framework.
- *Inversion of control:* In traditional programming style, the problem domain code controls the flow of the application execution. Inversion of control refers to the technique where reusable code controls the execution of the problem domain code, thus controlling the flow of the application execution.
- *Loose coupling:* This principle refers to the independence of the collaborating classes in the framework with which each collaborating class can be altered without influencing the other collaborating class.
- *Separation of concerns:* This principle refers to the need to classify the problem domain areas and deal with them in an isolated manner so that the concerns of one problem area do not influence the concerns of another problem area. The multitiered Java EE architecture we saw in Chapter 1 is driven by the principle of the separation of concerns.
- *Automating common functionalities:* A framework provides mechanisms for automated solutions to the mundane functionalities of the domain.

4.2   Model – View – Control (MVC) Framework

(Layka, 2014) trang 86:

The motivation for the Model-View-Controller (MVC) pattern has been around since the conception of object-oriented programming. Prior to MVC, the browser directly accessed JSP pages. In other words, JSP pages handled user requests directly. This was called a Model-1 architecture, as illustrated in Figure 2-31. A Model-1 architecture exhibited decentralized application control, which led to a tightly coupled and brittle presentation tier.
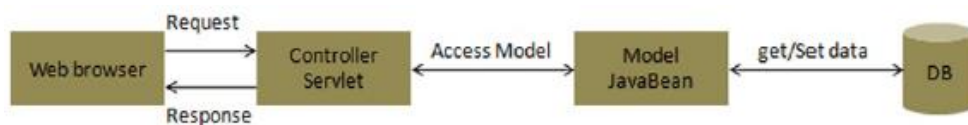
A Model-2 architecture for designing JSP pages is in actuality the MVC pattern applied to web applications. MVC originated in Smalltalk and has since made its way to Java community. Figure 2-32 shows the Model-2 (in other words, MVC) architecture. In Model-2, a controller handles the user request instead of another JSP page. The controller is implemented as a servlet. The following steps are executed when the user submits the request:

1. The controller servlet handles the user's request.
2. The controller servlet instantiates the appropriate JavaBeans based on the request.
3. The controller servlet communicates with the middle tier or directly to the database to retrieve the required data.
4. The controller sets the JavaBeans in one of the following contexts: request, session, or application.

5. The controller dispatches the request to the next view based on the request URL.
6. The view uses the JavaBeans from step 4 to display data.
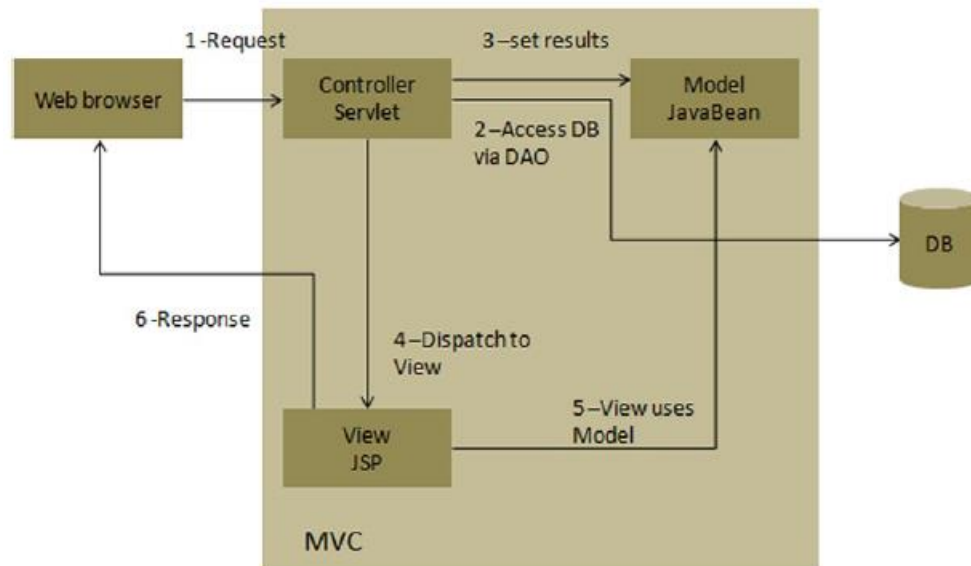
4.3 Cấu hình môi trường

- Tomcat: sử dụng Tomcat 7 hoặc Tomcat 8 đều được.
- Struts: Download the latest version of Struts2 binaries from http://struts.apache.org/download.cgi (các ví dụ chạy tốt với Struts 2.3.20)
- Copy các file lib cần thiết của Struts vào thư mục Lib của Tomcat:

commons-fileupload-x.y.z.jar
commons-io-x.y.z.jar
commons-lang-x.y.jar
commons-logging-x.y.z.jar
commons-logging-api-x.y.jar
freemarker-x.y.z.jar
javassist-.xy.z.GA
ognl-x.y.z.jar
struts2-core-x.y.z.jar
xwork-core.x.y.z.jar

4.4 Hello World với Struts

Eclipse: tạo project kiểu "Dynamic Web Project" như với JSP hay Servlet, tiếp theo là thiết lập các thành phần giao diện bằng JSP. Cuối cùng là kết nối các giao diện này theo logic xử lý của ứng dụng.

4.4.1 Xây dựng giao diện với JSP:

File giao diện đầu tiên là index.jsp gồm một form chứa text box để nhập tên người dùng vào tham số "name" và button "Say Hello" để gửi dữ liệu này lên server.

```
<%@ page contentType="text/html; charset=UTF-8" %>
<%@ taglib prefix="s" uri="/struts-tags" %>
```

```
<html>
    <head>
        <title>Name Collector</title>
    </head>
    <body>
        <h4>Enter your name </h4>
        <s:form action="hello">
            <s:textfield name="name" label="Your name"/>
            <s:submit/>
        </s:form>
    </body>
</html>
```

File giao diện thứ 2 là HelloWorld.jsp có chức năng nhận dữ liệu gửi lên từ index.jsp và hiển thị thông tin này.

```
<%@ page contentType="text/html; charset=UTF-8" %>
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
    <head>
        <title>HelloWorld</title>
    </head>
    <body>
        <h3>Custom Greeting Page</h3>
        <h4><s:property value="customGreeting"/></h4>
    </body>
</html>
```

Thêm một file JSP NoName.jsp để hiển thị thông tin trong trường hợp người dùng không nhập vào tên mà submit lên server luôn.

```
<%@ page contentType="text/html; charset=UTF-8" %>

<html>
  <head>
    <title>Hello World</title>
  </head>
  <body>
    Please input your name!
  </body>
</html>
```

4.4.2   *Kết nối các file giao diện theo logic của ứng dụng:*

Logic của ứng dụng rất đơn giản. Người sử dụng bắt đầu truy nhập ứng dụng với trang JSP index.jsp. Sau khi nhập tên mình vào text box và click buttton "Say Hello", trang JSP HelloWorld sẽ được hiển thị cùng với tên người sử dụng đã nhập vào. Trường hợp người dùng quên nhập vào tên mà submit ngay thì trang NoName.jsp sẽ được hiển thị.

Luồng kết nối các file giao diện JSP được Struts mô tả trong file cấu hình *struts.xml*. Mỗi ứng dụng web (ví dụ *my_struts*) cần khai báo một file *struts.xml* đặt trong thư mục WEB-IND/classes. Tạo file *struts.xml* cho ứng dụng *my_struts* với nội dung như sau:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts PUBLIC
  "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
  "http://struts.apache.org/dtds/struts-2.0.dtd">

<struts>
  <constant name="struts.devMode" value="true" />

  <package name="test" extends="struts-default">
    <action name="index">
      <result >/index.jsp</result>
    </action>
    <action name="hello"
        class="my_struts.HelloWorldAction"
        method="execute">
      <result name="success">/HelloWorld.jsp</result>
      <result name="no name">/NoName.jsp</result>
    </action>
  </package>
</struts>
```

Luồng kết nối giao diện bên trên mô tả 2 action "index" và "hello" tương ứng với 2 truy nhập tại các địa chỉ "http://localhost:8080/my_struts/index" và "http://localhost:8080/my_struts/hello". Với action "index", request sẽ được chuyển ngay đến trang giao diện *index.jsp* để xử lý và hiển thị kết quả. Với action "hello", request được chuyển cho một Java class (gọi là một Controller) my_struts.HelloWorldAction xử lý thông qua phương thức *execute()*. Tùy vào kết quả xử lý của phương thức này mà trang web hiển thị kết quả sẽ là HelloWorld.jsp hoặc NoName.jsp.

Struts được xây dựng dựa trên công nghệ Java Servlet nên ngoài file *struts.xml*, cần có file cấu hình *web.xml* đặt trong thư mục Web-INF. Struts cài đặt một filter đặc biệt có tên *org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter* có chức năng tiếp nhận tất cả các request gửi đến ứng dụng và dựa vào cấu hình luồng xử lý *struts.xml* mà chuyển tiếp request này đến các class xử lý tương ứng.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
  http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
  id="WebApp_ID" version="3.0">

  <display-name>Struts 2</display-name>

  <filter>
    <filter-name>struts2</filter-name>
    <filter-class>
      org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter
    </filter-class>
  </filter>

  <filter-mapping>
    <filter-name>struts2</filter-name>
```

```
    <url-pattern>/*</url-pattern>
  </filter-mapping>
</web-app>
```

4.4.3 Tạo Controller class HelloWorldAction

Controller class là một hoặc nhiều class Java cài đặt để xử lý toàn bộ phần logic của ứng dụng. Về bản chất, nó phân tích các thông tin và tạo ra một "bộ chuyển mạch" để phối hợp với luồng kết nối các file giao diện. Trong ứng dụng *HelloWorld*, Controller class đơn giản như sau.

```java
package my_struts;

public class HelloWorldAction{
        private String name;
        private String customGreeting;

        public String execute() throws Exception {
                setCustomGreeting( "Hi! my name is Struts Action Conrol... Hello " + getName() );
                if (name.length() > 0) return "success";
                return "no name";
        }

        public String getName() {
                return name;
        }
        public void setName(String name) {
                this.name = name;
        }

        public String getCustomGreeting() {
                return customGreeting;
        }
        public void setCustomGreeting(String customGreeting ) {
                this.customGreeting = customGreeting;
        }
}
```

Chạy Tomcat và truy nhập vào địa chỉ http://localhost:8080/my_struts/index với hai trường hợp (nhập đầy đủ tên hoặc không nhập tên) sẽ thấy struts điều phối các file giao diện JSP theo đúng thiết kế.

4.5 Bên trong Struts

(BROWN, DAVIS, & STANLICK, 2008) – trang 15

In this section, we'll detail processing a request within the framework. As you'll see, the framework has more than just its MVC components. We said that Struts 2 provides a cleaner implementation of MVC. These clean lines are only possible with the help of a few other key architectural components that participate in processing every request. Chief among these are the interceptors, OGNL, and the ValueStack. We'll learn what each of these does in

the following walkthrough of Struts 2 request processing. Figure 1.4 shows the request processing workflow.
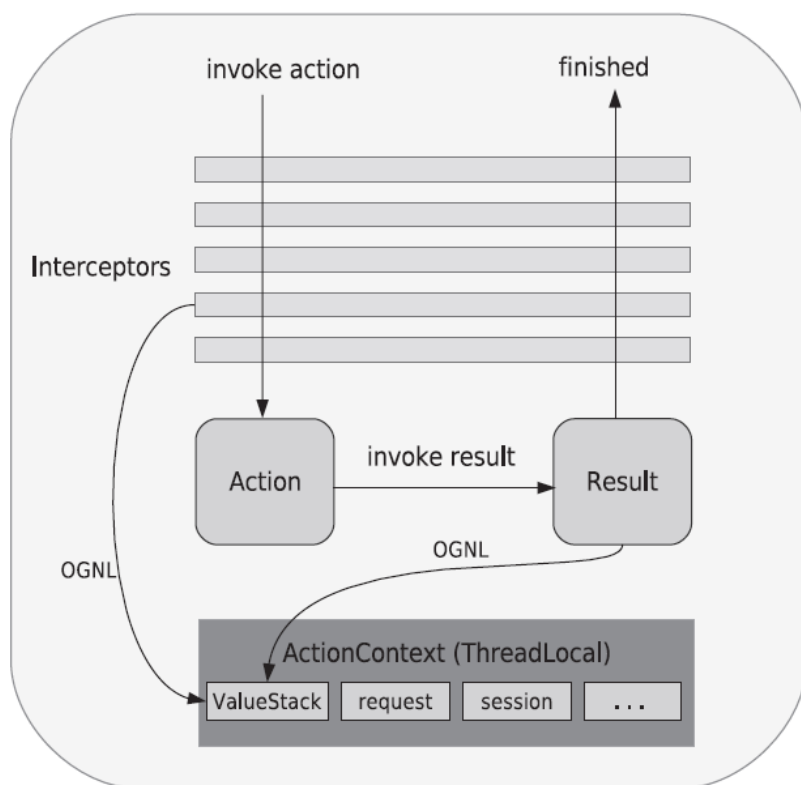
Figure 1.4 Struts 2 request processing uses interceptors that fire before and after the action and result.

The first thing we should consider is that the workflow of figure 1.4 still obeys the simpler MVC view of the framework that we saw earlier. In the figure, the FilterDispatcher has already done its controller work by selecting the appropriate action to handle the request. The figure demonstrates what really happens when the action is invoked by the controller. As you can see, a few extra parts are added to the MVC basics. We'll explain in the next paragraphs how the interceptors and the ActionContext aid the action and result in their processing of the request.

Figure 1.4 introduces the following new Struts 2 components: ActionContext, interceptors, the ValueStack, and OGNL. This diagram goes a long way toward showing what really happens in Struts 2. You could say that everything we'll discuss in this book is shown in this diagram. As interceptors come first in the request-processing cycle, we'll start with them. The name seems obvious, but what exactly do they intercept?

4.5.1 Interceptors

You may have noticed, while studying figure 1.4, that there is a stack of interceptors in front of the action. The invocation of the action must travel through this stack. This is a key part of the Struts 2 framework. We'll devote an entire chapter to this important component later in the book. At this time, it is enough to understand that most every action will have a stack of interceptors associated with it. These interceptors are invoked both before and after the action, though we should note that they actually fire after the result has executed. Interceptors don't necessarily have to do something both times they fire, but they do have the

opportunity. Some interceptors only do work before the action has been executed, and others only do work afterward. The important thing is that the interceptor allows common, cross-cutting tasks to be defined in clean, reusable components that you can keep separate from your action code.

What kinds of work should be done in interceptors? Logging is a good example. Logging should be done with the invocation of every action, but it probably shouldn't be put in the action itself. Why? Because it's not part of the action's own unit of work. It's more administrative, overhead if you will. Earlier, we charged a framework with the responsibility of providing built-in functional solutions to common domain tasks such as data validation, type conversion, and file uploads. Struts 2 uses interceptors to do this type of work. While these tasks are important, they're not specifically related to the action logic of the request. Struts 2 uses interceptors to both separate and reuse these cross-cutting concerns. Interceptors play a huge role in the Struts 2 framework. And while you probably won't spend a large percentage of your time writing interceptors, most developers will find that many tasks are perfectly solved with custom interceptors. As we said, we'll devote all of chapter 4 to exploring this core component.

4.5.2 The ValueStack & OGNL

While interceptors may not absorb a lot of your daily development energies, the ValueStack and OGNL will be constantly on your mind. In a nutshell, the ValueStack is a storage area that holds all of the data associated with the processing of a request. You could think of it as a piece of scratch paper where the framework does its work while solving the problems of request processing. Rather than passing the data around, Struts 2 keeps it in a convenient, central location—the ValueStack.

OGNL is the tool that allows us to access the data we put in that central repository. More specifically, it is an expression language that allows you to reference and manipulate the data on the ValueStack. Developers new to Struts 2 probably ask more questions about the ValueStack and OGNL than anything else. If you're coming from Struts 1, you'll find that these are a couple of the more exotic features of the new framework. Due to this, and the sheer importance of this duo, we'll treat them carefully throughout the book. In particular, chapters 5 and 6 describe the detailed function of these two framework components.

The tricky, and powerful, thing about the ValueStack and OGNL is that they don't belong to any of the individual framework components. Looking back to figure 1.4, note that both interceptors and results can use OGNL to target values on the ValueStack. The data in the ValueStack follows the request processing through all phases; it slices through the whole length of the framework. It can do this because it is stored in a ThreadLocal context called the ActionContext.

The ActionContext contains all of the data that makes up the context in which an action occurs. This includes the ValueStack but also includes stuff the framework itself will use internally, such as the request, session, and application maps from the Servlet API. You can access these objects yourself if you like; we'll see how later in the book. For now, we just
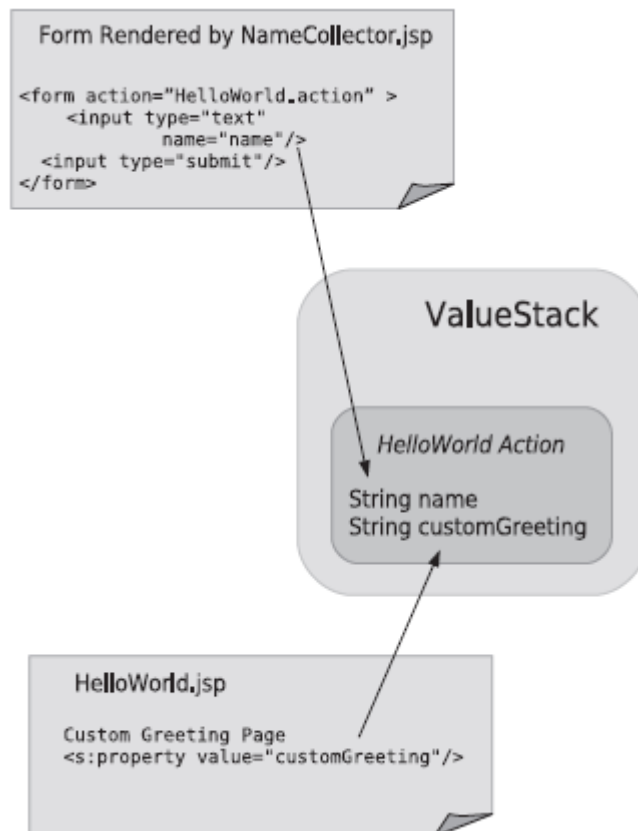
want to focus on the ActionContext as the ThreadLocal home of the ValueStack. The use of ThreadLocal makes the ActionContext, and thus the ValueStack, accessible from anywhere in the same thread of execution.

Since Struts 2's processing of each request occurs in a single thread, the ValueStack is available from any point in the framework's handling of a request. Typically, it is considered bad form to obtain the contents of the ActionContext yourself. The framework provides many elegant ways to interact with that data without actually touching the ActionContext, or the ValueStack, yourself. Primarily, you'll use OGNL to do this. OGNL is used in many places in the framework to reference and manipulate data in the ValueStack. For instance, you'll use OGNL to bind HTML form fields to data objects on the ValueStack for data transfer, and you'll use OGNL to pull data into the rendering of your JSPs and other result types. At this point, you just need to understand that the ValueStack is where your data is stored while you work with it, and that OGNL is the expression language that you, and the framework, use to target this data from various parts of the request-processing cycle.

4.5.3 Hello World làm việc thế nào

(BROWN, DAVIS, & STANLICK, 2008) – trang 35

You might still have questions about how the data gets from the front to the back of this process. Let's trace the path of data as it comes into, flows through, and ultimately exits the HelloWorld application. First, let's clear up some potential confusion regarding the location of data in the framework. In chapter 1, we learned that the framework provides something called the `ValueStack` for storing all of the domain data during the processing of a request. We also said that the framework uses a powerful expression language, OGNL, to reference and manipulate that data from various regions of the framework. But, as we've just learned, the action itself holds the domain data. In the case of the HelloWorld action, that data is held on JavaBeans properties exposed on the action itself. So, what gives? In short, both are true. The data is both stored in the action and in the `ValueStack`. Here's how. First, domain data is always stored in the action. We'll see variants on this, but it's essentially true. This is great because it allows convenient access to data from the action's `execute()` method. So that the rest of the framework can access the data, the action object itself is placed on the `ValueStack`.

```
Form Rendered by NameCollector.jsp

<form action="HelloWorld.action" >
    <input type="text"
            name="name"/>
  <input type="submit"/>
</form>
```

ValueStack

HelloWorld Action

String name
String customGreeting

```
HelloWorld.jsp

Custom Greeting Page
<s:property value="customGreeting"/>
```

The mechanics of the `ValueStack` are such that all properties of the action will then be exposed as toplevel properties of the `ValueStack` itself and, thus, accessible via OGNL. Figure 2.7 demonstrates how this works with the `Hello-World` action as an example. As figure 2.7 shows, the action holds the data, giving its own Java code convenient access. At the same time, the framework makes the properties of the action available on the `ValueStack` so that other regions of the framework can access the data as well. In terms of our HelloWorld application, thetwo most important places this occurs are on the incoming form and the outgoing result page. In the case of the incoming request, the form field name attribute is interpreted as an OGNL expression. The expression is used to target a property on the ValueStack; in this case, the name property from our action. The value from the form field is automatically moved onto that property by the framework. On the other end, the result JSP pulls data off the customGreeting property by likewise using an OGNL expression, inside a tag, to reference a property on the ValueStack. Obviously, this complicated process needs more than a quick sketch. We'll cover it fully, particularly in chapters 5 and 6.

That gives us as much as we need to know at this point. We've seen how to declare actions and results. We've also learned a bit about how the data moves through the framework. You might've noticed that we didn't declare any interceptors. Despite the importance of interceptors, the HelloWorld application declares none of them. It avoids declaring interceptors itself by using the default interceptor stack provided by the framework. This is common practice.

4.6 Làm việc với Structs action

(BROWN, DAVIS, & STANLICK, 2008) – trang 44

4.6.1 What does an action do?

Actions do three things. First, as you probably understand by now, an action's most important role, from the perspective of the framework's architecture, is encapsulating the actual work to be done for a given request. The second major role is to serve as a data carrier in the framework's automatic transfer of data from the request to the view. Finally, the action must assist the framework in determining which result should render the view that'll be returned in the request response. Let's see how the action component fulfills each of these various roles.

By the way, we're going to demonstrate our points in the coming paragraphs with examples from the HelloWorld application from chapter 2. But don't worry; we'll start building the real-world Struts 2 Portfolio in a few pages.

4.6.2 Actions encapsulate the Unit Of Work

Earlier in this book, we saw that the action fulfills the role of the MVC model for the framework. One of the central responsibilities of this role is the containment of business logic; actions use the `execute()` method for this purpose. The code inside this method should concern itself only with the logic of the work associated with the request. The following code snippet, from the previous chapter's HelloWorld application, shows the work done by the `HelloWorldAction`.

```
public String execute() {
setCustomGreeting( GREETING + getName() );
return "SUCCESS";
}
```

The action's work is to build a customized greeting for the user. As we can see, this action's `execute()` method does little else than build this greeting. In this case, the business logic amounts to little more than a concatenation. If it were much more complex, we'd probably have bumped that logic out to a business component and injected that component into the action. The use of dependency injection, which helps keep code such as actions clean and decoupled, is supported by the framework. We'll learn some techniques that utilize the framework's Spring integration for injecting these components later in the book. For now, just keep in mind that our actions hold the business logic, or at least the entry point to the business logic, and they should keep that logic as pure and brief as possible.

4.6.3 Actions provide locus for data transfer

Being the model component of the framework also means that the action is expected to carry the data around. While you might think this would make actions more complicated, it actually makes them cleaner. Since the data is held local to the action, it's always conveniently available during the execution of the business logic. There might be a bunch of JavaBeans properties adding lines of code to the action, but when the `execute()` method references the data in those properties, the proximity of the data makes that code all the more succinct.

Listing 3.1, also from `HelloWorldAction`, shows the code that allows that action to carry request data.

```
private String name;
public String getName() {
return name;
}
public void setName(String name) {
this.name = name;
}
private String customGreeting;
public String getCustomGreeting()
{
return customGreeting;
}
public void setCustomGreeting( String customGreeting ){
this.customGreeting = customGreeting;
}
```

The action merely implements JavaBeans properties for each piece of data that it wishes to carry. We saw this in action with the HelloWorld application. Request parameters from the form are moved to properties that have matching names. As we saw, the framework does this automatically. In this case, the `name` parameter from the name collection form will be set on the `name` property. In addition to receiving the incoming data from the request, these JavaBeans properties on the action will also expose the data to the result. The `HelloWorld` action's logic sets the custom greeting on the `customGreeting` property, which makes it available to the result as well.

In addition to these simple JavaBeans properties, there are a couple of other techniques for using the action as a data transfer object. We'll examine these alternatives later in this chapter, and will also examine the mechanisms by which the actual data transfer occurs. For now, we just want to recognize that the action serves as a centralized data transfer object that can be used to make the application data available in all tiers of the framework.

The use of actions as data transfer objects should probably ring some alarms in the minds of alert Struts 1 developers. In Struts 1, there's only one instance of a given action class. If this were still true, we couldn't use the action object itself as a data carrier for the request. In a multithreaded environment, such as a web application, it'd be problematic to store data in instance fields as we've seen. Struts 2 solves this problem by creating a new instance of an action for each request that maps to it. This fundamental difference allows Struts 2 objects to exist as dedicated data transfer objects for each request.

4.6.4   Actions return control string for result routing

The final duty of an action component is to return a control string that selects the result that should be rendered. Previous frameworks passed routing objects into the entry method of the action. Returning a control string eliminates the need for these objects, resulting in a cleaner signature and an action that is less coupled to specific routing code. The value of the return string must match the name of the desired result as configured in the

declarative architecture. For instance, the `HelloWorldAction` returns the string `"SUCCESS"`. As you can see from our XML declaration, `SUCCESS` is the name of the one of the result components.

```
<action name="HelloWorld" class="manning.chapterOne.HelloWorld">
<result name="SUCCESS">/chapterTwo/HelloWorld.jsp</result>
<result name="ERROR">/chapterTwo/Error.jsp</result>
</action>
```

The HelloWorld application has a simple logic for determining which result it will choose. In fact, it'll always choose the `"SUCCESS"` result. Most real-world actions will have a more complex determination process, and the result choices will almost always include some sort of error result to handle problems that might occur during the action's interaction with the model. Regardless of the complexity, actions must ultimately return a string that maps to one of the result components available for rendering the view for that action.

You should now realize what an action does, but before we design one, we need to create the packages to contain them. In the next section, we'll see how to organize our actions into packages and take our first glimpse at the Struts 2 Portfolio application, the main sample application for this book.

4.7   Xử lý giao diện: UI Tags & Results

(BROWN, DAVIS, & STANLICK, 2008) – trang 130

In part 2, we learned how the core of the framework processes each request. In particular, we learned how to write actions that contain the logic for each request, wrap that action logic with a stack of the appropriate interceptors, and take advantage of the framework's powerful data transfer and type conversion mechanisms. Though we've been using JSP pages to render simple views for our actions, we haven't gone into any of the details of the view layer. We now enter the part of the book that focuses on the view layer.

In Struts 2, the MVC view concerns are encapsulated in the result component. We've already become somewhat familiar with the result component even though we've said nothing regarding its details. In fact, actually developing results will be at least as rare as developing your own interceptors. Most of your development work will amount to little more than using built-in result types to hit JSP pages and Velocity templates. The built-in result types will handle the most common view-layer technologies, so you'll probably go through a lot of code before you find yourself writing your own. Nonetheless, we'll provide a detailed account of working directly with results in chapter 8. For now, we're going to focus on the Struts 2 tag libraries because, while secondary to the result component itself, they're the tools you'll have in hand for most of your development efforts. The most common view-layer technologies are probably JSP, Velocity, and FreeMarker. While each of these has its own tags or macros, the Struts 2 framework provides a high-level tag API that you can use on all three rendering platforms. In addition to being a portable tag API, these Struts 2 tags bring a lot to the table functionally. You'll find all of the features of any recently minted tag set, and more.

The Struts 2 tag libraries are divided into two groups: general-purpose tags and UI component tags. We'll start, in chapter 6, with the general-purpose tags. These tags provide all sorts of things from conditional logic to ValueStack manipulation to i18n help. We'll show you how the tags work, including an overview of their syntax and using OGNL to reference values on the ValueStack. We'll even include a primer on the most important parts of the OGNL expression language as used in the context of tags. The UI component tags, which we'll introduce in chapter 7, are perhaps the most impressive part of the Struts 2 tags. These tags not only generate HTML form fields, but wrap those fields in layers of functionality that tap into all of the framework's various features. And if you need to customize the HTML output of a given UI-oriented tag, such as a form tag, you can change its HTML source template and thus change the way it renders across all uses, enabling reusable customization. The full richness of the UI components can't be captured in a couple of sentences, but we think you'll find them alluring after reading about them.

4.7.1   Một số cấu trúc bên trong của Struts liên quan đến View

Before we talk about the details of how Struts 2 tags can help you dynamically pipe data into the rendering of your pages, let's talk about where that data comes from. While we focused on the data moving into the framework in the previous chapter, we'll now focus on the data leaving the framework. When a request hits the framework, one of the first things Struts 2 does is create the objects that'll store all the important data for the request. If we're talking about your application's domain-specific data, which is the data that you'll most frequently access with your tags, it'll be stored in the ValueStack. But processing a request requires more than just your application's domain data. Other, more infrastructural, data must be stored also. All of this data, along with the ValueStack itself, is stored in something called the ActionContext.

4.7.1.1 The ActionContext and OGNL

In the previous chapter, we used OGNL expressions to bind form field names to specific property locations on objects such as our action object. We already know that our action object is placed on something called the ValueStack and that the OGNL expressions target properties on that stack object. In reality, OGNL expressions can resolve against any of a set of objects. The ValueStack is just one of these objects, the default one. This wider set of objects against which OGNL expressions can choose to resolve is called the ActionContext. We'll now see how OGNL chooses which object to resolve against, as well as what other objects are available for accessing with OGNL.

The ActionContext contains all of the data available to the framework's processing of the request, including things ranging from application data to session- or application-scoped maps. All of your applicationspecific data, such as properties exposed on your action, will be held in the ValueStack, one of the objects in the ActionContext. All OGNL expressions must resolve against one of the objects contained in the ActionContext. By default, the ValueStack will be the one chosen for OGNL resolution, but you can specifically name one of the others, such as the session map, if you like.

The `ActionContext` is a key behind-the-scenes player in the Struts 2 framework. If you've worked with other web application frameworks, particularly Struts 1, then you might be asking, "Why do I need an `ActionContext`? Why have you made my life more complicated?" We've been trying to emphasize that the Struts 2 framework strives toward a clean MVC implementation. The `ActionContext` helps clean things up by providing the notion of a context for the execution of an action. By *context* we mean a simple container for all the important data and resources that surround the execution of a given action. A good example of the type of data we're talking about is the map of parameters from the request, or a map of session attributes from the Servlet Container. In Struts 1, most of these resources were accessed via the Servlet stuff handed into the execution of every action. We've already seen how clean the Struts 2 action object has become; it has no parameters in its method signature to tie it to any APIs that might have little to do with its task. So, really, your life is much less complicated, though at first it might not seem so.

Before we show you all of the specific things that the `ActionContext` holds, we need to discuss OGNL integration. As we've seen, OGNL expressions target properties on specific objects. The resolution of each OGNL expression requires a root object against which resolution of references will begin. Consider the following OGNL expression:

`user.account.balance`

Here we're targeting the `balance` property on the `account` object on the `user` object. But where's the `user` object located? We must define an initial object upon which we'll locate the `user` object itself. Every time you use an OGNL expression, you need to indicate which object the expression should start its resolution against. In Struts 2, each OGNL expression must choose its initial object from those contained in the `Action-Context`. Figure 6.1 shows the `ActionContext` and the objects it contains, any of which you can point your OGNL resolution toward.

As you can see, the `ActionContext` is full of juicy treasures. The most important of these is the `ValueStack`. As we've said, the `ValueStack` holds your application's

domain-specific data for a given action invocation. For instance, if you're updating a

student, you'll expect to find that student data on the `ValueStack`. We'll divulge more of the inner workings of the `ValueStack` in a moment. The other objects are all maps of important sets of data. Each of them has a name that indicates its purpose and should be familiar to seasoned Java web application developers, as they correspond to specific concepts from the Servlet API. For more information on where the data in these sets comes from, we recommend the Java Servlet Specification. The contents of

each of these objects is summarized in table 6.1.
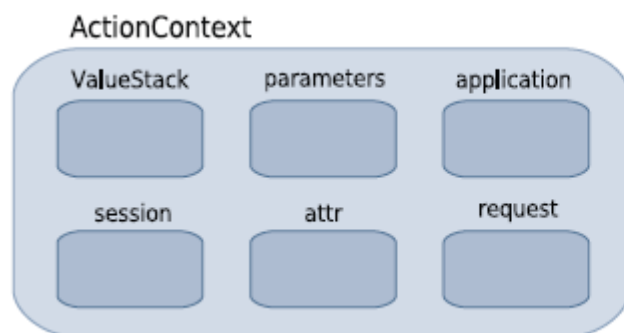
Figure 6.1 The `ActionContext` holds all the important data objects pertaining to a given action invocation; OGNL can target any of them.

Table 6.1   The names and contents of the objects and maps in the `ActionContext`

Name	Description
parameters	Map of request parameters for this request
request	Map of request-scoped attributes
session	Map of session-scoped attributes
application	Map of application-scoped attributes
attr	Returns first occurrence of attribute occurring in page, request, session, or application scope, in that order
ValueStack	Contains all the application-domain–specific data for the request

The `parameters` object is a map of the request parameters associated with the request being processed—the parameters submitted by the form, in other words. The `application` object is a map of the application attributes. The `request` and `session` objects are also maps of request and session attributes. By *attribute* we mean the Servlet API concept of an attribute. Attributes allow you to store arbitrary objects, associated with a name, in these respective scopes. Objects stored in application scope are accessible to all requests coming to the application. Objects stored in session scope are accessible to all requests of a particular session, and so forth. Common usage includes storing a user object in the session to indicate a logged-in user across multiple requests. The `attr` object is a special map that looks for attributes in the following locations, in sequence: page, request, session, and application scope. Now let's look at how we choose which object from the `ActionContext` our OGNL will resolve against.

4.7.1.2   The ValueStack: a virtual object

Back to that default root object of the `ActionContext`. Understanding the `ValueStack` is critical to understanding the way data moves through the Struts 2 framework. By now, you've got most of what you need. We've seen the `ValueStack` in action. When

Struts 2 receives a request, it immediately creates an `ActionContext`, a `ValueStack`

and an action object. As a carrier of application data, the action object is quickly placed on the `ValueStack` so that its properties will be accessible, via OGNL, to the far reaches of the framework.
First, these will receive the automatic data transfer from the incoming request parameters. As we saw in chapter 4, this occurs because the `params` interceptor sets those parameters on properties exposed on the `ValueStack`, upon which the action object sits. While other things, such as the model of the `ModelDriven` interface, may also be placed on the stack, what all data on the `ValueStack` has in common is that it's all specific to the application's domain. In MVC terms, the `ValueStack` is the request's view of the application's model data. There are no infrastructural objects, such as Servlet API or Struts 2 objects, on the `ValueStack`. The action is only there because of its role as domain data carrier; it's not there because of its action logic.
There's only one tricky bit about the `ValueStack`. The `ValueStack` pretends to be a single object when OGNL expressions are resolved against it. This virtual object contains all the properties of all the objects that've been placed on the stack. If multiple occurrences of the same property exist, those lowest down in the stack are hidden by the uppermost occurrence of a similarly named property. Figure 6.2 shows a `ValueStack` with several objects on it.
As you can see in figure 6.2, references to a given property resolve to the highest occurrence of that property in the stack. While this may seem complicated, it's actually not. As with most Struts 2 features, the flexibility and power to address complex use cases is there, but the common user can remain ignorant of such details.
Let's examine figure 6.2. As usual, the action object itself has been placed on the stack first. Then, a model object was added to the stack. This most likely has occurred

because the action implements `ModelDriven`. Sometime after that, another object, apparently some sort of random-number-making bean, was added to the stack. By *bean*

we simply mean a Java object that either serves as a data carrier or as a utility-providing
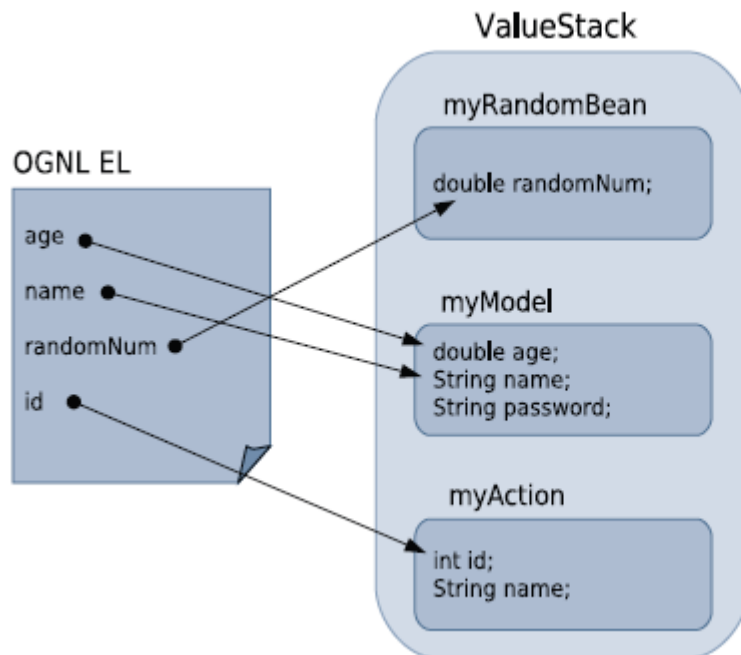
Figure 6.2 The `ValueStack` is the default object against which all OGNL expressions are resolved.

object. In other words, it's usually just some object whose properties you might want to access from your tags with OGNL expressions.

At present, we just want to see how the `ValueStack` appears as a single virtual object to the OGNL expressions resolving against it. In figure 6.2, we have four simple expressions. Each targets a top-level property. Behind the scenes, OGNL will resolve each of these expressions by asking the `ValueStack` if it has a property such as, for instance, `name`. The `ValueStack` will always return the highest-level occurrence of the `name` property in its stack of objects. In this case, the action object has a `name` property, but that property will never be accessed as long as the model object's `name` property

sits on top of it.

Just so you don't worry about it, we might as well discuss how that bean showed up on top of the stack. Prior to this point, we've just had stuff automatically placed on the `ValueStack` by the framework. So how did the bean get there? There are many ways to add a bean to the stack. Many of the most common ways occur within the tags that we'll soon cover. For instance, the `push` tag lets you push any bean you like onto the stack. You might do such a thing just before you wanted to reference that bean's data or methods from later tags. We'll demonstrate this with sample code when we cover those tags. With a clear view of where the data is and how to get to it, it's time to get back to the Struts 2 tags that are the focus of this chapter, and are the means of pulling data from the `ActionContext` and `ValueStack` into the dynamic rendering of your view

layer pages.

4.7.2   Struts UI tags

The Struts 2 tag API provides the functionality to dynamically create robust web pages by leveraging conditional rendering and integration of data from your application's domain model found on the `ValueStack`. Struts 2 comes with many different types of

Definition

When Java developers talk about beans in the context of view technologies, such as JSPs, they frequently mean something different than just a Java object that meets the JavaBeans standard. While these beans are most likely good JavaBeans as well, they don't have to be. The usage in this context more directly refers to the fact that the bean is a Java object that exposes data and/or utility methods for use in JSP tags

and the like. Many developers call any object exposed like this a "bean."
This nomenclature is a historical artifact. In the past, expression languages used in
tags couldn't call methods. Thus, they could only retrieve data from an object if it
were exposed as a JavaBean property. Since the OGNL expression language allows
you to call methods directly, you could completely ignore JavaBeans conventions and
still have data and utility methods exposed to your tags for use while rendering the
page. However, in order to keep your JSP pages free of complexity, we strongly recommend
following JavaBeans conventions and avoiding expression language method
invocation as long as possible.

138 CHAPTER 6 *Building a view: tags*

tags. For organizational purposes, they can be broken into four categories: *data tags*,
control-flow tags, *UI tags*, and *miscellaneous tags*. Since they are a complex topic all to
themselves, we'll leave the UI tags for chapter 7. This chapter examines the other
three categories.

Data tags focus on ways to extract data from the `ValueStack` and/or set values in
the `ValueStack`. Control-flow tags give you the tools to conditionally alter the flow of
the rendering process; they can even test values on the `ValueStack`. The miscellaneous
tags are a set of tags that don't quite fit into the other categories. These leftover
tags include such useful functionality as managing URL rendering and internationalization
of text. Before we get started, we need to make some general remarks about

the conventions that are applied across the usage of all Struts 2 tag APIs.

4.7.3   Hiển thị Result lên View

This chapter wraps up part 3. If you recall from the early chapters, the *result* is the
MVC view component of the Struts 2 framework. As the central figure of the view, you
might be wondering why we started with two chapters on the Struts 2 tag API and left
the result for last. Easy. For common development practice, you don't need to know
much about the result component itself. In fact, if you use JSP pages for your results,
you don't even need to know that the framework supports many different types of
results, because the default result type supports JSPs. But the framework comes with
support for many different kinds of results, and you can write your own results as
well. This chapter explores the details of this important Struts 2 component.

In order to make sure you know what they are and how they work, we start by
building a custom result that demonstrates a technique for developing Ajax applications
on the Struts 2 platform. Seeing how the framework can easily be adapted
to return nontraditional results, such as those required by Ajax clients, serves as a

perfect demonstration of the flexibility of the result component. It both teaches

you the internals of results and gives you an example of Struts 2 Ajax development.
After that, we go on to tour the built-in results that the framework provides for your
convenience. These include the default result that supports JSP pages, as well as alternative
page-rendering options such as Velocity or FreeMarker templates.

Let's start by refreshing ourselves on the Struts 2 architecture and the role played

by results in that architecture.

In a classic web application, these view concerns are generally equivalent to creating
an HTML page that's sent back to the client. The intelligent defaults of the framework
are in perfect tune with this usage. By default, the framework uses a result type that
works with JSPs to render these response pages. This result, the `dispatcher` result,
makes all the `ValueStack` data available to the executing JSP page. With access to that
data, the JSP can render a dynamic HTML page.

Thanks to the intelligent defaults, we've been happily using JSPs from our earliest
HelloWorld example, all the while oblivious to the existence of a variety of result
types. The following snippet shows how easy it is to use JSPs under the default settings
of the framework:

```
<action name="PortfolioHomePage" class=". . . PortfolioHomePage">
<result>/chapterEight/PortfolioHomePage.jsp</result>
</action>
```

DEFINITION

As the snippet demonstrates, you can get your JSPs up and running without knowing anything about what a result is. All you need to know is that a result is the element into which you stuff the location of your JSP page. And that's about all you need to know as long as your project stays the JSP course.

But what if you want to use Velocity or FreeMarker templates to render your HTML pages? Or what if you want to redirect to another URL rather than rendering a page for the client? These alternatives, which also follow the general request and response patterns of a classic web application, are completely supported by the built-in result types that come with the framework. Starting in section 8.2, we'll provide a tour of these commonly used results. If all you want to do is switch from JSPs to FreeMarker or Velocity, or redirect to another URL instead of rendering the HTML page yourself, you can skip ahead to the reference portion of this chapter.

If you want, however, to see how you can adapt results to nonstandard patterns of

usage, such as the nonclassic patterns of Ajax applications, then stick around.
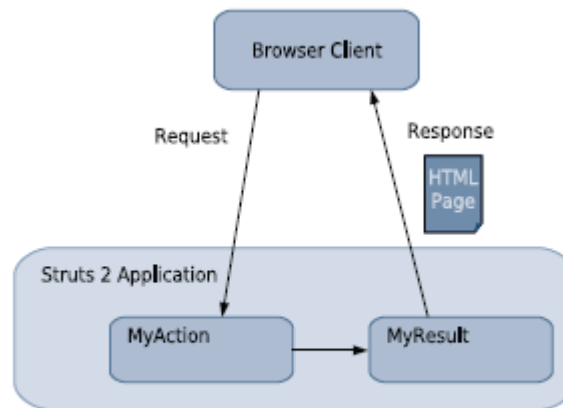
**Figure 8.1   Classic web applications return full HTML page responses to the client.**

As we mentioned, a classic web application returns full HTML page responses to the client. Figure 8.1 illustrates this pattern.

In figure 8.1, the client makes a request that maps to some action. This action, most likely, takes some piece of request data, conducts some business logic, then exposes the subsequent domain data on the `ValueStack`. The action then passes control to a result that renders a full HTML page, using the prepared data, to build the new HTML page. The key thing here is that the response is a full HTML page, which the client browser uses to rerender its entire window. The response sent back to the client in figure 8.1 is probably rendered by a JSP under the default dispatcher result type. As we've seen, the framework makes this classic

pattern of usage easy.

On the other hand, Ajax applications do something entirely different. Instead of requesting full HTML pages, they only want data. This data can come in many forms. Some Ajax applications want HTML fragments as their responses. Some want XML or JSON responses. In short, the content of an Ajax response can be in a variety of formats. Regardless

of their differences, they do share one
distinct commonality: none of them want
a full HTML page. Figure 8.2 illustrates a

typical Ajax request and response cycle.
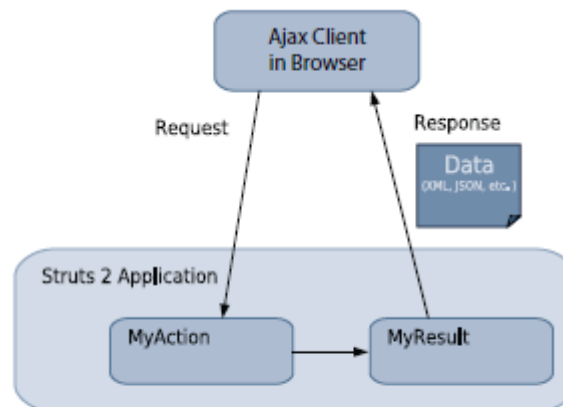
Figure 8.2 Ajax applications expect only data, such as JSON or XML, in the response.

When the Ajax client receives the response, it won't cause the browser to rerender
the entire HTML page. On the contrary, it carefully examines the data serialized in the
XML or JSON and uses that data to make targeted updates to the affected regions of
the current browser page. This is a different kind of response. Luckily, Struts 2 can easily handle this with
the flexibility of its result component.

4.8 Làm việc với Interceptors

(BROWN, DAVIS, & STANLICK, 2008) – trang 75

4.8.1 Why intercept requests?

Earlier in this book, we described Struts 2 as a second-generation MVC framework. We
said that this new framework leveraged the lessons learned by the first generation of MVC-
based frameworks to implement a super-clean architecture. Interceptors play a crucial role in
allowing the framework to achieve such a high level of separation of concerns. In this section,
we'll take a closer look at how interceptors provide a powerful tool for encapsulating the
kinds of tasks that have traditionally been an architectural thorn in the developer's side.

Cleaning up the MVC

From an architectural point of view, interceptors have immensely improved the level
of separation we can achieve when trying to isolate the various concerns of our web
applications. In particular, interceptors remove cross-cutting tasks from our action
components. When we try to describe the kinds of tasks that interceptors implement, we
usually say something like cross-cutting, or preprocessing and postprocessing. These terms
may sound vague now, but they won't by the time we finish this chapter.

Logging is a typical cross-cutting concern. In the past, you might've had a logging
statement in each of your actions. While this seemed a natural place for placing a logging
statement, it's not a part of the action's interaction with the model. In reality, logging is
administrative stuff that we want done for every request that the system processes. We call

this *cross-cutting* because it's not specific to a single action. It cuts across a whole range of actions. As software engineers, we should instantly see this as an opportunity to raise the task to a higher layer that can sit above, or in front of, any number of requests that require logging. The bottom line is that we have the opportunity to remove the logging from the action, thus creating cleaner separation of our MVC concerns.

Some of the tasks undertaken by interceptors are more easily understood as being preprocessing or postprocessing tasks. These are still technically cross-cutting; we recommend not worrying about the semantics of these terms. We present these new terms mostly to give you some ideas about the specific types of tasks handled by interceptors.

A good example of a preprocessing task would be data transfer, which we're already familiar with. This task is achieved with the `params` interceptor. Nearly every action will need to have some data transferred from the request parameters onto its domain-specific properties. This must be done before the action fires, and can be seen as mere preparation for the actual work of the action. From this aloof perspective, we can call it a preprocessing task. This is perfect for an interceptor. Again, this increases the purity of the action component by removing code that can't be strictly seen as part of a specific action's core work.

No matter whether we call the task cross-cutting or preprocessing, the conceptual mechanics of interceptors are clear. Instead of having a simple controller invoking an action directly, we now have a component that sits between the controller and the action. In Struts 2, no action is invoked in isolation. The invocation of an action is a layered process that always includes the execution of a stack of interceptors prior to and after the actual execution of the action itself. Rather than invoke the action's `execute()` method directly, the framework creates an object called an `ActionInvocation` that encapsulates the action and all of the interceptors that have been configured to fire before and after that action executes. Figure 4.1 illustrates the encapsulation of the entire action execution process in the `ActionInvocation` class.
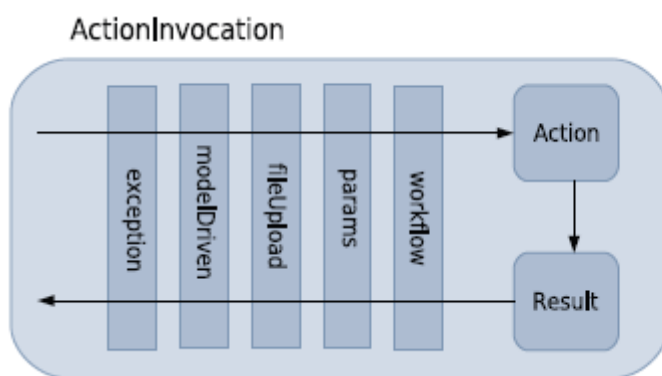
Figure 4.1 `ActionInvocation` encapsulates the execution of an action with its associated interceptors and results.

As you can see in figure 4.1, the invocation of an action must first travel through the stack of interceptors associated with that action. Here we've presented a simplified version of the `defaultStack`. The `defaultStack` includes such tasks as file uploading and transferring request parameters onto our action. Figure 4.1 represents the normal workflow; none of the interceptors have diverted the invocation. This action will ultimately execute and return a control string that selects the appropriate result. After the result executes, each of the

interceptors, in reverse order, gets a chance to do some postprocessing work. As we'll see, the interceptors have access to the action and other contextual values. This allows them to be aware of what's happening in the processing.

For instance, they can examine the control string returned from the action to see what result was chosen. One of the powerful functional aspects of interceptors is their ability to alter the workflow of the invocation. As we noted, figure 4.1 depicts an instance where none of the interceptors has intervened in the workflow, thus allowing the action to execute and determine which result should render the view. Sometimes, one of the interceptors will determine that the action shouldn't execute. In these cases, the interceptor can halt the workflow by itself returning a control string. Take the `workflow` interceptor, for example. As we've seen, this interceptor does two things. First, it invokes the `validate()` method on the action, if the action has implemented the `Validateable` interface. Next, it checks for the presence of error messages on the action. If errors are present, it returns a control string and, thus, stops further execution. The action will never fire. The next interceptor in the stack won't even be invoked. By returning the control string itself, the interceptor causes control to return back up the chain, giving each interceptor above the chance to do some postprocessing. Finally, the result that matches the returned control string will render the view. In the case of the `workflow` interceptor that has found error messages on the action, the control string is `"input"`, which typically maps back to the form page that submitted the invalid data.

As you might suspect, the details of this invocation process are thorny. In fact, they involve a bit of recursion. As with all recursion, it'll seem harmless once we look at the details, which we'll see shortly. But first we need to talk about the benefits we gain from using interceptors.

Reaping the benefits

Layering always makes our software cleaner, which helps with readability and testing and also provides flexibility. Once we've broken these cross-cutting, preprocessing, and postprocessing tasks into manageable units, we can do cool stuff with them. The two primary benefits we gain from this flexibility are reuse and configuration.

Everyone wants to reuse software. Perhaps this is the number-one goal of all software engineering. Reuse is a bottom-line issue from both business and engineering perspectives. Reuse means saving time, money, and maintainability. It makes everyone happy. And achieving it is simple. We just need to isolate the logic that we want to reuse in a cleanly separated unit. Once we've isolated the logic in an interceptor, we can drop it in anywhere we like, easily applying it to whole classes of actions. This is more exciting than clean architectural lines, but really it's the same thing. We've already been benefiting from code reuse by inheriting the `defaultStack`. Using the `defaultStack` allows us to reuse the data transfer code written by the Struts 2 developers, along with their validation code, their internationalization code, and so forth. In addition to the benefits of code reuse, the layering power of interceptors gives us another important benefit. Once we have these tasks cleanly encapsulated in interceptors, we can, in addition to reusing them, easily reconfigure their order and number.

While the `defaultStack` provides a common set of interceptors, arranged in a common sequence, to serve the common functional needs of most requests, we can rearrange them to meet varying requirements. We can even remove and add interceptors as we like. We can even do this on a per-action basis, but this is seldom necessary. In our Struts 2 Portfolio application, we'll develop an authentication interceptor and combine it with the `defaultStack` of interceptors that fires when the actions in our secure package are invoked. The flexible nature of interceptors allows us to easily customize request processing for the specific needs of certain requests, all while still taking advantage of code reuse.

Struts 2 is extremely flexible. This strength is what separates it from many of its competitors. But, as we've mentioned, this can also be confusing when you first begin to use the framework. Thankfully, Struts 2 provides a strong set of intelligent defaults that allow developers to build most standard functionality without needing to think about the many ways in which they can modify the framework and its core components. In the case of interceptors, one of the framework's most flexible components, the `defaultStack` should serve in the vast majority of cases.

Developing interceptors

Despite their importance, many developers won't write many interceptors. In fact, most of the common tasks of the web application domain have already been written and bundled into the `struts-default` package. Even if you never write an interceptor yourself, it's still important to understand what they are and how they do what they do.

If this chapter weren't core to understanding the framework, we would've placed it at the end of the book. We put this material here because we believe that understanding interceptors is absolutely necessary to successfully leveraging the power of the framework. First of all, you need to be familiar with the built-in interceptors, and you need to know how to arrange them to your liking. Second, debugging the framework can truly be confusing if you don't understand how the requests are processed. We think that interceptors ultimately provide a simpler architecture that can be more easily debugged and understood. However, many developers may find them counterintuitive at first.

With that said, when you do find yourself writing your own custom interceptors, you'll truly begin to enjoy the Struts 2 framework. As you develop your actions, keep your eyes out for any tasks that can be moved out to the interceptors. As soon as you do, you'll be hooked for life. But first, we should see how they actually work.

(to be continued)