SHRIMATI INDIRA GANDHI COLLEGE

Affiliated to Bharathidasan University| Nationally Accredited at 'A' Grade(3rd Cycle) by NAAC

An ISO 9001:2015 Certified Institution

Thiruchirrappalli

STUDY MATERIAL

WEB SERVICES(P22CSE1A)

DEPARTMENT OF COMPUTER SCIENCE, INFORMATION TECHNOLOGY AND COMPUTER APPLICATIONS

Prepared by,

MS.T.R.B.VIDHYA, M.S.I.T.,M.Phil.,M.C.A.,

ASST. PROF. IN COMPUTER SCIENCE,

SHRIMATI INDIRA GANDHI COLLEGE,

TIRUCHIRAPPALLI - 2

UNIT - IV

BUILDING REAL WORLD ENTERPRISE WEB SERVICES AND APPLICATIONS

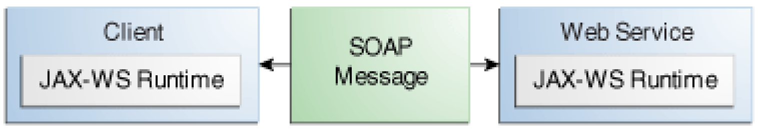
Any application can be put together in a haphazard manner, but these applications are almost never appropriate for a real world production environment. Real world enterprise applications must be easy to develop, even easier to customize and maintain, support transactional requirements, ...

SAMPLE SOURCE CODE TO DEVELOP WEB SERVICES

This section shows how to build and deploy a simple web service and two clients: an application client and a web client. The source code for the service is in the *tut-install*/examples/jaxws/helloservice-war/ directory, and the clients are in the *tut-install*/examples/jaxws/hello-appclient/ and *tut-install*/examples/jaxws/hello-webclient/ directories.

JAX-WS technology manages communication between a web service and a client.

Figure Communication between a JAX-WS Web Service and a Client



The starting point for developing a JAX-WS web service is a Java class annotated with the javax.jws.WebService annotation. The @WebService annotation defines the class as a web service endpoint.

A service endpoint interface or service endpoint implementation (SEI) is a Java interface or class, respectively, that declares the methods that a client can invoke on the service. An interface is not required when building a JAX-WS endpoint. The web service implementation class implicitly defines an SEI.

You may specify an explicit interface by adding the endpointInterface element to the @WebService annotation in the implementation class. You must then provide an interface that defines the public methods made available in the endpoint implementation class.

STEPS NECESSARY FOR CREATING A WEB SERVICE AND CLIENT

The basic steps for creating a web service and client are as follows.

1. Code the implementation class.
2. Compile the implementation class.
3. Package the files into a WAR file.
4. Deploy the WAR file. The web service artifacts, which are used to communicate with clients, are generated by GlassFish Server during deployment.

5. Code the client class.
6. Use the `wsimport` Maven goal to generate and compile the web service artifacts needed to connect to the service.
7. Compile the client class.
8. Run the client.

If you use NetBeans IDE to create a service and client, the IDE performs the `wsimport` task for you.

The sections that follow cover these steps in greater detail.

Requirements of a JAX-WS Endpoint

JAX-WS endpoints must follow these requirements.

- The implementing class must be annotated with either the `javax.jws.WebService` or the `javax.jws.WebServiceProvider` annotation.
- The implementing class may explicitly reference an SEI through the `endpointInterface` element of the `@WebService` annotation but is not required to do so. If no `endpointInterface` is specified in `@WebService`, an SEI is implicitly defined for the implementing class.
- The business methods of the implementing class must be public and must not be declared `static` or `final`.
- Business methods that are exposed to web service clients must be annotated with `javax.jws.WebMethod`.
- Business methods that are exposed to web service clients must have JAXB-compatible parameters and return types. See the list of JAXB default data type bindings in Types Supported by JAX-WS.
- The implementing class must not be declared `final` and must not be `abstract`.
- The implementing class must have a default public constructor.
- The implementing class must not define the `finalize` method.
- The implementing class may use the `javax.annotation.PostConstruct` or the `javax.annotation.PreDestroy` annotations on its methods for lifecycle event callbacks.

  The `@PostConstruct` method is called by the container before the implementing class begins responding to web service clients.

  The `@PreDestroy` method is called by the container before the endpoint is removed from operation.

Coding the Service Endpoint Implementation Class

In this example, the implementation class, `Hello`, is annotated as a web service endpoint using the `@WebService` annotation. `Hello` declares a single method named `sayHello`, annotated with the `@WebMethod` annotation, which exposes the annotated method to web

}

service clients. The sayHello method returns a greeting to the client, using the name passed to it to compose the greeting. The implementation class also must define a default, public, no-argument constructor.

```
package javaeetutorial.helloservice;



import javax.jws.WebService;

import javax.jws.WebMethod;



@WebService

public class Hello {

    private final String message = "Hello, ";



    public Hello() {

    }



    @WebMethod

    public String sayHello(String name) {

        return message + name + ".";

    }

}
```

Building, Packaging, and Deploying the Service

You can use either NetBeans IDE or Maven to build, package, and deploy the `helloservice-war` application.

The following topics are addressed here:

- To Build, Package, and Deploy the Service Using NetBeans IDE
- To Build, Package, and Deploy the Service Using Maven

To Build, Package, and Deploy the Service Using NetBeans IDE

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. From the File menu, choose Open Project.
3. In the Open Project dialog box, navigate to:

   ```
   tut-install/examples/jaxws
   ```
4. Select the `helloservice-war` folder.
5. Click Open Project.
6. In the Projects tab, right-click the `helloservice-war` project and select Run.

   This command builds and packages the application into a WAR file, `helloservice-war.war`, located in *tut-install*/examples/jaxws/helloservice-war/target/, and deploys this WAR file to your GlassFish Server instance. It also opens the web service test interface at the URL shown in To Test the Service without a Client.

Next Steps

You can view the WSDL file of the deployed service by requesting the URL `http://localhost:8080/helloservice-war/HelloService?wsdl` in a web browser. Now you are ready to create a client that accesses this service.

To Build, Package, and Deploy the Service Using Maven

1. Make sure that GlassFish Server has been started (see Starting and Stopping GlassFish Server).
2. In a terminal window, go to:

   ```
   tut-install/examples/jaxws/helloservice-war/
   ```
3. Enter the following command:

   ```
   mvn install
   ```
   This command builds and packages the application into a WAR file, `helloservice-war.war`, located in the `target` directory, and then deploys the WAR to GlassFish Server.

Next Steps

You can view the WSDL file of the deployed service by requesting the URL `http://localhost:8080/helloservice-war/HelloService?wsdl` in a web browser. Now you are ready to create a client that accesses this service.

Testing the Methods of a Web Service Endpoint

GlassFish Server allows you to test the methods of a web service endpoint.

The following topics are addressed here:

- To Test the Service without a Client

To Test the Service without a Client

To test the `sayHello` method of `HelloService`, follow these steps.

1. Open the web service test interface by entering the following URL in a web browser:

   ```
   http://localhost:8080/helloservice-war/HelloService?Tester
   ```
2. Under Methods, enter a name as the parameter to the `sayHello` method.
3. Click sayHello.

   This takes you to the `sayHello` Method invocation page.

   Under Method returned, you'll see the response from the endpoint.

A Simple JAX-WS Application Client

The `HelloAppClient` class is a stand-alone application client that accesses the `sayHello` method of `HelloService`. This call is made through a port, a local object that acts as a proxy for the remote service. The port is created at development time by the `wsimport` Maven goal, which generates JAX-WS portable artifacts based on a WSDL file.

The following topics are addressed here:

- Coding the Application Client
- Running the Application Client

Coding the Application Client

When invoking the remote methods on the port, the client performs these steps.

1. It uses the generated `helloservice.endpoint.HelloService` class, which represents the service at the URI of the deployed service's WSDL file:
2. import javaeetutorial.helloservice.endpoint.HelloService;

```
3. import javax.xml.ws.WebServiceRef;

4.

5. public class HelloAppClient {

6.     @WebServiceRef(wsdlLocation =

7.       "http://localhost:8080/helloservice-war/HelloService?WSDL")
```

```
    private static HelloService service;
```

8. It retrieves a proxy to the service, also known as a port, by invoking `getHelloPort` on the service:

   ```
   javaeetutorial.helloservice.endpoint.Hello port = service.getHelloPort();
   ```

   The port implements the SEI defined by the service.

9. It invokes the port’s `sayHello` method, passing a string to the service:

   ```
   return port.sayHello(arg0);
   ```

Here is the full source of `HelloAppClient.java`, which is located in the `*tut-install*/examples/jaxws/hello-appclient/src/main/java/javaeetutorial/hello/appclient/` directory:

```
package javaeetutorial.hello.appclient;



import javaeetutorial.helloservice.endpoint.HelloService;

import javax.xml.ws.WebServiceRef;



public class HelloAppClient {

    @WebServiceRef(wsdlLocation =

      "http://localhost:8080/helloservice-war/HelloService?WSDL")
```

```
    private static HelloService service;



    /**

     * @param args the command line arguments

     */

    public static void main(String[] args) {

        System.out.println(sayHello("world"));

    }



    private static String sayHello(java.lang.String arg0) {

        javaeetutorial.helloservice.endpoint.Hello port =

                service.getHelloPort();

        return port.sayHello(arg0);

    }

}
```

Running the Application Client

You can use either NetBeans IDE or Maven to build, package, deploy, and run the `hello-appclient` application. To build the client, you must first have deployed `helloservice-war`, as described in Building, Packaging, and Deploying the Service.

The following topics are addressed here:

- To Run the Application Client Using NetBeans IDE

To Run the Application Client Using NetBeans IDE

1. From the File menu, choose Open Project.
2. In the Open Project dialog box, navigate to:

   ```
   tut-install/examples/jaxws
   ```
3. Select the `hello-appclient` folder.
4. Click Open Project.
5. In the Projects tab, right-click the `hello-appclient` project and select Build.

   This command runs the `wsimport` goal, then builds, packages, and runs the client. You will see the output of the application client in the hello-appclient output tab:

   ```
   --- exec-maven-plugin:1.2.1:exec (run-appclient) @ hello-appclient ---


   Hello, world.
   ```

To Run the Application Client Using Maven

1. In a terminal window, go to:

   ```
   tut-install/examples/jaxws/hello-appclient/
   ```
2. Enter the following command:

   ```
   mvn install
   ```
   This command runs the `wsimport` goal, then builds, packages, and runs the client. The application client output looks like this:

   ```
   --- exec-maven-plugin:1.2.1:exec (run-appclient) @ hello-appclient ---


   Hello, world.
   ```

A Simple JAX-WS Web Client

`HelloServlet` is a servlet that, like the Java client, calls the `sayHello` method of the web service. Like the application client, it makes this call through a port.

The following topics are addressed here:

- Coding the Servlet
- Running the Web Client

Coding the Servlet

To invoke the method on the port, the client performs these steps.

1. It imports the `HelloService` endpoint and the `WebServiceRef` annotation:
2. `import javaeetutorial.helloservice.endpoint.HelloService;`
3. `...`

```
import javax.xml.ws.WebServiceRef;
```

4. It defines a reference to the web service by specifying the WSDL location:
5. `@WebServiceRef(wsdlLocation =`

```
  "http://localhost:8080/helloservice-war/HelloService?WSDL")
```

6. It declares the web service, then defines a private method that calls the `sayHello` method on the port:
7. `private HelloService service;`
8. `...`
9. `private String sayHello(java.lang.String arg0) {`
10. `    javaeetutorial.helloservice.endpoint.Hello port =`
11. `            service.getHelloPort();`
12. `    return port.sayHello(arg0);`

```
}
```

13. In the servlet, it calls this private method:

```
out.println("<p>" + sayHello("world") + "</p>");
```

The significant parts of the `HelloServlet` code follow. The code is located in the *tut-install*/examples/jaxws/hello-webclient/src/java/javaeetutorial/hello/ webclient/ directory.

```
package javaeetutorial.hello.webclient;



import javaeetutorial.helloservice.endpoint.HelloService;
import java.io.IOException;

import java.io.PrintWriter;

import javax.servlet.ServletException;

import javax.servlet.annotation.WebServlet;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import javax.xml.ws.WebServiceRef;



@WebServlet(name="HelloServlet", urlPatterns={"/HelloServlet"})

public class HelloServlet extends HttpServlet {

    @WebServiceRef(wsdlLocation =

      "http://localhost:8080/helloservice-war/HelloService?WSDL")

    private HelloService service;



    /**

     * Processes requests for both HTTP <code>GET</code>

     *   and <code>POST</code> methods.

     * @param request servlet request
     * @param response servlet response

     * @throws ServletException if a servlet-specific error occurs

     * @throws IOException if an I/O error occurs

     */

    protected void processRequest(HttpServletRequest request,

            HttpServletResponse response)

    throws ServletException, IOException {

        response.setContentType("text/html;charset=UTF-8");

        try (PrintWriter out = response.getWriter()) {



            out.println("<html lang=\"en\">");

            out.println("<head>");

            out.println("<title>Servlet HelloServlet</title>");

            out.println("</head>");

            out.println("<body>");

            out.println("<h1>Servlet HelloServlet at " +

                request.getContextPath () + "</h1>");

            out.println("<p>" + sayHello("world") + "</p>");

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

        }

    }



    // doGet and doPost methods, which call processRequest, and

    //   getServletInfo method



    private String sayHello(java.lang.String arg0) {

        javaeetutorial.helloservice.endpoint.Hello port =

                service.getHelloPort();

        return port.sayHello(arg0);

    }

}
```

Running the Web Client

You can use either NetBeans IDE or Maven to build, package, deploy, and run the `hello-webclient` application. To build the client, you must first have deployed `helloservice-war`, as described in Building, Packaging, and Deploying the Service.

The following topics are addressed here:

- To Run the Web Client Using NetBeans IDE
- To Run the Web Client Using Maven

To Run the Web Client Using NetBeans IDE

1. From the File menu, choose Open Project.
2. In the Open Project dialog box, navigate to:

```
tut-install/examples/jaxws
```

3. Select the `hello-webclient` folder.
4. Click Open Project.
5. In the Projects tab, right-click the `hello-webclient` project and select Build.

   This task runs the `wsimport` goal, builds and packages the application into a WAR file, `hello-webclient.war`, located in the `target` directory, and deploys it to GlassFish Server.

6. In a web browser, enter the following URL:

   ```
   http://localhost:8080/hello-webclient/HelloServlet
   ```

   The output of the `sayHello` method appears in the window.

To Run the Web Client Using Maven

1. In a terminal window, go to:

   ```
   tut-install/examples/jaxws/hello-webclient/
   ```

2. Enter the following command:

   ```
   mvn install
   ```

   This command runs the `wsimport` goal, then build and packages the application into a WAR file, `hello-webclient.war`, located in the `target` directory. The WAR file is then deployed to GlassFish Server.

3. In a web browser, enter the following URL:

   ```
   http://localhost:8080/hello-webclient/HelloServlet
   ```

   The output of the `sayHello` method appears in the window.

DEVELOPING WEB SERVICE APPLICATIONS

Develop and publish web service applications, which are modular applications that implement a services oriented architecture (SOA). These topics explain how to create and deploy web services, how to implement web service security, and how to test and validate web services.

- Learn about web service applications
  Web services are self-contained, modular applications that can be described, published, located, and invoked over a network. They implement a services oriented architecture (SOA), which supports the connecting or sharing of resources and data in a very flexible and standardized manner. Services are described and organized to support their dynamic, automated discovery and reuse.

- SOAP
  SOAP (formerly known as Simple Object Access Protocol) is a lightweight protocol for the exchange of information in a decentralized, distributed environment. A SOAP message is a transmission of information from a sender to a receiver. SOAP messages can be combined to perform request/response patterns.
- Java API for XML based web services
  Java API for XML-based web services (JAX-WS), which is also known as JSR-224, is the next generation web services programming model that extends the foundation provided by the Java API for XML-based RPC (JAX-RPC) programming model. Using JAX-WS, developing web services and clients is simplified with greater platform independence for Java applications by the use of dynamic proxies and Java annotations. The web services tools included in this product support JAX-WS 2.0, 2.1, and 2.2.
- JAXB
  Java Architecture for XML Binding (JAXB), which is also known as JSR-222, is a Java technology that provides an easy and convenient way to map Java classes and XML schema for simplified development of web services. JAXB leverages the flexibility of platform-neutral XML data in Java applications to bind XML schema to Java applications without requiring extensive knowledge of XML programming. The tools included in this workbench implement JAXB 2.0, 2.1, and 2.2 standards.
- Known problems and limitations for web service applications
  Various known problems and limitations apply when you are working with web service applications and WebSphere Developer Tools. Issues include, among others, problems when you use a secured WebSphere Application Server and when you use the web services wizards.
- Developing JAX-RS applications
  You can develop Java API for RESTful web services (JAX-RS) applications so that you can create Representational State Transfer (REST) services quickly.
- Tools for web services development
- Configuring a workspace for web services development
  Although you can begin web services development immediately upon creating a workspace, you might find it convenient to configure your workspace to optimize your development experience.
- Developing web services and clients
  You can create web services and clients by using the web services wizards, annotations, Ant tasks, or command-line tools.
- Web services: Editing, assembling, and securing tasks
  After you create a web service or client, you can do various assembly tasks, such as editing the web service deployment descriptors, adding handlers, and enabling security.
- Creating and editing JAX-WS web service handlers
  You can add JAX-WS logical or protocol handlers to intercept inbound and outbound messages to or from web services and their clients. You can select from any currently available JAX-WS web services and clients and start the Handler Creation Wizard. In the wizard, you provide the class name of the handler, the handler name, and an optional display name, and specify the type of handler. When finished, the wizard generates the skeleton handler code and updates the applicable deployment descriptor.
- Merged skeleton files for web services updates
  After you create a web service, you might want to change it. Although you cannot automatically propagate all your changes to all the required files, to retain your

changes while you update the web service, you can merge a generated skeleton file. You can then regenerate your web service, and your changes remain intact.

- Securing web services
  Web services security for WebSphere Application Server is based on the OASIS web services security (WSS) Version 1.0 specification, the Username token Version 1.0 profile, and the X.509 token Version 1.0 profile. These standards and profiles address how to provide protection for messages that are exchanged in a web service environment.
- Deploying web services
  Deploying a web service involves creating the code that makes your web service available to others. You can deploy a project, an EAR file, or an application client. If you created a web service by using the web services wizards, the deployment code is generated automatically.
- Testing and validating web services
  After you create a web service or client, you can test it using sample JSPs, the web services Explorer, or the Generic Service Client. You can also test the SOAP traffic that is passed by the service.

PORT ACCESS: SEAMLESS TRANSITIONS FROM ONE AREA TO ANOTHER

- In order to ensure seamless transitions from one area to another, a range of terminals to provide the functions known generically as PORT Access has been developed. Featuring user-friendly, high-resolution interfaces and timeless design, they are adaptable to a variety of use cases. Each terminal supports myPORT – Schindler's breakthrough mobile solution – giving tenants and building owners the flexibility to manage powerful digital features from a smartphone or tablet.

UNIT-V

How to deploy a java web application on tomcat server

What is a dynamic web application?

A dynamic web application produces web pages in real-time. Based on the client request, the server generates a custom response and sent to the client. To differentiate, static web pages send the same response to the same resource, where dynamic web apps can return different responses for the same resource. (not considering request parameters for the moment.)

For example, when you access your email, you get access to only your emails, and the server produces the response exclusively for you. The article How to set up a Dynamic Web Module with Maven and Eclipse explains how to create a basic, bare minimum maven web application. To recap, we named that web application SpringWeb. Now let's deploy the SpringWeb web application on the tomcat web server, enabling visitors to access it via a browser.

What is the Tomcat server?

Apache Tomcat is an open-source Java servlet container. Tomcat implements core Java enterprise specs, such as Java Servlets, JavaServer Pages (JSP), and WebSockets APIs.

Tomcat was first released in 1998 as an Apache Software Foundation project. Tomcat web server directly deal with client requests and responses. When we use Tomcat to deploy our web application, Tomcat handles requests from clients and responses back to clients. Our web application deals with Tomcat.

Creating a Java web application is not in the scope of this article. If you do not have a web application war file, follow Dynamic Web Project with Maven and Eclipse to create one.

Tomcat needs Java installed and the path set to run. This article explains

- How to install Java
- How to configure Java
- How to install Tomcat
- How to deploy a dynamic web application in Tomcat

In Windows, Mac, and Linus operating systems. First, let's start with Windows.

1. How to deploy a Java web app on Tomcat on Windows

To deploy the Tomcat server on Windows, we need to have Java installed and the JAVA_HOME variable set.

1.1 How to install Java on Windows

Before trying to install Java, ensure you do not have Java installed in your system. Open a command prompt and type the below command.

```
java -version
```

If Java is installed in your system and the path is set, you will see an output similar to :

```
C:\Program Files>java -version
OpenJDK version "1.8.0_131"
OpenJDK Runtime Environment (.................)
```

If that is the case, you can skip to the next step, How to set JAVA_HOME in Windows.

Otherwise, your command line output will look something like this:

```
java -version is not recognized as an internal or external command
```

Now we know at least the path is not set. The next step is to ensure that Java is installed. Without the path set, there are several ways to check if Java is installed.

- Go to control panel -> Programmes -> Programmes and features and type Java in the search box. If Java is installed, a Java icon will appear.
- Click Start Menu and type Java, and look for the Java icon.

If Java is installed, you can skip to the next step, How to set JAVA_HOME in Windows.

Download and install Java on windows#

Download Java from the Oracle site or the free version from the OpenJDK. Follow the installation steps. Make sure to note down the full installation path. My installation path is `c:\learnbestcoding\jdk18`

How to set JAVA_HOME in Windows

- Click Start and type environment variables
- Select Edit the system environment variables
- Click Environment variables
- Click new under System variables
- Enter JAVA_HOME for the variable name and the JDK installation path (c:\learnbestcoding\jdk18) for the variable value.
- Click ok. Restart the command window and type `echo %JAVA_HOME%`

How to install Tomcat on Windows

- Download Tomcat

- Extract the archive to your local hard drive. My Tomcat is in `c:\learnbestcoding\tomcat`. This location is called CATALINA_HOME.
- Create CATALINA_HOME environment variable. Variable value is `c:\learnbestcoding\tomcat`.

Now we have completed all the steps to run the Tomcat server on the Windows system.

How to deploy Tomcat on Windows

- Place the Java war file in the `CATALINA_HOME\webapps` folder. That is `c:\learnbestcoding\tomcat\webapps`
- Open a command prompt and navigate to the `CATALINA_HOME\bin` folder
- Run the server by typing `startup.bat`
- Open a browser and navigate to `http://localhost:8080/webapp`. Replace webapp with the actual context name of the web application.

How to deploy a Java web app on Tomcat on Mac

1.2 How to install Java on Mac

To ensure if you already have Java installed in your system,

- Open a terminal window and type `javac -version`. If Java is installed, it will output the installed version.
- Open a terminal window and type `which java`. If Java is installed, it will output the installation location.

If Java is installed, you can skip to the next step, How to install Tomcat on Mac.

Install Java with Homebrew

- Install Homebrew
- Install Java with `brew install openjdk@18`
- Confirm the installation with `brew info java`
- For the system Java wrappers to find this JDK, symlink it with `sudo ln -sfn /opt/homebrew/opt/openjdk/libexec/openjdk.jdk /Library/Java/JavaVirtualMachines/openjdk.jdk`
- Set the path with `export PATH="/opt/homebrew/opt/openjdk/bin:$PATH"`
- Create JAVA_HOME with `echo export "JAVA_HOME=/opt/homebrew/opt/openjdk" >> ~/.zshrc` or `echo export or "JAVA_HOME=/opt/homebrew/opt/openjdk" >> ~/.bash_profile`

How to install Tomcat on Mac

- Download Tomcat from https://tomcat.apache.org
- Extract the archive into your local hard drive. My Tomcat location is `/Users/learnbestcoding/SERVERS/apache-tomcat-10.0.14`. That is also your CATALINA_HOME.

- Set CATALINA_HOME with `export CATALINA_HOME=/Users/learnbestcoding/SERVERS/apache-tomcat-10.0.14`

Now the Tomcat server is ready to run on the Mac.

How to deploy Tomcat on Mac

- Place the Java war file in the `CATALINA_HOME\webapps` folder. That is `/Users/learnbestcoding/SERVERS/apache-tomcat-10.0.14/webapps`
- Open a command prompt and navigate to the `CATALINA_HOME\bin` folder
- Run the server by typing `startup.sh`
- Open a browser and navigate to `http://localhost:8080/webapp`. Replace webapp with the actual context name of the web application.
- To stop the server run `shutdown.sh`

How to deploy a Java web app on Tomcat on Linux

1.3 How to install Java on Linux

To ensure you don't have Java installed in your Linux system,

- Open a terminal window and type `javac -version`. If Java is installed, it will output the installed version. Otherwise, the output will be `java: command not found`.
- Open a terminal window and type `which java`. If Java is installed, it will output the installation location.

If Java is installed, you can skip to the next step, How to install Tomcat on Linux. But don't forget to set JAVA_HOME.

Install Java on Linux with apt or yum

- Run `sudo apt update` or `sudo yum update` (for aws)
- Install Java with `sudo apt-get install openjdk-18-jdk -y` (check FAQ for AWS Linux)
- Confirm the installation with the `java --version`
- Find the installation location with `which java`
- Set JAVA_HOME with export `JAVA_HOME=/usr/bin/java`

How to install Tomcat on Linux

- Create a directory to hold Tomcat server folder. I use `/home/learnbestcoding/tomcat` as my Tomcat location. `mkdir /home/learnbestcoding/tomcat`
- Download and extract Apache Tomcat server into `/home/learnbestcoding/tomcat directory`.
- That Tomcat installation directory is also called CATALINA_HOME. Set CATALINA_HOME variable with `export CATALINA_HOME=/home/learnbestcoding/tomcat`

How to deploy Tomcat on Linux

- Place the Java war file in the `CATALINA_HOME\webapps` folder. That is `/home/learnbestcoding/tomcat/webapps`

- Open a command prompt and navigate to the `CATALINA_HOME/bin` folder.
- Run the server by typing `./startup.sh`
- Open a browser and navigate to `http://localhost:8080/webapp`. Replace webapp with the actual context name of the web application.
- To stop the server run `./shutdown.sh`

FAQ

Q. sudo: apt: command not found

A. That usually happens in AWS EC2 instances. For AWS instances, use `sudo amazon-linux-extras install java-openjdk11 -y` with your preferred JDK version.

Q. mkdir: cannot create directory 'tomcat': Permission denied

A. This happens when the base folder is not in your home directory. Use `sudo mkdir tomcat`

Q. Cannot find /home/ec2-user/tomcat/bin/setclasspath.sh This file is needed to run this program

A. This can be due to the CATALINA_HOME variable pointing to the wrong folder. Make sure the CATALINA_HOME points to your actual tomcat server folder. The `unset CATALINA_HOME` will solve this error, but it will also remove the CATALINA_HOME variable. The correct solution is to fix the CATALINA_HOME path.

Java SOAP Webservice using Axis 2 and Tomcat Tutorial with examples

PUBLISHED ON October 01, 2014 **BY:** Prakash Hari Sharma **IN:**Web Services

Web services are application components which communicate using open protocols. Using Web Services we can publish our application's functions to everyone. This tutorial provides step by step instructions to develop Web Services using Axis2 Web Services / SOAP / WSDL engine and Eclipse IDE. Let's start.

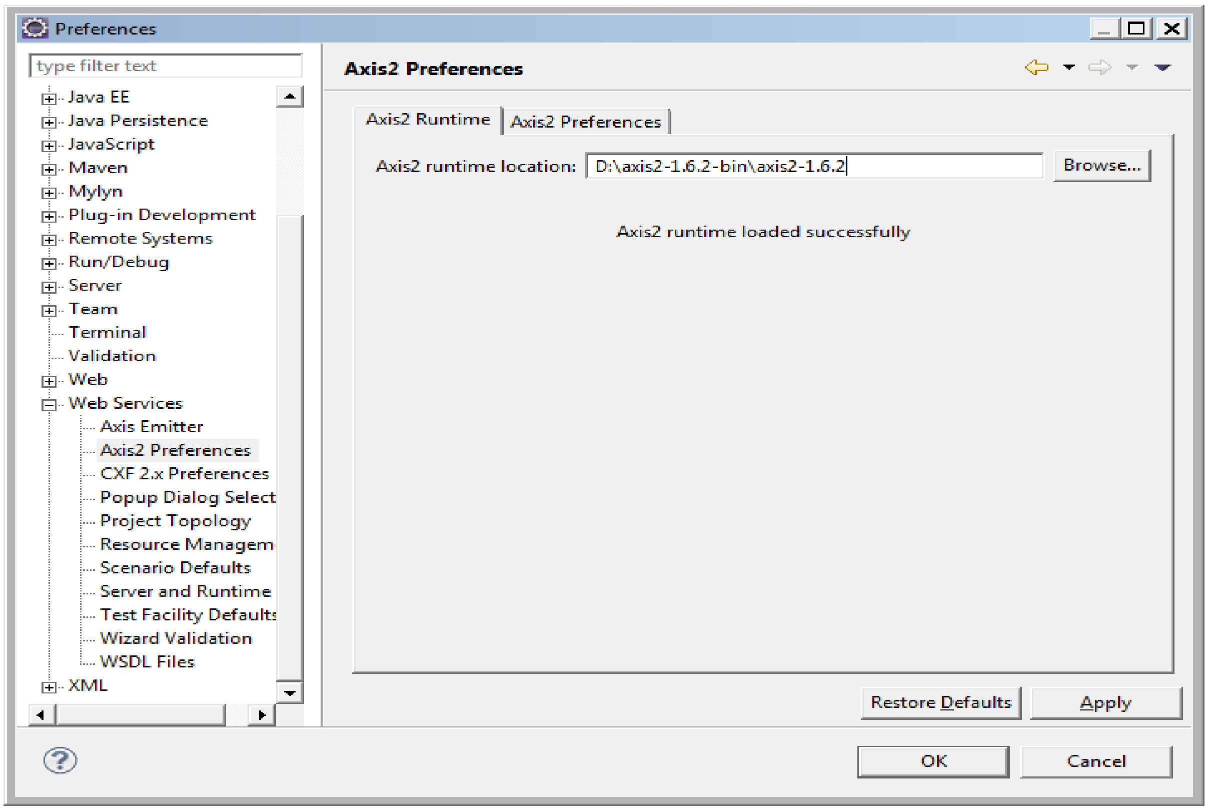
Axis 2

Axis 2 is a web service/SOAP/WSDL engine provided by Apache. It is a java based implementation. Axis 2 provides complete object model and modular architecture. Using Axis 2 you can easily create a web service from a plain java class, send SOAP messages, receive SOAP message.
We have used below tools for this tutorials

1. Apache Tomcat 7.05.55 Download from this website http://tomcat.apache.org/
2. axis2-1.6.2 Download from this website http://axis.apache.org/axis2/java/core/
3. Eclipse IDE - Kepler for Web Developers. Download from this website http://www.eclipse.org/

Set Axis runtime in Eclipse

- Go to Window Menu.
- Click on Preferences.
- Expand Web Services option.
- Click Axis2 Preferences.
- Give your axis2 location in the Axis2 runtime location box.

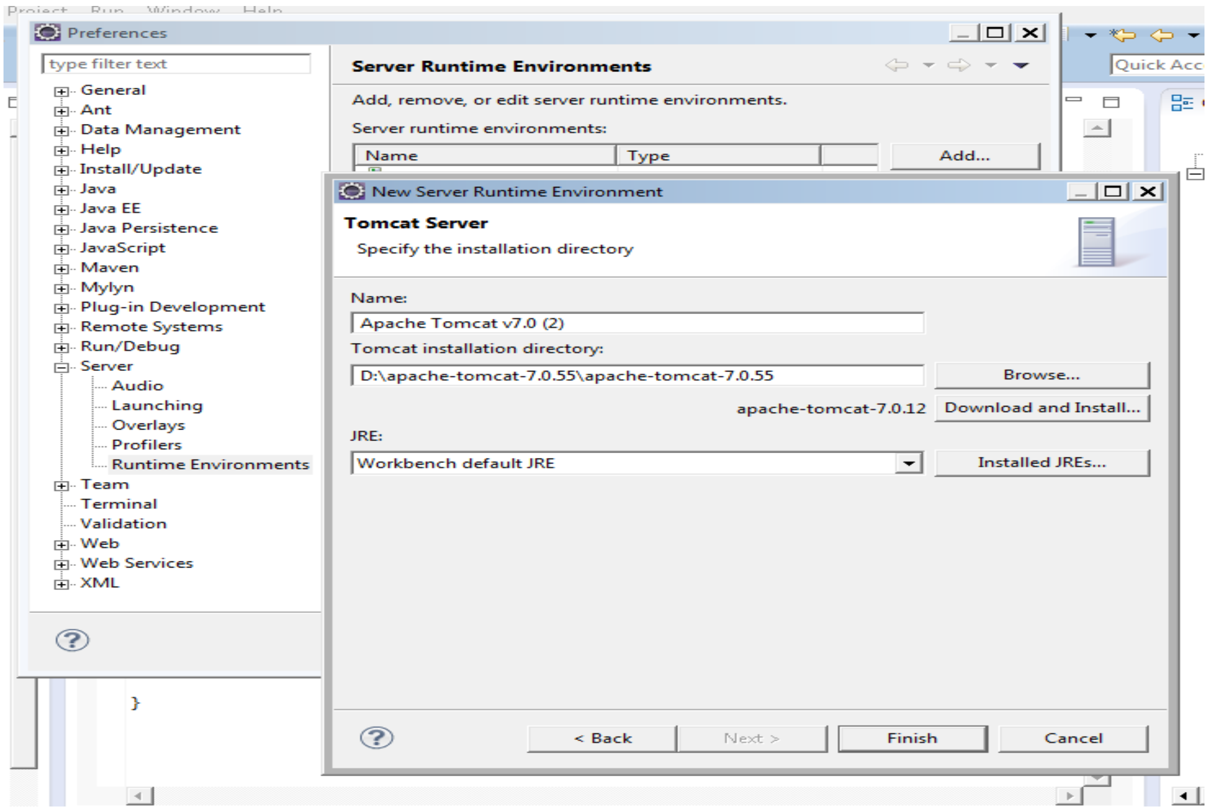


Axis2 runtime location: D:\axis2-1.6.2-bin\axis2-1.6.2

Axis2 runtime loaded successfully

Set Tomcat runtime in Eclipse

- Go to Window Menu.
- Click on Preferences.
- Expand Server option.
- Click on ADD.
- Select Apache Tomcat v7.0.
- Click Next
- Give your Tomcat location in the Tomcat installation directory box.



Server Runtime Environments

Add, remove, or edit server runtime environments.

Server runtime environments:

New Server Runtime Environment

Tomcat Server

Specify the installation directory

Name: Apache Tomcat v7.0 (2)

Tomcat installation directory: D:\apache-tomcat-7.0.55\apache-tomcat-7.0.55

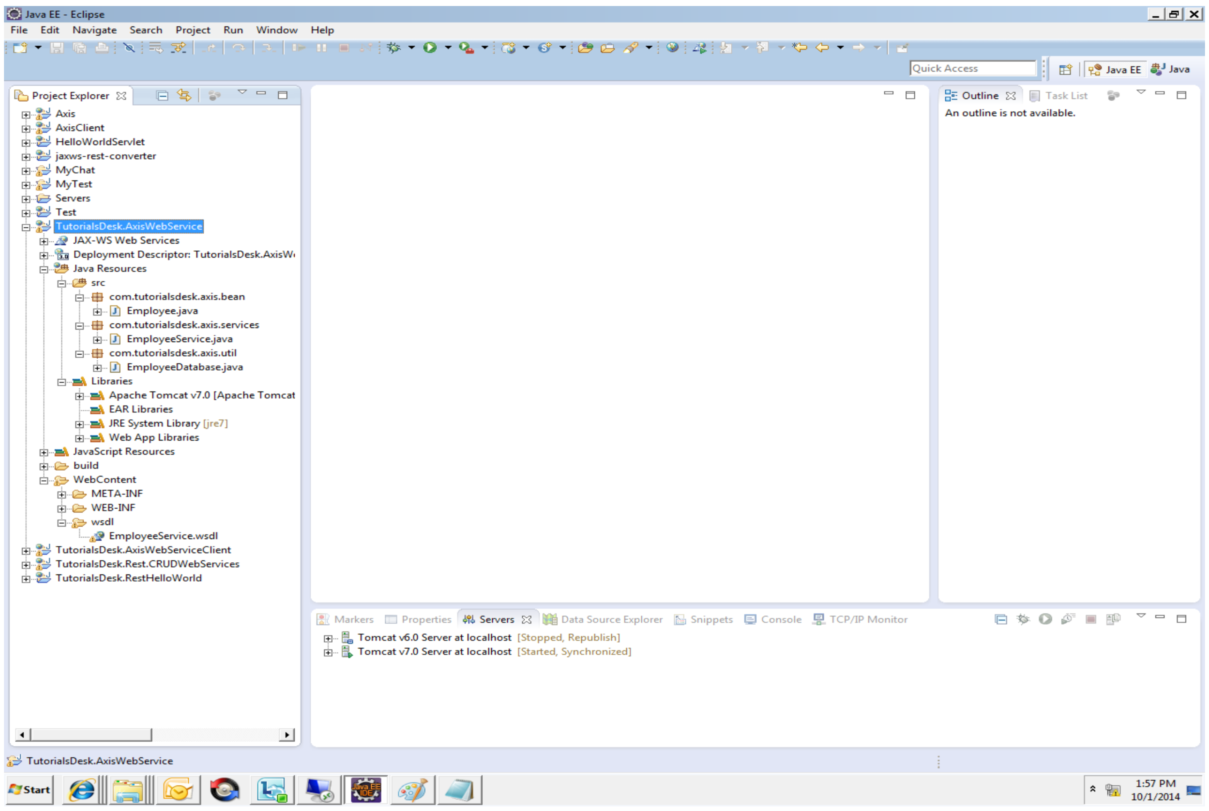
apache-tomcat-7.0.12

JRE: Workbench default JRE

Creating a web service from a plain java class in eclipse

Let's develope a simple web application to demonstrate Java SOAP Webservices using Axis 2 and Tomcat. This Sample Application is all about querying information from a Web Application. For this, let us assume that we have a Employee Database maintained in the backend and we publish the various services to the Client in the WSDL file. Both types of clients, Browser Client and Console Client are included in the sample Application.

STEP 1 : Create a new Dynamic Web Project called TutorialsDesk.AxisWebService.


Java EE - Eclipse
File Edit Navigate Search Project Run Window Help
Quick Access
Java EE
Java
Project Explorer
Axis
AxisClient
HelloWorldServlet
jaxws-rest-converter
MyChat
MyTest
Servers
Test
TutorialsDesk.AxisWebService
JAX-WS Web Services
Deployment Descriptor: TutorialsDesk.AxisW
Java Resources
src
com.tutorialsdesk.axis.bean
Employee.java
com.tutorialsdesk.axis.services
EmployeeService.java
com.tutorialsdesk.axis.util
EmployeeDatabase.java
Libraries
Apache Tomcat v7.0 [Apache Tomcat
EAR Libraries
JRE System Library [jre7]
Web App Libraries
JavaScript Resources
build
WebContent
META-INF
WEB-INF
wsdl
EmployeeService.wsdl
TutorialsDesk.AxisWebServiceClient
TutorialsDesk.Rest.CRUDWebServices
TutorialsDesk.RestHelloWorld
Outline
Task List
An outline is not available.
Markers Properties Servers Data Source Explorer Snippets Console TCP/IP Monitor
Tomcat v6.0 Server at localhost [Stopped, Republish]
Tomcat v7.0 Server at localhost [Started, Synchronized]
TutorialsDesk.AxisWebService
Start
1:57 PM
10/1/2014

STEP 2 : Create a package com.tutorialsdesk.axis.bean under TutorialsDesk.AxisWebService.

STEP 3 : Create class Employee.java under package com.tutorialsdesk.axis.bean as given below

```
package com.tutorialsdesk.axis.bean;



import java.io.Serializable;



public class Employee implements Serializable {

 private static final long serialVersionUID = -1129402159048345204L;

 String Name;

 String Department;

 int Age;

 double Salary;
```

```
public Employee() {

}

public Employee(String name, String department, int age, double salary) {
 super();
 Name = name;
 Department = department;
 Age = age;
 Salary = salary;
}

public String getName() {
 return Name;
}

public void setName(String name) {
 Name = name;
}

public String getDepartment() {
 return Department;
}

public void setDepartment(String department) {
  Department = department;
 }

 public int getAge() {
  return Age;
 }

 public void setAge(int age) {
  Age = age;
 }

 public double getSalary() {
  return Salary;
 }

 public void setSalary(double salary) {
  Salary = salary;
 }

}
```

STEP 4 : Create a package com.tutorialsdesk.axis.util under TutorialsDesk.AxisWebService.

STEP 5 : Create class EmployeeDatabase.java under package com.tutorialsdesk.axis.util as given below

```
package com.tutorialsdesk.axis.util;


import java.util.*;

import com.tutorialsdesk.axis.bean.Employee;


public class EmployeeDatabase {


    private static List<employee> employees;

    public static List<employee> list(){

        return employees;

    }


    public static Employee getEmployee(String name){

        Iterator<employee> iterator = employees.iterator();

        while (iterator.hasNext()){

         Employee employee = (Employee) iterator.next();

            if (employee.getName().equals(name)){

                return employee;

            }

        }

        return null;

    }


    public static Employee getEmployeeData(String name){

        Iterator<employee> iterator = employees.iterator();
        while (iterator.hasNext()){

         Employee employee = (Employee) iterator.next();

            if (employee.getName().equals(name)){

                return employee;

            }

        }

        return null;

    }

    static {

     initEmployees();

    }


    static void initEmployees(){

     employees = new ArrayList<employee>();

     employees.add(new Employee("Rahul", "HR", 25, 15000.00));

     employees.add(new Employee("Zuzana", "Sales", 32, 48000.00));

     employees.add(new Employee("Martin", "Engineering",22, 32000.00));

     employees.add(new Employee("Sachin", "Engineering",25, 65000.00));

     employees.add(new Employee("Ondrej", "Operations",26, 25000.00));

    }

}
```

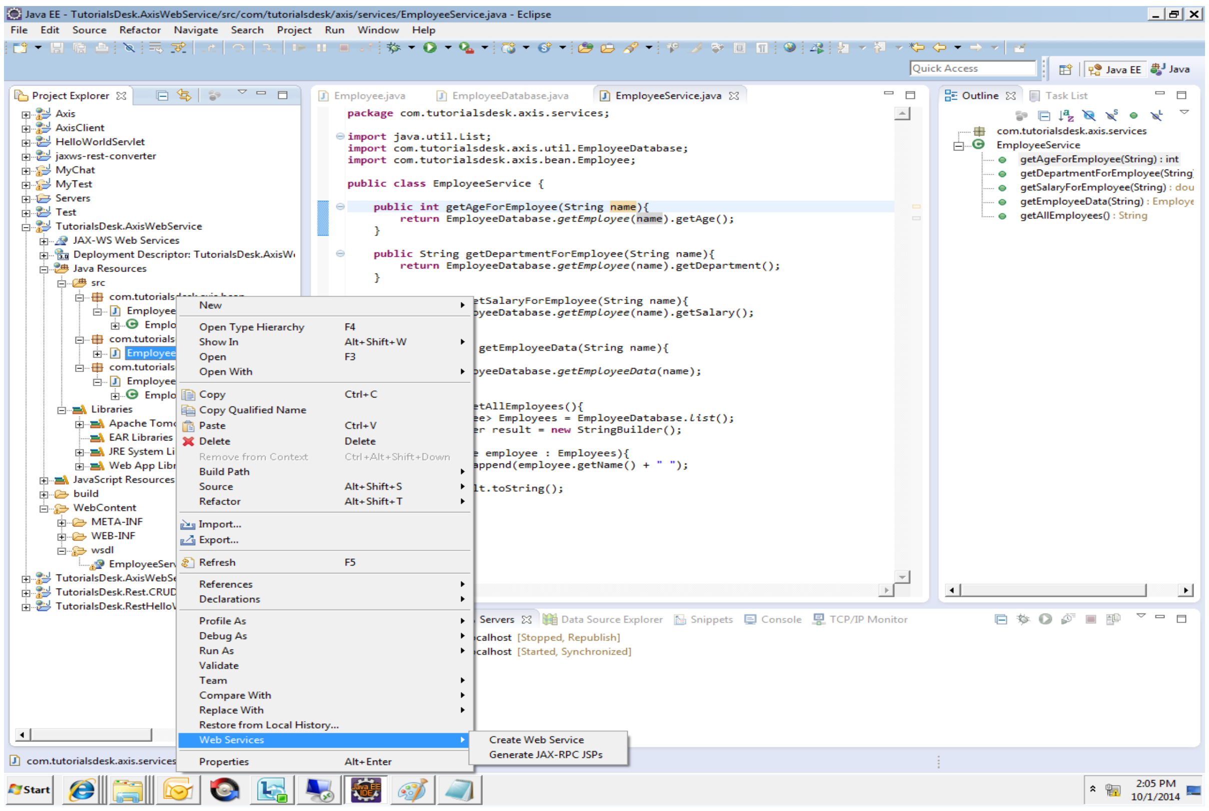
STEP 6 : Create a package com.tutorialsdesk.axis.services under TutorialsDesk.AxisWebService.

STEP 7 : Create class EmployeeService.java under package com.tutorialsdesk.axis.services as given below

```
package com.tutorialsdesk.axis.services;



import java.util.List;

import com.tutorialsdesk.axis.util.EmployeeDatabase;

import com.tutorialsdesk.axis.bean.Employee;



public class EmployeeService {



    public int getAgeForEmployee(String name){

        return EmployeeDatabase.getEmployee(name).getAge();

    }



    public String getDepartmentForEmployee(String name){

        return EmployeeDatabase.getEmployee(name).getDepartment();

    }



    public double getSalaryForEmployee(String name){

        return EmployeeDatabase.getEmployee(name).getSalary();

    }



    public Employee getEmployeeData(String name){



     return EmployeeDatabase.getEmployeeData(name);
    }


    public String getAllEmployees(){

        List<employee%gt; Employees = EmployeeDatabase.list();

        StringBuilder result = new StringBuilder();


        for(Employee employee : Employees){

            result.append(employee.getName() + " ");

        }

        return result.toString();

    }


}
```

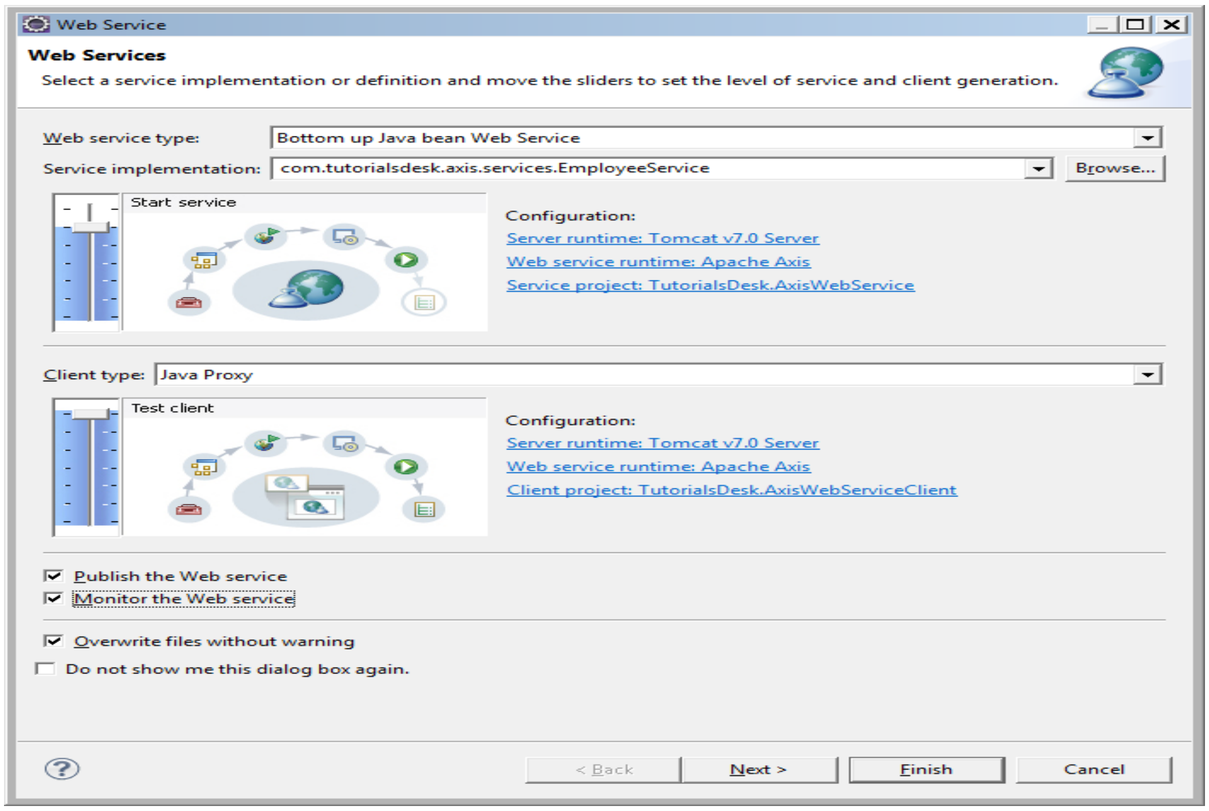
STEP 8 : In Project folder right click on EmployeeService.java

STEP 9 : Click on Web Services - > Create Web Service.



```
package com.tutorialsdesk.axis.services;

import java.util.List;
import com.tutorialsdesk.axis.util.EmployeeDatabase;
import com.tutorialsdesk.axis.bean.Employee;

public class EmployeeService {

    public int getAgeForEmployee(String name){
        return EmployeeDatabase.getEmployee(name).getAge();
    }

    public String getDepartmentForEmployee(String name){
        return EmployeeDatabase.getEmployee(name).getDepartment();
    }

etSalaryForEmployee(String name){
oyeeDatabase.getEmployee(name).getSalary();

 getEmployeeData(String name){

oyeeDatabase.getEmployeeData(name);

etAllEmployees(){
ee> Employees = EmployeeDatabase.list();
er result = new StringBuilder();

e employee : Employees){
append(employee.getName() + " ");

lt.toString();
```

STEP 10 : Select options **publish the web service** and **Monitor the web service**.



Web Service

Web Services

Select a service implementation or definition and move the sliders to set the level of service and client generation.

Web service type: Bottom up Java bean Web Service

Service implementation: com.tutorialsdesk.axis.services.EmployeeService

Start service

Configuration:
Server runtime: Tomcat v7.0 Server
Web service runtime: Apache Axis
Service project: TutorialsDesk.AxisWebService

Client type: Java Proxy

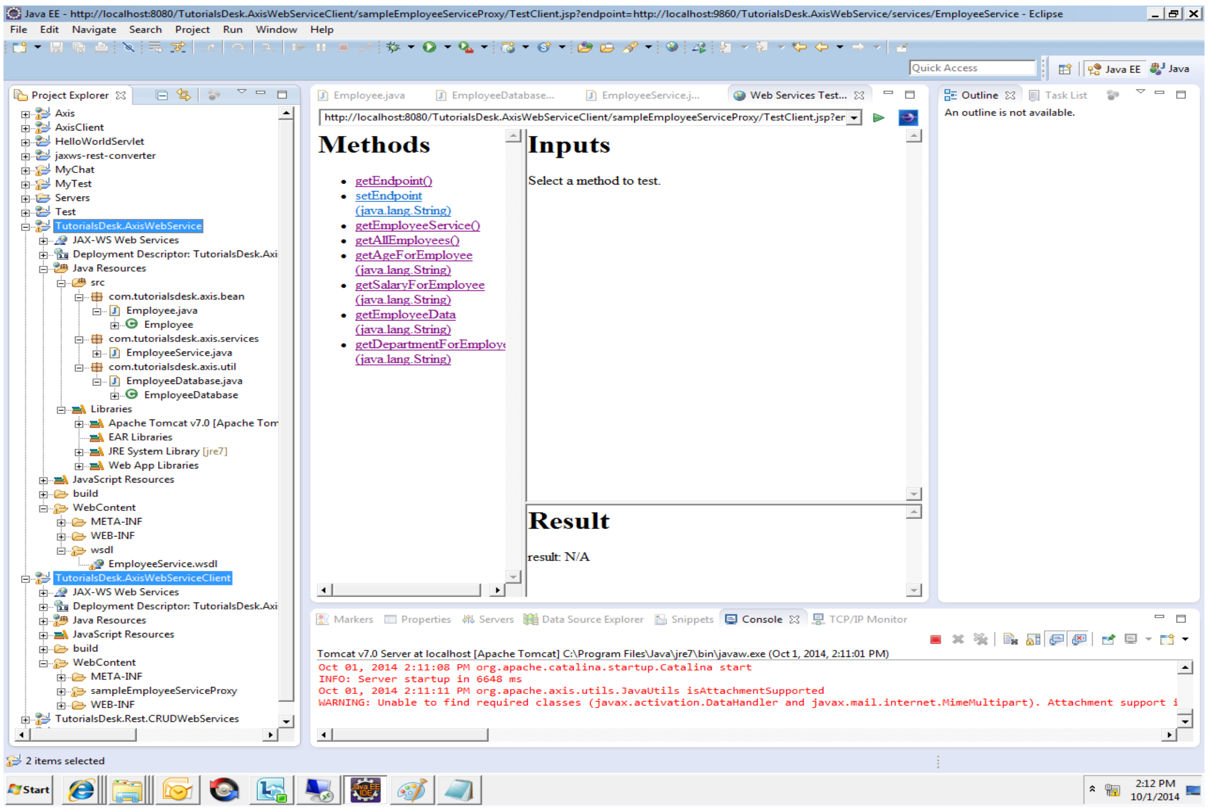
Test client

Configuration:
Server runtime: Tomcat v7.0 Server
Web service runtime: Apache Axis
Client project: TutorialsDesk.AxisWebServiceClient

☑ Publish the Web service
☑ Monitor the Web service
☑ Overwrite files without warning
☐ Do not show me this dialog box again.

STEP 11 : Click on **Finish**.

STEP 12 : It may take some time to finish all processes and you should see new project “TutorialsDesk.AxisWebServiceClient” created. Here is a final project structure.


Java EE - http://localhost:8080/TutorialsDesk.AxisWebServiceClient/sampleEmployeeServiceProxy/TestClient.jsp?endpoint=http://localhost:9860/TutorialsDesk.AxisWebService/services/EmployeeService - Eclipse

```
Project Explorer
+ Axis
+ AxisClient
+ HelloWorldServlet
+ jaxws-rest-converter
+ MyChat
+ MyTest
+ Servers
+ Test
- TutorialsDesk.AxisWebService
    + JAX-WS Web Services
    + Deployment Descriptor: TutorialsDesk.Axi
    - Java Resources
        - src
            - com.tutorialsdesk.axis.bean
                - Employee.java
                    + Employee
            - com.tutorialsdesk.axis.services
                + EmployeeService.java
            - com.tutorialsdesk.axis.util
                - EmployeeDatabase.java
                    + EmployeeDatabase
        - Libraries
            + Apache Tomcat v7.0 [Apache Tom
              EAR Libraries
            + JRE System Library [jre7]
            + Web App Libraries
    + JavaScript Resources
    + build
    - WebContent
        + META-INF
        + WEB-INF
        - wsdl
            EmployeeService.wsdl
- TutorialsDesk.AxisWebServiceClient
    + JAX-WS Web Services
    + Deployment Descriptor: TutorialsDesk.Axi
    + Java Resources
    + JavaScript Resources
    + build
    - WebContent
        + META-INF
        + sampleEmployeeServiceProxy
        + WEB-INF
+ TutorialsDesk.Rest.CRUDWebServices
```

http://localhost:8080/TutorialsDesk.AxisWebServiceClient/sampleEmployeeServiceProxy/TestClient.jsp?er

Methods

- getEndpoint()
- setEndpoint (java.lang.String)
- getEmployeeService()
- getAllEmployees()
- getAgeForEmployee (java.lang.String)
- getSalaryForEmployee (java.lang.String)
- getEmployeeData (java.lang.String)
- getDepartmentForEmploye (java.lang.String)

Inputs

Select a method to test.

Result

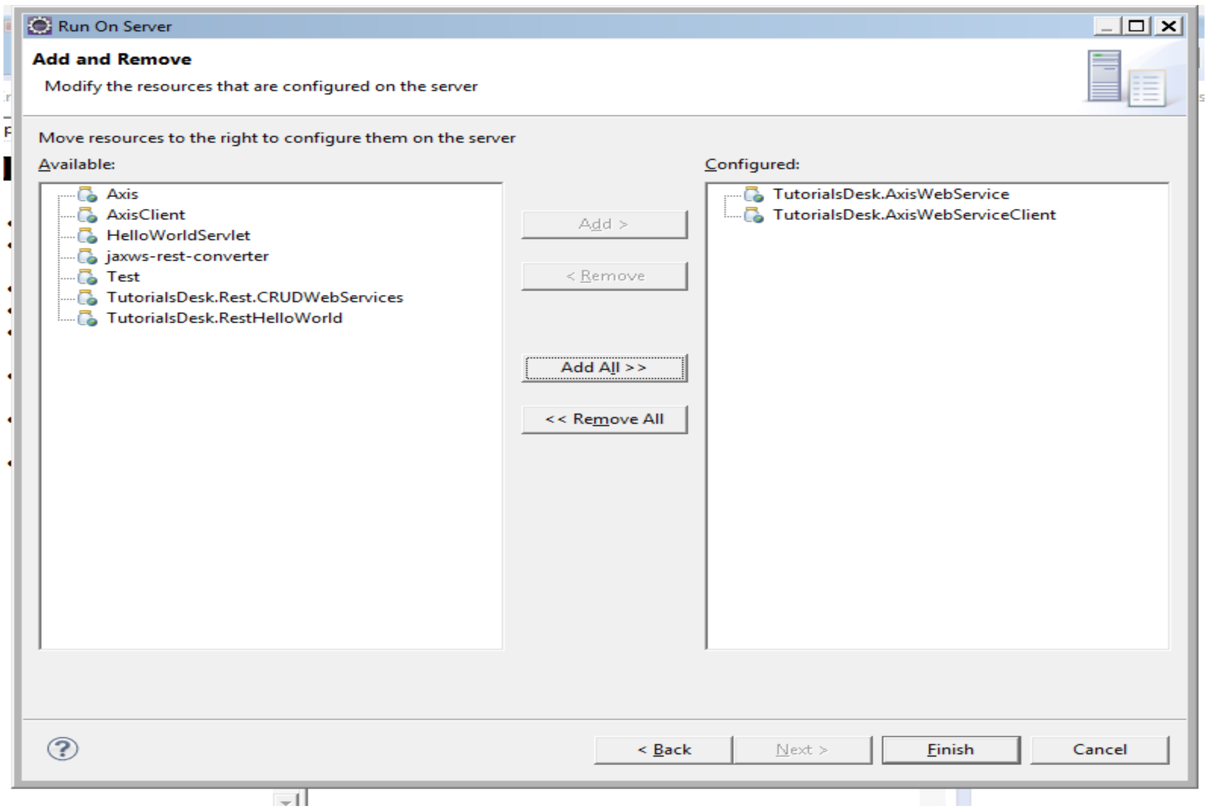
result: N/A

Outline: An outline is not available.

```
Tomcat v7.0 Server at localhost [Apache Tomcat] C:\Program Files\Java\jre7\bin\javaw.exe (Oct 1, 2014, 2:11:01 PM)
Oct 01, 2014 2:11:08 PM org.apache.catalina.startup.Catalina start
INFO: Server startup in 6648 ms
Oct 01, 2014 2:11:11 PM org.apache.axis.utils.JavaUtils isAttachmentSupported
WARNING: Unable to find required classes (javax.activation.DataHandler and javax.mail.internet.MimeMultipart). Attachment support i
```

2 items selected

STEP 13 : “TutorialsDesk.AxisWebService” and “TutorialsDesk.AxisWebServiceClient” both projects should be automatically deployed to server.


Run On Server

Add and Remove

Modify the resources that are configured on the server

Move resources to the right to configure them on the server

Available:

- Axis
- AxisClient
- HelloWorldServlet
- jaxws-rest-converter
- Test
- TutorialsDesk.Rest.CRUDWebServices
- TutorialsDesk.RestHelloWorld

Configured:

- TutorialsDesk.AxisWebService
- TutorialsDesk.AxisWebServiceClient

Invoking the Web Services

Use the following URL's to access the various web services operations

- http://localhost:8080/TutorialsDesk.AxisWebService/services/EmployeeService?method=getAllEmployees
- http://localhost:8080/TutorialsDesk.AxisWebService/services/EmployeeService?method=getAgeForEmployee&name=Martin
- http://localhost:8080/TutorialsDesk.AxisWebService/services/EmployeeService?method=getSalaryForEmployee&name=Ondrej
- http://localhost:8080/TutorialsDesk.AxisWebService/services/EmployeeService?method=getDepartmentForEmployee&name=Rahul
- http://localhost:8080/TutorialsDesk.AxisWebService/services/EmployeeService?method=getEmployeeData&name=Zuzana

Creating a Console-based Client

Let us see how to access the Web Service using the Axis Client API with a normal Console Client. Following is the code snippet for the same.

```
package com.tutorialsdesk.axis.client;



import java.net.URL;



import org.apache.axis.client.Service;

import org.apache.axis.client.Call;

public class EmployeeServiceClient {



    public static void main(String[] args) {



        try{

            URL url = new
URL("http://localhost:8080/TutorialsDesk.AxisWebService/services/EmployeeS
ervice");



            Service service = new Service();
            Call call  = (Call)service.createCall();

            call.setTargetEndpointAddress(url);

            Object result = call.invoke("getAllEmployees", new
Object[]{});

            System.out.println(result);



            result = call.invoke("getAgeForEmployee", new
Object[]{"Martin"});

            System.out.println(result);



            result = call.invoke("getSalaryForEmployee", new
Object[]{"Ondrej"});

            System.out.println(result);



            result = call.invoke("getDepartmentForEmployee", new
Object[]{"Rahul"});

            System.out.println(result);



        }catch(Exception exception){

            exception.printStackTrace();

        }

    }

}
```

Hope we are able to explain you **Java SOAP Webservices using Axis 2 and Tomcat** , if you have any questions or suggestions please write to us using contact us form.(Second Menu from top left).

REST in Practice by Jim Webber, Savas Parastatidis, Ian Robinson

Chapter 1. The Web As a Platform for Building Distributed Systems

THE WEB HAS RADICALLY TRANSFORMED THE WAY we produce and share information. Its international ecosystem of applications and services allows us to search, aggregate, combine, transform, replicate, cache, and archive the information that underpins today's digital society. Successful despite its chaotic growth, it is the largest, least formal integration project ever attempted—all of this, despite having barely entered its teenage years.

Today’s Web is in large part the human Web: human users are the direct consumers of the services offered by the majority of today’s web applications. Given its success in managing our digital needs at such phenomenal scale, we’re now starting to ask how we might apply the Web’s underlying architectural principles to building other kinds of distributed systems, particularly the kinds of distributed systems typically implemented by “enterprise application” developers.

Why is the Web such a successful application platform? What are its guiding principles, and how should we apply them when building distributed systems? What technologies can and should we use? Why does the Web model feel familiar, but still different from previous platforms? Conversely, is the Web always the solution to the challenges we face as enterprise application developers?

These are the questions we’ll answer in the rest of this book. Our goal throughout is to describe how to build distributed systems based on the Web’s architecture. We show how to implement systems that use the Web’s predominant application protocol, HyperText Transfer Protocol (HTTP), and which leverage REST’s architectural tenets. We explain the Web’s fundamental principles in simple terms and discuss their relevance in developing robust distributed applications. And we illustrate all this with challenging examples drawn from representative enterprise scenarios and solutions implemented using Java and .NET.

The remainder of this chapter takes a first, high-level look at the Web’s architecture. Here we discuss some key building blocks, touch briefly on the REpresentational State Transfer (REST) architectural style, and explain why the Web can readily be used as a platform for connecting services at global scale. Subsequent chapters dive deeper into the Web’s principles and discuss the technologies available for connecting systems in a web-friendly manner.

Architecture of the Web

Tim Berners-Lee designed and built the foundations of the World Wide Web while a research fellow at CERN in the early 1990s. His motivation was to create an easy-to-use, distributed, loosely coupled system for sharing documents. Rather than starting from traditional distributed application middleware stacks, he opted for a small set of technologies and architectural principles. His approach made it simple to implement applications and author content. At the same time, it enabled the nascent Web to scale and evolve globally. Within a few years of the Web's birth, academic and research websites had emerged all over the Internet. Shortly thereafter, the business world started establishing a web presence and extracting web-scale profits from its use. Today the Web is a heady mix of business, research, government, social, and individual interests.

This diverse constituency makes the Web a chaotic place—the only consistency being the consistent variety of the interests represented there; the only unifying factor the seemingly never-ending thread of connections that lead from gaming to commerce, to dating to enterprise administration, as we see in Figure 1-1.
Despite the emergent chaos at global scale, the Web is remarkably simple to understand and easy to use at local scale. As documented by the World Wide Web Consortium (W3C) in its "Architecture of the World Wide Web," the anarchic architecture of today's Web is the culmination of thousands of simple, small-scale interactions between agents and resources that use the founding technologies of HTTP and the URI.[1]

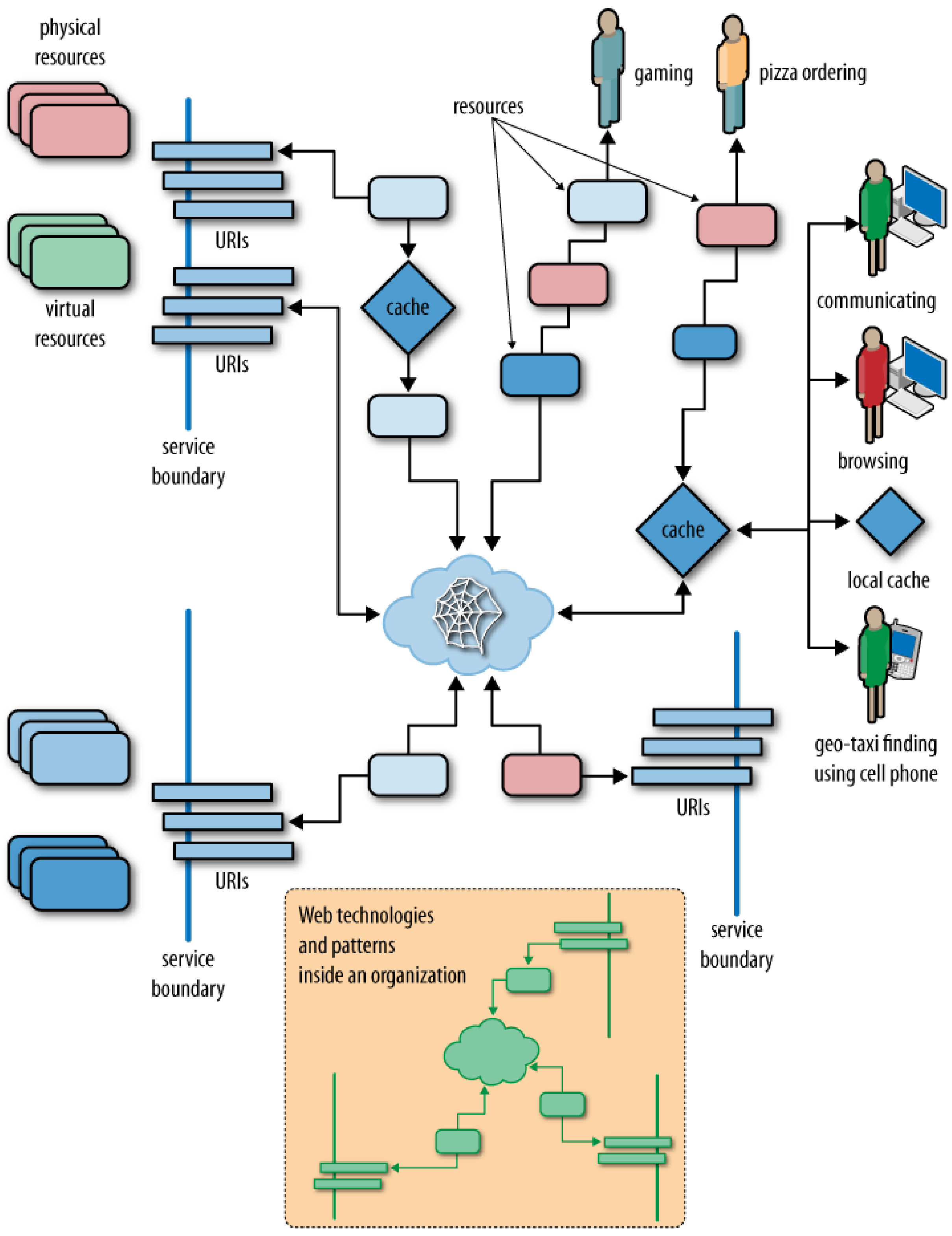


Figure 1-1. The Web

The Web's architecture, as portrayed in Figure 1-1, shows URIs and resources playing a leading role, supported by web caches for scalability. Behind the scenes, service boundaries support isolation and independent evolution of functionality, thereby encouraging loose coupling. In the enterprise, the same architectural principles and technology can be applied.

Traditionally we’ve used middleware to build distributed systems. Despite the amount of research and development that has gone into such platforms, none of them has managed to become as pervasive as the Web is today. Traditional middleware technologies have always focused on the computer science aspects of distributed systems: components, type systems, objects, remote procedure calls, and so on.

The Web’s middleware is a set of widely deployed and commoditized servers. From the obvious—web servers that host resources (and the data and computation that back them)—to the hidden: proxies, caches, and content delivery networks, which manage traffic flow. Together, these elements support the deployment of a planetary-scale network of systems without resorting to intricate object models or complex middleware solutions.

This low-ceremony middleware environment has allowed the Web’s focus to shift to information and document sharing using hypermedia. While hypermedia itself was not a new idea, its application at Internet scale took a radical turn with the decision to allow broken links. Although we’re now nonplussed (though sometimes annoyed) at the classic “404 Page Not Found” error when we use the Web, this modest status code set a new and radical direction for distributed computing: it explicitly acknowledged that we can’t be in control of the whole system all the time.

Compared to classic distributed systems thinking, the Web’s seeming ambivalence to dangling pointers is heresy. But it is precisely this shift toward a web-centric way of building computer systems that is the focus of this book.

Thinking in Resources

Resources are the fundamental building blocks of web-based systems, to the extent that the Web is often referred to as being “resource-oriented." A resource is anything we expose to the Web, from a document or video clip to a business process or device. From a consumer's point of view, a resource is anything with which that consumer interacts while progressing toward some goal. Many real-world resources might at first appear impossible to project onto the Web. However, their appearance on the Web is a result of our abstracting out their useful *information* aspects and presenting these aspects to the digital world. A flesh-and-blood or bricks-and-mortar resource becomes a web resource by the simple act of making the information associated with it accessible on the Web. The generality of the resource concept makes for a heterogeneous community. Almost anything can be modeled as a resource and then made available for manipulation over the network: "Roy's dissertation," "the movie *Star Wars*," "the invoice for the books Jane just bought," "Paul's poker bot," and "the HR process for dealing with new hires" all happily coexist as resources on the Web.

Resources and Identifiers

To use a resource we need both to be able to identify it on the network and to have some means of manipulating it. The Web provides the Uniform Resource Identifier, or URI, for just these purposes. A URI uniquely identifies a web resource, and at the same time makes it addressable, or capable of being manipulated using an application protocol such as HTTP (which is the predominant protocol on the Web). A resource's URI distinguishes it from any other resource, and it's through its URI that interactions with that resource take place.

The relationship between URIs and resources is many-to-one. A URI identifies only one resource, but a resource can have more than one URI. That is, a resource can be identified in more than one way, much as humans can have multiple email addresses or telephone numbers. This fits well with our frequent need to identify real-world resources in more than one way.

There's no limit on the number of URIs that can refer to a resource, and it is in fact quite common for a resource to be identified by numerous URIs, as shown in Figure 1-2. A resource's URIs may provide different information about the location of the resource, or the protocol that can be used to manipulate it. For example, the Google home page (which is, of course, a resource) can be accessed via both http://www.google.com and http://google.com URIs.

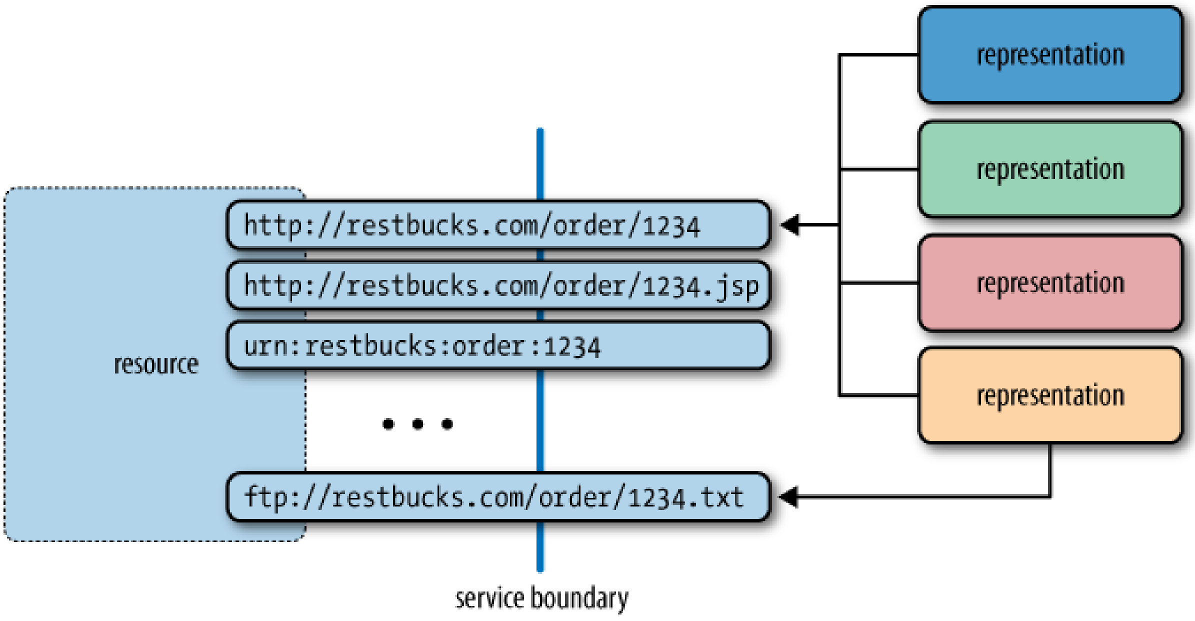


Figure 1-2. Multiple URIs for a resource

NOTE

Although several URIs can identify the same resource, the Web doesn't provide any way to compute whether two different URIs actually refer to the same resource. As developers, we should never assume that two URIs refer to different resources based merely on their syntactic differences. Where such comparisons are important, we should draw on Semantic Web technologies, which offer vocabularies for declaring resource identity sameness. We will discuss some useful techniques from semantic computing later in the book.

A URI takes the form *<scheme>:<scheme-specific-structure>*. The *scheme* defines how the rest of the identifier is to be interpreted. For example, the *http* part of a URI such as *http://example.org/reports/book.tar* tells us that the rest of the URI must be interpreted according to the HTTP scheme. Under this scheme, the URI identifies a resource at a machine that is identified by the hostname *example.org* using DNS lookup. It's the responsibility of the machine "listening" at *example.org* to map the remainder of the URI, *reports/book.tar*, to the actual resource. Any authorized software agent that understands the HTTP scheme can interact with this resource by following the rules set out by the HTTP specification (RFC 2616).

NOTE

Although we're mostly familiar with HTTP URIs from browsing the Web, other forms are supported too. For example, the well-known FTP scheme[2] suggests that a URI such as *ftp://example.org/reports/book.txt* should be interpreted in the following way: *example.org* is the Domain Name System (DNS) name of the computer that knows File Transfer Protocol (FTP), *reports* is interpreted as the argument to the `CWD` (Change Working Directory) command, and *book.txt* is a filename that can be manipulated through FTP commands, such as `RETR` for retrieving the identified file from the FTP server. Similarly, the *mailto* URI scheme is used to identify email addresses: mailto:enquiries@restbucks.com.

The mechanism we can use to interact with a resource cannot always be inferred as easily as the HTTP case suggests; the URN scheme, for example, is not associated with a particular interaction protocol.

In addition to *URI*, several other terms are used to refer to web resource identifiers. Table 1-1 presents a few of the more common terms, including *URN* and *URL*, which are specific forms of URIs, and *IRI*, which supports international character sets.

Table 1-1. Terms used on the Web to refer to identifiers

Term	Comments
URI (Uniform Resource Identifier)	This is often incorrectly referred to as a "Universal" or "Unique" Resource Id
IRI (International Resource Identifier)	This is an update to the definition of *URI* to allow the use of international ch

Term	Comments
URN (Uniform Resource Name)	This is a URI with "urn" as the scheme, used to convey unique names in a pa the URN's structure. For example, a book's ISBN can be captured as a uniqu
URL (Uniform Resource Locator)	This is a URI used to convey information about the way in which one intera example, http://google.com identifies a resource on the Web with which co obsolete, since not all URIs need to convey interaction-protocol-specific inf is still widely in use.
Address	Many think of resources as having "addresses" on the Web and, as a result,

URI VERSUS URL VERSUS URN

URLs and URNs are special forms of URIs. A URI that identifies the mechanism by which a resource may be accessed is usually referred to as a URL. HTTP URIs are examples of URLs.

If the URI has *urn* as its scheme and adheres to the requirements of RFC 2141 and RFC 2611,[3] it is a URN. The goal of URNs is to provide globally unique names for resources.

Resource Representations

The Web is so pervasive that the HTTP URI scheme is today a common synonym for both identity and address. In the web-based solutions presented in this book, we'll use HTTP URIs exclusively to identify resources, and we'll often refer to these URIs using the shorthand term *address*.
Resources must have at least one identifier to be addressable on the Web, and each identifier is associated with one or more *representations*. A representation is a transformation or a view of a resource's state at an instant in time. This view is encoded in one or more transferable formats, such as XHTML, Atom, XML, JSON, plain text, comma-separated values, MP3, or JPEG.

For real-world resources, such as goods in a warehouse, we can distinguish between the actual object and the logical "information" resource encapsulated by an application or service. It's the information resource that is made available to interested parties through projecting its representations onto the Web. By distinguishing between the “real” and the “information” resource, we recognize that objects in the real world can have properties that are not captured in any of their representations. In this book, we’re primarily interested in representations of information resources, and where we talk of a resource or “underlying resource,” it’s the information resource to which we’re referring.

Access to a resource is always mediated by way of its representations. That is, web components *exchange* representations; they never access the underlying resource directly—the Web does not support pointers! URIs relate, connect, and associate representations with their resources on the Web. This separation between a resource and its representations promotes loose coupling between backend systems and consuming applications. It also helps with scalability, since a representation can be cached and replicated.

NOTE

The terms *resource representation* and *resource* are often used interchangeably. It is important to understand, though, that there is a difference, and that there exists a one-to-many relationship between a resource and its representations.

There are other reasons we wouldn’t want to directly expose the state of a resource. For example, we may want to serve different views of a resource’s state depending on which user or application interacts with it, or we may want to consider different quality-of-service characteristics for individual consumers. Perhaps a legacy application written for a mainframe requires access to invoices in plain text, while a more modern application can cope with an XML or JSON representation of the same information. Each representation is a view onto the same underlying resource, with transfer formats negotiated at runtime through the Web’s *content negotiation* mechanism.

The Web doesn’t prescribe any particular structure or format for resource representations; representations can just as well take the form of a photograph or a video as they can a text file or an XML or

JSON document. Given the range of options for resource representations, it might seem that the Web is far too chaotic a choice for integrating computer systems, which traditionally prefer fewer, more structured formats. However, by carefully choosing a set of appropriate representation formats, we can constrain the Web's chaos so that it supports computer-to-computer interactions.

Resource *representation formats* serve the needs of service consumers. This consumer friendliness, however, does not extend to allowing consumers to control how resources are identified, evolved, modified, and managed. Instead, services control their resources and how their states are represented to the outside world. This encapsulation is one of the key aspects of the Web's loose coupling.

The success of the Web is linked with the proliferation and wide acceptance of common representation formats. This ecosystem of formats (which includes HTML for structured documents, PNG and JPEG for images, MPEG for videos, and XML and JSON for data), combined with the large installed base of software capable of processing them, has been a catalyst in the Web's success. After all, if your web browser couldn't decode JPEG images or HTML documents, the human Web would have been stunted from the start, despite the benefits of a widespread transfer protocol such as HTTP.

To illustrate the importance of representation formats, in Figure 1-3 we've modeled the menu of a new coffee store called Restbucks (which will provide the domain for examples and explanations throughout this book). We have associated this menu with an HTTP URI. The publication of the URI surfaces the resource to the Web, allowing software agents to access the resource's representation(s).

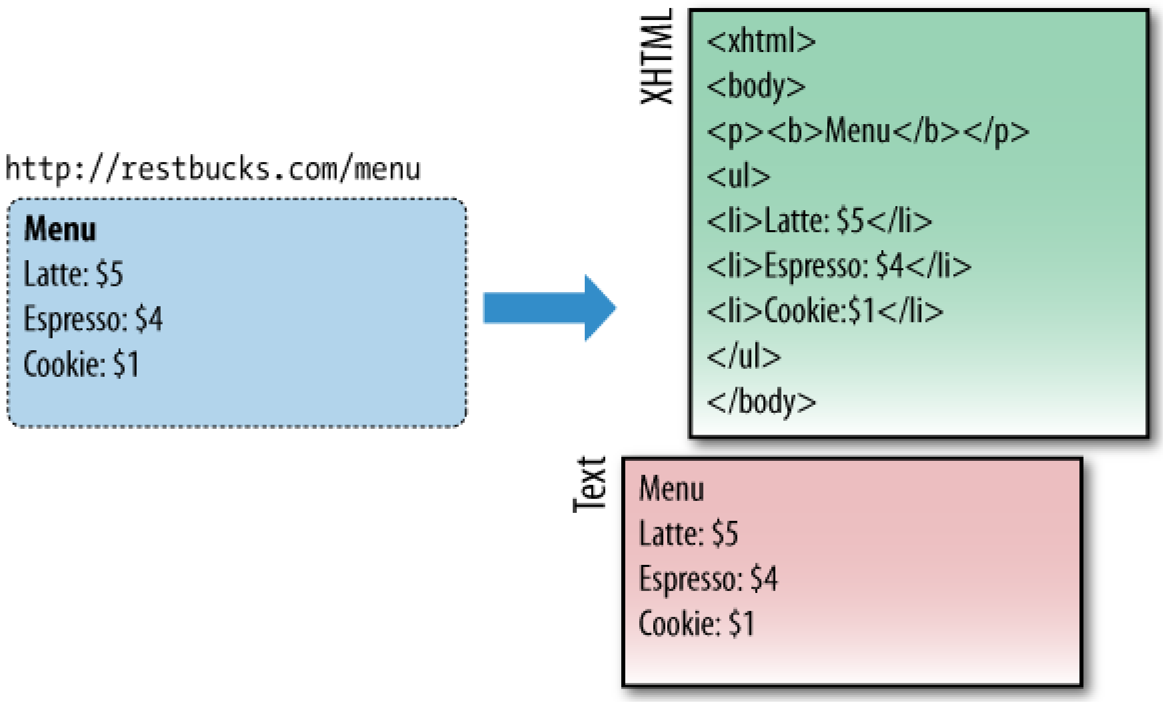


Figure 1-3. Example of a resource and its representations

In this example, we have decided to make only XHTML and text-only representations of the resource available. Many more representations of the same announcement could be served using formats such as PDF, JPEG, MPEG video, and so on, but we have made a pragmatic decision to limit our formats to those that are both human- and machine-friendly.

Typically, resource representations such as those in Figure 1-3 are meant for human consumption via a web browser. Browsers are the most common computer agents on the Web today. They understand protocols such as HTTP and FTP, and they know how to render formats such as (X)HTML and JPEG for human consumption. Yet, as we move toward an era of computer systems that span the Web, there is no reason to think of the web browser as the only important software agent, or to think that humans will be the only active consumers of those resources. Take Figure 1-4 as an example. An order resource is exposed on the Web through a URI. Another software agent consumes the XML representation of the order as part of a business-to-business process. Computers interact with one another over the Web, using HTTP, URIs, and representation formats to drive the process forward just as readily as humans.

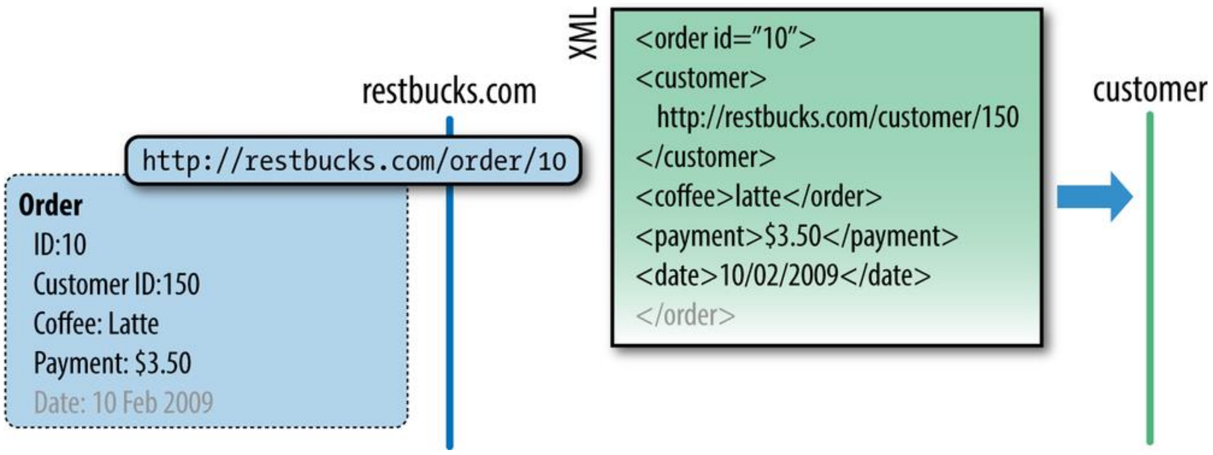


Figure 1-4. Computer-to-computer communication using the Web

Representation Formats and URIs

There is a misconception that different resource representations should each have their own URI—a notion that has been popularized by the Rails framework. With this approach, consumers of a resource terminate URIs with *.xml* or *.json* to indicate a preferred format, requesting http://restbucks.com/order.xml or *http://example.org/order.json* as they see fit. While such URIs convey intent explicitly, the Web has a means of negotiating representation formats that is a little more sophisticated.

NOTE

URIs should be opaque to consumers. Only the issuer of the URI knows how to interpret and map it to a resource. Using extensions such as *.xml, .html*, or *.json* is a historical convention that stems from the time when web servers simply mapped URIs directly to files.

In the example in Figure 1-3, we hinted at the availability of two representation formats: XHTML and plain text. But we didn't specify two separate URIs for the representations. This is because there is a one-to-many association between a URI and its possible resource representations, as Figure 1-5 illustrates.

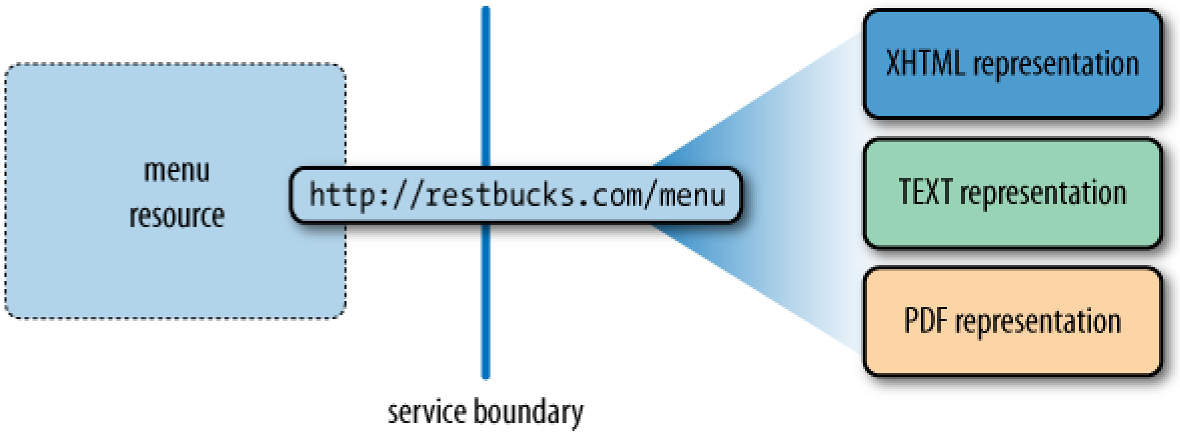


Figure 1-5. Multiple resource representations addressed by a single URI

Using *content negotiation*, consumers can negotiate for specific representation formats from a service. They do so by populating the HTTP `Accept` request header with a list of media types they're prepared to process. However, it is ultimately up to the owner of a resource to decide what constitutes a good representation of that resource in the context of the current interaction, and hence which format should be returned.

The Art of Communication

It's time to bring some threads together to see how resources, representation formats, and URIs help us build systems. On the Web, resources provide the subjects and objects with which we want to interact, but how do we act on them? The answer is that we need verbs, and on the Web these verbs are provided by HTTP methods.[4] The term *uniform interface* is used to describe how a (small) number of verbs with well-defined and widely accepted semantics are sufficient to meet the requirements of most distributed applications. A collection of verbs is used for communication between systems.

NOTE

In theory, HTTP is just one of the many interaction protocols that can be used to support a web of resources and actions, but given its pervasiveness we will assume that HTTP is *the* protocol of the Web.

In contemporary distributed systems thinking, it's a popular idea that the set of verbs supported by HTTP—`GET`, `POST`, `PUT`, `DELETE`, `OPTIONS`, `HEAD`, `TRACE`, `CONNECT`, and `PATCH`—forms a sufficiently general-purpose protocol to support a wide range of solutions.

NOTE

In reality, these verbs are used with differing frequencies on the Web, suggesting that an even smaller subset is usually enough to satisfy the requirements of many distributed applications.

In addition to verbs, HTTP also defines a collection of response codes, such as `200 OK`, `201 Created`, and `404 Not Found`, that coordinate the interactions instigated by the use of the verbs. Taken together, verbs and status codes provide a general framework for operating on resources over the network.

Resources, identifiers, and actions are all we need to interact with resources hosted on the Web. For example, Figure 1-6 shows how the XML representation of an order might be requested and then delivered using HTTP, with the overall orchestration of the process governed by HTTP response codes. We'll see much more of all this in later chapters.

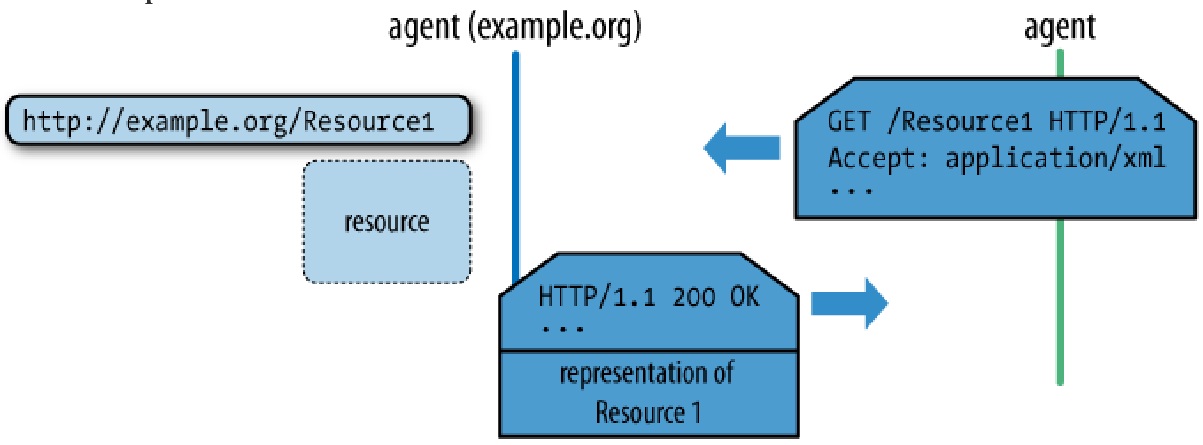


Figure 1-6. Using HTTP to "GET" the representation of a resource

From the Web Architecture to the REST Architectural Style

Intrigued by the Web, researchers studied its rapid growth and sought to understand the reasons for its success. In that spirit, the Web's architectural underpinnings were investigated in a seminal work that supports much of our thinking around contemporary web-based systems.

As part of his doctoral work, Roy Fielding generalized the Web's architectural principles and presented them as a framework of constraints, or an *architectural style*. Through this framework, Fielding described how distributed information systems such as the Web are built and operated. He described the interplay between resources, and the role of unique identifiers in such systems. He also talked about using a limited set of operations with uniform semantics to build a ubiquitous infrastructure that can support any type of application.[5] Fielding referred to this architectural style as *REpresentational State Transfer*, or REST. REST describes the Web as a distributed hypermedia application whose linked resources communicate by exchanging representations of resource state.

Hypermedia

The description of the Web, as captured in W3C's "Architecture of the World Wide Web"[6] and other IETF RFC[7] documents, was heavily influenced by Fielding's work. The architectural abstractions and constraints he established led to the introduction of *hypermedia as the engine of application state*. The latter has given us a new perspective on how the Web can be used for tasks other than information storage and retrieval. His work on REST demonstrated that the Web is an application platform, with the REST architectural style providing guiding principles for building distributed applications that scale well, exhibit loose coupling, and compose functionality across service boundaries.

The idea is simple, and yet very powerful. A distributed application makes forward progress by transitioning from one state to another, just like a state machine. The difference from traditional state machines, however, is that the possible states and the transitions between them are not known in advance. Instead, as the application reaches a new state, the next possible transitions are discovered. It's like a treasure hunt.

NOTE

We’re used to this notion on the human Web. In a typical e-commerce solution such as Amazon.com, the server generates web pages with links on them that corral the user through the process of selecting goods, purchasing, and arranging delivery.

This is hypermedia at work, but it doesn’t have to be restricted to humans; computers are just as good at following protocols defined by state machines.

In a hypermedia system, application states are communicated through representations of uniquely identifiable resources. The identifiers of the states to which the application can transition are embedded in the representation of the current state in the form of *links*. Figure 1-7 illustrates such a hypermedia state machine.

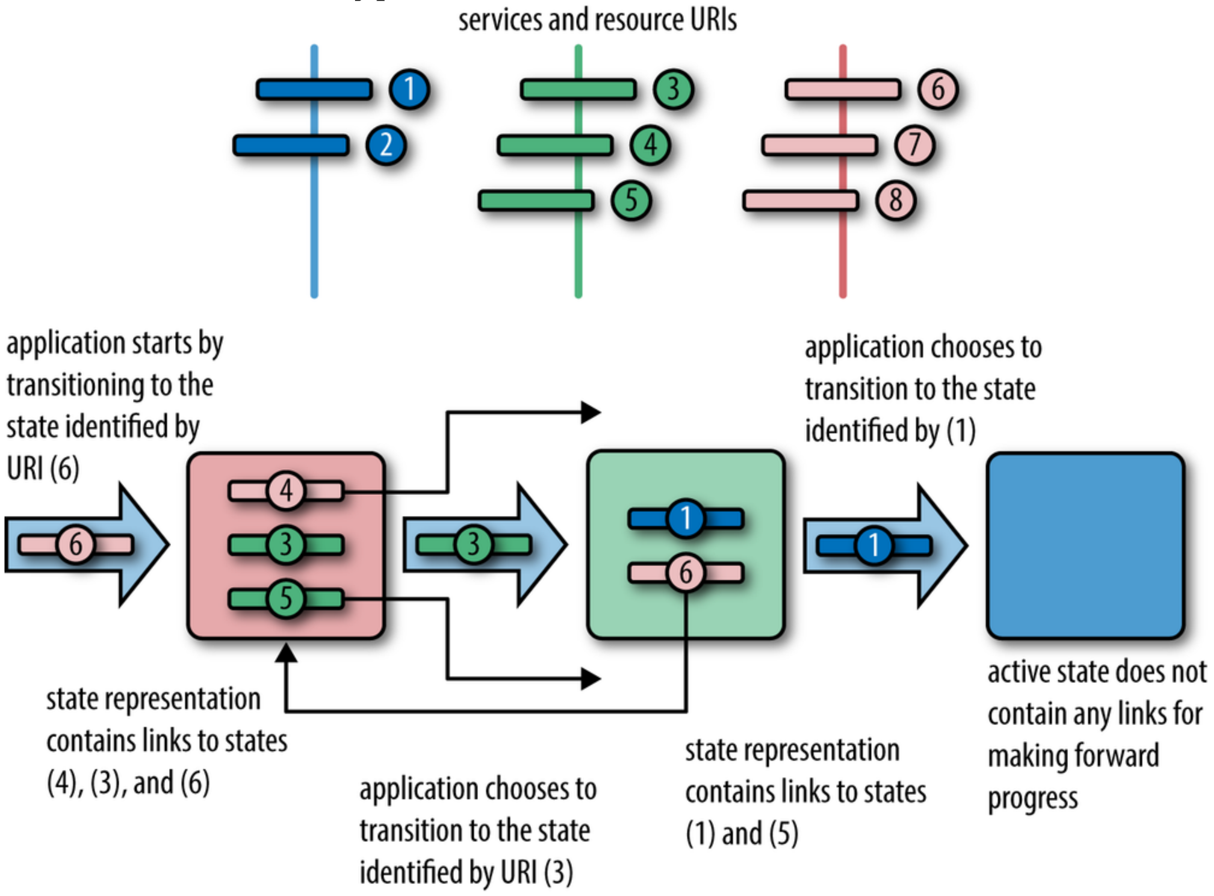

services and resource URIs
application starts by transitioning to the state identified by URI (6)
application chooses to transition to the state identified by (1)
state representation contains links to states (4), (3), and (6)
application chooses to transition to the state identified by URI (3)
state representation contains links to states (1) and (5)
active state does not contain any links for making forward progress

Figure 1-7. Example of hypermedia as the engine for application state in action

This, in simple terms, is what the famous *hypermedia as the engine of application state* or *HATEOAS* constraint is all about. We see it in action every day on the Web, when we follow the links to other pages within our browsers. In this book, we show how the same principles can be used to enable computer-to-computer interactions.

REST and the Rest of This Book

While REST captures the fundamental principles that underlie the Web, there are still occasions where practice sidesteps theoretical guidance. Even so, the term *REST* has become so popular that it is almost impossible to disassociate it from any approach that uses HTTP.[8] It's no surprise that the term *REST* is treated as a buzzword these days rather than as an accurate description of the Web's blueprints.

The pervasiveness of HTTP sets it aside as being special among all the Internet protocols. The Web has become a universal "on ramp," providing near-ubiquitous connectivity for billions of software agents across the planet. Correspondingly, the focus of this book is on the Web as it is used in practice—as a distributed application platform rather than as a single large hypermedia system. Although we are highly appreciative of Fielding's research, and of much subsequent work in understanding web-scale systems, we'll use the term *web* throughout this book to depict a warts-'n-all view, reserving the REST terminology to describe solutions that embrace the REST architectural style. We do this because many of today's distributed applications on the Web do not follow the REST architectural tenets, even though many still refer to these applications as "RESTful."

The Web As an Application Platform

Though the Web began as a publishing platform, it is now emerging as a means of connecting distributed applications. The Web as a platform is the result of its architectural simplicity, the use of a widely implemented and agreed-upon protocol (HTTP), and the pervasiveness of common representation formats. The Web is no longer just a successful large-scale information system, but a platform for an ecosystem of services.

But how can resources, identifiers, document formats, and a protocol make such an impression? Why, even after the dot-com bubble, are we still interested in it? What do enterprises—with their innate tendency toward safe middleware choices from established vendors—see in it? What is new that changes the way we deliver functionality and integrate systems inside and outside the enterprise?

As developers, we build solutions on top of platforms that solve or help with hard distributed computing problems, leaving us free to work on delivering valuable business functionality. Hopefully, this book will give you the information you need in order to make an informed decision on whether the Web fits your problem domain, and whether it will help or hinder delivering your solution. We happen to believe that the Web is a sensible solution for the majority of the distributed computing problems encountered in business computing, and we hope to convince you of this view in the following chapters. But for starters, here are a number of reasons we're such web fans.

Technology Support

An application platform isn't of much use unless it's supported by software libraries and development toolkits. Today, practically all operating systems and development platforms provide some kind of support for web technologies (e.g., .NET, Java, Perl, PHP, Python, and Ruby). Furthermore, the capabilities to process HTTP messages, deal with URIs, and handle XML or JSON payloads are all widely implemented in web frameworks such as Ruby on Rails, Java servlets, PHP Symfony, and ASP.NET MVC. Web servers such as Apache and Internet Information Server provide runtime hosting for services.

Scalability and Performance

Underpinned by HTTP, the web architecture supports a global deployment of networked applications. But the massive volume of blogs, mashups, and news feeds wouldn't have been possible if it wasn’t for the way in which the Web and HTTP constrain solutions to a handful of scalable patterns and practices.

Scalability and performance are quite different concerns. Naively, it would seem that if latency and bandwidth are critical success factors for an application, using HTTP is not a good option. We know that there are messaging protocols with far better performance characteristics than HTTP’s text-based, synchronous, request-response behavior. Yet this is an inequitable comparison, since HTTP is not just another messaging protocol; it’s a protocol that implements some very specific application semantics. The HTTP verbs (and `GET` in particular) support caching, which translates into reduced latency, enabling massive horizontal scaling for large aggregate throughput of work.

NOTE

As developers ourselves, we understand how we can believe that asynchronous message-centric solutions are the most scalable and highest-performing options. However, existing high-performance and highly available services on the Web are proof that a synchronous, text-based request-response protocol can provide good performance and massive scalability when used correctly.

The Web combines a widely shared vision for how to use HTTP efficiently and how to federate load through a network. It may sound incredible, but through the remainder of this book, we hope to demonstrate this paradox beyond doubt.

Loose Coupling

The Web is loosely coupled, and correspondingly scalable. The Web does not try to incorporate in its architecture and technology stack any of the traditional quality-of-service guarantees, such as data consistency, transactionality, referential integrity, statefulness, and so on. This deliberate lack of guarantees means that browsers sometimes try to retrieve nonexistent pages, mashups can’t always access information, and business applications can’t always make immediate progress. Such failures are part of our everyday lives, and the Web is no different. Just like us, the Web needs to know how to cope with unintended outcomes or outright failures.

A software agent may be given the URI of a resource on the Web, or it might retrieve it from the list of hypermedia links inside an HTML document, or find it after a business-to-business XML message interaction. But a request to retrieve the representation of that resource is never guaranteed to be successful. Unlike other contemporary distributed systems architectures, the Web's blueprints do not provide any explicit mechanisms to support information integrity. For example, if a service on the Web decides that a URI is no longer going to be associated with a particular resource, there is no way to notify all those consumers that depend on the old URI–resource association.

This is an unusual stance, but it does not mean that the Web is neglectful—far from it. HTTP defines response codes that can be used by service providers to indicate what has happened. To communicate that "the resource is now associated with a new URI," a service can use the status code `301 Moved Permanently` or `303 See Other`. The Web always tries to help move us toward a successful conclusion, but without introducing tight coupling.

Business Processes

Although business processes can be modeled and exposed through web resources, HTTP does not provide direct support for such processes. There is a plethora of work on vocabularies to capture business processes (e.g., BPEL,[9] WS-Choreography[10]), but none of them has really embraced the Web's architectural principles. Yet the Web—and hypermedia specifically—provides a great platform for modeling business-to-business interactions.

Instead of reaching for extensive XML dialects to construct choreographies, the Web allows us to model state machines using HTTP and hypermedia-friendly formats such as XHTML and Atom. Once we understand that the states of a process can be modeled as resources, it’s simply a matter of describing the transitions between those resources and allowing clients to choose among them at runtime.

This isn’t exactly new thinking, since HTML does precisely this for the human-readable Web through the `<a href= “…” >` tag. Although implementing hypermedia-based solutions for computer-to-computer systems is a new step for most developers, we’ll show you how to embrace this model in your systems to support loosely coupled business processes (i.e., behavior, not just data) over the Web.

Consistency and Uniformity

To the Web, one representation looks very much like another. The Web doesn’t care if a document is encoded as HTML and carries weather information for on-screen human consumption, or as an XML document conveying the same weather data to another application for further processing. Irrespective of the format, they’re all just resource representations.

The principle of uniformity and least surprise is a fundamental aspect of the Web. We see this in the way the number of permissible operations is constrained to a small set, the members of which have well-understood semantics. By embracing these constraints, the web community has developed myriad creative ways to build applications and infrastructure that support information exchange and application delivery over the Web.

Caches and proxy servers work precisely because of the widely understood caching semantics of some of the HTTP verbs—in particular, `GET`. The Web’s underlying infrastructure enables reuse of software tools and development libraries to provide an ecosystem of middleware services, such as caches, that support performance and scaling. With plumbing that understands the application model baked right into the network, the Web allows innovation to flourish at the edges, with the heavy lifting being carried out in the cloud.

Simplicity, Architectural Pervasiveness, and Reach

This focus on resources, identifiers, HTTP, and formats as the building blocks of the world's largest distributed information system might sound strange to those of us who are used to building distributed applications around remote method invocations, message-oriented middleware platforms, interface description languages, and shared type systems. We have been told that distributed application development is difficult and requires specialist software and skills. And yet web proponents constantly talk about simpler approaches.

Traditionally, distributed systems development has focused on exposing custom behavior in the form of application-specific interfaces and interaction protocols. Conversely, the Web focuses on a few well-known network actions (those now-familiar HTTP verbs) and the application-specific interpretation of resource representations. URIs, HTTP, and common representation formats give us reach—straightforward connectivity and ubiquitous support from mobile phones and embedded devices to entire server farms, all sharing a common application infrastructure.

Web Friendliness and the Richardson Maturity Model

As with any other technology, the Web will not automatically solve a business's application and integration problems. But good design practices and adoption of good, well-tested, and widely deployed patterns will take us a long way in our journey to build great web services.

You'll often hear the term *web friendliness* used to characterize good application of web technologies. For example, a service would be considered "web-friendly" if it correctly implemented the semantics of HTTP `GET` when exposing resources through URIs. Since `GET` doesn't make any service-side state changes that a consumer can be held accountable for, representations generated as responses to `GET` *may* be cached to increase performance and decrease latency.

Leonard Richardson proposed a classification for services on the Web that we'll use in this book to quantify discussions on service maturity.[11] Leonard's model promotes three levels of service maturity based on a service's support for URIs, HTTP, and hypermedia (and a fourth level where no support is present). We believe this taxonomy is important because it allows us to ascribe general architectural patterns to services in a manner that is easily understood by service implementers.

The diagram in Figure 1-8 shows the three core technologies with which Richardson evaluates service maturity. Each layer builds on the concepts and technologies of the layers below. Generally speaking, the higher up the stack an application sits, and the more it employs instances of the technology in each layer, the more mature it is.

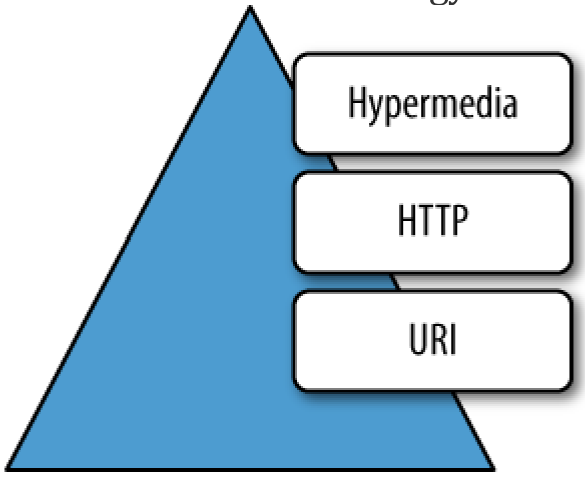


Figure 1-8. The levels of maturity according to Richardson's model

Level Zero Services

The most basic level of service maturity is characterized by those services that have a single URI, and which use a single HTTP method (typically POST). For example, most Web Services (WS-*)-based services use a single URI to identify an endpoint, and HTTP POST to transfer SOAP-based payloads, effectively ignoring the rest of the HTTP verbs.[12]

NOTE

We can do wonderful, sophisticated things with WS-*, and it is not our intention to imply that its level zero status is a criticism. We merely observe that WS-* services do not use many web features to help achieve their goals.[13]

XML-RPC and Plain Old XML (POX) employ similar methods: HTTP POST requests with XML payloads transmitted to a single URI endpoint, with replies delivered in XML as part of the HTTP response. We will examine the details of these patterns, and show where they can be effective, in Chapter 3.

Level One Services

The next level of service maturity employs many URIs but only a single HTTP verb. The key dividing feature between these kinds of rudimentary services and level zero services is that level one services expose numerous logical resources, while level zero services tunnel all interactions through a single (large, complex) resource. In level one services, however, operations are tunneled by inserting operation names and parameters into a URI, and then transmitting that URI to a remote service, typically via HTTP GET.

NOTE

Richardson claims that most services that describe themselves as “RESTful” today are in reality often level one services. Level one services can be useful, even though they don’t strictly adhere to RESTful constraints, and so it’s possible to accidentally destroy data by using a verb (GET) that should not have such side effects.

Level Two Services

Level two services host numerous URI-addressable resources. Such services support several of the HTTP verbs on each exposed resource. Included in this level are Create Read Update Delete (CRUD) services, which we cover in Chapter 4, where the state of resources, typically representing business entities, can be manipulated over the network. A prominent example of such a service is Amazon’s S3 storage system.

NOTE

Importantly, level two services use HTTP verbs and status codes to coordinate interactions. This suggests that they make use of the Web for robustness.

Level Three Services

The most web-aware level of service supports the notion of hypermedia as the engine of application state. That is, representations contain URI links to other resources that might be of interest to consumers. The service leads consumers through a trail of resources, causing application state transitions as a result.

NOTE

The phrase *hypermedia as the engine of application state* comes from Fielding's work on the REST architectural style. In this book, we'll tend to use the term *hypermedia constraint* instead because it's shorter and it conveys that using hypermedia to manage application state is a beneficial aspect of large-scale computing systems.

GET on Board

Can the same principles that drive the Web today be used to connect systems? Can we follow the same principles driving the human Web for computer-to-computer scenarios? In the remainder of this book, we will try to show why it makes sense to do exactly that, but first we'll need to introduce our business domain: a simple coffee shop called Restbucks.