Article *Tutorial covering JSP 2.2 and Servlets 3.0 with OpenSource Resin Servlet Container: Part 1* has been updated.

View Edit

Bill Digman
Bio

Tutorial covering JSP 2.2 and Servlets 3.0 with OpenSource Resin Servlet Container: Part 1

Java EE: Servlet 3.0 and JSP 2.2 tutorial

This tutorial focuses on using Servlet's and JSP the right way. Servlet and JSP have evolved over the years, and now there is often more than one way to do things. For example, this tutorial uses EL and JSTL not JSP scriptlets, it uses JSPs in a Model 2/MVC style not in a model 1 style, etc. Consider it a tutorial that focuses only on the best practices and not the legacy ways to do things.

There are other tutorials on this JSP and Servlets, but this tutorial is going to be different in that you can follow along with Eclipse. Also instead of focusing on JSF, we are going to focus on JSP and Servlets as the main view technology.

Java EE, JSP and Servlets have added a lot of features that are in other frameworks, making those other frameworks less relevant then they were before Java EE garnered these extra abilities. Even is you decide to use JSF, Struts, Stripes, Spring MVC, JSF, etc., this tutorial should help you have a better understanding of the JSP/Servlets core that they build on.

We are going to start by building a simple bookstore. We will progressively add more features to the bookstore and as we do we will use more of Java EE/CDI, JSP and Servlets. For this tutorial, I am going to use Resin 4.0.x, but you could use one of several Servlet/JSP Containers that support CDI.

Tutorial Style

1. Not going to list every option of every tag, configuration, etc. (maybe later).
2. Not going to teach you something you are suppose to avoid anyway or that is deprecated
3. Build something real enough to ensure that things actually work and we don't miss something common
4. Don't build something too real so that the concepts are hard to ascertain outside of the domain of the example
5. Provide cook books on how to do common tasks like I18n, etc.
6. IDE, Performance testing, Debugging, etc. from the start!

I can go on and on about each point. I've seen a lot of tutorial and books out there, and they drag you down with detail you don't need. (You can alway google it later). Or, they skip out very important things because their examples are too simple.

1. Building a simple listing in JSP: covers model 2, Servlets, JSP intro.
2. Java EE Servlet tutorial: Adding create, update and delete to the bookstore listing: covers more interactions.
3. Java EE Servlet tutorial: Using JSPs to create header, footer area, formatting, and basic CSS for bookstore
4. Java EE Servlet tutorial: Adding MySQL and JDBC to bookstore example
5. Java EE Servlet/JSP tutorial: Adding an error page, logging, and other forms of_debugging

I can go on and on about each point. I've seen a lot of tutorial and books out there, and they drag you down with detail you don't need. (You can alway google it later). Or, they skip out very important things because their examples are too simple.

Java EE Servlet/JSP tutorial: Building a simple listing in JSP

This cookbook in the Java EE Servlet tutorial covers building a simple listing in JSP and Servlets. This tutorial is part of Java EE Tutorial covering JSP_2.2, and Servlets 3.0.
This cookbook assumes very little knowledge of HTML, Java and JSP. It does not cover HTML and Java at length, but it does point you in the right direction. You will find that you can get started with Java Servlets and JSP (Java Server Pages) quite easily.
Feel free to use whatever IDE you would like. Eclipse is the 800 pound gorilla in the Java space so we have some specific instructions to help you get started with Eclipse.
If you are new to Java and/or Java development, I suggest starting with Eclipse. It is the dominant IDE mindshare wise.
Any IDE that supports Java EE will support creating a war file. The war file is a binary distribution file that you can copy to the Java EE server, and the server will automatically deploy your web application (war is short for web application archive file).

Contents

- Setting up CDI
  - CDI META-INF/beans.xml
- Deploying with Eclipse
- Deploying from the command line
- Cookbooks and Tutorials

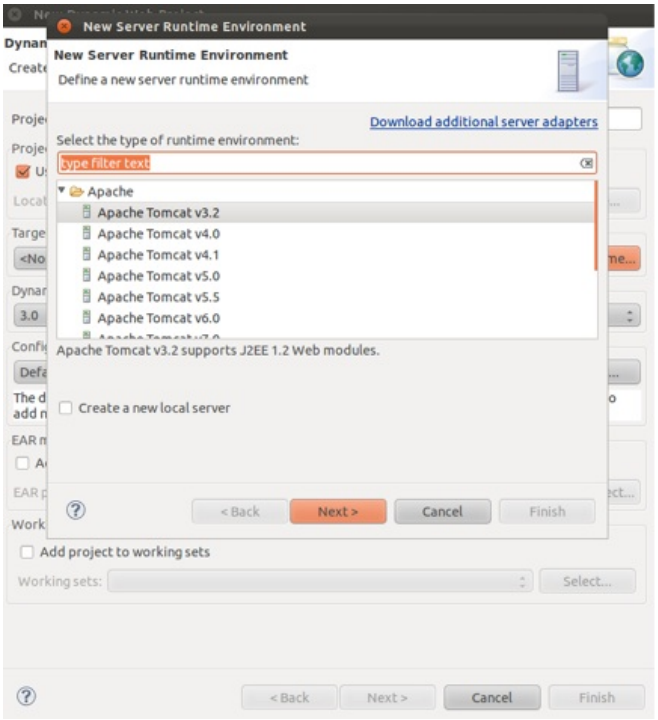
Optional: Getting started Eclipse Java EE IDE

First go here to get Eclipse for Java EE: Install Guide for Eclipse for Java EE Indigo or higher.

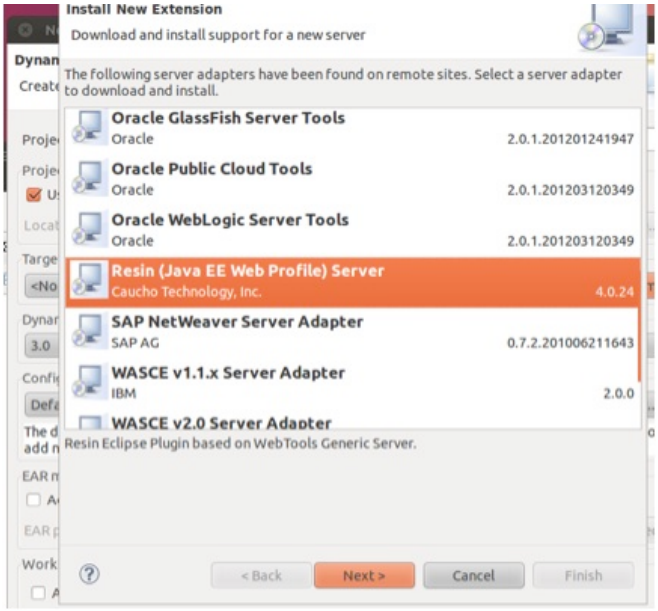
Using Eclipse Indigo or higher

Install Resin (lightweight, fast, easy to use Java EE 6 Web Profile server) plugin:

1. Go to File menu -> New Project -> Dynamic Web Project
2. In New Dynamic Web Project Dialog-> New Runtime...
3. In New Runtime Dialog -> Download Additional Server Adapters -> Select Resin (Java EE Web Profile) 4.0.x



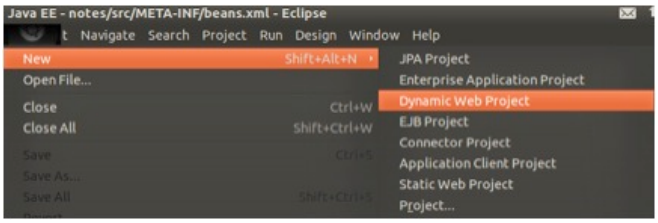
4. (Click Next and Ok until it installs Resin runtime)



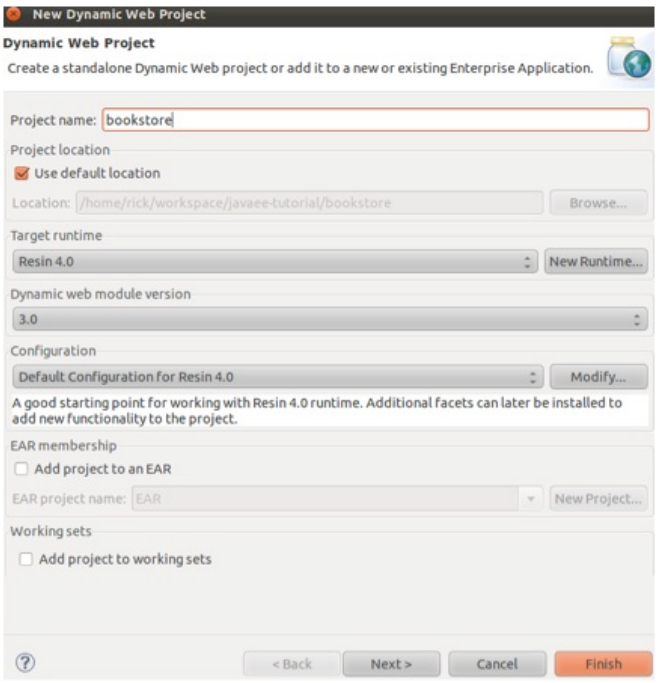
5. (Eclipse needs to restart)

Setup new Web Project in Eclipse

1. File -> New Project -> Dynamic Web Project



2. In New Dynamic Web Project Dialog-> New Runtime...->Select Resin->Check create local server checkbox
3. Step 2 of New Runtime...->Click Download and Install (you only have to do this once)
4. Fill out project name etc. (bookstore).



Resources

For each cookbook of the tutorial our goal is to provide a slide deck, and video to accompany the wiki page. The idea is between the three resources (wiki, slide deck and video or videos), you might find what you are looking for if you missed something. I don't recommend skipping this slidedeck as it is usually done after the wiki and I catch stuff (concepts) that I missed in the text.

- Slide Deck I Java EE Servlet/JSP Tutorial- Cookbook 1
- YouTube Video I Java EE Servlets/JSP Tutorial - Cookbook 1 - Part 1
- YouTube Video I Java EE Servlets/JSP Tutorial - Cookbook 1 - Part 2

What is going to be in the project

The project will be a basic CRUD (create read update delete) listing for a BookStore.

Creating model and Repository objects for books

Create a new Java class as follows:

Eclipse: Right Click "Java Resources" in Project Explorer -> New -> Class -> Package com.bookstore -> Class Name -> Book (That is the last time I will tell you how to create a class in Eclipse.)

Add title, description, price, pubDate and id properties, and the toString and cloneMe methods as follows:

Book model class

```
01. package com.bookstore;
02.
03. import java.math.BigDecimal;
04. import java.util.Date;
05.
06. public class Book implements Cloneable {
07.
08.    private String title;
09.    private String description;
10.    private BigDecimal price;
11.    private Date pubDate;
12.    private String id;
13.
14.    public Book(String id, String title, String description, BigDecimal
          price, Date pubDate) {
15.       this.id = id;
16.       this.title = title;
17.       this.description = description;
18.       this.price = price;
19.       this.pubDate = pubDate;
20.
21.    }
22.
23.    public Book () {
24.
25.    }
26.
27.    public String getTitle() {
28.       return title;
29.
30.    }
31.
32.    public void setTitle(String title) {
33.       this.title = title;
```

```
    }

    public String getDescription() {
        return description;

    }

    public void setDescription(String description) {
        this.description = description;

    }

    public BigDecimal getPrice() {
        return price;

    }

    public void setPrice(BigDecimal price) {
        this.price = price;

    }

    public Date getPubDate() {
        return pubDate;

    }

    public void setPubDate(Date pubDate) {
        this.pubDate = pubDate;

    }

    public String getId() {
        return id;

    }

    public void setId(String id) {
        this.id = id;

    }

    public Book cloneMe() {
        try {
            return (Book) super.clone();
        } catch (CloneNotSupportedException e) {
            return null;
        }
    }

    @Override
    public String toString() {
        return "Book [title=" + title + ", description=" + description
            + ", price=" + price + ", pubDate=" + pubDate + ", id=" + id
            + "]";
    }


}
```

Are you new to Java?

If you are new to Java and the above seems foreign to you, I suggest you read up on Java a bit Official Java tutorial. Read the first three trails (Getting Started, Learning the Java Language and Essential Java Classes), and then skip ahead to Java Beans. Skimming is ok. If you have programmed before, most things you will pick up on your own.

Reminder: The methods public String getId() and public void setId(String id) would define a property called id.

just going to use the collection API to create a Repository object (some people call this a DAO -- data access object). The Repository object encapsulates how an object gets persisted, queried and updated (CRUD operations).

We know we are going to later use JDBC/RDBMS (MySQL), JTA/RDBMS, JCache, MongoDB, etc. instead of the collection API so let's define an interface so we can swap these in quickly as we get to these tutorials. The interface will define the contract with our Repository object that we can later swap out with other implementations.

Eclipse: Right Click "Java Resources" in Project Explorer -> New -> Interface -> Package = com.bookstore -> Class Name = Book -> Click Finish. (That is the last time I will tell you how to create an interface in Eclipse.)

BookRepository interface

```
01. package com.bookstore;
02.
03. import java.util.List;
04.
05. public interface BookRepository {
06.    Book lookupBookById(String id);
07.
08.    void addBook(String title, String description,
09.       String price, String pubDate);
10.
11.    void updateBook(String id, String title,
12.       String description, String price, String pubDate);
13.
14.    void removeBook(String id);
15.
16.
17.    List<Book> listBooks();
18.
19. }
```

Next create a class called BookRepositoryImpl as follows (don't study it too much unless you want to, it just simulates access to database):

```
001. package com.bookstore;
002.
003. import java.text.SimpleDateFormat;
004. import java.util.ArrayList;
005. import java.util.Collections;
006. import java.util.Comparator;
007. import java.util.Date;
008. import java.util.HashMap;
009. import java.util.List;
010. import java.util.Map;
011. import java.math.BigDecimal;
012.
013. import javax.enterprise.context.ApplicationScoped;
014.
015. @ApplicationScoped
016. public class BookRepositoryImpl implements BookRepository {
017.
018.    private SimpleDateFormat dateFormat = new
          SimpleDateFormat("MM/dd/yyyy");
019.    private int count;
020.    private Map<String, Book> idToBookMap = new HashMap<String, Book>();
021.
022.    public BookRepositoryImpl() {
023.       synchronized (this) {
024.          books(book("War and Peace", "blah blah blah", "5.50",
             "5/29/1970"),
025.             book("Pride and Prejudice", "blah blah blah", "5.50",
                "5/29/1960"),
```

```
            book("book3", "blah blah blah", "5.50", "5/29/1960"),
            book("book4", "blah blah blah", "5.50", "5/29/1960"),
            book("book5", "blah blah blah", "5.50", "5/29/1960"),
            book("book6", "blah blah blah", "5.50", "5/29/1960"),
            book("book7", "blah blah blah", "5.50", "5/29/1960"),
            book("book8", "blah blah blah", "5.50", "5/29/1960"),
            book("book9", "blah blah blah", "5.50", "5/29/1960"),
            book("Java for dummies", "blah blah blah", "1.99",
              "5/29/1960"));
      }
   }

   private Book book(String title, String description, String aPrice,
         String aPubDate) {

      Date pubDate = null;
      BigDecimal price = null;

      try {
         price = new BigDecimal(aPrice);
      }catch (Exception ex) {
      }

      try {
         pubDate = dateFormat.parse(aPubDate);
      }catch (Exception ex) {
      }

      return new Book("" + (count++), title, description, price,
        pubDate);

   }

   private void books(Book... books) {
      for (Book book : books) {
         doAddBook(book);
      }
   }

   private void doAddBook(Book book) {
      synchronized (this) {
         this.idToBookMap.put(book.getId(), book);
      }
   }

   @Override
   public Book lookupBookById(String id) {
      synchronized (this) {
         return this.idToBookMap.get(id).cloneMe();
      }
   }

   @Override
   public void addBook(String title, String description, String price,
         String pubDate) {
      doAddBook(book(title, description, price, pubDate));
   }

   @Override
   public void updateBook(String id, String title, String description,
         String price, String pubDate) {
      Book book = book(title, description, price, pubDate);
      synchronized (this) {
         book.setId(id);
         this.idToBookMap.put(id, book);
      }
   }

   private List<Book> doListBooks() {
      List<Book> books;
```

```
098.          books = new ArrayList<Book>(this.idToBookMap.size());
099.          for (Book book : this.idToBookMap.values()) {
100.             books.add(book.cloneMe());
101.          }
102.       }
103.       return books;
104.    }
105.
106.    public List<Book> listBooks() {
107.
108.       List<Book> books = doListBooks();
109.
110.       Collections.sort(books, new Comparator<Book>() {
111.          public int compare(Book bookA, Book bookB) {
112.             return bookA.getId().compareTo(bookB.getId());
113.          }
114.       });
115.       return books;
116.    }
117.
118.    @Override
119.    public void removeBook(String id) {
120.       synchronized(this) {
121.          this.idToBookMap.remove(id);
122.       }
123.    }
124. }
```

This BookRepositoryImpl is a fairly basic class. It is largely based on the collections API. You can find out more information about the Java collections API at this tutorial trail. A full discussion of the collection API is out of scope for this tutorial, and this class is mainly just for testing, later we will store items in the databases and such.

One interesting thing to note is the use of this new Java EE annotation @ApplicationScoped as follows:

```
1. import javax.enterprise.context.ApplicationScoped;
2.
3. @ApplicationScoped
4. public class BookRepositoryImpl implements BookRepository {
```

Annotations allow you to add meta-data to Java objects. (To learn more about annotations see this annotations tutorial.).

The ApplicationScoped specifies that a bean is application scoped. Scoping defines a lifecycle for how long an object will be around. ApplicationScoped means it will be around for the complete lifecycle of the Web Application. This annotations has slightly different meanings depending on whether it is used with EJBs or Servlets. This annotation is part of the CDI support added to Java EE 6 to handle Java Dependency Injection. To learn more about CDI go see this tutorial Java Dependency Injection and this one part 2 both written by the same author that is writing this tutorial now. :)

Now that we have a model (Book, BookRepository), lets define our web controller (Servlet) and view (JSPs).

Servlets / JSP

Servlet Background

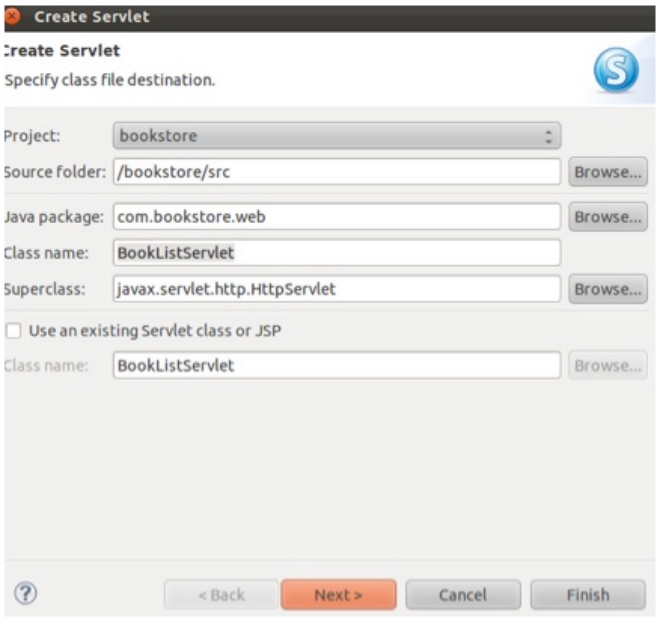
A servlet is a class that handles HTTP requests. A Java web application consists of one or more servlet bundled together with any JSPs and Java classes it needs into a war file (Web Application Archive file).

Servlets run inside of a container like Caucho's Resin Servlet/JSP Container. End users typically use a Java Web Application through a web browser like Apple Safari, Google Chrome, Mozilla FireFox or heaven forbid Internet Explorer.

Next we will create our first Servlet.

Creating your first Servlet

Eclipse: Right Click "Java Resources" in Project Explorer -> New -> Servlet -> Package = com.bookstore.web -> Class Name = BookListServlet -> Click Next -> Remove URL mapping, create new mapping /book/



(That is the last time I will tell you how to create a Servlet in Eclipse.)

Note: This application uses the REST style URL mappings so things that end in / imply you are working with a list (like a directory of files). Thus the URI /book/ implies a collection of books since we want to show a list of books this is a good URI.

Modify the Servlet to only handle the doGet method, change the doGet method to forward to a JSP that we have not created yet that lives in WEB-INF.

BookListServlet listing

```
import java.io.IOException;


import javax.inject.Inject;
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 com.bookstore.BookRepository;

@WebServlet("/book/")
public class BookListServlet extends HttpServlet {

   @Inject
   private BookRepository bookRepo;

   protected void doGet(HttpServletRequest request, HttpServletResponse
      response) throws ServletException, IOException {
      request.setAttribute("books", bookRepo.listBooks());
      getServletContext().getRequestDispatcher("/WEB-INF/pages/book-
         list.jsp").forward(request, response);
   }

}
```

WEB-INF is a meta directory folder for war files. JSPs in the WEB-INF folder can never be loaded directly from a browser. This allows us to force all JSPs to first go through our Servlet tier (controllers) which is essential for a model 2 architecture (a form of MVC tailored for HTTP applications), which we will use throughout the tutorial. Consider putting JSPs in WEB-INF a best practice.

The annotations @WebServlet("/book/") allow us to map this Servlet to handle requests for the URI /book/ as follows:

```
...

@WebServlet("/book/")
public class BookListServlet extends HttpServlet {
```

This WebServlet annotation is a new feature of Servlets 3.0, prior to this, all Servlet configuration went in WEB-INF/web.xml. Anything that avoids a lot of XML configuration is a good thing. Also, the annotation puts the configuration near the things it configuring making it easier to read and more cohesive to understand.

This Servlet is going to use the repository model object to look up a list of BookRepository books. We used Java dependency injection to inject the BookRepository into the Servlet with the @Inject annotation as follows:

```
@Inject
private BookRepository bookRepo;
```

The Servlet is typically used as a controller. It talks to the model (BookRepository to get a list of Book),

The doGet method gets called when somebody loads the page from the browser and corresponds to the HTTP GET.

The doGet method uses the request object (HttpServletRequest request) to put the list of books into **request scope** using setAttribute as follows:

```
    protected void doGet(HttpServletRequest request, HttpServletResponse
       response) throws ServletException, IOException {
        request.setAttribute("books", bookRepo.listBooks());
        getServletContext().getRequestDispatcher("/WEB-INF/pages/book-
           list.jsp").forward(request, response);
    }
```

Java EE / Servlets scopes (page, request, conversation, session, application)

Servlets and Java EE have various **scopes** as follows (from least to greatest): page, request, conversation, session, application. Page scope is around for just for that page (10 to 200 miliseconds). Request scope is around for one HTTP request (1/2 second to several seconds). Session scope is around for the entire user session (5 minutes to 90 minutes). Conversation scope is around for one workflow (user registration, shopping cart, etc.). Application scope is around from the time that the application server starts the web application until it shuts it down (days, months, years, good for helper classes and reference/config data). Scopes are buckets to put your objects in. When the scope lifecycle ends, it gets rid of all the objects in the scope. This allows you to put objects in a location where multiple resources (Servlets, JSPs, Tag files, etc.) can access them to render pages.

By putting the books list into **request scope**, we make it available for the JSP page to access the book list to render the book listing. You could use the response object (HttpServletResponse response), and render the listing directly, but the code would be ugly and hard to change the HTML. Instead, we are going to dispatch the request to the JSP to actually render the listing.

After the doGet method puts books into request scope, it forwards the rest of the rendering to the book-list.jsp as follows:

```
getServletContext().getRequestDispatcher("/WEB-INF/pages/boo
   list.jsp").forward(request, response);
```

The book-list JPS uses Unified EL and JSTL to render the books as HTML to the end user.

Creating your first JSP

JSP pages are also Servlets. JSP is a templating language to define Servlets that allows you to focus on the HTML instead of the Java code. A JSP is like a Servlet turned inside out. Essentially a JSP page is translated and compiled into a servlet. JSP is similar to ASP and PHP in concept and scope. ASP predates JSP and early JSP and ASP use a lot of the same concepts (JSP is a bit like a Java clone of ASP, ASP was a reaction to Cold Fusion, PHP was the first Open Source Cold Fusion like thing so you could say that they are all cousins). JSP is closest in concept to ASP.

Templates/JSTL versus Java scriptlets

JSP allows you to freely mix and match Java code and HTML. However, that is called Java Scriptlets and that is frowned upon.

JSP started to adopt more of a classic templating approach Freemarker, Velocity and Smarty (PHP based templating) approach to templating. Those templating engines don't allow mixing the programming language with the templating language. This allows the templating language to be a simple view logic language, and keeps the templates smaller and more readable.

JSP uses JSTL and the Unified EL to accomplish things that Smarty, Freemarker and Velocity accomplish. We will use this JSTL/EL approach because it is consider a best practice and it makes the code easier to read.

/WEB-INF/pages/book-list.jsp listing

```
   pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix ="c" %>
<!DOCTYPE HTML>

<html>
<head>
<title>Book listing</title>
</head>
<body>


<table>
   <tr>
      <th>Title</th>
      <th>Description</th>
      <th>Price</th>
      <th>Publication Date</th>
      </tr>

   <c:forEach var="book" items="${books}">
      <tr>
         <td>${book.title}</td>
         <td>${book.description}</td>
         <td>${book.price}</td>
         <td>${book.pubDate}</td>
      </tr>
   </c:forEach>
</table>

</body>
</html>
```

Let's break this down some. First we setup the page using this JSP page directive:

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

The above just says how we want the characters encoded and what the mime type of the page is. Consider it boiler plate for now.

Next we import the JSTL core library, under the tag c as follows:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix ="
```

HTML background

Most of the page is boiler plate and simple HTML. If you are new to HTML, try this HTML tutorial.

JSTL c:forEach

The part of the page defines a table and uses the JSTL tag c:forEach to iterate through the books and display them as follows:

```
<c:forEach var="book" items="${books}">
   <tr>
      <td>${book.title}</td>
      <td>${book.description}</td>
      <td>${book.price}</td>
      <td>${book.pubDate}</td>
   </tr>
</c:forEach>
```

The above literally says iterate over the list of books and render a new row for each book in books. Then it creates a column in each row and outputs the values of the book properties into each column for each property (the title property, the description property, the price property, the pubDate property) to each column.

The syntax ${books}, ${book.title}, ${book.price}, ${book.pubDate} is Unified EL. Unified EL + JSTL

designers (this has been my experience anyway).

${books} is the same books that we put into request scope in the BookListServlet.doGet method.

```
1. //doGet method
2.       request.setAttribute("books", bookRepo.listBooks());
```

Setting up CDI

To setup CDI, you need to create a beans.xml file and put it in META-INF. META-INF is a special directory for jar files and Java classpath entries. It contains meta information. The beans.xml can be completely blank, i.e., you could create it like this:

```
1. $ pwd
2. ~/workspace/javaee-tutorial/notes/src/META-INF
3.
4. $ touch beans.xml
```

To get Eclipse to quit complaining about the blank file not having the right format, I went ahead and created boiler plate beans.xml file as follows:

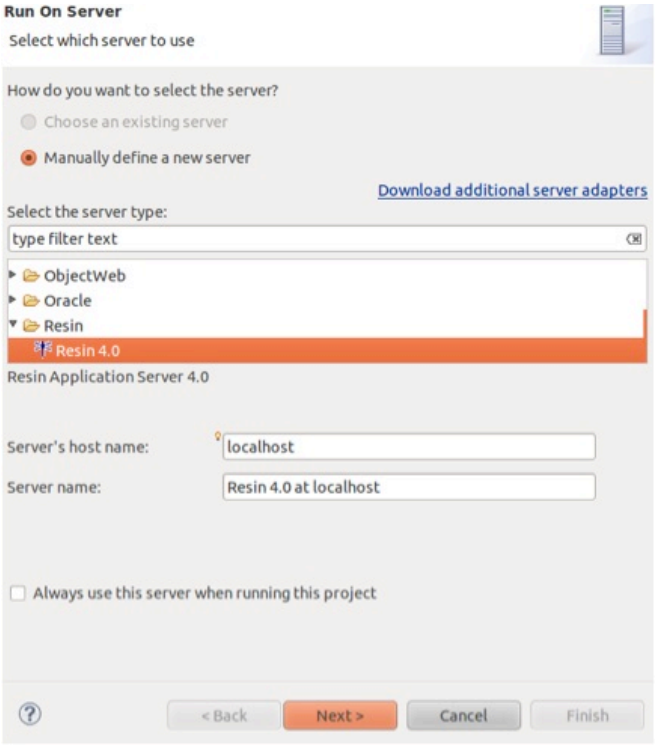
CDI META-INF/beans.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://java.sun.com/xml/ns/javaee"
3.   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
       http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
5. </beans>
```

Note: the @Inject Java dependency injection will not work without this file.

Deploying with Eclipse

1. Right click the BookListServlet in the Project Explorer
2. Choose Run As->Run On Server...
3. "Select Manually Define a New Server"
4. Choose Resin->Resin 4.0 from the Server List



5. Click "Download additional Server adapters"

Deploying from the command line

1. Do a standard install for your operating system.
2. Export the war file from your project (In Eclipse Right click project -> Export -> WAR file).
3. Copy war file to {$resin_root}/webapps (on Unix this for a standard install this is /var/www/webapps).
4. Start Resin. (on Unix this is sudo /etc/init.d/resin start or resinctl start)
5. With your favorite browser go to URL http://localhost:8080/bookstore/book/.

This should load the book page which will look like the last two screenshots.

Bill Digman is a Java EE / Servlet enthusiast and Open Source enthusiast who loves working with Caucho's Resin Servlet Container, a Java EE Web Profile Servlet Container.

Caucho's Resin OpenSource Servlet Container

Java EE Web Profile Servlet Container

Tags: CDI Java java ee jsp servlet Tutorial