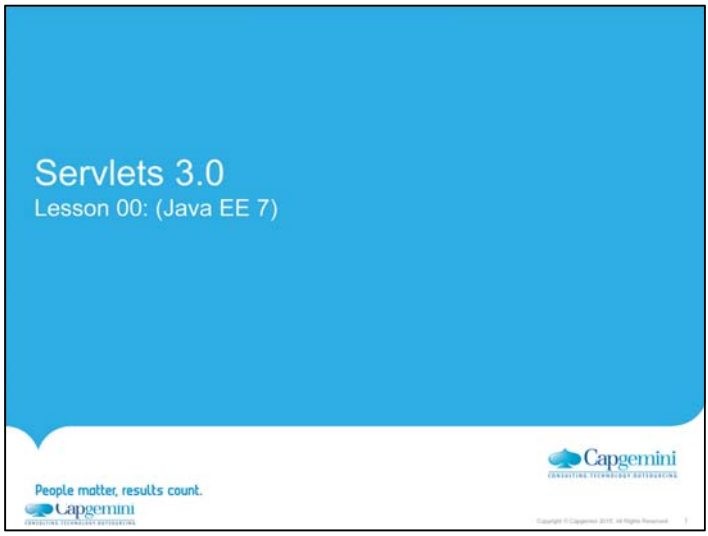
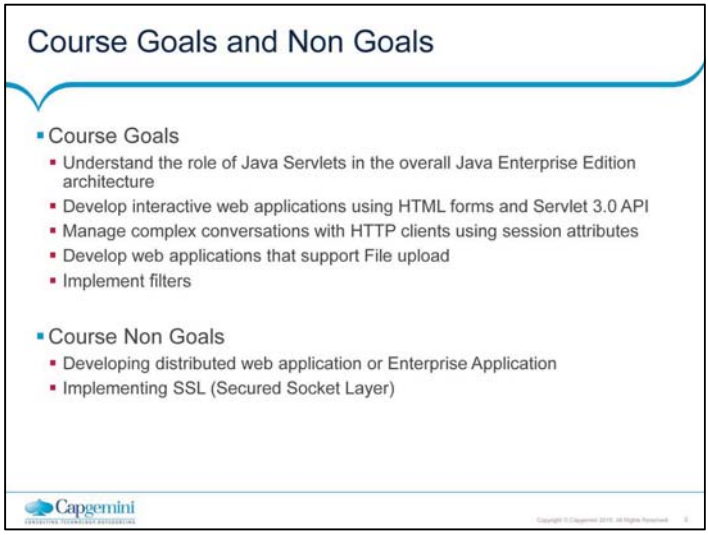
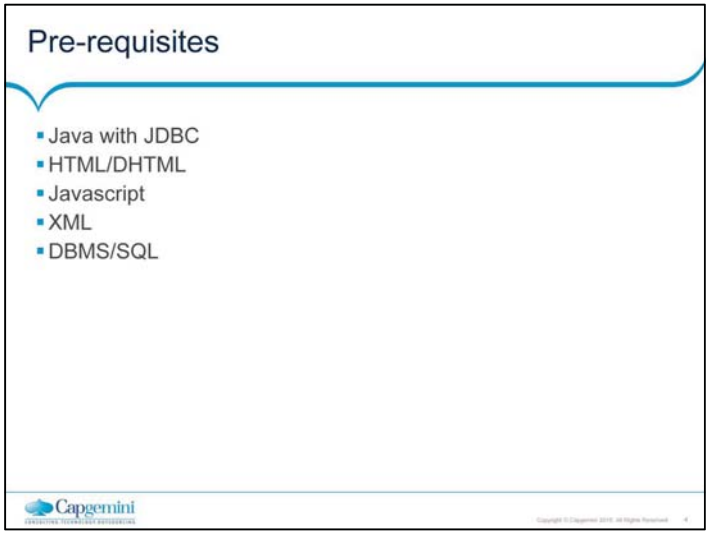


Document History

Date	Course Version No.	Software Version No.	Developer / SME	Change Record Remarks
16-Feb-2015	6.0	3.0	Yukti Valecha	Upgraded from Servlets 2.5 to Servlets 3.0
May 2016	6.1	3.0	Anjulata	Refinement as per new TOC

Capgemini

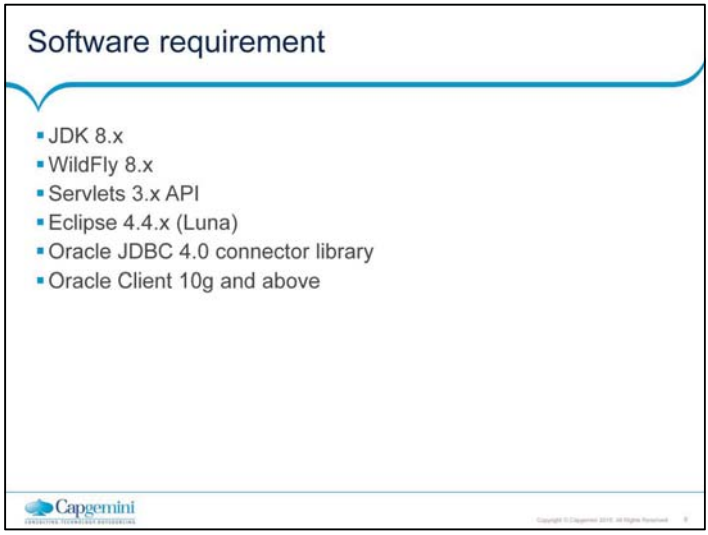

Course Goals and Non Goals
Course Goals
Understand the role of Java Servlets in the overall Java Enterprise Edition architecture
Develop interactive web applications using HTML forms and Servlet 3.0 API
Manage complex conversations with HTTP clients using session attributes
Develop web applications that support File upload
Implement filters
Course Non Goals
Developing distributed web application or Enterprise Application
Implementing SSL (Secured Socket Layer)
Capgemini

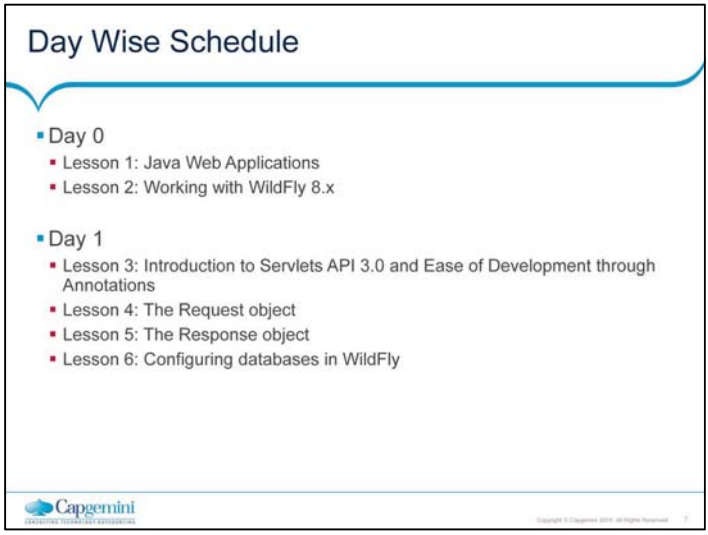

Pre-requisites
Java with JDBC
HTML/DHTML
Javascript
XML
DBMS/SQL
Capgemini

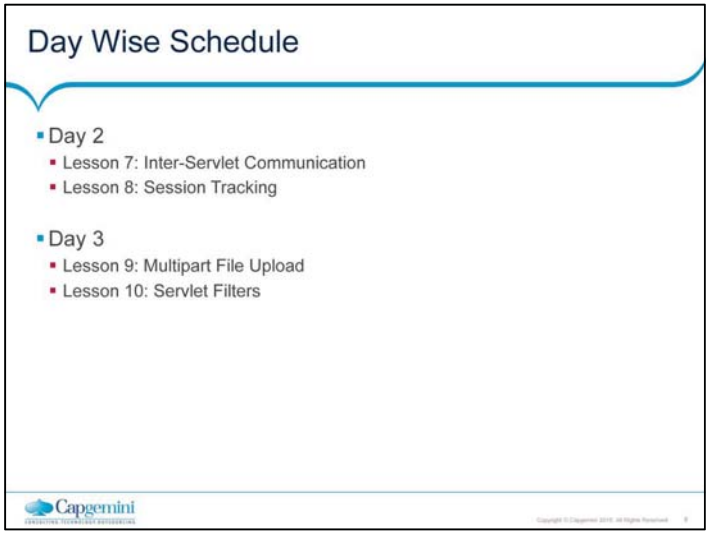
Intended Audience

- Web Authors
- Web Developers

Capgemini


Software requirement
JDK 8.x
WildFly 8.x
Servlets 3.x API
Eclipse 4.4.x (Luna)
Oracle JDBC 4.0 connector library
Oracle Client 10g and above
Capgemini


Day Wise Schedule
Day 0
Lesson 1: Java Web Applications
Lesson 2: Working with WildFly 8.x
Day 1
Lesson 3: Introduction to Servlets API 3.0 and Ease of Development through Annotations
Lesson 4: The Request object
Lesson 5: The Response object
Lesson 6: Configuring databases in WildFly
Capgemini


Day Wise Schedule
Day 2
Lesson 7: Inter-Servlet Communication
Lesson 8: Session Tracking
Day 3
Lesson 9: Multipart File Upload
Lesson 10: Servlet Filters
Capgemini

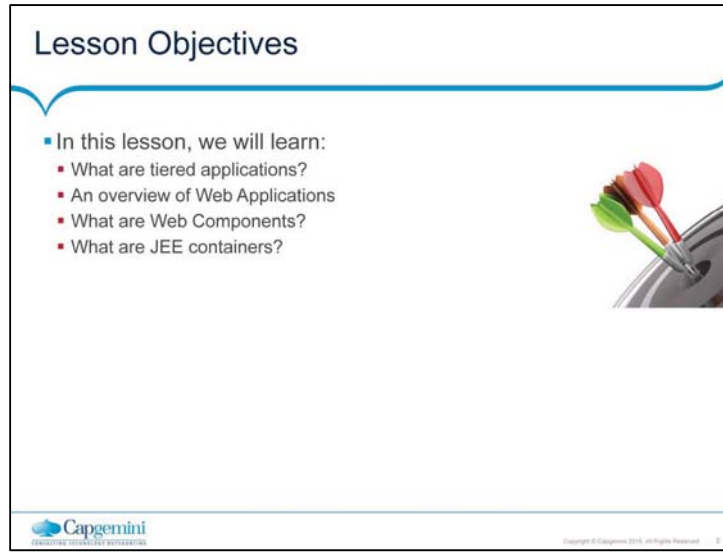
References

- Books
  - Java EE 6 Server Programming for Professionals
  - Head First Servlets and JSP
- Websites:
  - http://docs.coreservlets.com/servlet-3.0-api/
  - http://pdf.coreservlets.com/

Capgemini

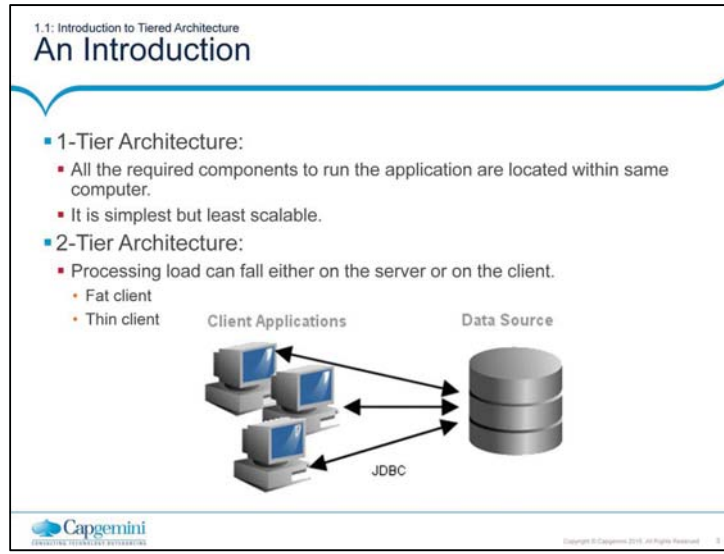
Java Servlets 3.0

Lesson 01: Java Web Applications



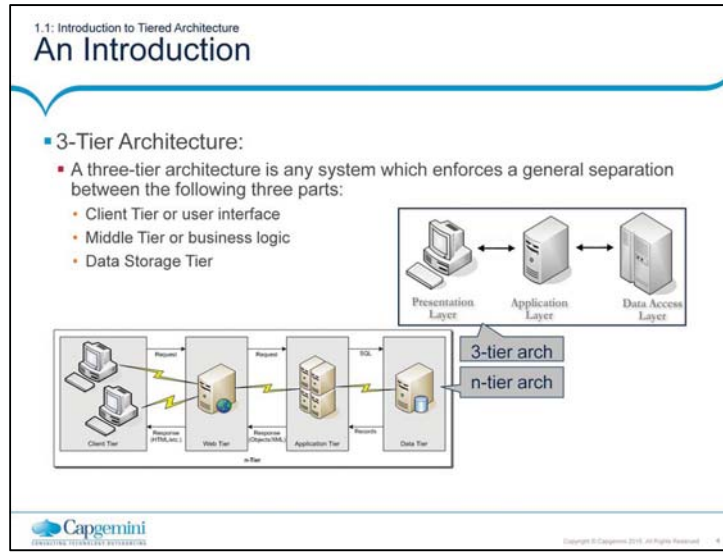
Lesson Objectives:
This lesson introduces Web application concepts. The lesson contents are:
Lesson 01: Java Web Applications

1.1: Introduction to tiered architecture
1.2: Web applications – an overview
1.3: What are Web components?
1.4: What are JEE containers?



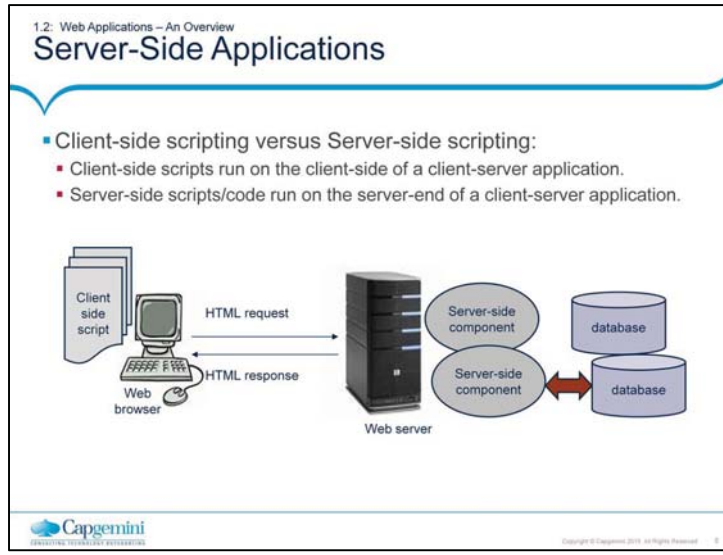
Introduction to Tiered Architecture:

- Tiered architecture can be classified as 1-Tier, 2-Tier, and 3-Tier or n-tier.
- **1-Tier Architecture** is the simplest, single tier on single user, and is the equivalent of running an application on a personal computer. All the required components, that is User interface, business logic, and data storage to run the application are located within the computer. They are the easiest to design, but the least scalable. This is because they are not part of a network – they are not useful for designing web applications.
- **2-Tier application or architecture** is a client server application where the processing load can fall either on the server or on the client.
  - When the processing load falls on the client, the server simply acts as a controller between the data and the client. Such a client is called as a **fat** client and imposes a lot of memory and bandwidth on the client's machine. Thus in this type of architecture the business logic and the presentation layer is located on the client machine and the data layer is on the server machine.
  - Problems in this approach? As the number of clients connecting to the server increases, there will be multiple requests to the server which will be time consuming. Also client machine should have sufficient processing power.
  - Another approach is that the client is a **thin** client and the business logic and the data layer are located on the server machine.
  - Problems in this approach? Multiple clients connecting to the sever may overload the server and this will make processing each client request very slow. To overcome all these problems one should plan a 3-tier architecture.
- The figure in the above slide shows a typical client/server application where Java programs access database server for executing SQL calls.



Introduction to Tiered Architecture:

- **3-Tier Architecture:** 3-Tier Architecture is most commonly used to build web applications. In this model, the browser acts like a client. Middleware or an application server contains the business logic, and database servers handle data functions. This approach separates business logic from display and data. The figure in the above slide shows a 3-tier application where Presentation, Business logic, and Data management is distributed across three separate tiers.
  Middle Layer acts as a bridge between the clients and the server, handling all the requests that come in from the client, routing appropriate requests to database, and taking care of the application business logic. The middle layer typically is an application server such as JBOSS, Websphere Application Server, Weblogic, WildFly
- **N-Tier Architecture:** The 3-tier architecture does not specialize functional layers. It is fine for prototypical or very simple web applications, but it does not measure up to the complexity demanded of web applications. The application server is still too broad, with too many functions grouped together. This reduces flexibility and scalability. N-Tier Architectures provide finer granularity, which provides more modules to choose from as the application is separated into smaller functions.
  Usually N-Tier Architecture begins as a 3-Tier model and is expanded. Here the application logic components can be divided and made available on multiple middle layers. Various layers of multi-tier architecture may or may not be distributed across various machines.



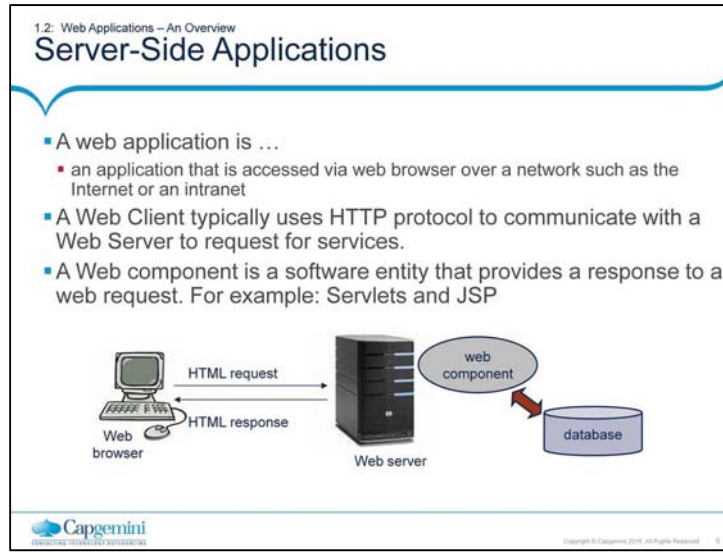
Web Applications – An Overview:

- **Desktop applications** are those that have stand-alone applications running on them. They are maintenance heavy.
- **Server-side applications** on the other hand run on the server side of a client-server system. The server is then able to access server side resources like databases, server components, and provide these services to multiple clients concurrently.

Client-side scripting versus Server-side scripting:

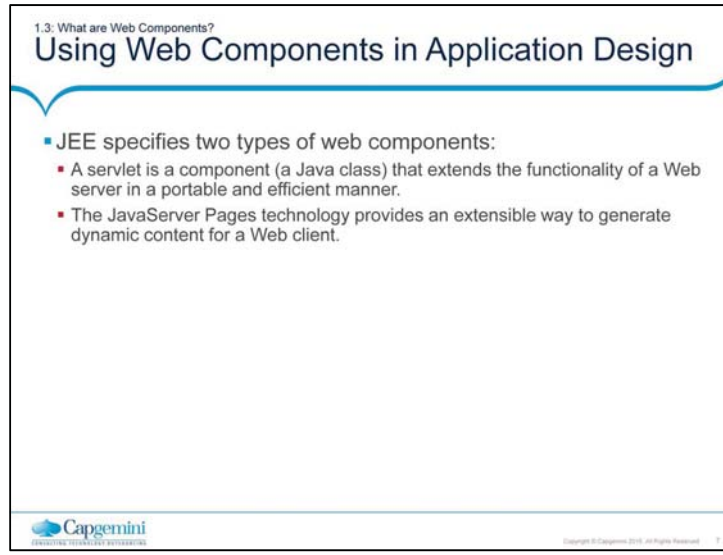
- **Client-side scripts** run on the client-side of a client-server application. For example, in a web application, we have written Javascript that runs on the browser to perform presentation validation!
- **Server-side scripts** run on the server-end of a client-server application. For example, when a HTML form is submitted, the data may be persisted on a database at the server side!

  From a web application perspective, there are many advantages to this:
  - The response is in HTML form, so complex code that is executed to generate the response stays at the server-side.
  - Server is able to access server side resources like databases, server components, which ensures centralized control.



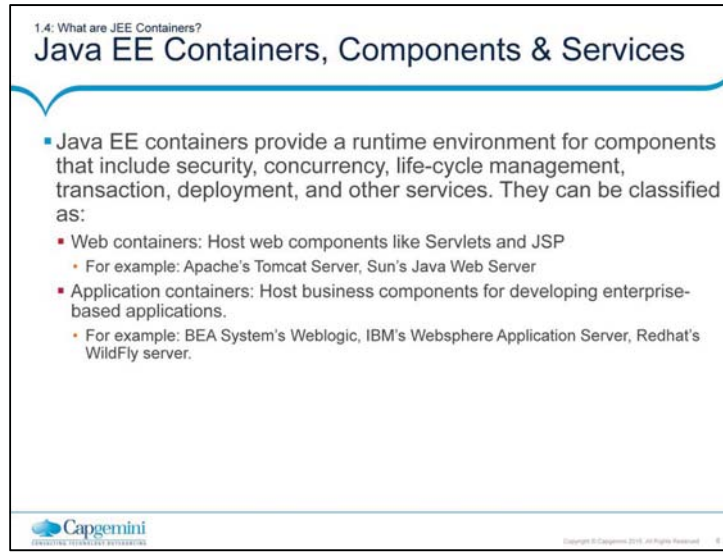
Web Applications – An Overview:

- A web application relieves the developer of the responsibility of building a client for a specific type of computer or a specific operating system. Since the client runs in a web browser, the user could be using an IBM-compatible or a Mac. They can be running Windows XP or Windows Vista using popular browsers like Internet Explorer, Netscape Navigator, or Firefox.
- **Web applications** commonly use a combination of **server-side script** (such as ASP, JSP, PHP) and **client-side script** (such as HTML, JavaScript) to develop the application.
  - The client-side script deals with the presentation of the information.
  - The server-side script/code deals with business logic, storing and retrieving the information.
- **Web clients (browser)** typically requests for services to a web server, and the web server either sends a static HTML page or dynamically generates a response to the request.
- **Web server** is a computer program that delivers (serves) content, such as a web page, using the Hypertext Transfer Protocol. The term web server can also refer to the computer or virtual machine running the program.
- Web components present in the Web Server are responsible for generating appropriate dynamic response. The JEE platform specifies two types of Web components:
  - Servlets
  - JavaServer Pages (JSP) pages



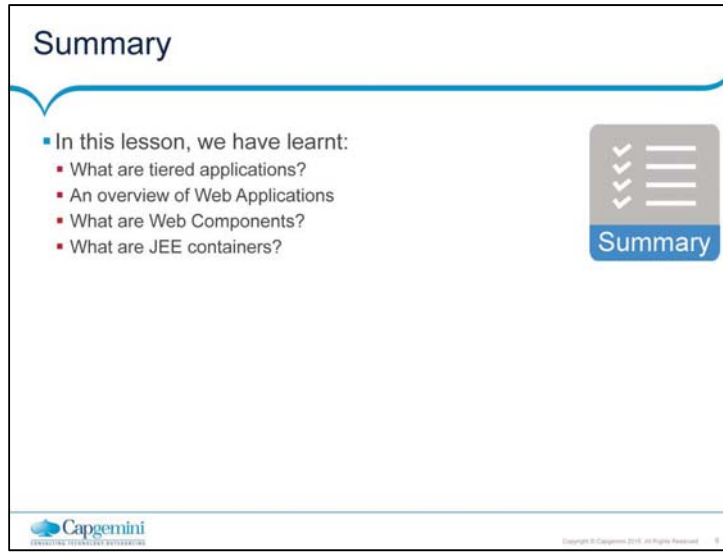
What are Web Components?

- A Web component typically generates the user interface for a Web-based application.
- **Servlets:** A Web server hosts Java servlet classes that execute within a server (web) container. When a servlet receives a request from a client, it generates a response, possibly by invoking business logic in Enterprise Beans or by querying a database directly. It then sends the response – as an HTML or XML document – to the requestor.
- **Java Server Pages technology (JSP):** It uses XML-like tags and scriptlets written in the Java programming language to encapsulate the logic that generates the content for the page.
  - By separating the page logic from its design and display, and supporting a reusable component-based design, JSP technology makes it faster and easier than ever to build web-based applications.
  - We shall be covering servlets in greater detail in this course and shall be covering JSP as a subsequent course.



What are JEE Containers?

- Once an application is built, it is packaged into one or more standard units for deployment to any Java EE platform. This **JEE unit** is now ready to be deployed. Deployment typically involves using a platform's deployment tool to specify location-specific information, such as a list of local users, who can access it, and the name of the local database. Once deployed on a local platform, the application is ready to run.
- Containers provide a runtime environment for components that includes security, concurrency, life-cycle management, transaction, deployment, and other services. Since the container handles these services, the developers need not be concerned about taking care of these aspects in the code. They can then focus on the business logic!
- JEE Containers are classified as:
  - **Web component containers (servers):** Allow us to test and deploy web components. They come in three flavors: Standalone, Add-on, Embeddable. We shall be seeing more on this topic in lesson-2.
  - **Application servers:** They offer server-side support for developing enterprise-based applications. Most Java-based application servers support servlets,JSP, and the rest of JEE specification. For example: BEA System's Weblogic, IBM's Websphere Application Server, Redhat's WildFly server


Summary
In this lesson, we have learnt:
What are tiered applications?
An overview of Web Applications
What are Web Components?
What are JEE containers?
Summary
Capgemini

Review Questions

- Question 1: In a 2-tiered application with a fat client, the business logic and the data layer are located on the server machine.
  - True/False
- Question 2: A 2-tiered or client/server application can also be a web application.
  - True/False
- Question 3: ___ acts as a bridge between the clients and the server, handling all the incoming requests, taking care of the application business logic.

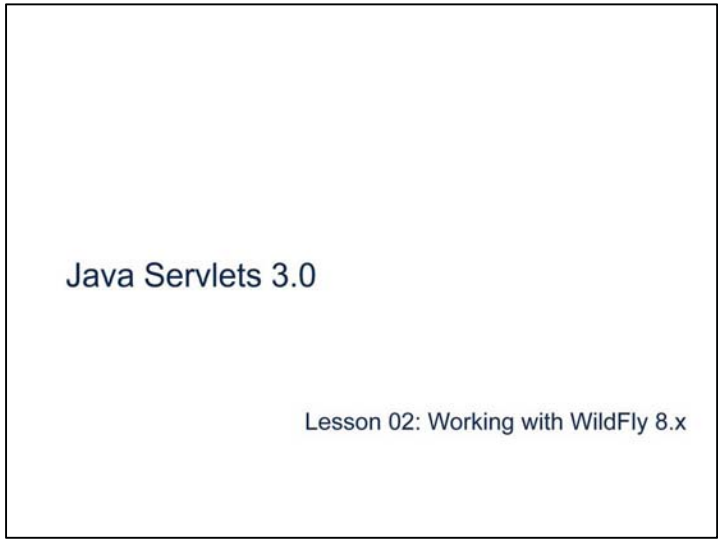
Capgemini

Review Questions

- Question 4: A ___ is a software entity that provides a response to a web request.
  - Option 1: Web Component
  - Option 2: HTML page
  - Option 3: Javascript
  - Option 4: Web Server

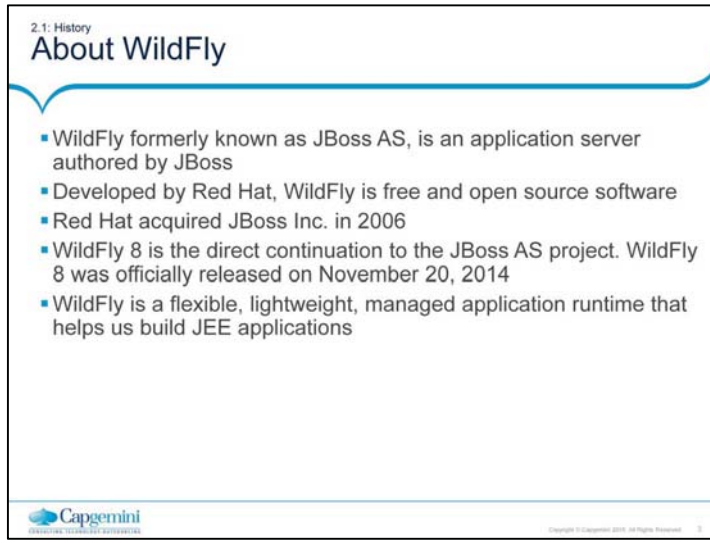
- Question 5: WildFly is actually an example of a:
  - Option 1: Web Server
  - Option 2: Application server

Capgemini

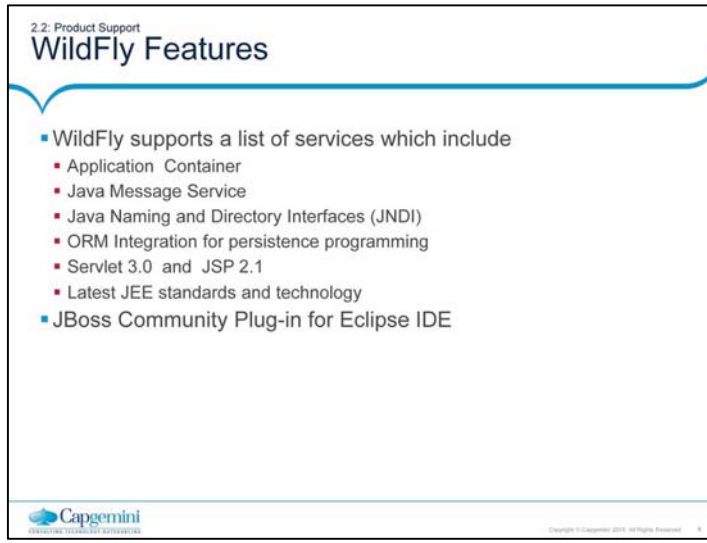

Java Servlets 3.0
Lesson 02: Working with WildFly 8.x

Lesson Objectives:
This lesson introduces Web application concepts. The lesson contents are:
Lesson 02: Working with WildFly 8.x

- 2.1: History
- 2.2: Product Support
- 2.3: Features
- 2.4: Installing WildFly
- 2.5: Working with WildFly using Eclipse
- 2.6: Accessing WildFly Homepage

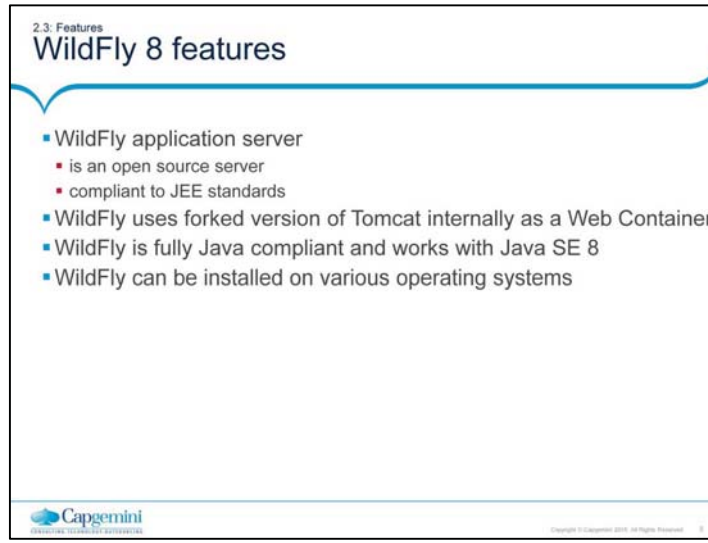


In 1999, JBoss project was started by March Fleury which launched JBoss Application Server. He started this project with intention of taking forward his research interest in middleware.
It was taken ahead by forming the JBoss group in 2001 which provided expert technical support services. Then in 2004, JBoss Group formed a corporation know as JBoss Inc. The JBoss Inc. was owned by employees and was backed by Matrix Partners, Accel Partners and Intel.
JBoss Inc. provides Middleware technology that offers the lowest cost of ownership by the use of open source software licenses. It is also backed up by expert technical support services.
Further in April 2006, JBoss Inc. was acquired by Red Hat.



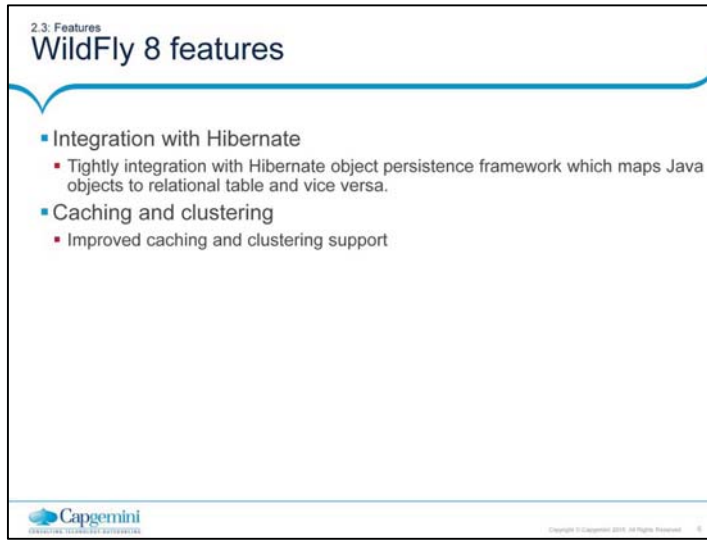
WildFly has become popular and is a safe choice for deploying JEE applications because it provides a set of Java Enterprise products.
The following list of products are supported in WildFly
WildFly Application Server – A server for deploying Enterprise applications
Java Messaging Service – It is the asynchronous message queuing system
Java Naming and Directory Interfaces (JNDI) – is a Java API for a directory service that allows Java clients to discover and look up data and objects via a name
Hibernate-ORM (Hibernate in short) is an object relational mapping library for the Java language, providing a framework for mapping an object oriented domain model to a traditional relational database
WildFly Web Server – Uses Tomcat internally, this web server supports Servlets/JSP, ASP.NET, PHP & CGI also.
Also included are JBoss Community which consists of JBoss Plug-in for Eclipse IDE and JBoss developer studio.

We will be using WildFly as a Web Server. Deploying Web application is covered in further courses



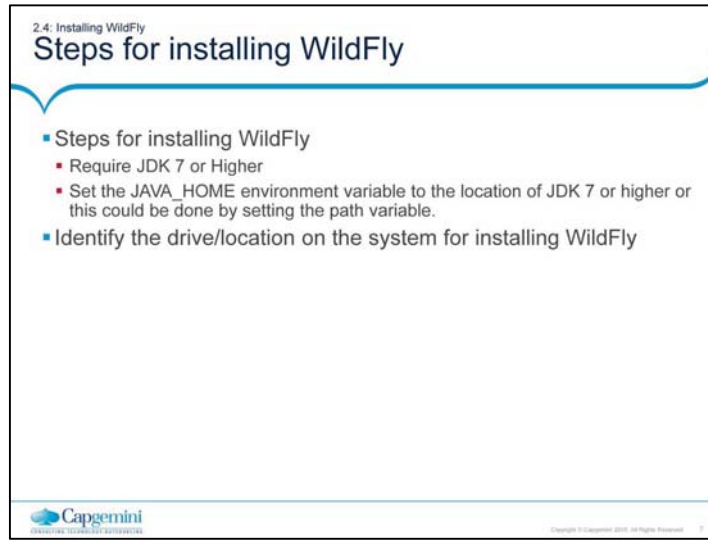
WildFly application server is an open source server and it complies to JEE 7 standards. It is the industry's first officially certified application server.
WildFly has full support for Service Oriented Architecture and JEE Web Services. It also supports asynchronous messaging using Java Messaging Service(JMS). JMS is a standard API for sending & receiving messages.
In addition, WildFly 8.0 supports JSP2.x,Servlet 3.x and EJB 3.x. Java Connector Architecture (JCA) which is a standard architecture for connecting the JEE platform to heterogeneous enterprise information systems.
JBoss 4 uses forked version of Tomcat as a Web Container. It is fully Java compliant and works with JEE 7 as well as Java SE 8. WildFly can be installed on various operating systems like Windows, Linux etc.

Forked version of Tomcat implies that copy of tomcat was used (original source) and independent development was started on that to create a branch(fork) of tomcat.

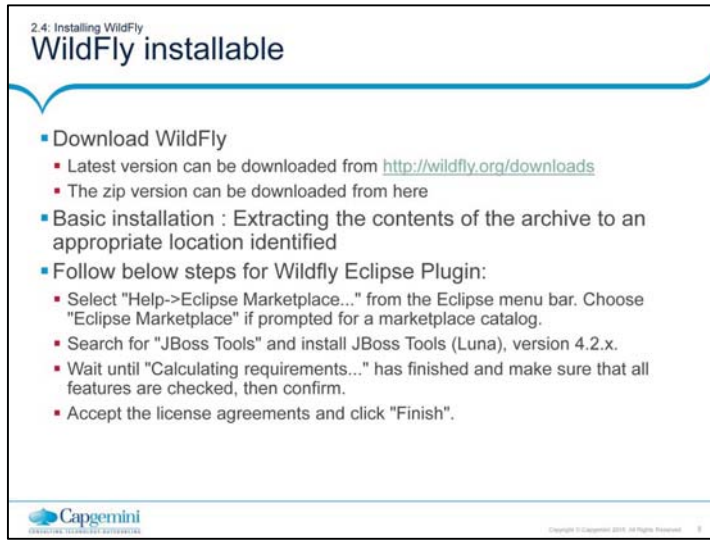


Integration with Hibernate – It is an object persistence framework which maps Java objects to relational tables and vice versa. Hibernate was acquired by WildFly and now it maintains and provides support for Hibernate. Hibernate deployer is available by default to provide hibernate framework libraries support to all applications

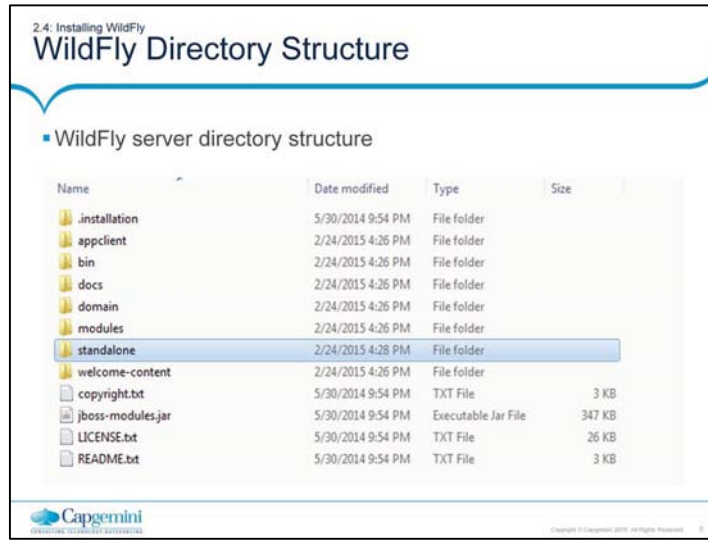
Caching and clustering – WildFly 8.0 has improved caching and clustering support. WildFly cache can be replicated. The WildFly cache instances can be distributed across JVM's. To support the caching features WildFly takes support of different cache loaders. SleepyCat Berkeley DB, generic JDBC data source and file system cache loaders are available.
The sessions information and SSO security context is replicated across clustered servers. Therefore if one server fails, the users are automatically moved on to another servers without losing any of their information.



Before installing WildFly ensure that the system satisfies certain pre-requisites.
Check the availability of JDK 8. Check the version of JDK by using
the command
java –version
Set the JAVA_HOME variable to the location of JDK 8 or setting path variable
Identify the drive/location on the system for installing WildFly
(Note : It does not matter where WildFly is installed on the system. However do not install WildFly in a directory that contains spaces, it can cause problems sometimes)

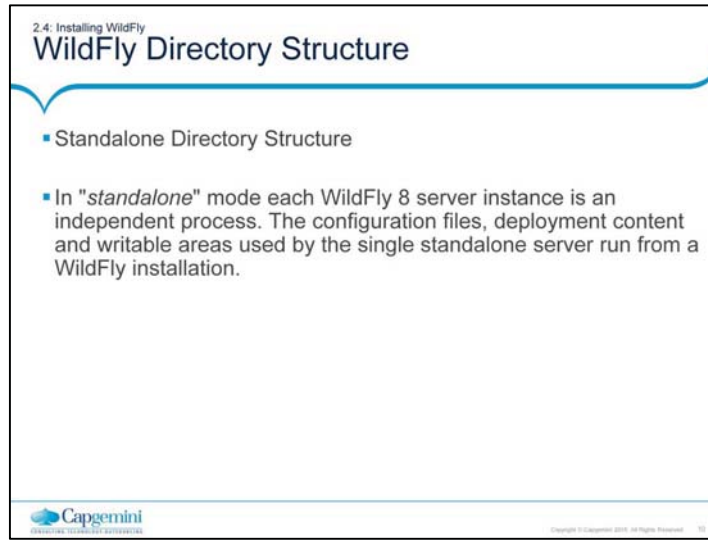


WildFly Server is freely downloadable. An evaluation copy could also be used
WildFly archive(Zip, tar) can be downloaded from http://wildfly.org/downloads
Once the required version is downloaded basic installation can be done. Extract the contents of the archive in the appropriate location identified on the system.



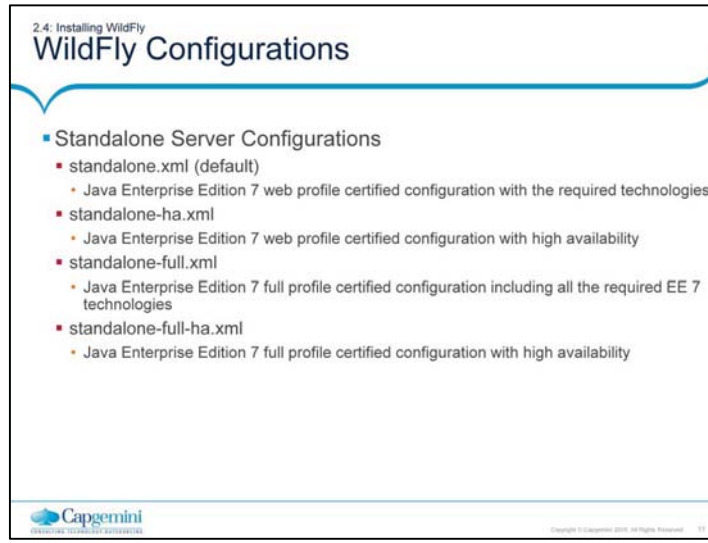
Once WildFly is available on the system, it will create the directory that contains server configurations , JARs and other directories which help in functioning of WildFly.
The figure on the slide shows the WildFly directory structure with below diagram

DIRECTORY	DESCRIPTION
appclient	Configuration files, deployment content
bin	Startup scripts, startup configuration files and various command line utilities like Vault, add-user
docs/schema	XML schema definition files
docs/examples/configs	Example configuration files representing specific use cases
domain	Configuration files, deployment content, and writable areas used by the domain mode processes run from this installation.
modules	WildFly 8 is based on modular class loading architecture. The various modules used in the server are stored here.
Standalone	Configuration files, deployment content, and writable areas used by the single standalone server run from this installation
welcome-content	Default Welcome Page content



The details of the standalone directory is present in below diagram

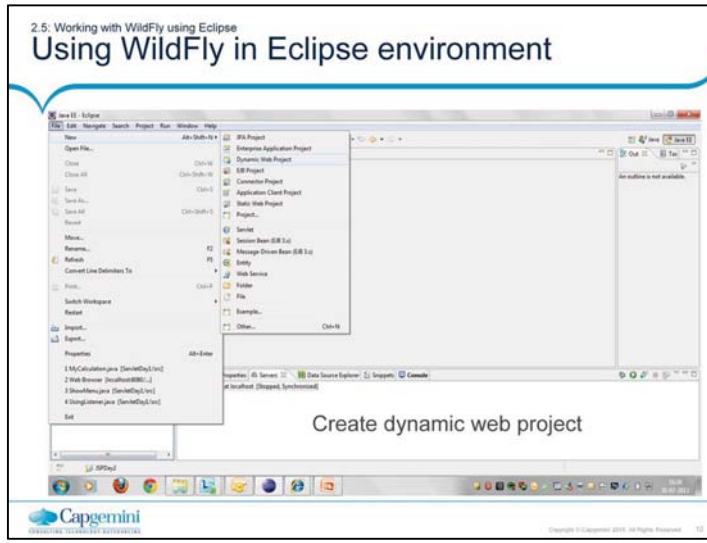
DIRECTORY	DESCRIPTION
configuration	Configuration files for the standalone server that runs off of this installation. All configuration information for the running server is located here and is the single place for configuration modifications for the standalone server.
data	Persistent information written by the server to survive a restart of the server
deployments	End user deployment content can be placed in this directory for automatic detection and deployment of that content into the server's runtime.
lib/ext	Location for installed library jars referenced by applications using the Extension-List mechanism
log	standalone server log files
tmp	location for temporary files written by the server
tmp/auth	Special location used to exchange authentication tokens with local clients so they can confirm that they are local to the running AS (Application Server)process



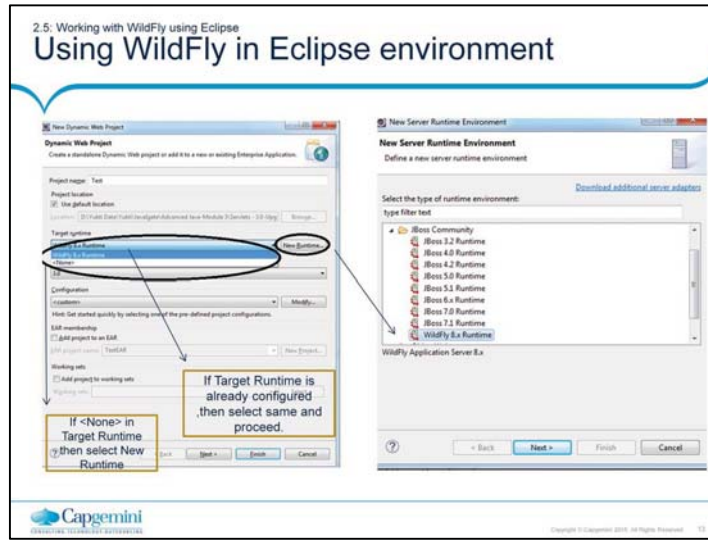
We also have the Domain Server Configurations

domain.xml
Java Enterprise Edition 7 full and web profiles available with or without high availability

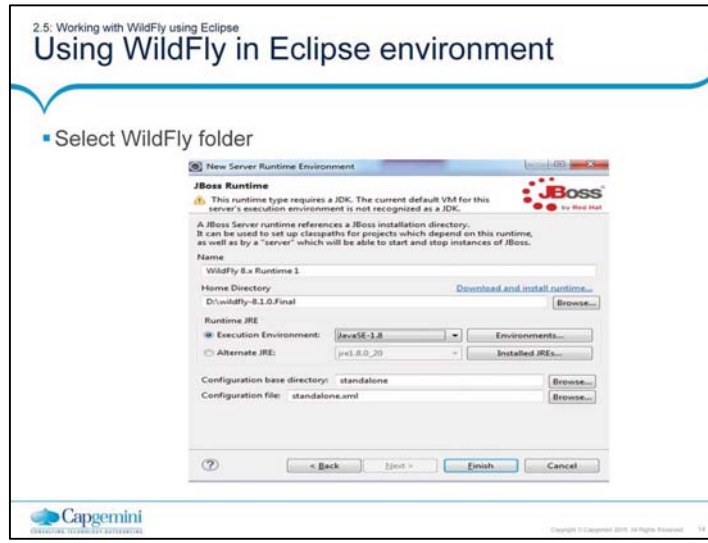
Important to note is that the domain and standalone modes determine how the servers are managed not what capabilities they provide.



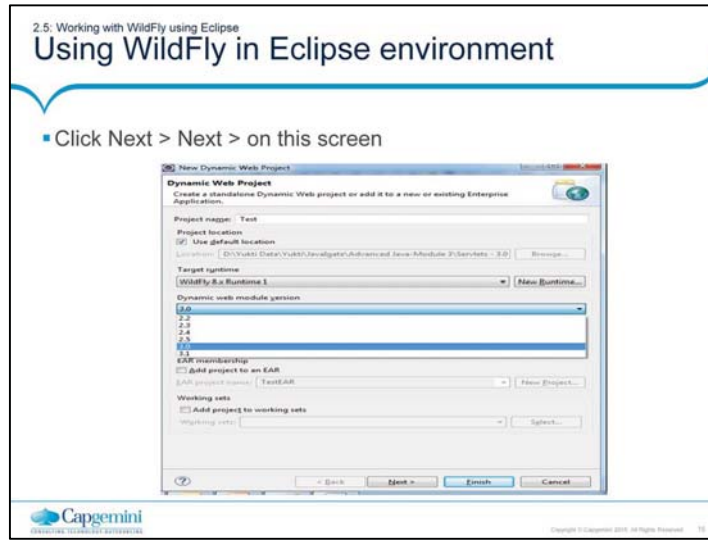
Select Dynamic Web Project



Click New Runtime to add server



Select WildFly folder

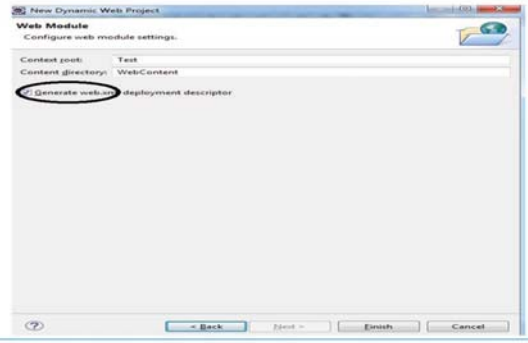


Select Dynamic web module version 3.0

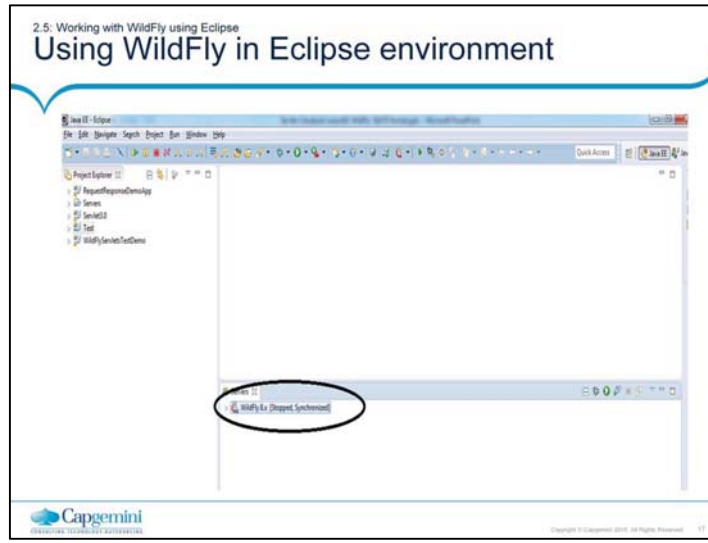
2.5: Working with WildFly using Eclipse

Using WildFly in Eclipse environment

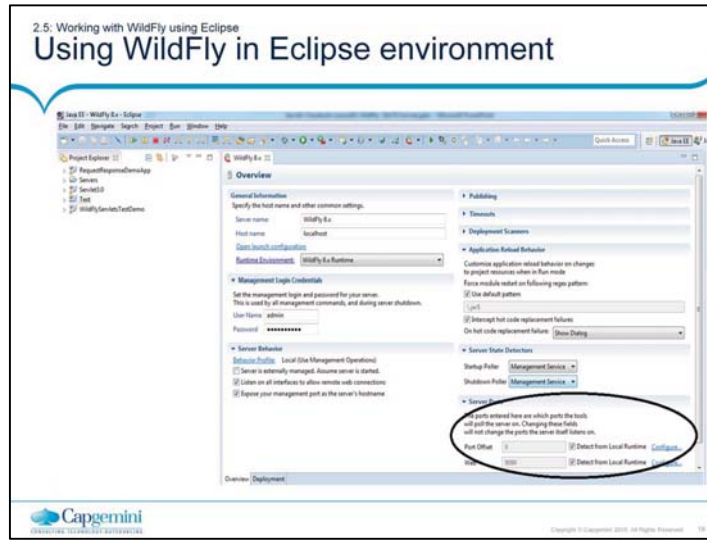
- Click on check box for web.xml and then on Finish



Capgemini



Double click highlighted link to start server



Default port number of WildFly is 8080. In the above configuration localhost port number has been set to 9090. (could be also set from standalone.xml file (D:\wildfly-8.1.0.Final\standalone\configuration)).

See below diagram

standalone.xml

```
<socket-binding-group name="standard-sockets" default-interface="public"
port-offset="${jboss.socket.binding.port-offset:0}">
    <socket-binding name="management-http" interface="management" port=
    "${jboss.management.http.port:9990}"/>
    <socket-binding name="management-https" interface="management" port=
    "${jboss.management.https.port:9993}"/>
    <socket-binding name="ajp" port="${jboss.ajp.port:8009}"/>
    <socket-binding name="http" port="${jboss.http.port:9090}"/>
    <socket-binding name="https" port="${jboss.https.port:8443}"/>
    <socket-binding name="txn-recovery-environment" port="4712"/>
    <socket-binding name="txn-status-manager" port="4713"/>
    <outbound-socket-binding name="mail-smtp">
        <remote-destination host="localhost" port="25"/>
```

2.6: Accessing WildFly Homepage

WildFly - Home Page

- Start the WildFly server
- The home page of WildFly can be accessed by specifying the URL http://localhost:9090.
- The home page can be seen

Capgemini

Add the notes here.

Review Questions

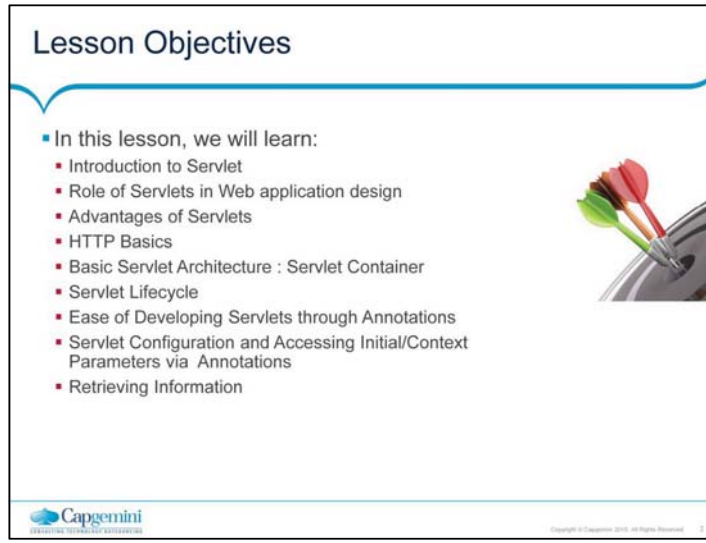
- Question 1: If an independent instance of WildFly required then which mode should be selected
  - Option 1: Basic
  - Option 2: Domain
  - Option 3: Standalone
  - Option 4: None of the above
- Question 2: All configuration information for running the server is located here and it is single place for configuration modifications for the standalone server
  - Option 1: data
  - Option 2: configuration
  - Option 3: log
  - Option 4: conf

Capgemini

Add the notes here.

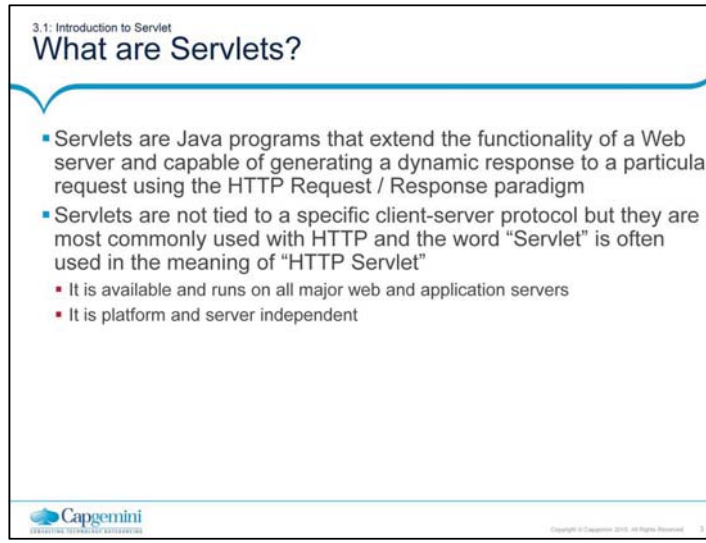
Java Servlets 3.0

Lesson 03: Introduction to Servlets API and Ease of Development through Annotations

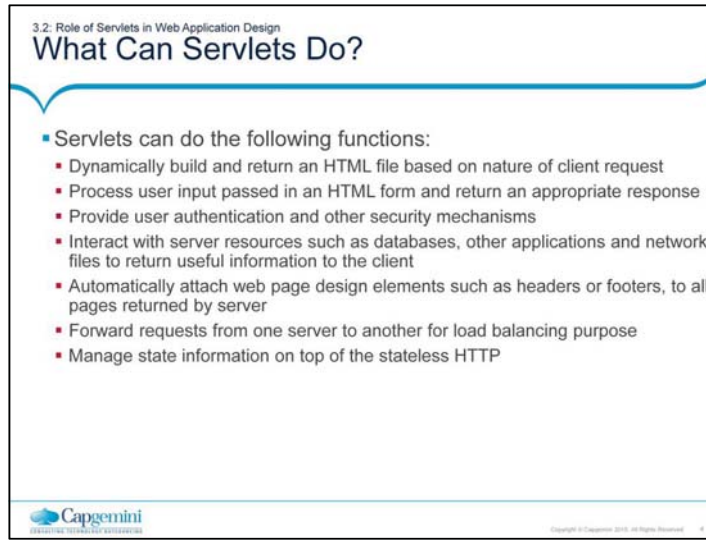


Lesson Objectives:
This lesson introduces Servlets 3.0 and ease of development through Annotations , The lesson contents are:

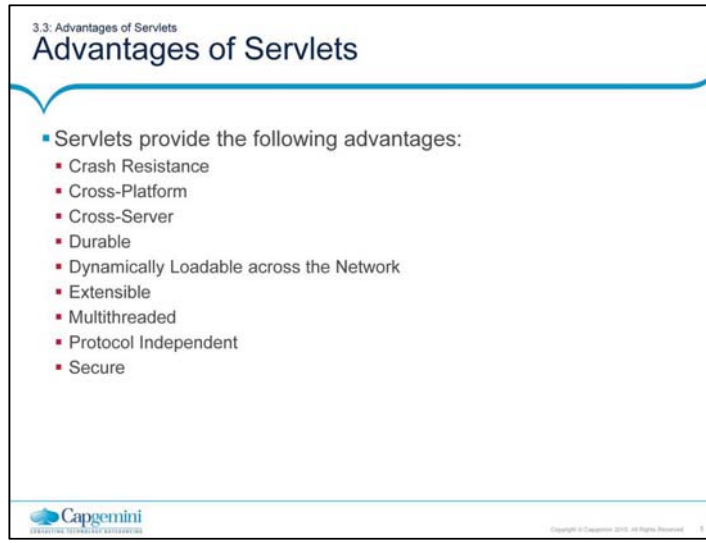
3.1: Introduction to Servlet
3.2: Role of Servlets in Web Application Design
3.3: Advantages of Servlets
3.4: HTTP Basics
3.5: Basic Servlet Architecture: Servlet Container
3.6: Servlet Lifecycle
3.7: Elements of Web Application
3.8: Ease of Developing Servlets through Annotations
3.9: Servlet Configuration and Accessing Initial / Context Parameters via Annotations
3.10: Retrieving Information



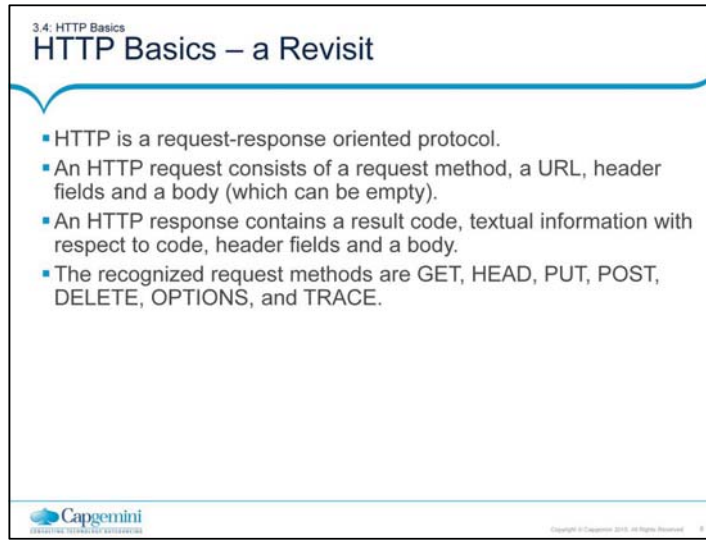
Introduction to Servlet:
Java servlets are a key component of server-side Java Development. Servlets are modules of Java code that run in a server application to answer client requests. These are small, pluggable extensions to server that enhance the server's functionality. Servlets allow developers to extend and customize any Java-enabled server.
When servlets are used to generate dynamic content for a web page or otherwise extend the functionality of a web server, is like creating a web application!
Since servlets are written in the highly portable Java language and follow a standard framework, they provide a means to create sophisticated server extensions in a server and operating system independent way.
Although servlets can respond to any type of request, they are commonly used to extend the applications hosted by web servers. For such applications, Java Servlet technology defines HTTP-specific servlet classes. We shall see more on this in the coming sections.
Servlets first came on the scene around 1997. As of March 26 2010, the current version of the servlet specification is 3.0. The minimum platform requirement for Servlet 3.0 is JDK 1.6.



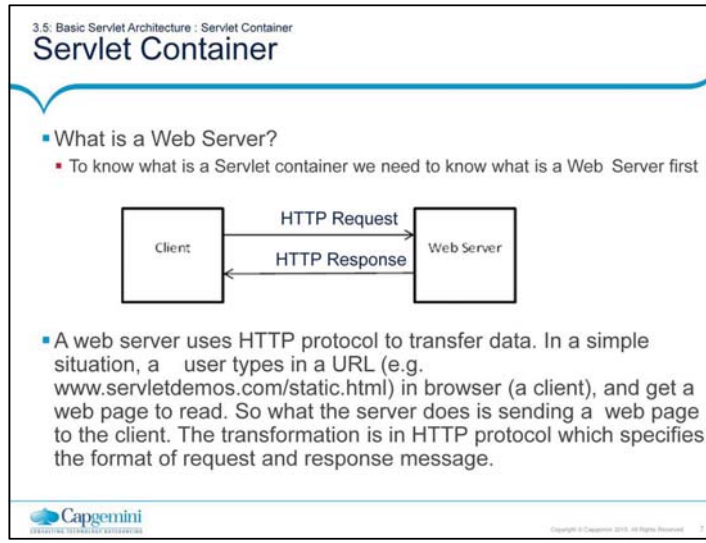
Role of Servlets in Web Application Design:
Servlets can read explicit data sent in by the client in browser. This data can come from an HTML page into which user has entered data. Servlets can also read implicit request data which is sent by browser as part of request header.
Servlets can dynamically generate response and send content back to client. This process may involve talking to databases, executing another server-side component, and so on.
The response sent can be in from of pure HTML, plain text, XML, GIF images, or even as compressed data.



Advantages of Servlets over CGI:
Servlets provide the following advantages:
Compiled: Servlets are compiled into Java byte-codes. This improves performance through compile-time code optimization. Server-side JIT compilers dramatically improve the performance of JVM.
Compilation also offers the advantages of strong error and type checking. Since many errors are flushed out during the compilation, servlets are more stable and easier to develop and debug.
Crash Resistance: The JVM does not allow servlets direct access to memory locations, thereby eliminating crashes that results from invalid memory accesses. In addition, before execution, the JVM verifies that compiled Java class files are valid and do not perform illegal operations. Finally, rather than crashing, the JVM will propagate an exception up the calling chain until it is caught. Thus, a poorly written or malicious servlet cannot crash the server.
Cross-Platform: Since servlets are written in Java, they enjoy the same cross platform support as any program. This “write once, run anywhere” capability allows servlets to be easily distributed throughout the enterprise without rewriting for each platform.
Cross-Server: Servlets can be run on virtually every popular web server that is in use today. More than a dozen software vendors currently provide native support for servlets within their products. For those servers that do not currently offer native servlet support, there are many third party add-ons that allow these servers to load and run servlets.



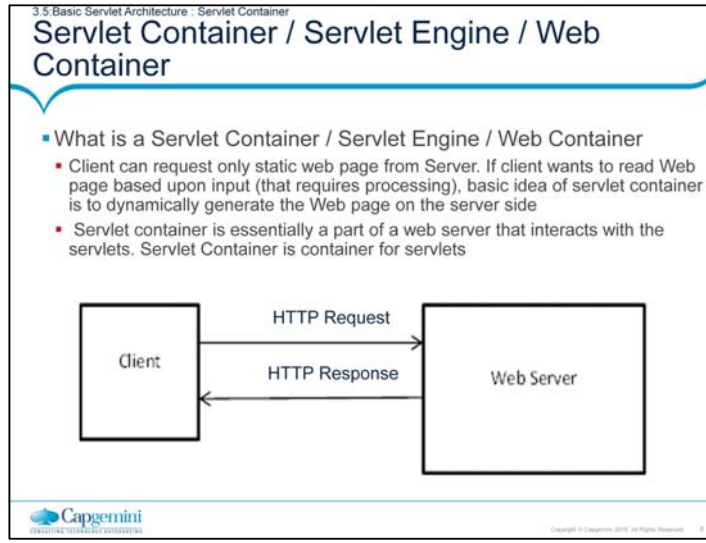
Basic Servlet Architecture:
HTTP Basics – a Revisit:
HTTP is the protocol that is used by a WWW client (for example: a browser) to send a request to a Web Server.
HTTP is a request-response oriented stateless protocol. An HTTP request consists of a request method, a URL, header fields, and a body (which can be empty). An HTTP response contains a result code, textual information w.r.t code, header fields and a body.
When client request a URL in a Web Browser, the default method is GET for the request. With a GET request, the input parameters are appended to the URL. A GET request does not have a body.
With a POST request the input parameters are transmitted in the body.
The response should contain a body with the response data and header fields, which describe the body (especially Content-Type and Content-Encoding).



Since we using WildFly 8.x which is an Application server which will host the Servlet Container to handle the web components.

We could also use the Web server for the same purpose.

Since HTTP is a web oriented protocol, thus we have taken the term to be as “Web Server”



When a request is sent, it is handled by the web server. If response is available within the web server it will be returned by the server to the client.

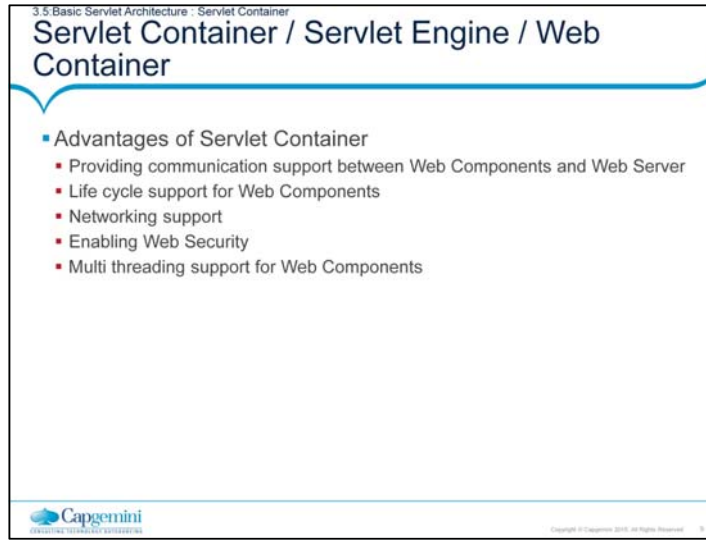
If response is not available and requires some processing then request will be processed by the Web Container.

The web container will then process the request and handover the response back to web server. The web server will then handover response back to client.

The web container delegates the request processing to web components like Servlets and JSP.

For example: If we request for a page as www.gmail.com , then same page is returned as response for all users. **This is a static content**

Whereas after entering your login credentials respective inbox details per user is retrieved as a response. **This is dynamic content**



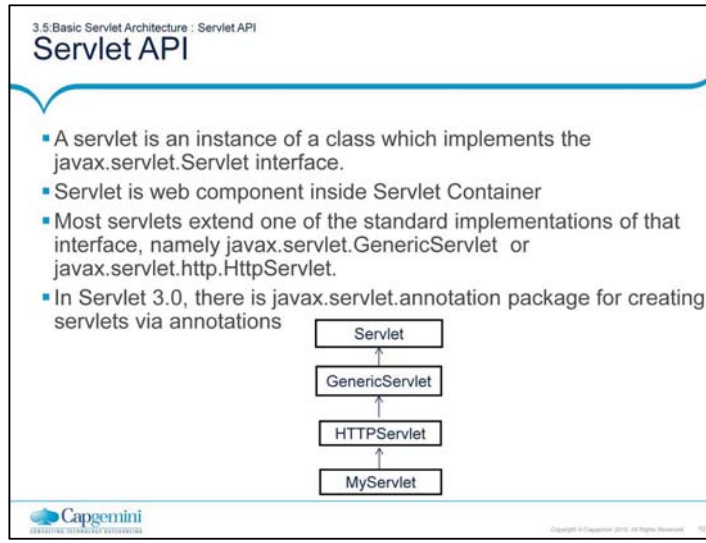
Providing communication support between Web Components and Web Server: Servlet as a web component does not communicate with Web Server directly. All the communication happens via Web container

Life cycle support for Web Components: Life cycle of Web components (Servlets and JSP) is managed by the container

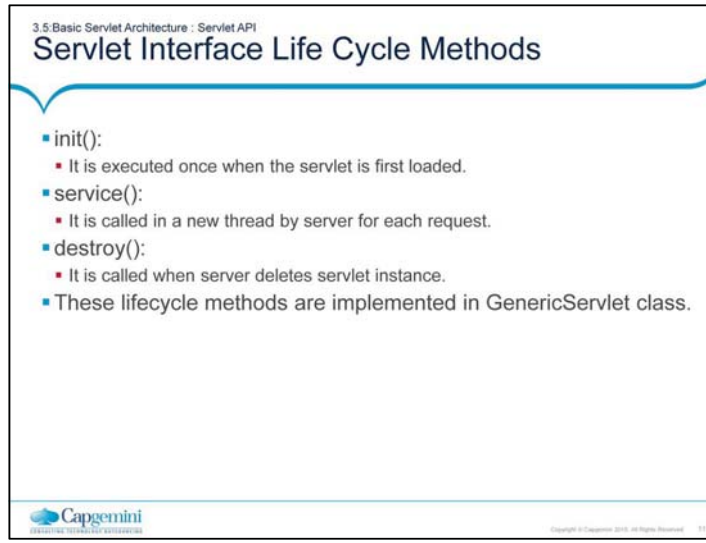
Networking support: Web components do not need to open network and socket connections all is taken care by Web server

Enabling Web Security: Access to Web components can be secured by granting privileges.

Multi Threading support: Web components are multi-thread, for every request there is one thread generated. Thus all threads are managed by container. We do not need to do explicit multi-threading.



Basic Servlet Architecture:
Servlet API:
Servlets use classes and interfaces from three packages: javax.servlet, javax.servlet.http.
A Servlet, in its most general form, is an instance of a class which implements the javax.servlet.Servlet interface. Most Servlets extend one of the standard implementations of that interface, namely javax.servlet.GenericServlet and javax.servlet.http.HttpServlet.
As of Servlet 3.0, make use of package javax.servlet.annotation to create Servlets.
A protocol-independent servlet should subclass GenericServlet, while an HTTP servlet should subclass HTTPServlet, which is itself a subclass of GenericServlet! (refer to the above figure).
Notice that the classes do not belong to the core Java API. They are extensions to the core API and hence javax!



Basic Servlet Architecture:
Servlet Interface Life Cycle Methods:
The Servlet interface defines methods to initialize a servlet, to service requests, and to remove a servlet from the server. These are known as life-cycle methods and are called in the following sequence:

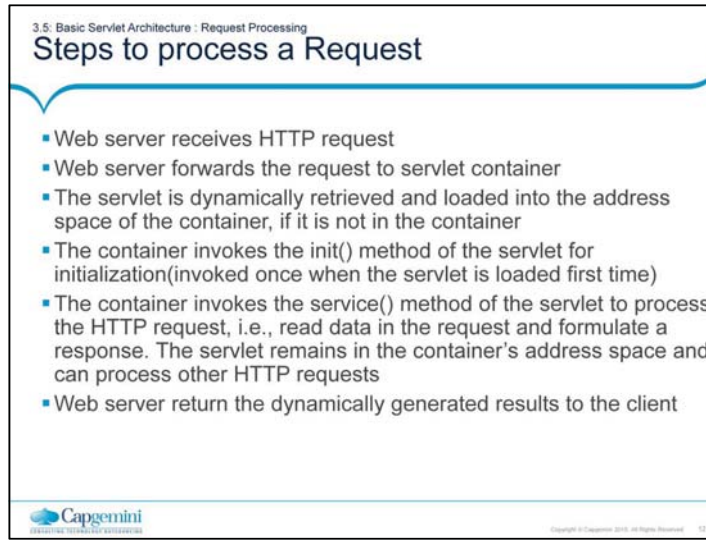
The init() method is guaranteed to be called only once during the Servlet's lifecycle. The Servlet performs one-time setup configurations in this method. It stores the ServletConfig[*] object so that it can be retrieved later by calling the Servlet's getServletConfig() method (This is handled by GenericServlet). The ServletConfig object contains Servlet parameters and a reference to the Servlet's ServletContext[*].

service() method gets called every time a new request comes in. The method is called concurrently (that is, multiple threads may call this method at the same time) so it should be implemented in a thread-safe manner.

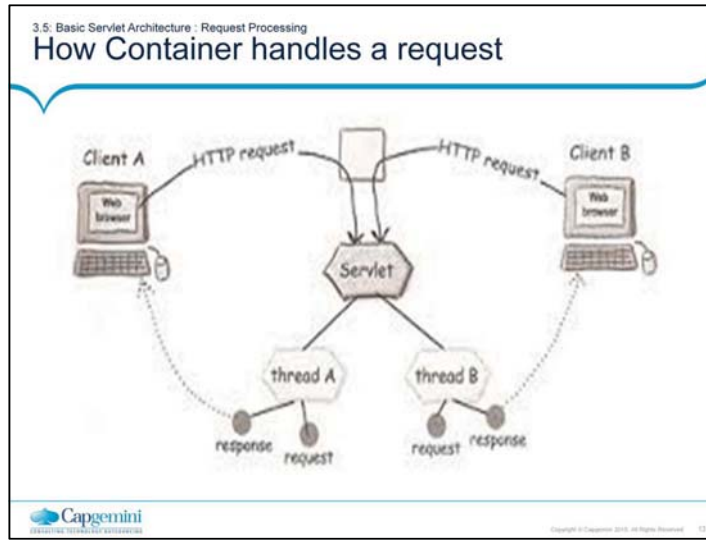
When servlet needs to be unloaded (for example: since a new version should be loaded or the server is shutting down), the destroy() method is called. This method too is guaranteed to be called only once during the Servlet's lifecycle. All resources which were allocated in init() should be released in destroy().

We shall be covering entire lifecycle of a typical servlet in the next section.

[*] : The ServletConfig and ServletContext interfaces are explained in the next slide.



From the life cycle of a servlet object, we can see that servlet classes are loaded to container by class loader dynamically. Each request is in its own thread, and a servlet object can serve multiple threads at the same time. When it is no longer being used, it should be garbage collected by JVM.
Like any Java program, the servlet runs within a JVM. To handle the complexity of HTTP requests, the servlet container comes in. The servlet container is responsible for servlets' creation, execution and destruction.

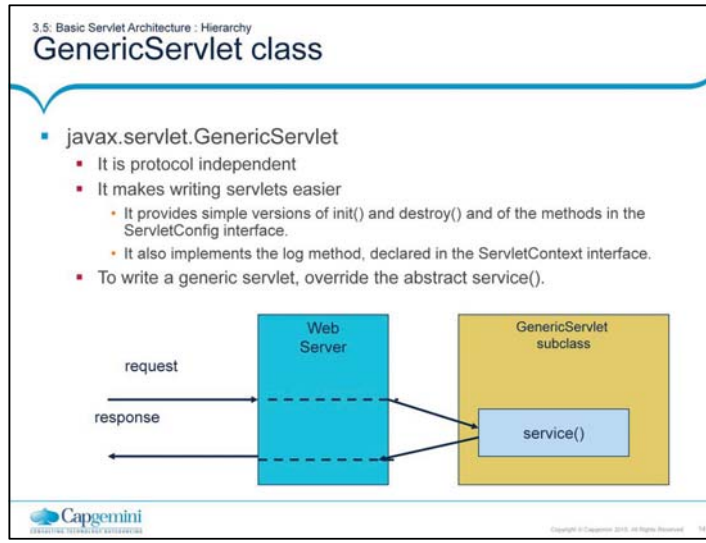


Every request coming to the container from the client for a particular servlet will be treated as a separate thread.

Each thread will have its own copy of the request / response objects.

After the processing of a particular request is completed the threads would be garbage collected. But the servlet instance would still be there in memory.

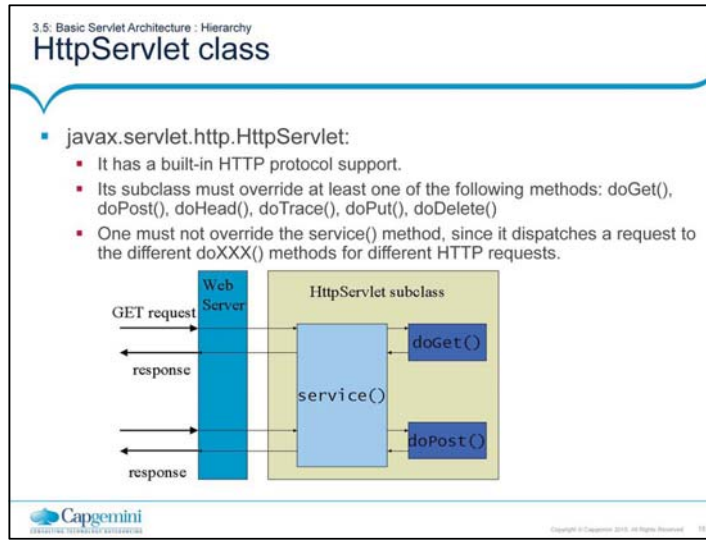
Thus for processing multiple requests multiple threads are created.



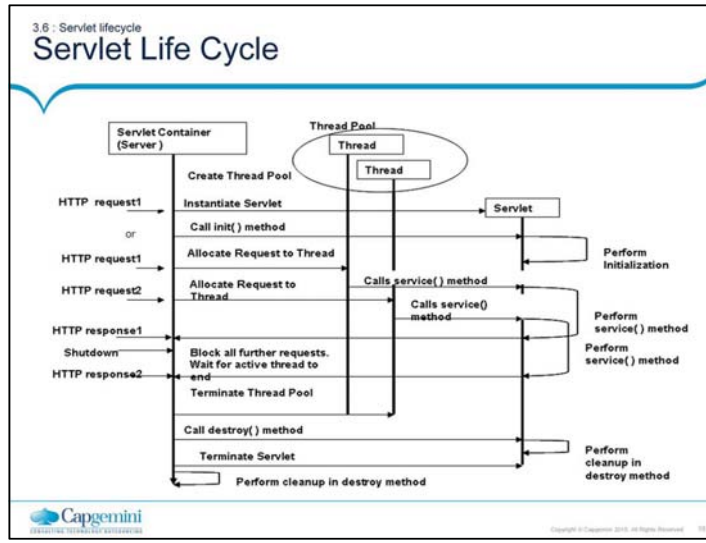
Basic Servlet Architecture:
GenericServlet:
All the lifecycle methods are implemented in GenericServlet class. Thus if GenericServlet class is extended to create a servlet, all the methods may be overridden to provide implementations. Optionally, just override the service() method to handle the request in the manner required!

ServletContext interface :
It defines a set of methods that a servlet uses to communicate with its servlet container, for example, to get the MIME type of a file, dispatch requests, or write to a log file.
There is one context per “web application” per JVM.
ServletContext attributes can be used to share information among a group of servlets.
The ServletContext object is contained within the ServletConfig object, which the Web server provides the servlet when the servlet is initialized.
ServletConfig interface:
This is a servlet configuration object used by a servlet container used to pass information to a servlet during initialization.
It is implemented by GenericServlet.
All of its initialization parameters can be set in deployment descriptor. Or else they can be passed via annotations. The ServletConfig parameters are specified for a particular servlet and are unknown to other servlets.

ServletContext and ServletConfig will be discussed in later sessions.

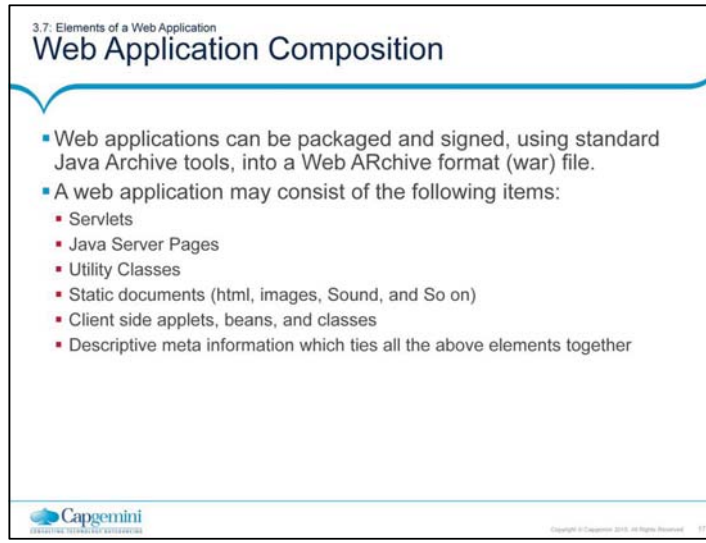


Basic Servlet Architecture:
HttpServlet:
init() and destroy() can be overridden to manage resources that are held for the life of the servlet. Servlet can implement getServletInfo() to provide information about itself.
The service() method of HttpServlet dispatches a request to different Java methods for different HTTP request methods. It recognizes the standard HTTP/1.1 methods like GET, HEAD, PUT, POST, DELETE, OPTIONS and TRACE. Other methods are answered with a Bad Request HTTP error.
Do not override the service() method because it handles setup and dispatching to all the doXXX() methods. Since service() method dispatches to doXXX() methods, it passes its request and response objects to these methods.
An HTTP servlet generally overrides doGet() to handle GET requests and doPost() to handle POST type of requests.

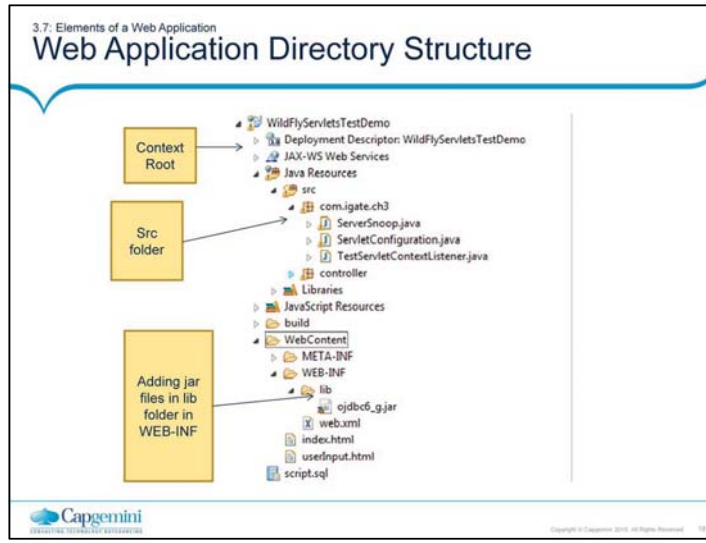


Basic Servlet Architecture:
Servlet Life Cycle:
The process by which a server invokes a servlet can be broken down into the nine steps described below and as shown in the figure in the above slide:

The server loads the servlet when the client first requests it or, if configured to do so, at server start-up. The servlet may be loaded from either a local or remote location using the standard Java class loading facility. This step is equivalent to the following code:
Class c= Class.forName("com.igate.Myservlet");
The server creates one or more instances of the servlet class. Depending on the implementation, the server may create a single instance that services all requests through multiple threads or create a pool of instances from which one is chosen to service each new request. This step is equivalent to the following Java code:
Servlet s= (Servlet) c.newInstance();
The server constructs a ServletConfig object that provides initialization information to the servlet.
The server calls the servlet's init(ServletConfig cfg) method. The init() method is guaranteed to finish execution prior to the servlet processing the first request. If the server created multiple servlets instances (step 2), then the init method is called one time for each instance.



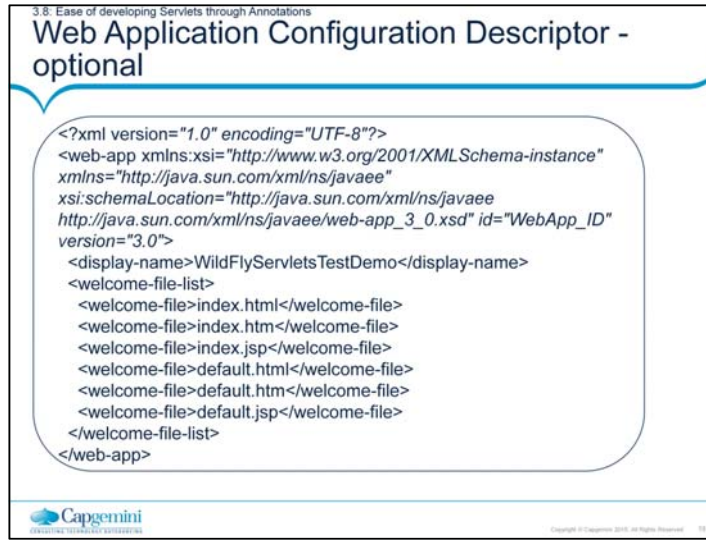
Elements of a Web Application:
Web Archive (WAR) files provide a convenient way of bundling Web applications in a single file. Having a single file instead of many small files makes it easier to transfer the web application from server to server.
A WAR file is really just a JAR file with a .war extension. We can use the jar command to create it.
For example: jar cvf myWebApp.war *
However, we can also use the ANT tool or IDE like Eclipse to automate the process!
A Web application is a collection of web elements that can be bundled and run on multiple containers (servers) from multiple vendors.
A web application is rooted at a specific path within a web server called context root (see figure on the next slide).
For example, a catalog application can be located at http://www.mycorp.com/catalog. All requests that start with this prefix will be routed to the ServletContext, which represents the catalog application.
The servlet container must enforce a one to one correspondence between a web application and a ServletContext. A ServletContext object can be viewed as a Servlet's view onto its application.



Elements of a Web Application:
Directory Structure:
A web application exists as a structured hierarchy of directories. The root of this hierarchy (called Context Root) serves as a document root for serving files that are part of this context. For example, for a web application located at /Catalog in a web server, the index.html file located at the base of the web application hierarchy can be served to satisfy a request to /Catalog/index.html.
A special directory exists within the application hierarchy named “WEB-INF”. This directory contains all things related to the application that are not in the context root of the application. Note that the WEB-INF node is not part of the public document tree of the application. No file contained in the WEB-INF directory may be served directly to a client.
The contents of the WEB-INF directory are:

web.xml: It is a deployment descriptor. As of Servlets 3.0, the creation of this file can be omitted.
classes directory: It is for servlet and utility classes. The classes in this directory are used by the application class loader to load classes from.
lib directory: It contains all .jar files. It may contains servlets, beans, and other utility classes useful to the web application. All such archive files are used by the web application class loader to load classes.
src: It is optional and will contain source code.

See the figure given in the above slide. The context-root here is called servlets-demo, which contains all web elements for a specific web application. For example: To invoke index.html, the url will be: http://<server-name>:<port-no>/servlet-demo/html/index.html.
Note: If an IDE like Eclipse Luna 4.4 is used, the folder structure might be a bit different in the IDE!

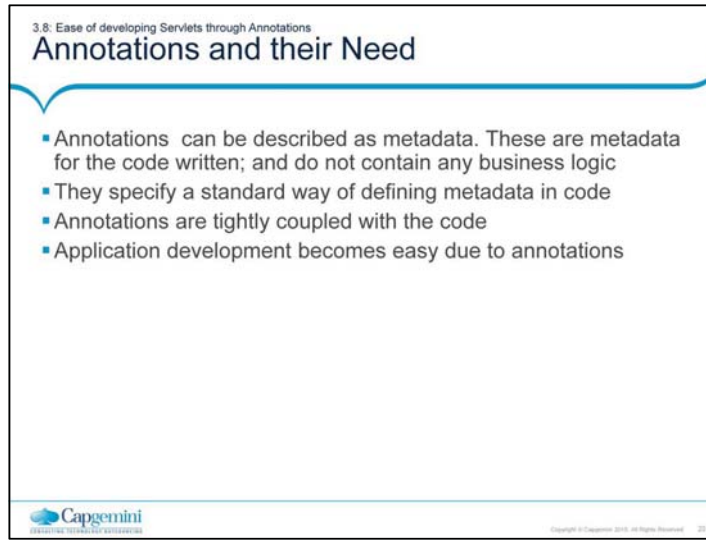


Elements of a Web Application:
Refer to Lesson 02 : To understand the steps for selecting the checkbox for web.xml

Web Application Configuration Descriptor (web.xml): is optional since Servlets 3.0
A deployment descriptor is an XML document with an .xml extension. It defines a component's deployment settings. The information provided by a deployment descriptor is declarative and therefore it can be modified without changing the source code of a bean. The Java EE server reads the deployment descriptor at run time and acts upon the component accordingly.
The web.xml file in the WEB-INF directory is a deployment descriptor. The following types of configuration and deployment information exist in the web application deployment descriptor:

- ServletContext Init Parameters
- Session Configuration
- Servlet / JSP Definitions
- Servlet / JSP Mappings
- Mime Type Mappings
- Welcome File list
- Error Pages
- Security

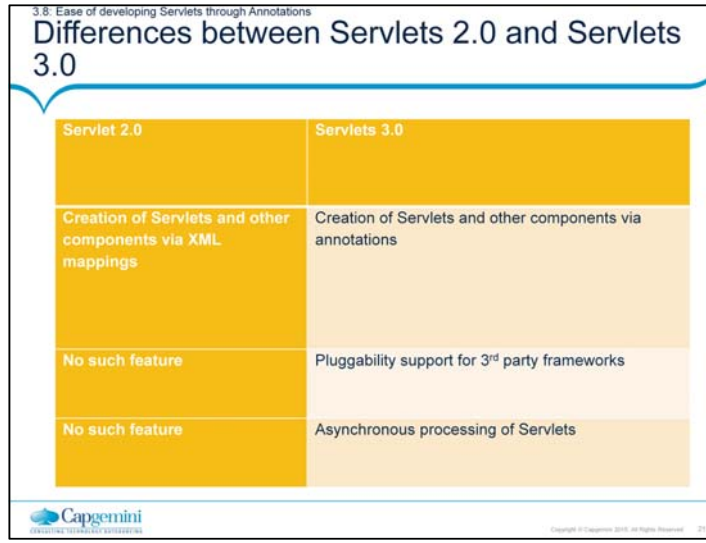
All of these types of information are conveyed in the deployment descriptor. A partial entry is shown in the code listing in the above slide.



Why Were Annotations Introduced?

Prior to annotation (and even after) XML were extensively used for metadata and somehow a particular set of Application Developers and Architects thought XML maintenance was getting troublesome. They wanted something which could be coupled closely with code instead of XML which is very loosely coupled (in some cases almost separate) from code. Interesting point is XML configurations were introduced to separate configuration from code.

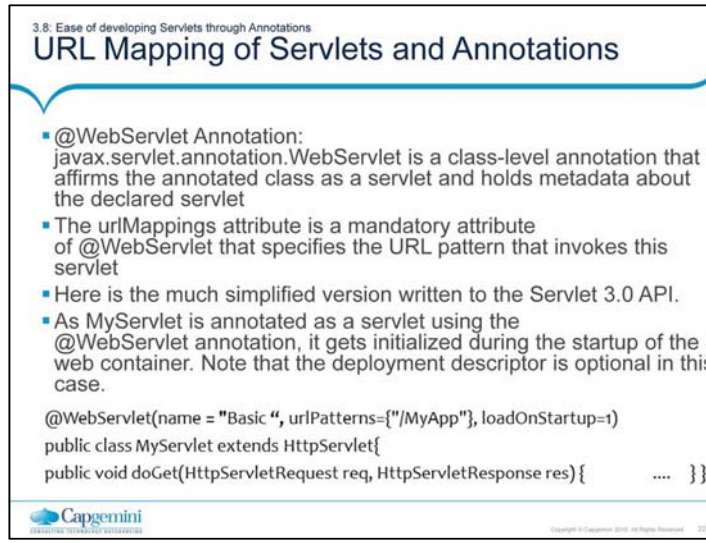
Suppose, you want to set some application wide constants/parameters. In this scenario, XML would be a better choice because this is not related with any specific piece of code. If you want to expose some method as a service, annotation would be a better choice as it needs to be tightly coupled with that method and developer of the method must be aware of this.
Another important factor is that annotation defines a standard way of defining metadata in code. Prior to annotations people also used their own ways to define metadata. Some examples are – using marker interfaces, comments, transient keywords etc. Each developer decided his own way to decide metadata, but annotation standardized things. These days most frameworks use combination of both XML and Annotations to leverage positive aspects of both.



Pluggability Support for 3rd Party frameworks and Asynchronous processing of Servlets - details of which are shared in the Appendix.

Creation of Servlets and other components via XML mappings – details of which are shared in the Appendix.

In the subsequent slides we would be seeing creation of Servlets and other components via annotations.


3.8: Ease of developing Servlets through Annotations
URL Mapping of Servlets and Annotations
@WebServlet Annotation: javax.servlet.annotation.WebServlet is a class-level annotation that affirms the annotated class as a servlet and holds metadata about the declared servlet
The urlMappings attribute is a mandatory attribute of @WebServlet that specifies the URL pattern that invokes this servlet
Here is the much simplified version written to the Servlet 3.0 API.
As MyServlet is annotated as a servlet using the @WebServlet annotation, it gets initialized during the startup of the web container. Note that the deployment descriptor is optional in this case.
@WebServlet(name = "Basic ", urlPatterns={"/MyApp"}, loadOnStartup=1)
public class MyServlet extends HttpServlet{
public void doGet(HttpServletRequest req, HttpServletResponse res){ .... }}
Capgemini

When a request arrives, the container matches the URL in the request with the servlet's URL Patterns and if the URL pattern matches, the corresponding servlet will be invoked to serve the request. All other attributes of this annotation are optional, with reasonable defaults.

The name attribute specifies the name of servlet. It could be different from Servlet class, URL Pattern. Rather is just a dummy name given to servlets. This attribute is not mandatory.

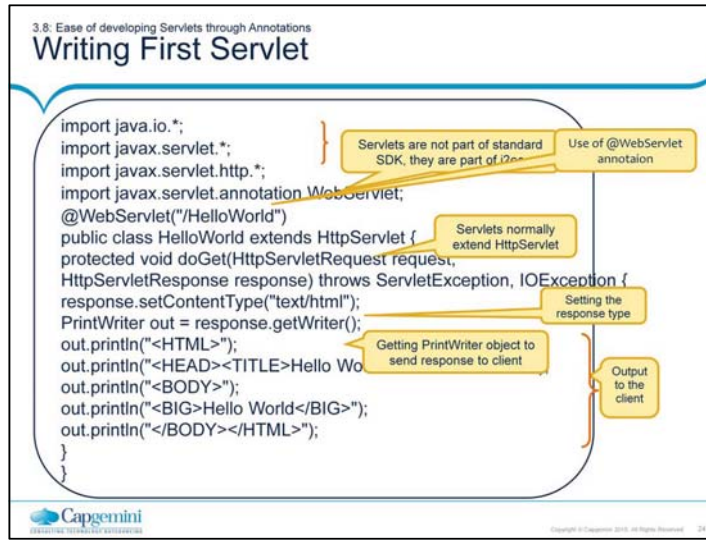
It also has loadOnStartup attribute, this is used to specify - If you want to do something on the startup of the servlet (i.e. the element loadOnStartup with value greater or equal to zero, 0), you need put the code in a init method or in the constructor of the servlet: As we know the init() method is the first life cycle method of servlet.

loadOnStartup also specifies that it will be the first servlet to execute before the client request comes for the servlet and the init() method would run.

For comparison, a code snippet for writing a Java servlet using the old Servlet 2.5 API is shown below. In Servlet 2.5, the web container will initialize the servlet only if you configure the details of the servlet in the deployment descriptor

```
public class MyServlet extends HttpServlet {
public void doGet (HttpServletRequest req, HttpServletResponse res)
{               ....

}
}
Deployment descriptor (web.xml)
<web-app>
                    <servlet>
                    <servlet-name>MyServlet</servlet-name>
                    <servlet-class>samples.MyServlet</servlet-class>
                    </servlet>
                    <servlet-mapping>
                    <servlet-name>MyServlet</servlet-name>
                    <url-pattern>/MyApp</url-pattern>
                    </servlet-mapping>
</web-app>
```



Elements of a Web Application:
The above servlet prints a simple hello world.

This code snippet is using @WebServlet annotation.
Here the URL pattern mapping is embedded within the same annotation. The class also extends HttpServlet and overrides the doGet(request, response) method.

Make use of the annotation @WebServlet above the class name which will create the servlet – HelloWorld inside the container. The servlet will now be referred by the name "HelloWorld". The "/HelloWorld" is the URL Pattern of servlet, that would be used to invoke servlet from web browser.

URL pattern could be any logical name example: "/Hello". It need not be necessarily same name of Servlet. A servlet could have multiple URL patterns.

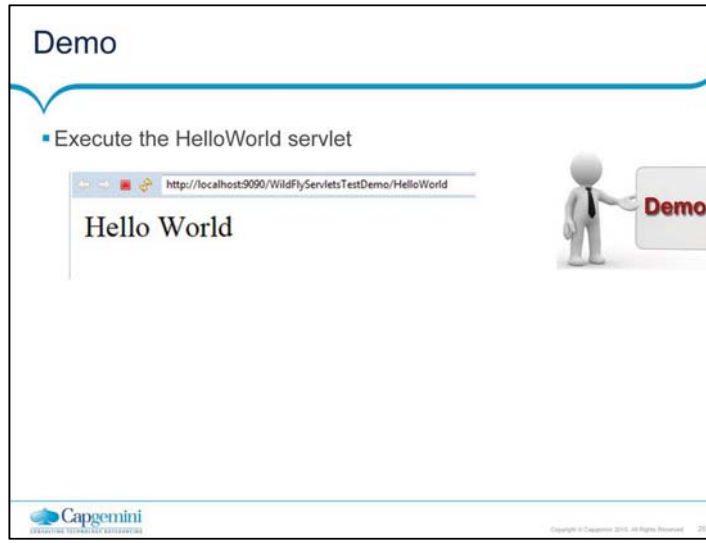
The doGet() method is overridden to service all incoming requests. Line 8 uses HttpServletResponse object to set the content type of the response. Line 9 uses the response object to retrieve a PrintWriter object to send responses. The remaining lines dynamically generate the HTTP response back to the browser client!
Following are minimum steps needed to create any servlet:

Write servlet
- Import the necessary Java packages.
- Inherit from GenericServlet or the HTTP convenience class HttpServlet.
- Override the service method or the doXXX() methods.
- Save the file with a .java filename extension.

2. Deploy the servlet in a container. The process for deploying (installing) a servlet into a webserver (container) varies from webserver to webserver.
Test the servlet by invoking the servlet from a JDK1.8-compatible browser. The URL typically will be of the form:
- http://host_name:port/servletcontext/servlet/servlet_class_name
- For example: http://localhost:9090/WildFlyServletsTestDemo/HelloWorld,

where WildFlyServletsTestDemo is the context root and HelloWorld is the name of the servlet.

If the server is installed on the same machine as the browser, then the server name can also be referred to as localhost.



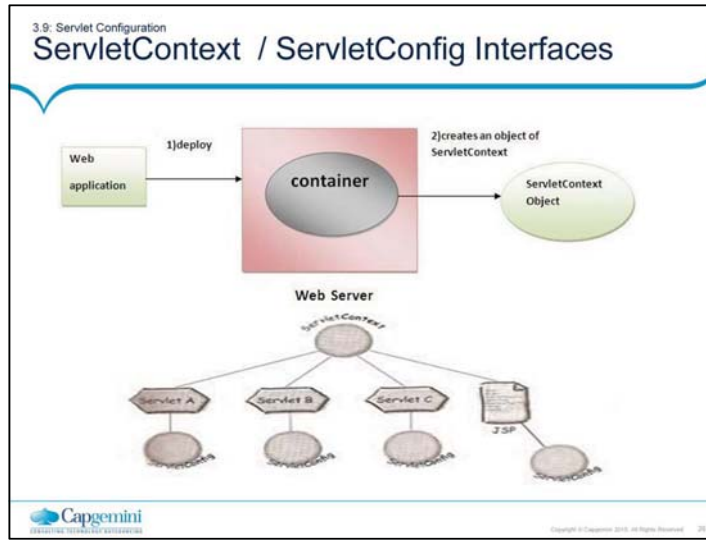
Note:
We shall use Eclipse Luna 4.4 with WildFly 8.x
To create a web application in Eclipse, follow the given steps:

In Project Explorer, right-click and select New → Dynamic Web Project.
Give a name for the project, say WildFlyServletsTestDemo. Ensure target runtime selected is WildFly 8.x, configure the server.
Check on the check box to create (web.xml – deployment descriptor) as it is optional under Servlets 3.0 and click Finish.
Create servlet class under src folder and within package controller . Refer HelloWorld.java class. At this point compile servlet, but since we are using IDE, this process is automatic. The servlet-api.jar is already part of build path, since we are using WildFly.
As we make use of annotation thus no changes required in web.xml.
Start the browser in Eclipse and run the servlet by invoking it with the following URL:

http://localhost:9090/WildFlyServletsTestDemo/HelloWorld

Hello World should be seen on the browser window!

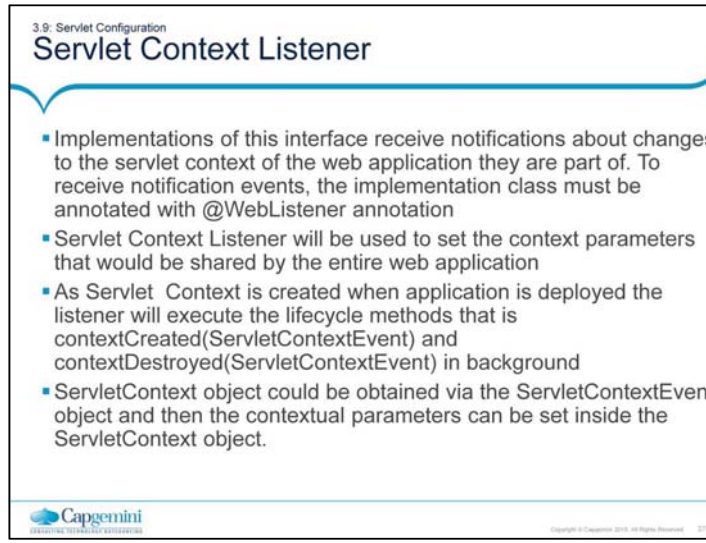
The web application could be packaged into a .WAR file.
Right click and select context root → Export → WAR File. Select the destination and click Finish.
This WAR file could be dropped into an web server like Wildfly and Tomcat (into its webapps folder) and invoke servlet using any browser!



The Container is responsible for creating the ServletContext object when the application gets deployed. ServletContext Object can be shared among multiple web components.

ServletConfig object will be created by the container for every servlet. This object cannot be shared between multiple web components.

More on these interfaces in the next subsequent slides



ServletContextListener is a listener which would get notified for the changes in the ServletContext once the ServletContext object gets created.

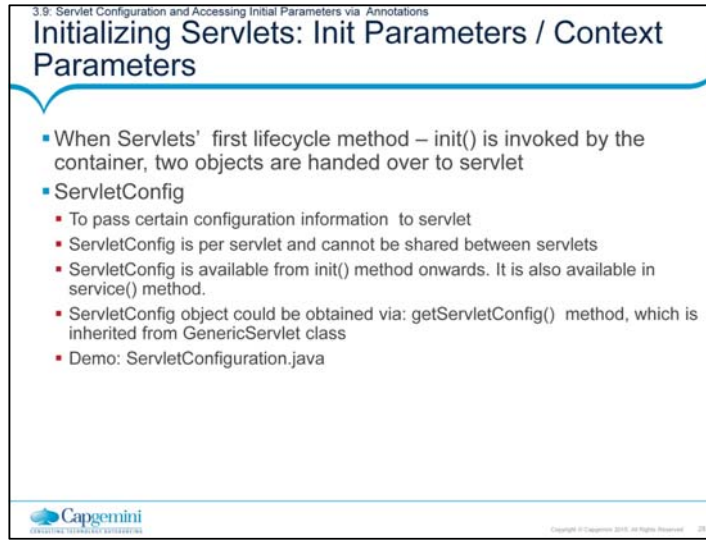
As ServletContext is created per Web application , data can be configured in listener which can be shared throughout the application.

Once the Listener is implemented need to override the life cycle methods as contextCreated() and contextDestroyed(), each of them taking a parameter of type ServletContextEvent; this object can be used to obtain the ServletContext object. (ServletContextEvent.getServletContext())

Refer below listing to have an idea about the Listener:

```
@WebListener
public class TestServletContextListener implements ServletContextListener {
public void contextDestroyed(ServletContextEvent sc) {
}
public void contextInitialized(ServletContextEvent sc) {
}

}
```



There are 2 versions of init:
init()
init(ServletConfig config)

Both these methods could be overridden. If we override parameterized version of init(ServletConfig config) there is a need to give a call explicitly to super.init(config). By seeing this call the container would invoke the parameterized init() from the GenericServlet hierarchy and then give a call for the default version of init() handing over the ServletConfig and ServletContext objects.

Even if we override default version of init(), the container would still invoke the parameterized init(ServletConfig config) from the GenericServlet hierarchy and then give a call for the default version of init() handing over the ServletConfig and ServletContext objects.

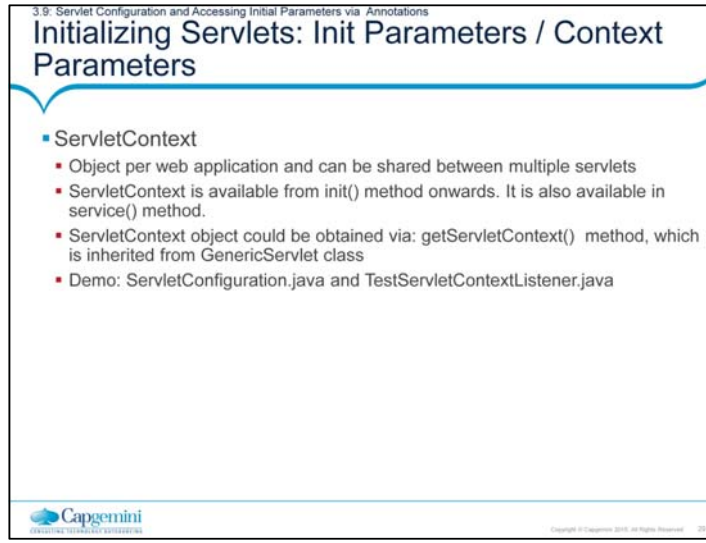
This is due to Servlet Hierarchy discussed in previous slides.

Since Servlet 3.0 we have made use of annotation to declare init parameter. Example:
@WebInitParam(name = "name", value = "My Servlet")

For Servlet Context parameters, there is a need to configure a Listener :
ServletContextListener – which will implement the lifecycle methods of ServletContext.
And then the contextual parameters could be set.

Refer Demo TestServletContextListener.java for same.

The context parameters could be then retrieved inside a servlet.



There are 2 versions of init:
init()
init(ServletConfig config)

Both these methods could be overridden. If we override parameterized version of init(ServletConfig config) there is a need to give a call explicitly to super.init(config). By seeing this call the container would invoke the parameterized init() from the GenericServlet hierarchy and then give a call for the default version of init() handing over the ServletConfig and ServletContext objects.

Even if we override default version of init(), the container would still invoke the parameterized init(ServletConfig config) from the GenericServlet hierarchy and then give a call for the default version of init() handing over the ServletConfig and ServletContext objects.

This is due to Servlet Hierarchy discussed in previous slides.

Since Servlet 3.0 we have made use of annotation to declare init parameter. Example:
@WebInitParam(name = "name", value = "My Servlet")

For Servlet Context parameters, there is a need to configure a Listener :
ServletContextListener – which will implement the lifecycle methods of ServletContext.
And then the contextual parameters could be set.

Refer Demo TestServletContextListener.java for same.

The context parameters could be then retrieved inside a servlet.

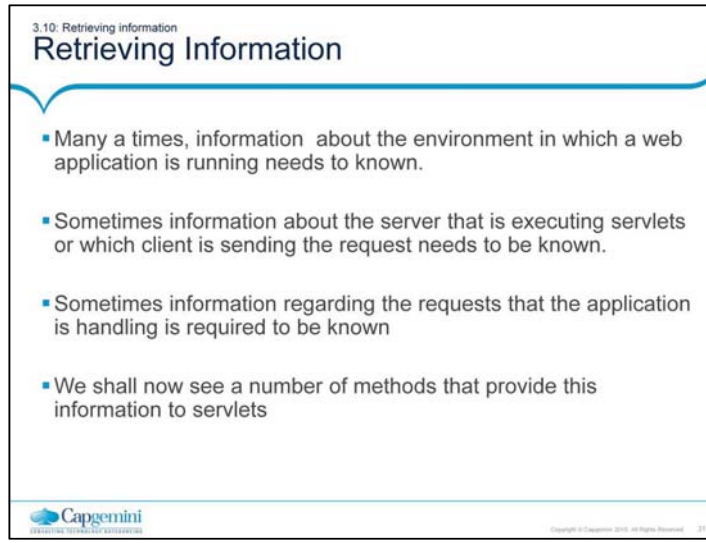
Note:
Refer com.igate.ch3.ServletConfiguration.java class. Notice that we have used GenericServlet to create the servlet.

For the ServletContext , a Listener is created called TestServletContextListener that would implement the Lifecycle methods of ServletContext.

```
public class InitSnoop extends GenericServlet {
  public void service(ServletRequest req, ServletResponse res)
          throws ServletException, IOException {
    res.setContentType("text/plain");
    PrintWriter out = res.getWriter();
    out.println("Init Parameters:");
    //retrieves all init parameters from web.xml
    Enumeration params = getInitParameterNames();
    while (params.hasMoreElements()) {
        String name = (String) params.nextElement();
        out.println(name + ": " + getInitParameter(name));
    }
  }
}
```

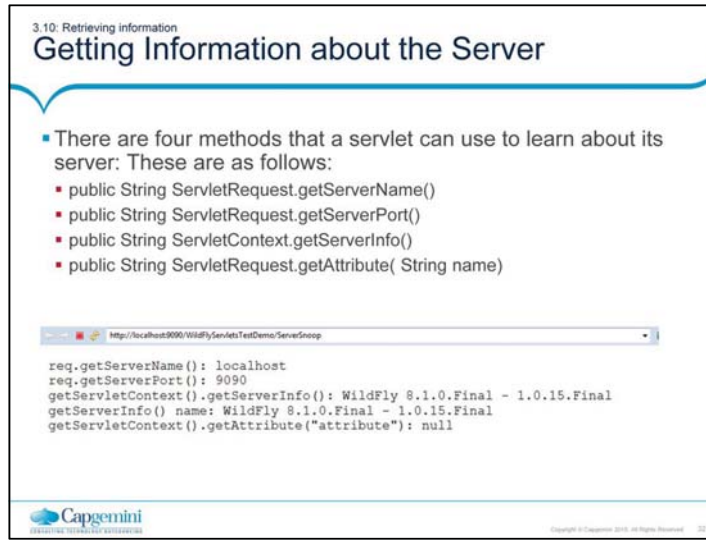
Notice how the getInitParameter() and getInitParameterNames() have been used to retrieve the appropriate init information. Invoke this servlet as: http://localhost:8080/Servlet-Demo/initsnoop

The browser shows the init parameters with values as shown above



In the Servlet Hierarchy, there is a Listener associated with different objects like ServletContext, HttpSession etc; that would be invoked when these objects are created. A Listener is a java class that would implement the ServletContextListener interface and would implement the lifecycle methods of ServletContext:
contextInitilized(ServletContextEvent se)
contextDestroyed(ServletContextEvent se)

We are using the annotation @WebListener above the Listener class name that would instruct the container to create the ServletContextListener and invoked it when a ServletContext object is created by Container, Data or configuration information that needs to be shared across multiple servlets could be set up in the ServletContext here. ServletContext is obtained via the ServletContextEvent.getServletContext() method.



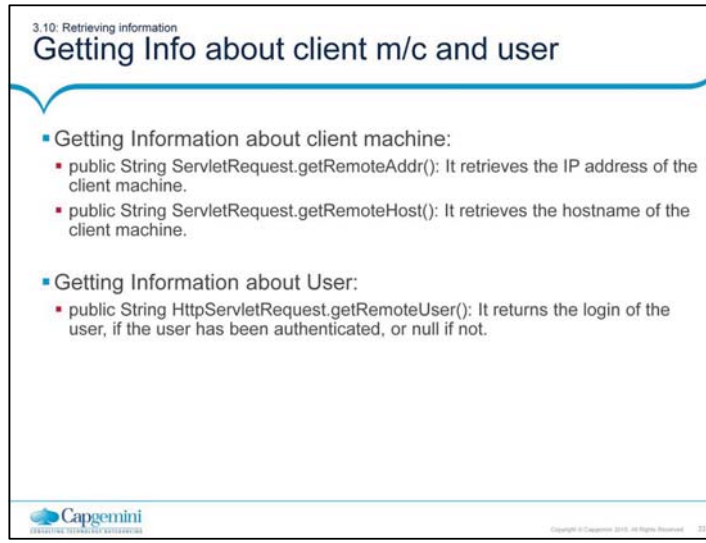
Retrieving Information:
Getting Information about the Server:
A servlet can find out much about the server in which it is executing. It can learn the hostname, listening port, and server software, among other things. A servlet can display this information to a client, use it to customize its behavior based on a particular server package, or even use it to explicitly restrict the machine on which the servlet will run.

getServerName() method returns the host name of the server to which the request was sent.

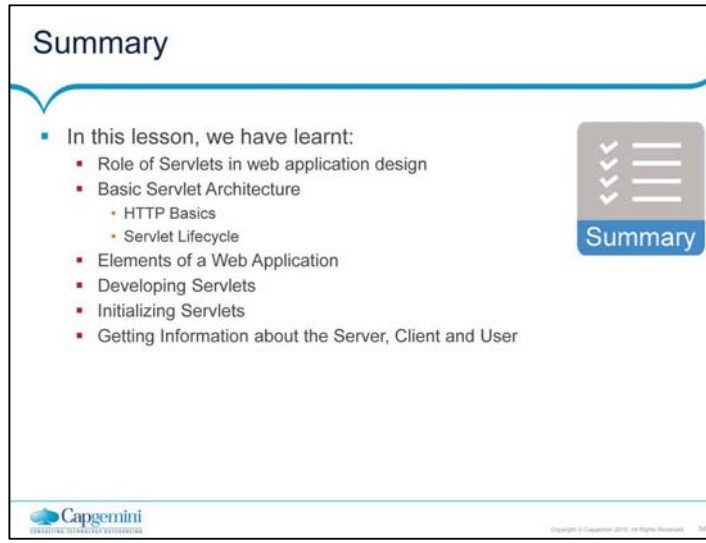
getServerPort() method returns the port number to which the request was sent

getServerInfo() method returns the name and version of the server software as: JBossWeb/2.0.1.GA.

getAttribute() returns the value of the named server attribute as an Object or null if the attribute does not exist. The attributes are server dependent. We shall see this in detail in the session on inter-servlet communication.



Retrieving Information:
Getting Info About client m/c and user:
A servlet has the ability to find out about the client machine and for pages requiring authentication, about the actual user. This information can be used for logging access data, associating information with individual users, or restricting access to certain clients.
A servlet can use getRemoteAddr() and getRemoteHost() to retrieve the IP address and hostname of the client machine, respectively.
The information comes from the socket that connects the server to the client, so the remote address and hostname may be that of a proxy server. An example remote address might be “192.26.80.1320” while an example of remote host might be “dist.engr.com”.
The method getRemoteUser () of the HttpServletRequest gives the username of the client. With the remote user’s name, a servlet can save information about each client. Over the long term, it can remember each individual’s preferences. For the short tern, it can remember the series of pages viewed by the client and use them to add a sense of state to a stateless HTTP protocol. A simple servlet that uses getRemoteUser() can greet its clients by name and remember when each user last logged in.



Summary:
This lesson introduced us to servlets, web applications. We have seen how to create simple servlets, install on server and execute it. We learnt about the servlet lifecycle and the Servlet API. We also saw how to initialize servlets using init() method and how to find information about the server, the client making the request and the user.

Review Questions

- Question 1: Which Method in a HttpServlet is not recommended to be overridden?
  - Option 1: init
  - Option 2: destroy
  - Option 3: service
  - Option 4: doGet
  - Option 5: doPost

- Question 2: What is the superclass of HttpServlet?
  - Option 1: GenericServlet
  - Option 2: Servlet
  - Option 3: SuperHttpServlet

Capgemini

Review Questions

- Question 3: Servlets are:
  - Option 1: Server-side components
  - Option 2: Client-side components
  - Option 3: Neither Client-side or Server-side components
  - Option 4: Data tier components

- Question 4: When a servlet receives an HTTP GET Request for the first time, which of these methods will be called? (Assume the servlet is not preloaded)
  - Option 1: init
  - Option 2: doPost
  - Option 3: processRequest
  - Option 4: doGet
  - Option 5: service

Capgemini

Review Questions

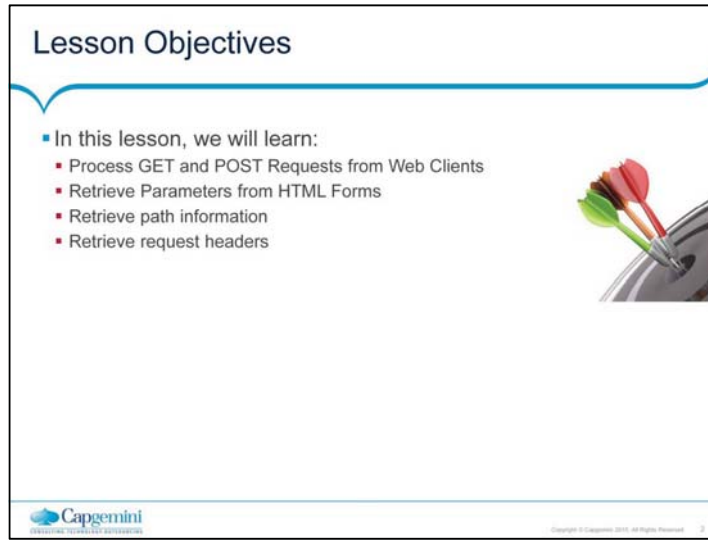
- Question 5: What is the significance of using annotation @WebServlet("/HelloWorld") above public class HelloWorld extends HttpServlet {}
  - Option 1: Instruct container to create Servlet Hello with URL Pattern "/HelloWorld"
  - Option 2: Instruct container to create Servlet HelloWorld with URL Pattern "/Hello"
  - Option 3: Instruct container to create Servlet HelloWorld with URL Pattern "/HelloWorld"
  - Option 4: No significanc

Review Questions

- Question 6: How to pass Initialization parameters to servlet
  - Option 1: creating mapping in xml file
  - Option 2: making use of annotation @WebInitParam
  - Option 3: making use of annotation @WebContextParam
  - Option 4: configuring a listener

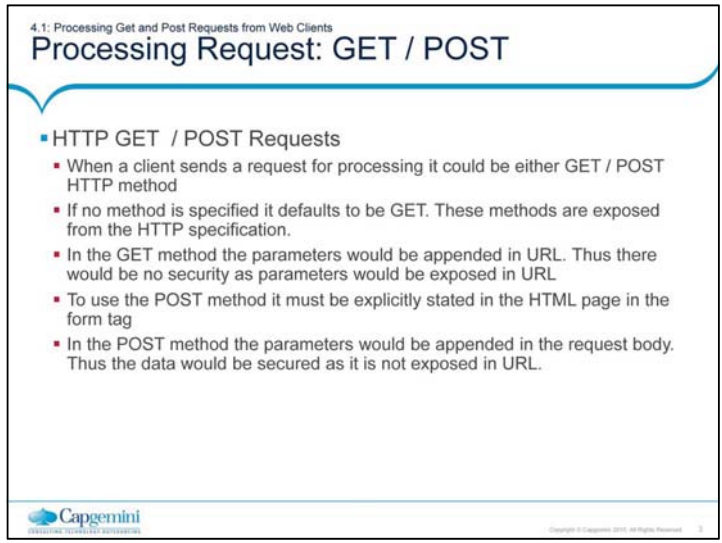
Capgemini


Java Servlets 3.0
Lesson 04: Request Object



Lesson Objectives:
This lesson introduces the Servlet request object. The lesson contents are:
Lesson 04: Request Object

- 4.1: Processing Get and Post Requests from Web Clients
- 4.2: Retrieve Parameters from HTML Forms
- 4.3: Retrieving Path Information
- 4.4: Retrieve Request Headers

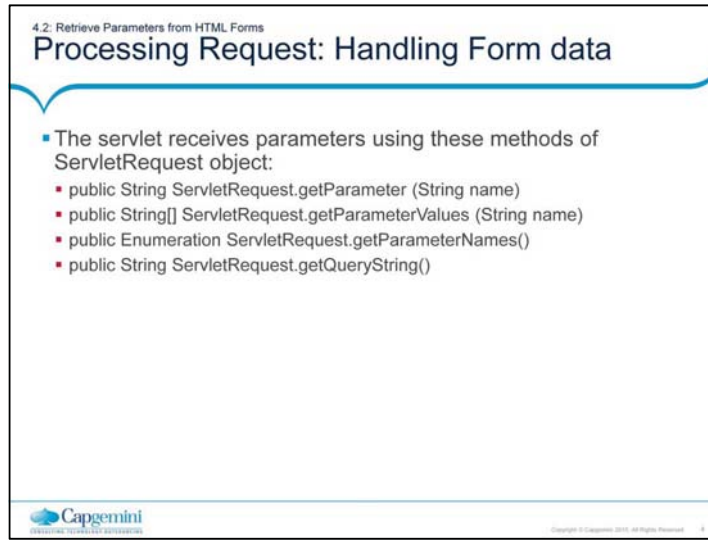


Need to explain that 2 methods could be used in Servlet to process the request (GET / POST).

The methods are present in the HTTP specification which are exposed in Servlet API.

Differences between them being:
GET : not secured and POST: secured
GET: data gets appended in URL and POST: data is appended as a part of request body
GET: limitation of data to be appended and POST: unlimited data could be appended

Other methods of the Http Specification are:
PUT : Replaces all current representations of the target resource with the uploaded content
HEAD : Same as GET, but transfers the status line and header section only.
DELETE : Removes all current representations of the target resource given by a URI.
TRACE : Performs a message loop-back test along the path to the target resource
OPTIONS : Describes the communication options for the target resource
CONNECT : Establishes a tunnel to the server identified by a given URI



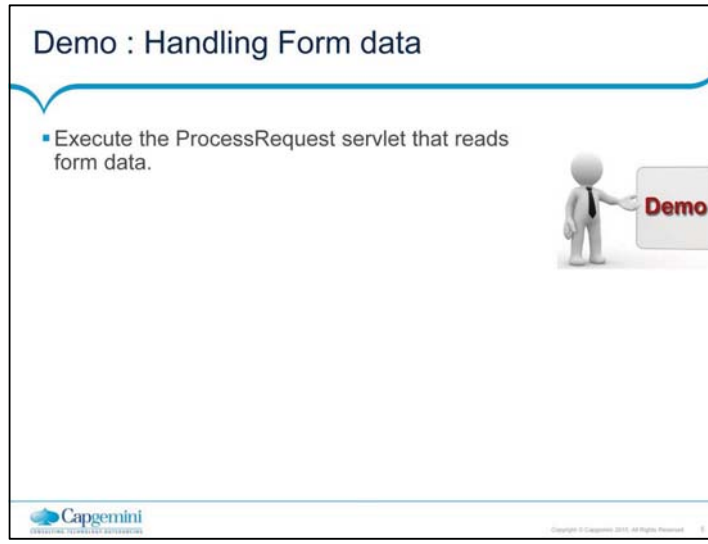
Processing Get and Post Requests from Web Clients:

Request – Information / data / parameters sent from the client (browser) to Server for processing and formulating a response.
When a servlet accepts a request from a client (service(Request , Response) method), the container creates and hands over two objects, ServletRequest and ServletResponse. The ServletRequest object encapsulates the communication from the client to the server, while the ServletResponse object encapsulates the communication from the server back to the client.

Since we are using HTTP protocol, the container would handover HttpServletRequest and HttpServletResponse objects to the service() method of HttpServlet class. Thias is done by downcasting ServletRequest to HttpServletRequest and ServletResponse to HttpServletResponse.

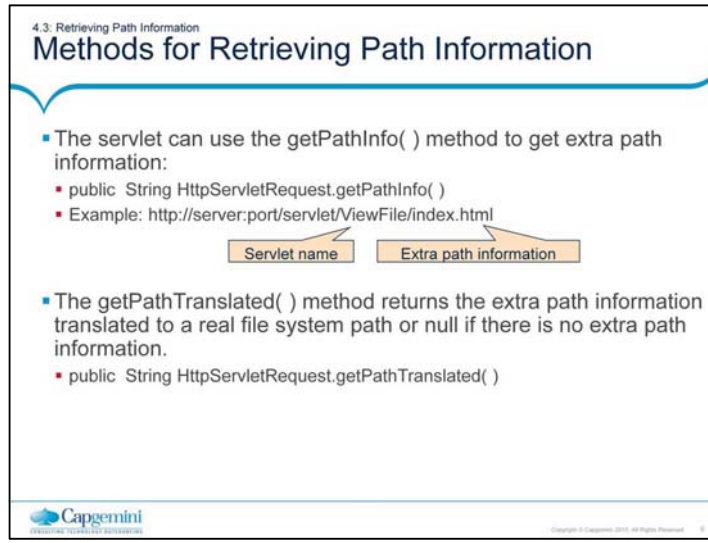
Request parameters:
The ServletRequest (or HttpServletRequest object if using HttpServlet) contains a lot of information. We shall see how servlet processes form parameters.

public String ServletRequest.getParameter (String name): This returns the value of the named parameter as a String or null if parameter wasn't specified.
public String[] ServletRequest.getParameterValues (String name): It returns values of named parameter as a String array if parameter could have multiple values, as in the case of list element with multiple select.
public Enumeration ServletRequest.getParameterNames(): It returns a list of parameter names.
public String ServletRequest.getQueryString(): It returns the raw query string of the request.



Note:
Refer the com.igate.ch4.ProcessRequest.java class and userInput.html for data to be submitted from html page.

Invoke this servlet as follows:
http://localhost:9090/RequestResponseDemoApp/userInput.html.
Make appropriate choices and key in data. On form submit, the ProcessRequest servlet retrieves all form parameters and displays them.



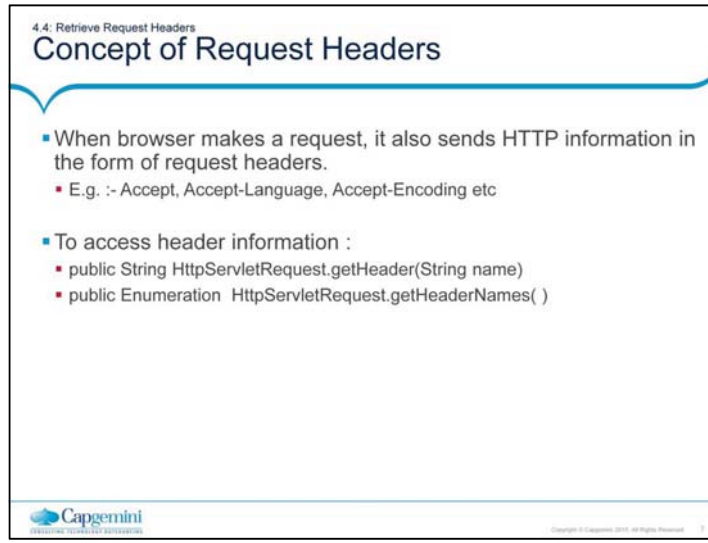
Retrieving Path Information:
In addition to parameters, an HTTP request can include something called “extra path information”. This extra path information can be used to indicate a file on the server or some piece of information that the servlet can use for something. This path information is encoded in the URL of an HTTP request. An example URL looks as follows:

http://server:port/servlet/ViewFile/index.html

This invokes the ViewFile servlet, passing “/index.html” as extra path information. A servlet can access this path information using the getPathInfo( ) method. String pathname = request.getPathInfo();
If the extra path information is the name of a file, then a servlet can get the actual file system location of the file using getPathTranslated( ) method:

String filename = request.getPathTranslated()

This method returns the extra path information translated to a real file system path or null if there is no extra path information.
Demo for this would be seen later during the File Upload concepts discussion.

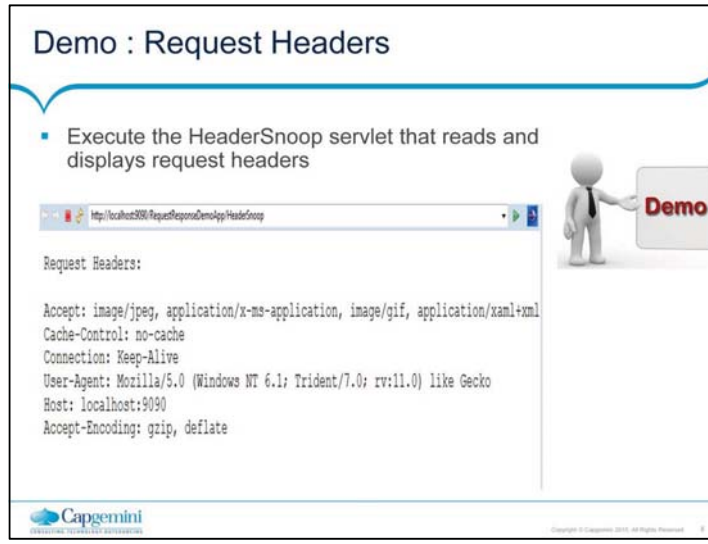


Request Headers:
When the client makes a request it also sends some HTTP information in the form of request headers. These are indirectly set by the browser, contain some extra information about the request and can be used by servlet to customize responses to the browser. Some examples of HTTP request headers are given below:

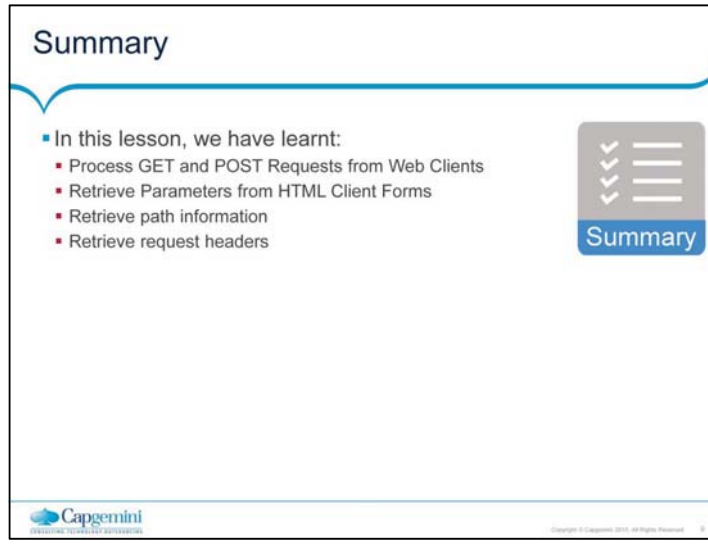
Accept: It specifies the MIME type that client accepts
Accept-Language: It specifies the language(s) that client can receive
Accept-Encoding: It specifies the encoding format that client can use
User-Agent: It gives information about client software

Header information can be accessed through the request object using several methods:

public String HttpServletRequest.getHeader(String name): It returns the value of the header as a String or null if header was not sent as part of request.
public String HttpServletRequest.getDateHeader(String name): It returns the value of the specified request header as a long value that represents a Date object.
public String HttpServletRequest.getIntHeader(String name): It returns the value of the specified request header as an int.
public Enumeration HttpServletRequest.getHeaderNames( ): It returns an enumeration of all the header names this request contains.



Note: Refer com.igate.ch4.HeaderSnoop.java class.
Invoke this servlet as follows:
http://localhost:9090/RequestResponseDemoApp/HeaderSnoop
Output could be seen as shown in the figure given in the above slide.


Summary
In this lesson, we have learnt:
Process GET and POST Requests from Web Clients
Retrieve Parameters from HTML Client Forms
Retrieve path information
Retrieve request headers
Summary
Capgemini

Review – Match the Following

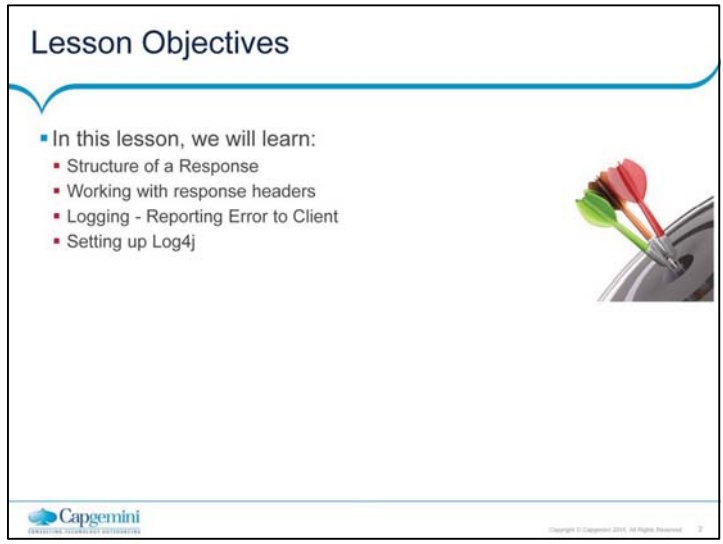
1.getPathInfo()	A.Returns the means by which HTTP request was made
2.getQueryString()	B.Returns extra path info
3.getRemoteHost ()	C.Translates given path into real path
4.getRealPath()	D.Returns string following the URL
5.getContentType ()	E.Returns the fully qualified name of the client
6.getMethod ()	F.Returns the MIME type of the body of the request

Capgemini

 10

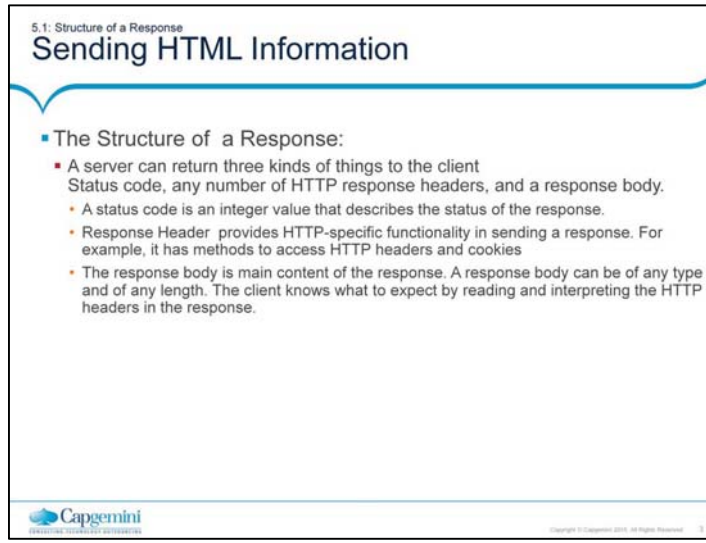
To Participants : refer to Appendix-B and extra API's for the Request object.


Java Servlets 3.0
Lesson 05:The Response Object



Lesson Objectives:
This lesson introduces Servlet's Response object. The lesson contents are as follows:
Lesson 05: The response object

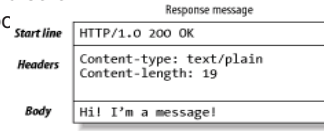
- 5.1: Structure of a Response
- 5.2: Working with Response Headers
- 5.3: Logging – Reporting Messages / Error to Client
- 5.4: Setting up Log4j

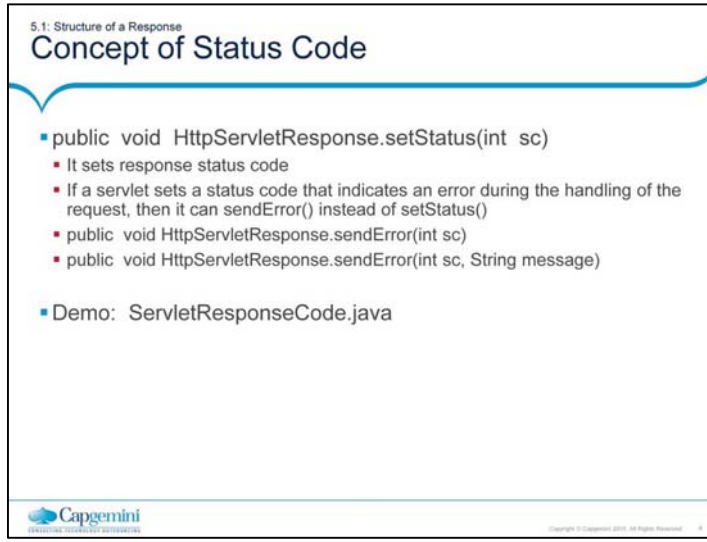


Structure of a Response:
As part of Servlet lifecycle, we saw how response object is obtained. We saw examples of servlets where we used response object to set response type (response.setContentType()) and obtain the PrintWriter object to send response to browser (response.getWriter()). We shall see the HttpServletResponse object in more detail.
When a web server responds to a request, the response typically consists of a status line, some response headers and the response body itself.

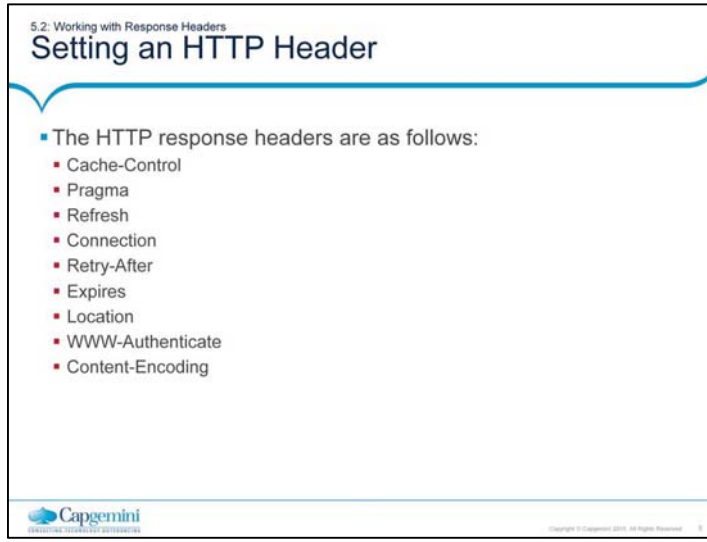
The status line consists of HTTP version, a status code, and a very short textual message corresponding to the status code.
The response headers are optional except for “content-type” which specifies the MIME type of the document being sent.
The response body contains the actual response that the browser lays out.

Servlets can perform a number of important tasks by manipulating the status line and the response headers. For example, Servlets can redirect the user to other sites, indicate the document type (example: image, PDF) to browser and tell the user that authentication is required to access any pages, and so on.
More details on Respc ext subsequent slides - notes page.



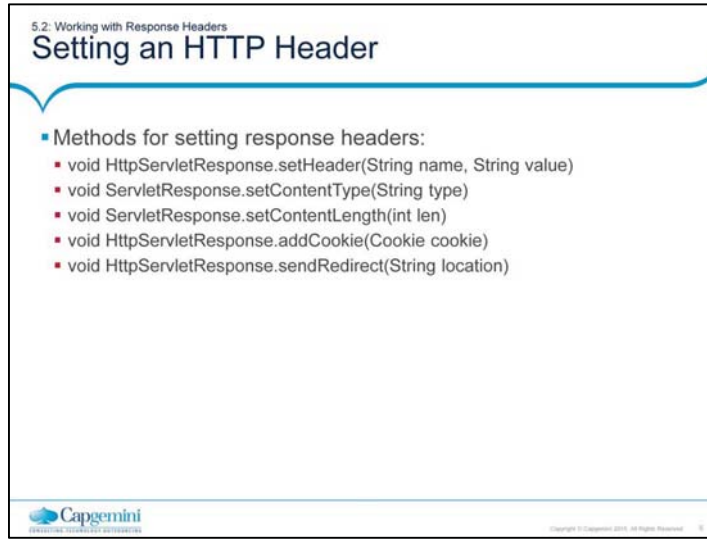


Concept of Status Codes:
As mentioned on the previous page, the status line consists of HTTP version, a status code, and a short textual message corresponding to the status code. The HTTP version is set by server and message is directly associated with status code. Therefore a servlet needs to only set the status code.
HTTP status codes are returned by the server to the client to determine the outcome of a request. The status code can indicate success or failure, or it can tell the client software to take further action to finish the request. A status code works behind the scenes and is interpreted by the browser software. If a servlet does not specifically set the status code, the server steps in and sets its value to the default 200 “OK” status.
Status codes are three-digit integer values, the first digit of which specifies one of five classes of response. So we have 1xx: Informational, 2xx: Success, 3xx: Redirection, 4xx: Client Error, and 5xx: Server Error.
Status codes are defined as mnemonic constants in the HttpServletResponse interface. For example: SC_OK for status code 200 and SC_NOT_FOUND for status code 404! For a full list, refer the HttpServletResponse class in the Servlet 3.0 documentation.
A Servlet can use the setStatus() method to set response status code. Remember, the setStatus( ) method should be called before the servlet returns any of its response body. A servlet can use sendError(int sc) (sc indicates the status code to indicate error during handling of request. In this case, the server generates and sends a server-specific page describing the error.)



Setting Response Headers:
In Lesson-4, we saw how client browser sends HTTP information in the form of request headers. Similarly, a servlet can set response headers to provide more information about its response. HTTP version 1.0 and 1.1 define a large number of request and response headers, description of which is beyond the scope of this course. Some of the most commonly used HTTP headers are listed below:

Cache-Control: It tells all caching mechanisms from server to client whether the browser may cache the object
For example: Cache-Control: no-cache
Pragma: The HTTP 1.0 equivalent of Cache-Control, with no-cache as its only possible value
Connection: It is used to indicate whether the server is willing to maintain an open connection to the client. If so, its value is set to keep-alive.
Retry-After: If an entity is temporarily unavailable, this instructs the client to try again after a specified period of time (in seconds)
For example: Retry-After: 120
Expires: It gives the date/time after which the response is considered stale. For example: Expires: Thu, 01 Dec 2009 16:00:00 GMT
Location: It is used in redirection, or when a new resource has been created. Its value must be a fully qualified URL
WWW-Authenticate: It indicates the authentication scheme that should be used to access the requested entity. WWW-Authenticate: Basic
Content-Encoding: The type of encoding used on the data.
For example: Content-Encoding: gzip
Refresh: It is used in redirection, or when a new resource has been created

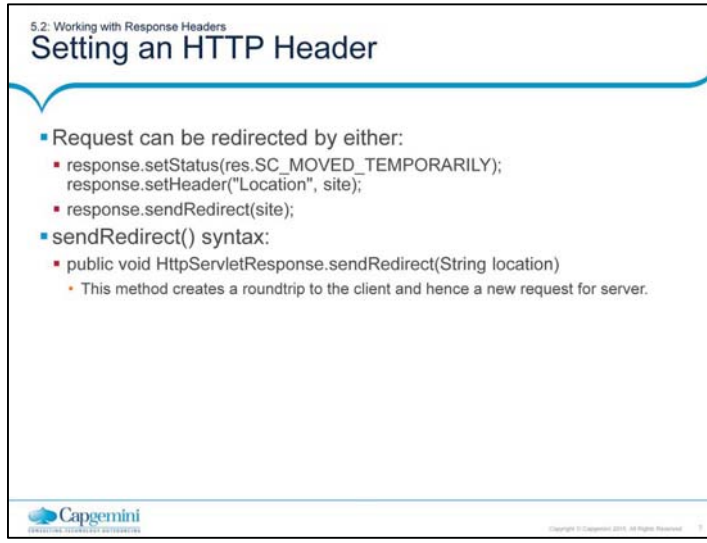


Setting Response Headers:
HTTP response headers can be used to give forwarding location, specify cookies, supply the page modification date, instruct the browser to reload the page after a designated interval, designate the type of document being generated, and so on.
Methods for setting arbitrary response headers are as follows:

The setHeader(String name, String value) method sets the value of the named header as a string. For example:
res.setHeader("Location", new_location);
Use the setDateHeader(String name, long date) method, if there is a need to specify a timestamp for a header. It sets the value of the named header with the given name and date-value
Use the setIntHeader(String name, int value) method to specify an integer value for a header
The containsHeader() method returns a boolean indicating whether the named response header has already been set

Other than these methods, there are some additional methods to set response headers:

setContentType: Sets the Content- Type header
setContentLength: Sets the Content- Length header
addCookie: Adds a value to the Set- Cookie header
sendRedirect: Changes status code to redirection i.e. 300 and a specifies the client to request another page
(Refer to Slide 6.2 – Client-side dispatch for further details on sendRedirect)



Redirecting a Request:
One of the useful things a servlet can do using status codes and a header is redirecting a request. This is done by sending instructions to the client, to use another URL in the response. Redirection is generally used when a document moves (to send the client to the new location), for load balancing (so one URL can distribute the load to several different machine), or for simple randomization (choosing a destination at random).
The actual redirection happens in two lines:

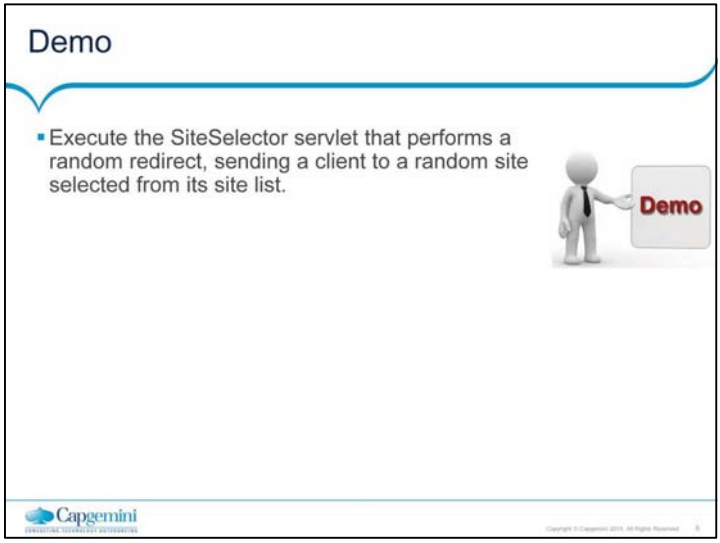
```
res.setStatus(res.SC_MOVED_TEMPORARILY);
res.setHeader("Location", site);
```

The first line sets the status code to indicate a redirection is to take place, while the second line gives the new location.
These two lines can be simplified to one using the sendRedirect( ) convenience method:

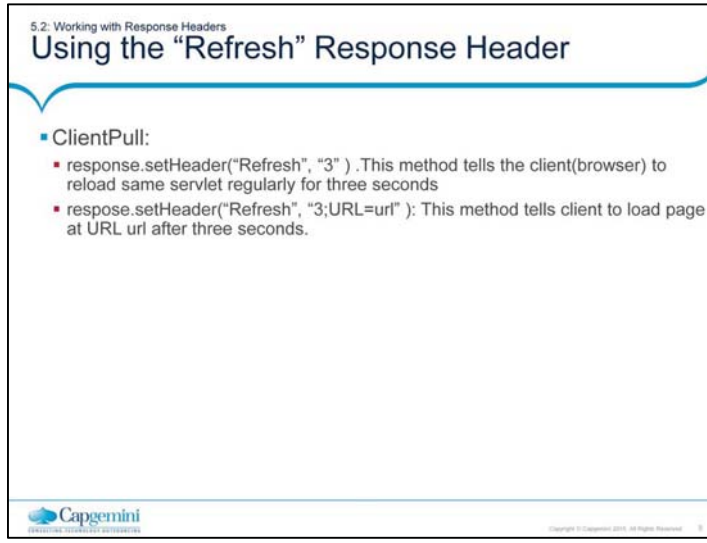
```
public void HttpServletResponse.sendRedirect(String location)
throws IOException.
```

This method redirects the response to the specified location, automatically setting the status code and location headers. For our example, the two lines simply become:

```
res.sendRedirect(site);
```



Note:
Refer to com.igate.ch4.SiteSelector.java class. A partial listing is shown below:
Invoke this servlet as:
http://localhost:9090/RequestResponseDemoApp/SiteSelector
The servlet randomly picking one site from the vector list and displays.



ClientPull:
Client pull is similar to redirection, with one major difference – the browser actually displays the content from the first page and waits for some specified amount of time before retrieving and displaying the content from the next page. It is called client pull because the client is responsible for pulling the content from the next page.
There are two reasons for using client pull.

First, to communicate to the client that requested page has been moved before the next page is automatically loaded

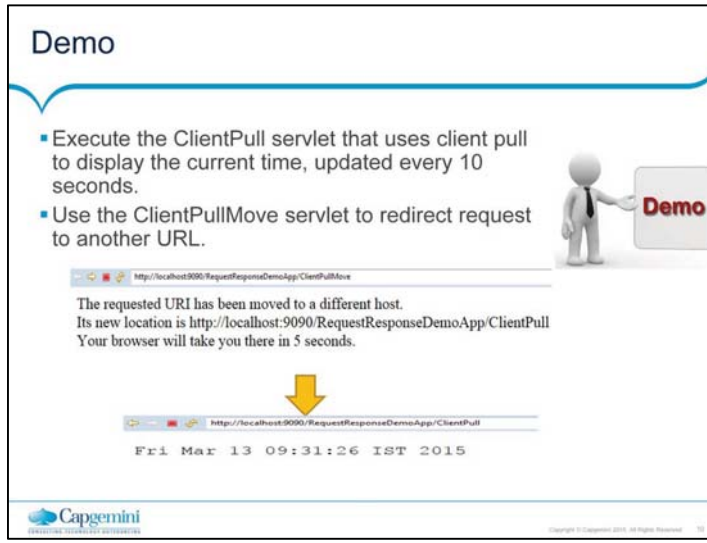
Second, pages can be retrieved in sequence, making it possible to present a slow-motion page animation.

Client pull information is sent to the client using the Refresh HTTP header. This header’s value specifies the number of seconds to display the page before pulling the next one, and it optionally includes a URL string that specifies the URL from which to pull. If no URL is given the same URL is used. Some examples follow:

A call to setHeader() that tells the client to reload this same servlet after showing its current content for three seconds:

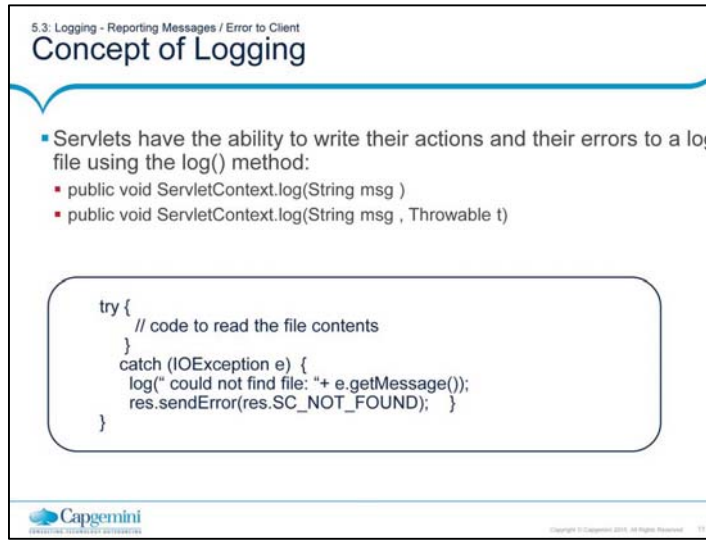
```
setHeader(“Refresh”, “3” );
setHeader(“Refresh”, “3;URL=https://ispace.igate.com”);
```

method call tells client to display IGATE’s home page after three seconds.



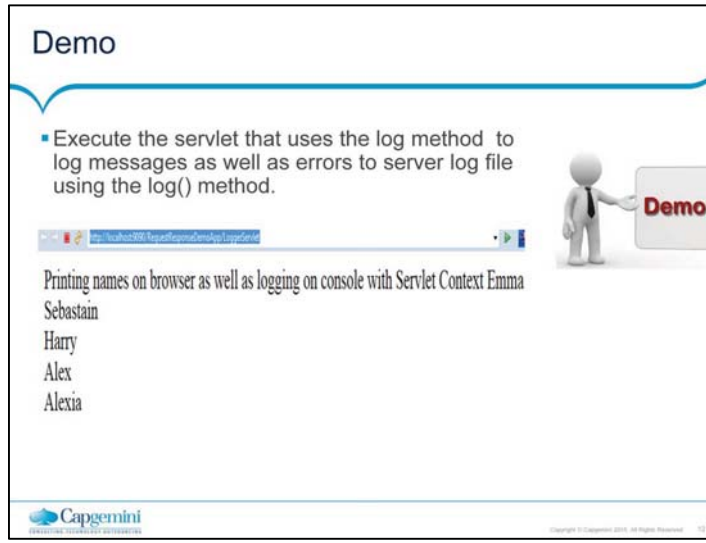
Note:
Refer the following classes:
com.igate.ch4.ClientPull.java
com.igate.ch4.ClientPullMove.java classes.
A partial listing of both these servlets is shown below:

Invoke the ClientPullMove servlet as:
http://localhost:9090/RequestResponseDemoApp/ ClientPullMove
See the servlet ClientPullMove redirecting to ClientPull servlet after 5 seconds. The ClientPull servlet displays updated time after every 10 seconds.



Logging:
Logging of messages is a common requirement in all applications. In a server environment, it is a critical feature due to the distributed multi-user interaction that is characteristic of a server. Many users interact simultaneously with an application server and some degree of logging of the interactions is essential for support.
The log() method aids debugging by providing a way to track a servlet's actions. It also offers a way to save a complete description of any errors encountered by the servlet. The description can be the same as the one given to the client, or it can be more exhaustive and detailed.
The single-argument method writes the given message to a servlet log, which is usually event log file. The two-argument version writes an explanatory message and a stack trace for a given Throwable exception to the servlet log file. The name and type of the servlet log file is specific to the servlet container, usually an event log.
Refer to the code snippet in the above slide.

In the catch block, log() is used to record on the server, when requested files do not exist, while returning a simple "404 Not Found" page to the client.



Note:
Refer the com.igate.ch4.LoggerServlet.java class. A partial listing is shown below:
Invoke this servlet as:

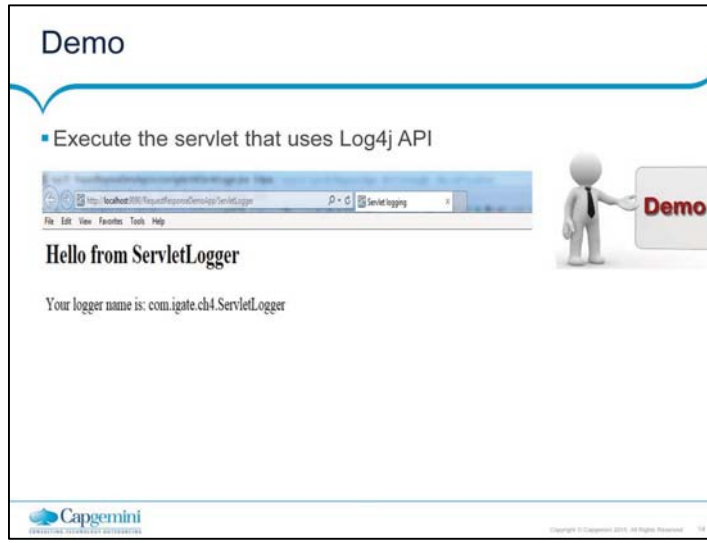
http://localhost:9090/RequestResponseDemoApp/LoggerServlet

The servlet checks for the filename to be read and prints out the contents. For WildFly, this is the server.log file found in the wildfly-8.1.0.Final\standalone\log folder. The output for this servlet is as seen in the figure above.

The partial contents of the log file are: This shows servlet has logged the usernames in server.log file.

2015-03-16 05:23:00,940 INFO [io.undertow.servlet] (default task-1) LoggerServlet: Servlet context logging the users>>>Emma
2015-03-16 05:23:00,940 INFO [io.undertow.servlet] (default task-1) LoggerServlet: Servlet context logging the users>>>Sabastian
2015-03-16 05:23:00,950 INFO [io.undertow.servlet] (default task-1) LoggerServlet: Servlet context logging the users>>>Harry
2015-03-16 05:23:00,950 INFO [io.undertow.servlet] (default task-1) LoggerServlet: Servlet context logging the users>>>Alex
2015-03-16 05:23:00,950 INFO [io.undertow.servlet] (default task-1) LoggerServlet: Servlet context logging the users>>>Alexia


5.4: Setting up Log4j
Log4j with Servlet
Log4j can be setup with Servlets for standard logging facility
Need to refer to Log4j package under Apache Software License
Need to place the jar file (log4j.jar) under WEB-INF directory of the web application created
Create a Log4j properties file and place it in the web application's WEB-INF directory
This is typically a text file with name / value pairs for configuring log4j elements like loggers, appenders and layouts
In servlets logger could be used
Capgemini

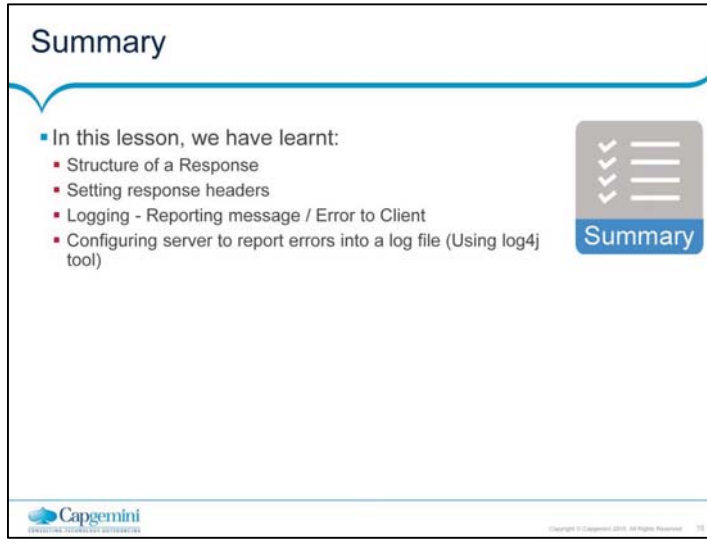


Refer the com.igate.ch4.ServletLogger.java class. A partial listing is shown below:

Invoke this servlet as follows:

http://localhost:9090/RequestResponseDemoApp/ServletLogger

The servlet first executes the init() method and logs the message to both – console and log.out file (under D:\usr\home\log4j). It then logs an info and a debug level message. The log.out file will contain both, but console will only show the info-level message. The output for this servlet is as seen in the figure above.


Summary
In this lesson, we have learnt:
Structure of a Response
Setting response headers
Logging - Reporting message / Error to Client
Configuring server to report errors into a log file (Using log4j tool)
Summary
Capgemini

Review Questions

- Question 1: The sendRedirect method defined in the HttpServlet class is equivalent to invoking the setStatus method with the following parameter and a Location header in the URL.

  - Option 1: SC_OK
  - Option 2: SC_MOVED_TEMPORARILY
  - Option 3: SC_NOT_FOUND
  - Option 4: SC_INTERNAL_SERVER_ERROR
  - Option 5: SC_BAD_REQUEST

Capgemini

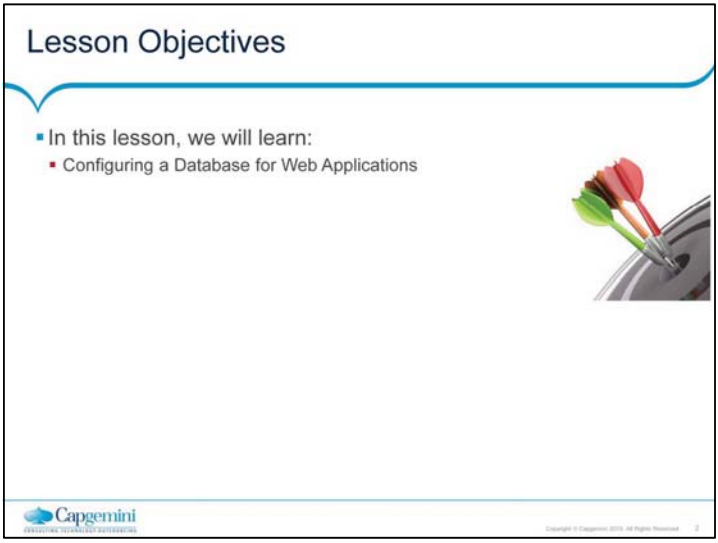
Review Questions

- Question 2: Browsers that support content encoding feature, indicate that they do so by setting the ___ request header.
  - Option 1: Cache-Control
  - Option 2: Encoding-true
  - Option 3: Accept-Encoding
  - Option 4: Content-Encoding

- Question 3: The ___ method aids debugging by providing a way to track a servlet's actions.
  - Option 1: debug()
  - Option 2: log()
  - Option 3: logFile()

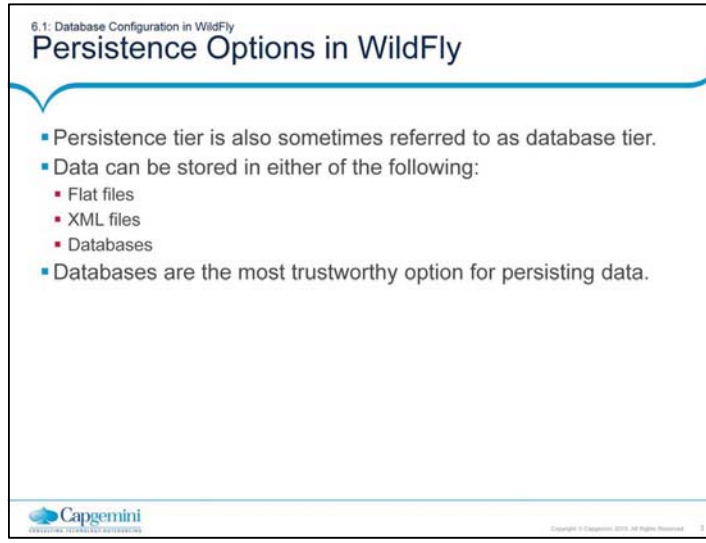
Capgemini


Java Servlets 3.0
Lesson 06: Configuring Database in WildFly

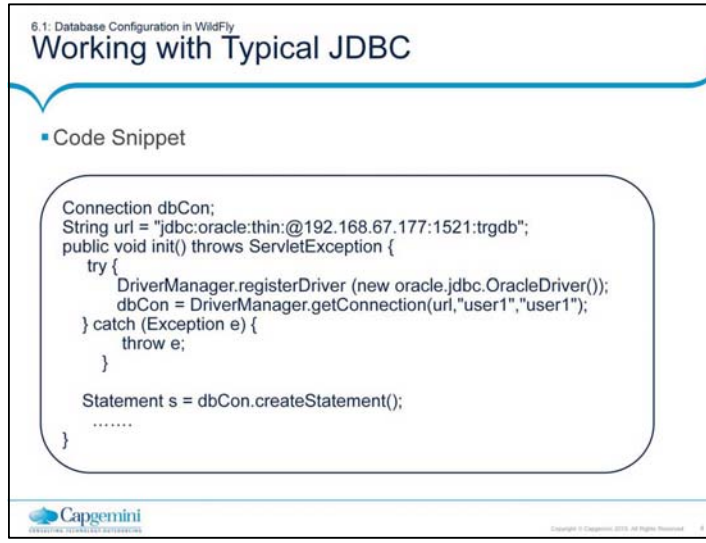


Lesson Objectives:
This lesson introduces database configuration in WildFly.

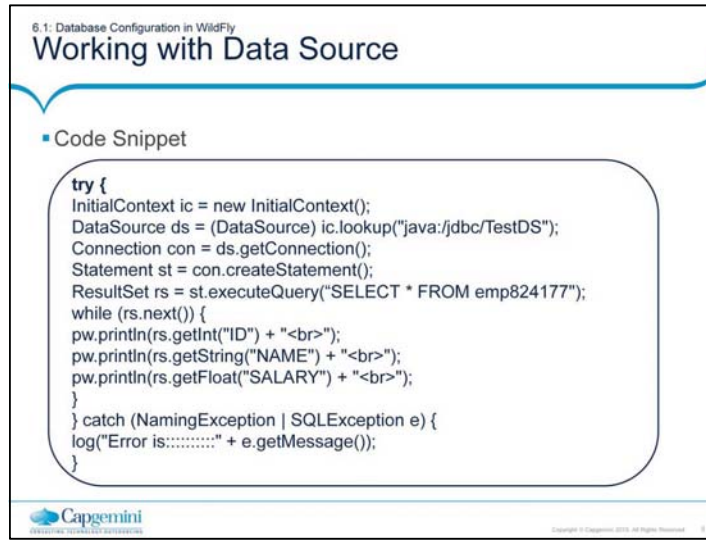
The lesson contents are as follows:
Lesson 06: Configurinag Database in WildFly
6.1: Database configuration in WildFly



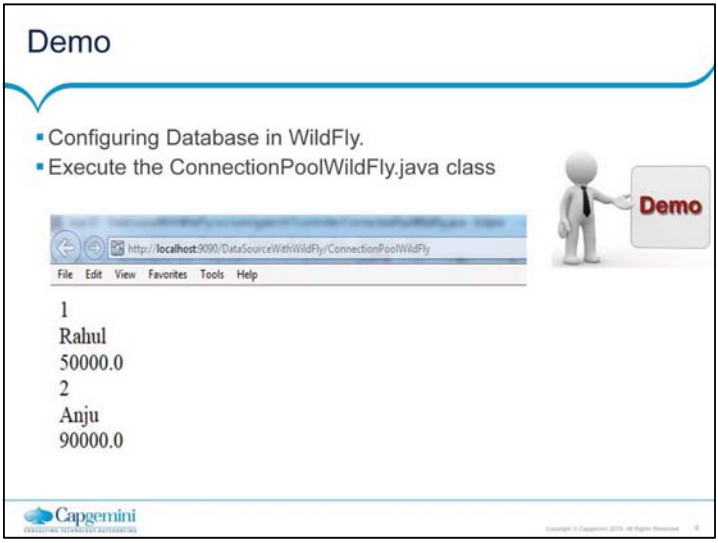
Persistence Options in WildFly:
Till now we have focused on designing simple web application, the focus was on the web tier. However, for any web application, the data storage is an integral part. Now, let us move to the persistence tier, which is also sometimes referred to as database tier. This is where application data is stored for a long term / persistently.
While working with JEE applications, data can be stored in Flat files, XML files, or Database. Databases are the most trustworthy options for persisting data.
Here, we all need to accept one simple fact. While dealing with relational databases in a JEE application, all paths some way or the other eventually lead to JDBC.
We are already familiar with JDBC.



Working with Typical JDBC:
In the previous chapter (Request-Response-DBServlet), already typical JDBC code in a Servlet is shown to interact with the database. The above slide shows a snippet of the code.
While this code definitely works, it does have some drawbacks:
Every time that we connect and disconnect from the database, the physical database connection is created and destroyed. This leads to an overhead.
If there are any transactions related to the applications, then they need to be managed by developer.
If the application deals with only local transactions within the database, it is fine. However, in case it interacts with other resources, then managing the transactions and other aspects is difficult within the application.



Working with DataSource:
While working with an application server, gives a lot of benefits. In the previous slide, we mentioned a few drawbacks with the typical JDBC approach of programming in a web application. However, these issues can be taken care of very easily in an Application Server like WildFly.
The above slide makes use of datasource, which is the change that is required to obtain Database connections. Instead of importing java.sql.DriverManager, import javax.sql.DataSource. Furthermore, since we would be using JNDI naming for looking up the DataSource, need to import javax.naming.InitialContext and javax.naming.NamingException.
Using the DataSource has its own advantages:
When obtaining a Database connection using DataSource, no need to create a new connection. During startup WildFly creates a DB connection pool managed by DataSource. So essentially when accessing an existing connection from the pool and when the connection is closed, it is just returned back to the pool so someone else can use it.
When we use Container Managed DataSource (In this case JBoss Server Managed), all database access for a particular Transaction commits or rollback automatically. No need to manage the transaction anymore.
Let us now take some time and understand the code.



Note:
Steps to configure the database for your application in WildFly
Database Configuration:

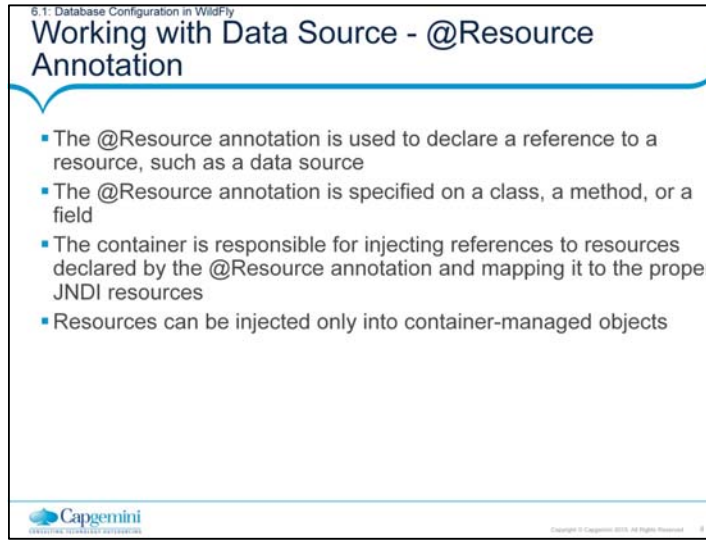
Refer to Lab Book: Steps to configure Database connection pool in WildFly in Lab 4:

Demo

- Partial listing of standalone.xml

```
<datasource jta="false" jndi-name="java:/jdbc/TestDS" pool-
name="TestDS" enabled="true" use-ccm="false">
<connection-
url>jdbc:oracle:thin:@localhost:1522:XE</connection-url>
<driver-class>oracle.jdbc.OracleDriver</driver-class>
<driver>ojdbc6_g.jar</driver>
<security>
        <user-name>system</user-name>
        <password>igate</password>
</security>
<validation>
         <validate-on-match>false</validate-on-match>
        <background-validation>false</background-validation>
</validation>
<statement>
        <share-prepared-statements>false</share-prepared-
statements>                </statement>
</datasource>
```

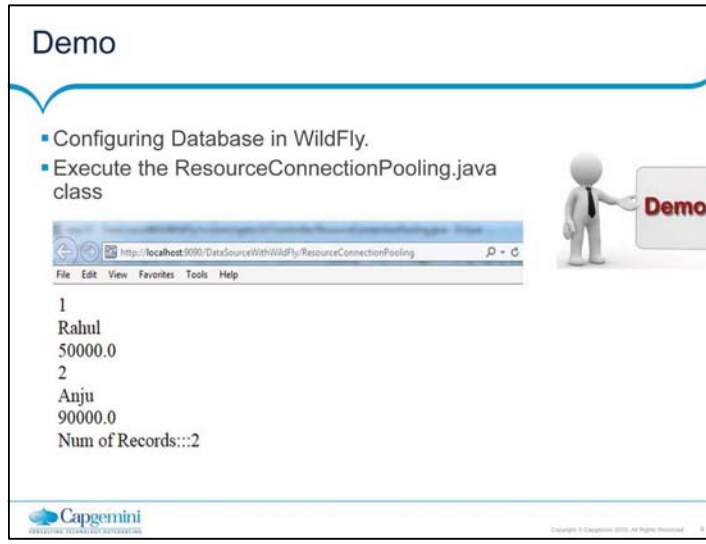
Capgemini



@Resource annotation is used to inject a data source into a component that needs to make a connection to the data source, as is done when using JDBC technology to access a relational database:

The container injects this data source prior to the component's being made available to the application. The data source JNDI mapping is inferred from the field name (example: TestDS) and the type, javax.sql.DataSource.

Container managed objects are the web components (Servlets / JSPs).Resources can be injected only into container-managed objects. Resource injection is convenient for the developer. Since a container must have control over the creation of a component it can perform the injection into a component. As a result, resources cannot be injected into objects such as simple JavaBeans components



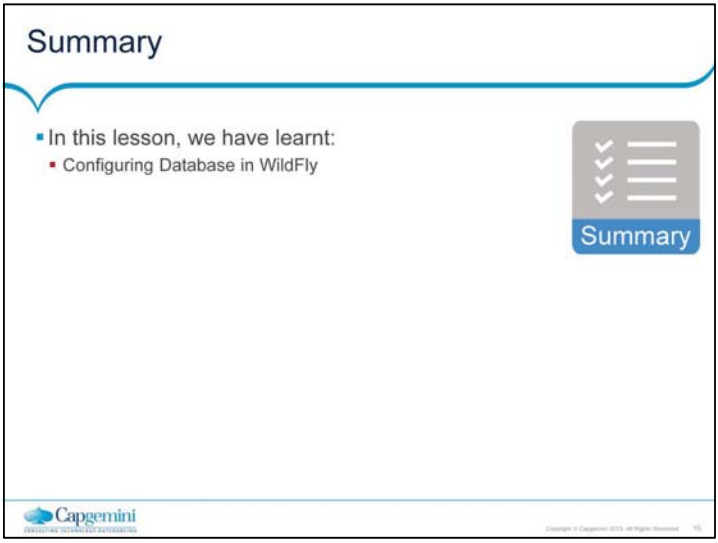
Execute the following servlet:

http://localhost:9090/DataSourceWithWildFly/ResourceConnectionPoolig

Partial listing of code is given below: which applies the field – level annotation

```
@Resource(lookup="java:/jdbc/TestDS")
DataSource ds;

try {
Connection con = ds.getConnection();
-------
} catch (SQLException e1) {
log("Error is::::::::::" + e1.getMessage());
}
```



Add the notes here.

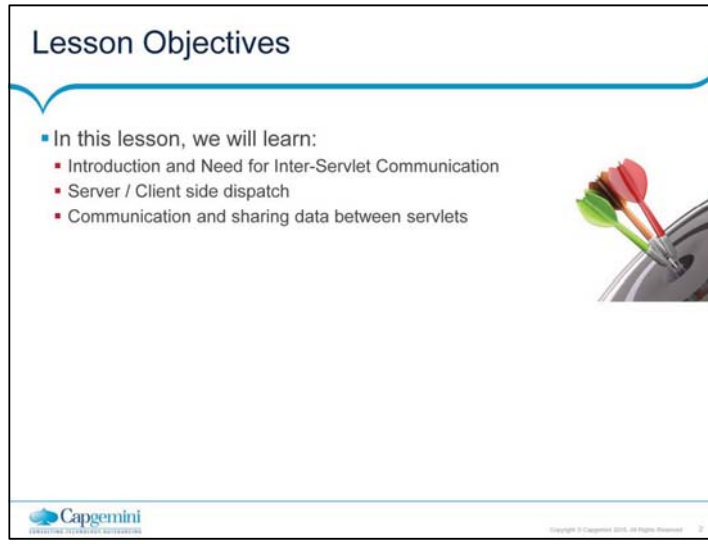
Review Questions

- Question 1: The package to be imported for using DataSource is ___.
- Question 2: The Datasource lookup happens in standalone.xml
  - True/ False

Capgemini

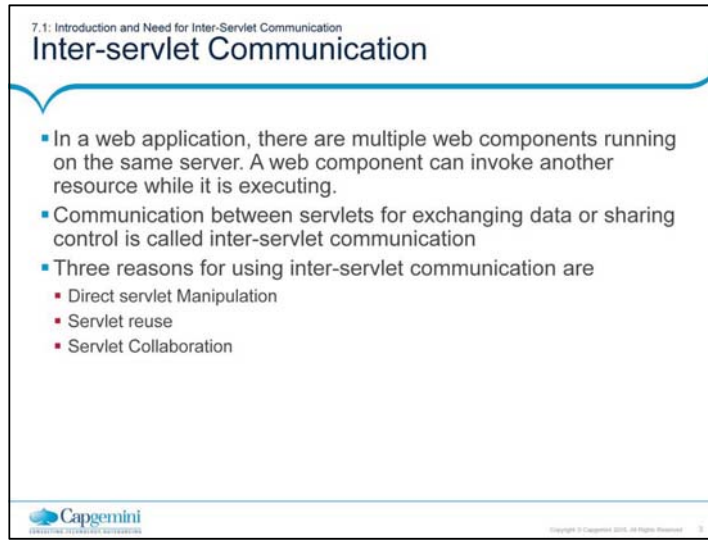
Add the notes here.


Java Servlets 3.0
Lesson 7: Inter-Servlet Communication



Lesson Objectives:
This lesson introduces Inter-Servlet communication. The lesson contents are:
Lesson 07: Inter-Servlet communication

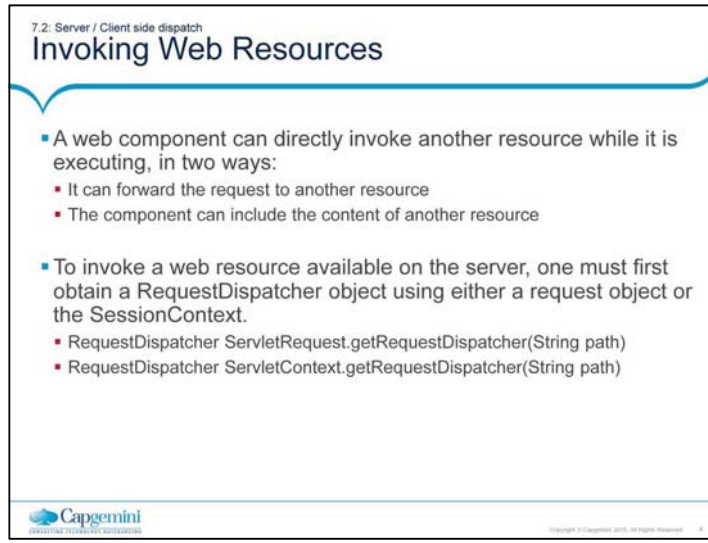
- 7.1: Introduction and Need for Inter-Servlet Communication
- 7.2: Server / Client side dispatch
- 7.3: Communication and sharing data between servlets



Introducing Inter-Servlet Communication:
The communication between servlets is called Inter-Servlet Communication. In fact, servlet being a web component can invoke another web component like html, jsp and other servlets too.
Servlets running together in the same server communicate with each other in several ways.
The three major reasons for using inter-servlet communication are:

Direct servlet manipulation : allows gaining access to the other currently loaded servlets and performing certain tasks (through the ServletContext object)
Servlet reuse : allows the servlet to reuse the public methods of another servlet
Servlet collaboration : requires to communicate with each other by sharing specific information (through method invocation). Basically servlets collaborate with each another for exchanging data and sharing control.

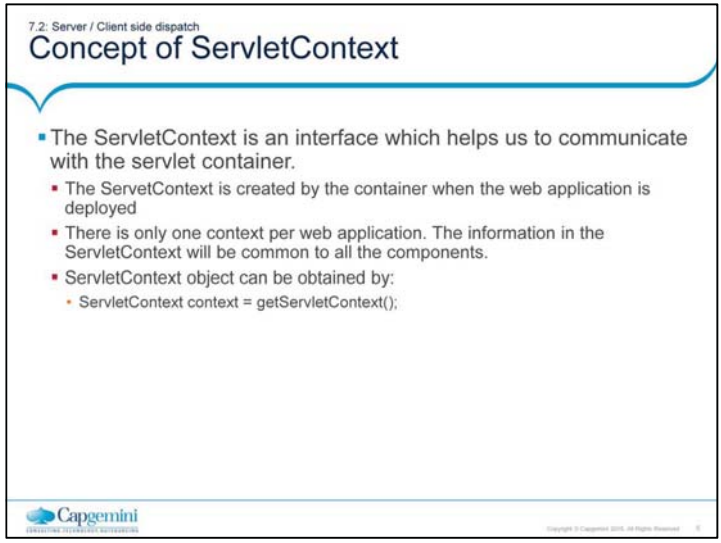
In this lesson, we shall be focusing on servlet collaboration



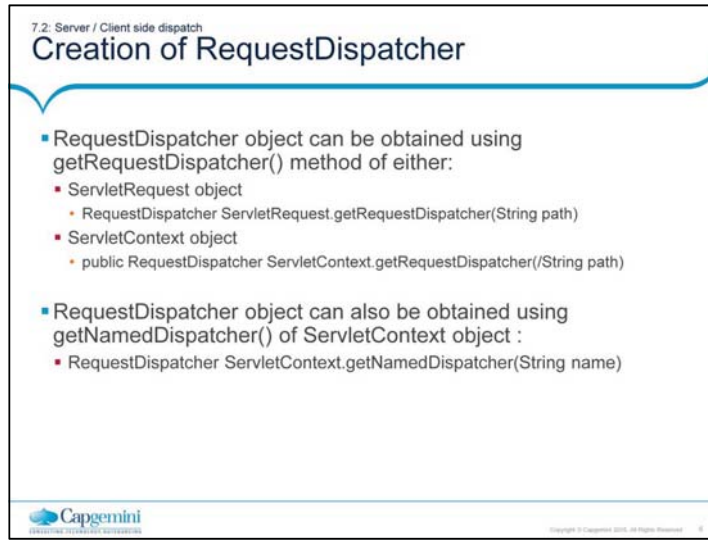
Introducing Inter-Servlet Communication:
Invoking other Web Resource:
RequestDispatcher is an interface, implementation of which defines an object that receives requests from the client and sends them to any resource (such as a servlet, HTML file, or JSP file) on the server. The servlet container creates the RequestDispatcher object which is used as a wrapper around a server resource located at a particular path or given by a particular name.
This interface is intended to wrap servlets, but a servlet container can create RequestDispatcher objects to wrap any type of resource.

RequestDispatcher object can be obtained from either a Request Object or the ServletContext object as seen in the method signatures above.

Let us first see the concept of dispatching with ServletContext object



RequestDispatcher Interface:
ServletContext:
There is only one ServletContext for the entire web application and the components of the web application can share it. The information in the ServletContext will be common to all the components.
Remember that each servlet will have its own ServletConfig. The ServetContext is created by the container when the web application is deployed and after that only the context is available to each servlet in the web application

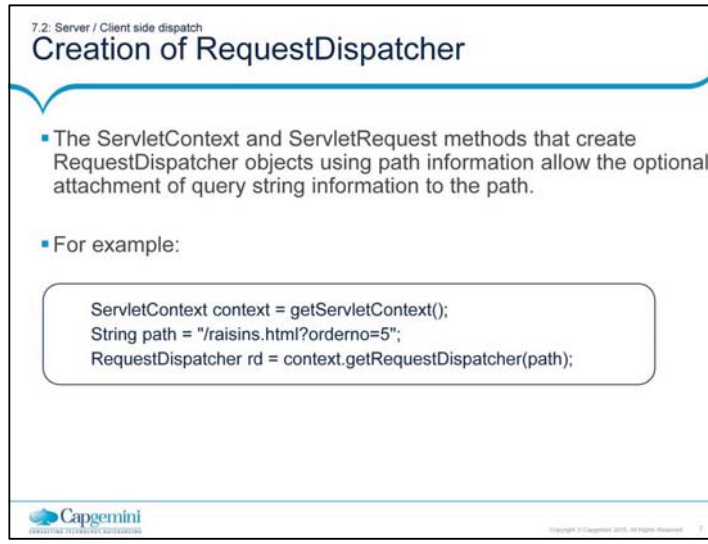


RequestDispatcher Interface:
Creation of RequestDispatcher:
The ServletRequest.getRequestDispatcher() can take a relative path while ServletContext.getRequestDispatcher() cannot (It can only take relative to the current context's root).
For example, with ServletRequest below statements are valid:
request.getRequestDispatcher("./html/first.html") - evaluated relative to the path of the request
request.getRequestDispatcher("/html/first.html") - evaluated relative to the root

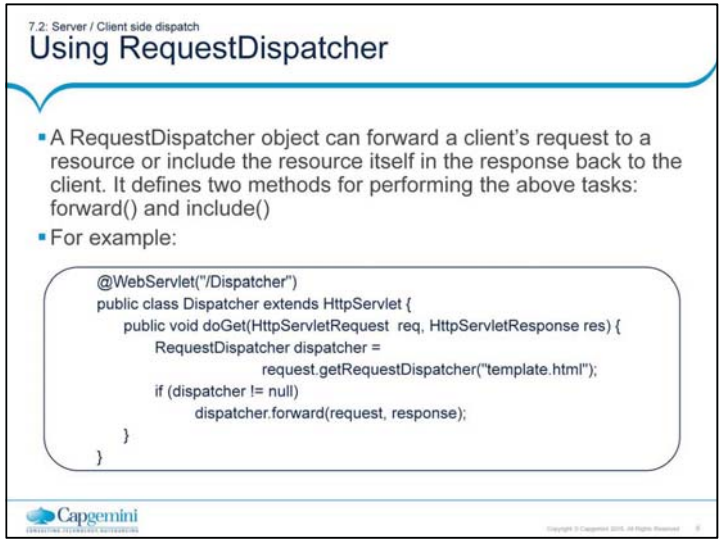
With ServletContext only :
context.getRequestDispatcher("/html/first.html") is valid but not context.getRequestDispatcher("./html/first.html") i.e. it can not evaluate a path other than context root.

The ServletContext.getNamedDispatcher() method takes a String argument indicating the name of a servlet known to the ServletContext. If a servlet is found, then it is wrapped with a RequestDispatcher object and the object is returned. If no servlet is associated with the given name, then the method returns null. (A servlet instance can determine its name using ServletConfig.getServletName()).

RequestDispatcher allows for direct communication between web resources.



RequestDispatcher Interface:
Creation of RequestDispatcher:
Parameters specified in the query string used to create the RequestDispatcher take precedence over other parameters of the same name passed to the included servlet.
The parameters associated with a RequestDispatcher are scoped to apply only for the duration of the include or forward call.


7.2: Server / Client side dispatch
Using RequestDispatcher
A RequestDispatcher object can forward a client's request to a resource or include the resource itself in the response back to the client. It defines two methods for performing the above tasks: forward() and include()
For example:
@WebServlet("/Dispatcher")
public class Dispatcher extends HttpServlet {
public void doGet(HttpServletRequest req, HttpServletResponse res) {
RequestDispatcher dispatcher =
request.getRequestDispatcher("template.html");
if (dispatcher != null)
dispatcher.forward(request, response);
}
}
Capgemini

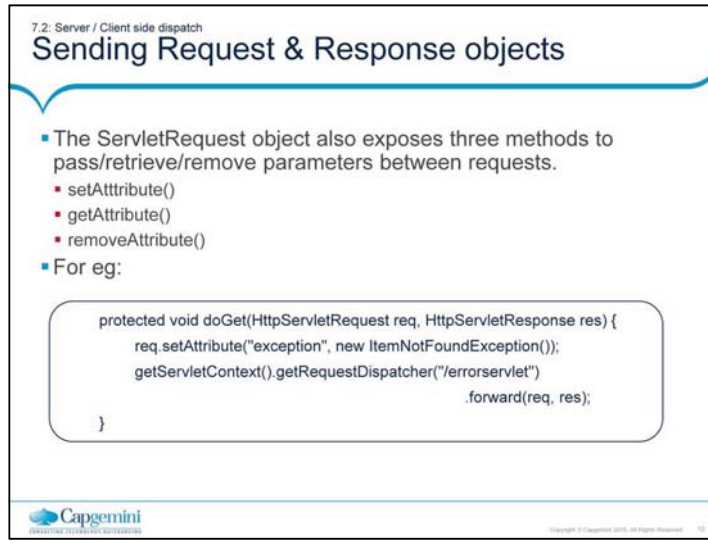
RequestDispatcher Interface:
Using RequestDispatcher:
A RequestDispatcher object can forward a client's request to a resource or include the resource itself in the response back to the client. A resource can be another servlet, HTML file or a JSP file. Methods are:
public void forward(ServletRequest req, ServletResponse resp)
public void include(ServletRequest req, ServletResponse resp)

The forward() method of the RequestDispatcher interface may be called by the calling servlet only when no output has been committed to the client. If non-committed output data exists in the response buffer, the content must be cleared before the target servlet's service method is called. Else, an IllegalStateException will be thrown.
In the case of forward() , once another web resource is being invoked , we cannot access the previous web resource.
include() includes the content of a resource in the response. The included servlet cannot change the response status code or set headers; any attempt to make a change is ignored.
The target servlet of the include() method has access to all aspects of the request object, but its use of the response object is more limited. It can only write information to the ServletOutputStream or Writer of the response object and commit a response by writing content past the end of the response buffer, or by explicitly calling the flushBuffer() method of the ServletResponse interface. It cannot set headers or call any method that affects the headers of the response.
In the case of include() , once another web resource is being invoked , we can come back with the response embedded to access the previous web resource.

Working with the RequestDispatcher is called as Server side dispatch. This is because the container is responsible to either forward request to other servlet or include the response of another servlet.

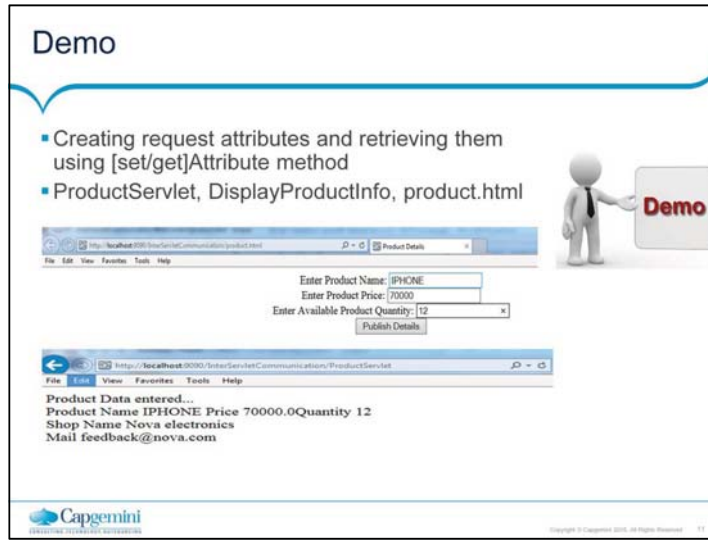
In both of these methods URL pattern does not change. Thus client is not aware which servlet is processing the request.

We introduce multiple servlets here as a single servlet may not be able to complete the request processing life cycle, or we may require it for load balancing

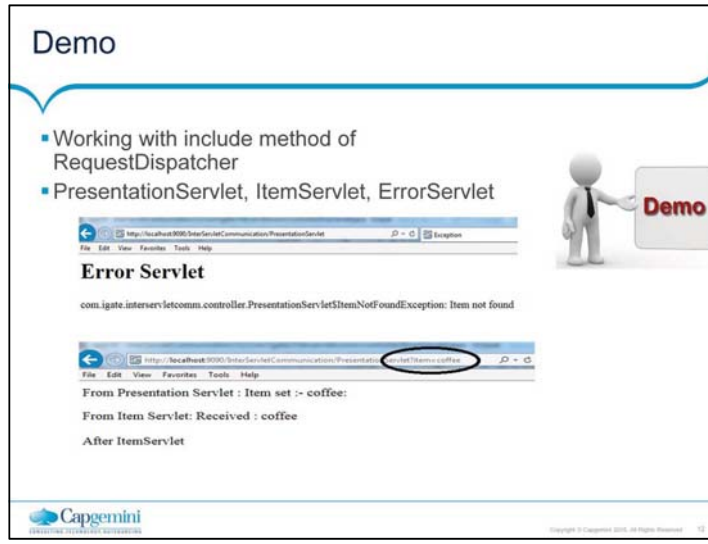


Communication with Server Resources:
Notice that the forward() and include() methods have the Request and Response objects as parameters. Thus one servlet can easily pass its Request and Response objects to the target servlet, and thus any request parameters too. However, if user wishes to pass more request parameters and also between just a request, use the setAttribute() method of the ServletRequest object. The ServletRequest object also exposes the getAttribute() and removeAttribute() methods, the likes of which has been discussed earlier.
See the code snippet in the slide above. The setAttribute() method sets an attribute named “exception” to a new instance of an exception – ItemNotFoundException. The forward() method while carrying request information will also carry this new attribute to the target servlet!

Thus to communicate among servlets and pass data between them we can make use of Request object and its API methods mentioned in the slide.



Note:
Refer to ProductServlet, DisplayProductInfo, product.html partial code below:
Execute the code by invoking the product.html first and fill in required fields. Data is fetched into ProductServlet and then forwarded to DisplayProductInfo servlet



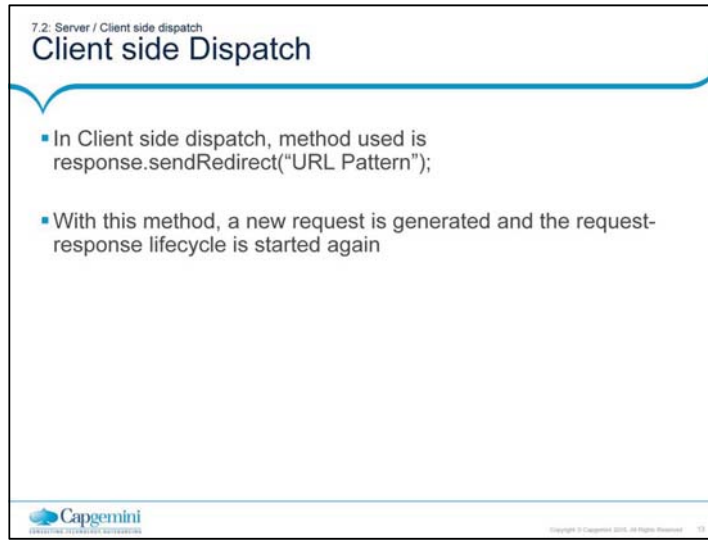
Note:
Refer to PresentationServlet, ItemServlet, ErrorServlet partial code below:
Execute the code by invoking the PresentationServlet with no item appended as query parameter.

Then execute PresentationServlet with item appended as query parameter

http://localhost:9090/InterServletCommunication/PresentationServlet

By appending item as query parameter
http://localhost:9090/InterServletCommunication/PresentationServlet?item=coffee

Here the response of ItemServlet is included in PresentationServlet



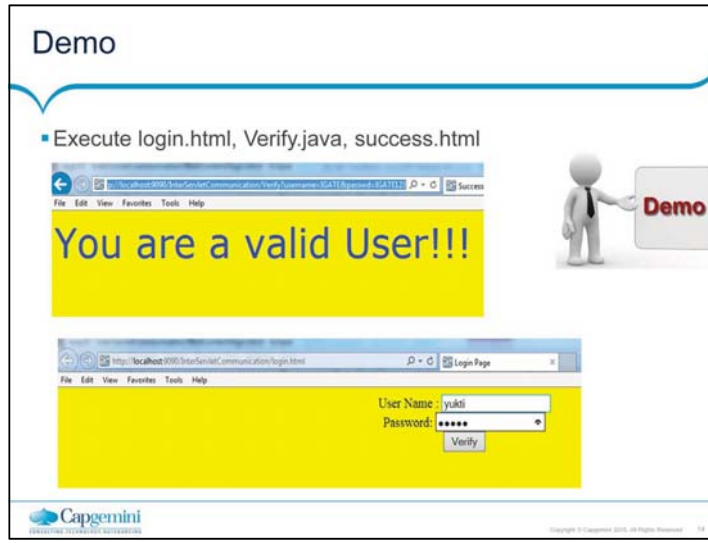
Client Side Dispatch

Make use of method, response.sendRedirect("URL Pattern");

In this, a servlet may not be able to process the request completely and sends it back to the browser with HTTP Response Code as 300 for Redirection.

This will result in a new URL to be created and start of another Request-Response lifecycle.

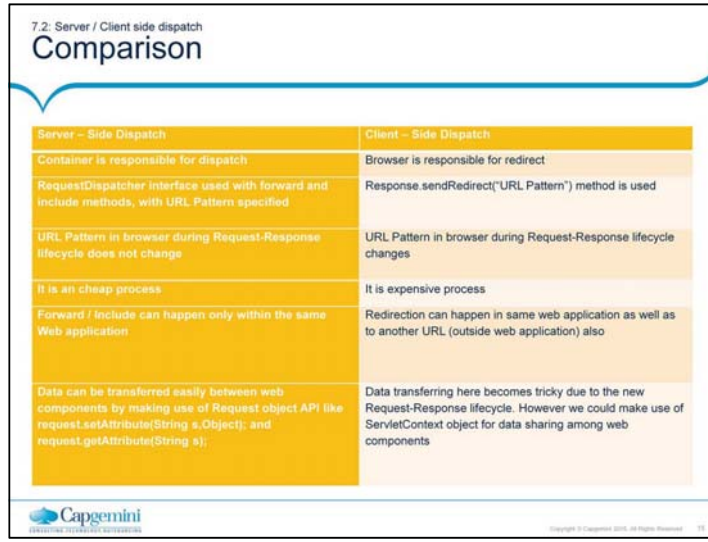
The URL pattern will change in browser as contrast to Server side dispatch

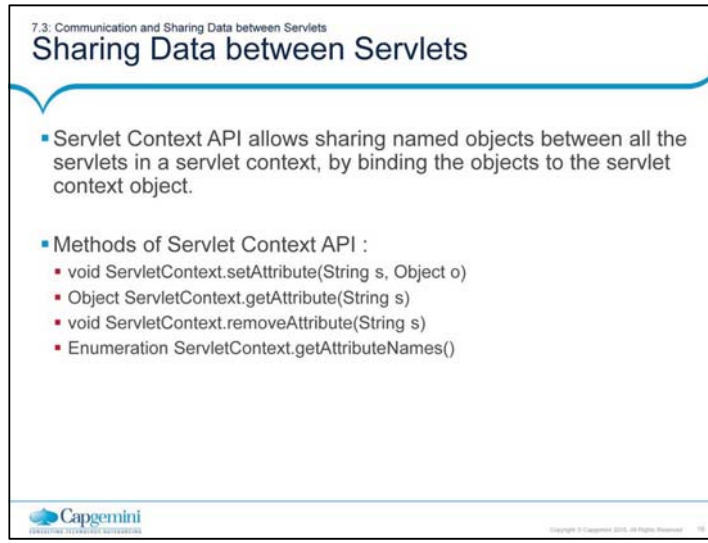


Run the URL Pattern: http://localhost:9090/InterServletCommunication/login.html

If user credentials are valid, display Success.html page

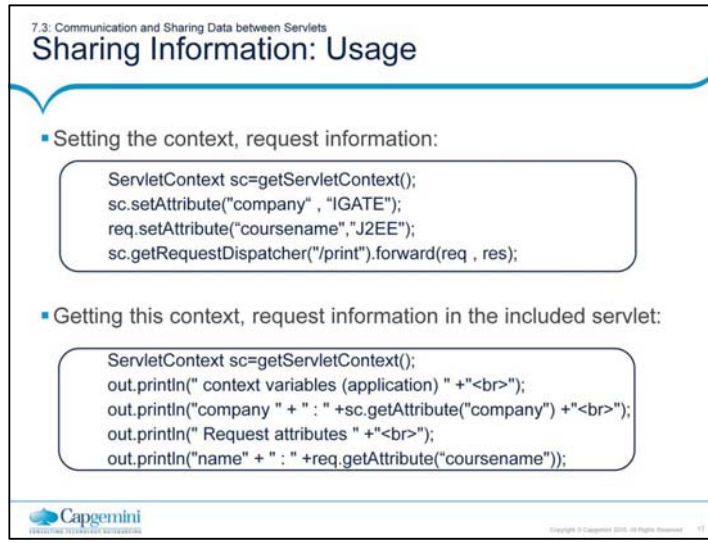
And if invalid user credentials display -> login.html back


7.2: Server / Client side dispatch
Comparison
Server – Side Dispatch
Client – Side Dispatch
Container is responsible for dispatch
Browser is responsible for redirect
RequestDispatcher interface used with forward and include methods, with URL Pattern specified
Response.sendRedirect("URL Pattern") method is used
URL Pattern in browser during Request-Response lifecycle does not change
URL Pattern in browser during Request-Response lifecycle changes
It is an cheap process
It is expensive process
Forward / Include can happen only within the same Web application
Redirection can happen in same web application as well as to another URL (outside web application) also
Data can be transferred easily between web components by making use of Request object API like request.setAttribute(String s,Object); and request.getAttribute(String s);
Data transferring here becomes tricky due to the new Request-Response lifecycle. However we could make use of ServletContext object for data sharing among web components
Capgemini

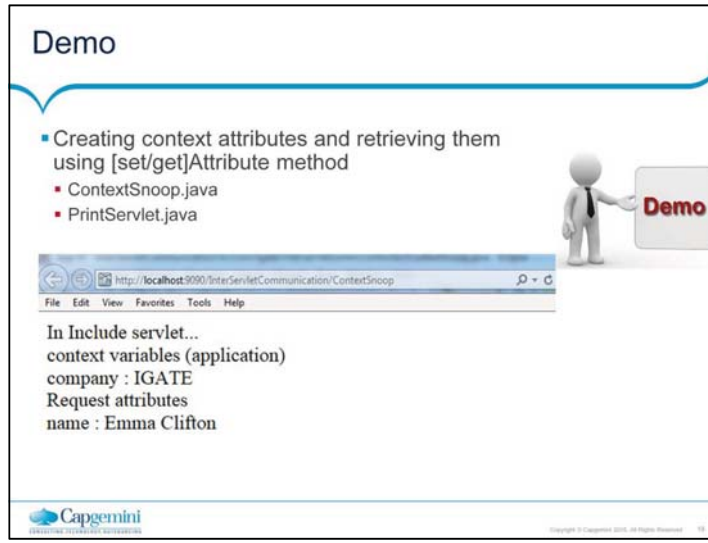


Sharing Data between Servlets:
One of the reasons for inter-servlet communication that we listed is to share data between the servlets. Servlet API offers way of sharing named objects between all the Servlets in a Servlet context (and also other contexts, as we'll see in next page) by binding the objects to the ServletContext object which is shared by several Servlets.
The ServletContext class has several methods for accessing the shared objects:
public void setAttribute(String name, Object object) adds a new object or replaces an old object by the specified name. The attribute name should follow the same naming convention as a package name.
Just like a custom ServletRequest attribute, an object which is stored as a ServletContext attribute should also be serializable to allow attributes to be shared by Servlets which are running in different JVMs on different machines in a load-balancing server environment.

public Object getAttribute(String name) returns the named object or null if the attribute does not exist.

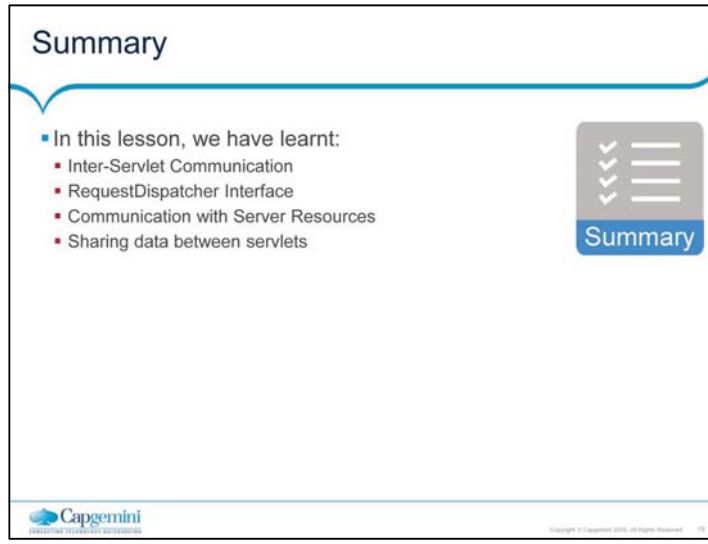


Sharing Data between Servlets:
Sharing Information: Usage
In addition to the user-defined attributes, there may also be predefined attributes that are specific to the Servlet engine and provide additional information about a Servlet(Context)'s environment.
public Enumeration getAttributeNames() returns an Enumeration of the names of all available attributes.
public void removeAttribute(String name) removes the attribute with the specified name if it exists.
The separation of Servlets into Servlet contexts depends on the Servlet engine. The ServletContext object of a Servlet with a known local URI can be retrieved with the method public ServletContext getContext(String uripath) of the Servlet's own ServletContext. This method returns null if there is no Servlet for the specified path or if this Servlet is not allowed to get the ServletContext for the specified path due to security restrictions.



Note:
Refer to the ContextSnoop Servlet and Include servlet; partial code of both given below:
ContextSnoop.java
PrintServlet.java :

Execute the ContextSnoop servlet as :
http://localhost:9090/InterServletCommunication/ContextSnoop
Notice how the ContextSnoop servlet first sets the request-level and ServletContext-level attributes and then forwards to print servlet!
For the curious, invoke the Print servlet directly; and see what happens



Add the notes here.

Review Question

- Question 1: Which of the following can be used to obtain RequestDispatcher object?
  - Option 1: ServletConfig.getRequestDispatcher()
  - Option 2: ServletRequest.getRequestDispatcher()
  - Option 3: ServletResponse.getRequestDispatcher()
  - Option 4: ServletContext.getRequestDispatcher()

Capgemini

Add the notes here.

Review Question

- Question 2: What is the difference between include() and forward() of RequestDispatcher ?
  - Option 1: Forward() transfers a request from a servlet to another resource on the server while include() embeds the content of a resource in the response
  - Option 2: Forward() embeds the content of a resource in the response while include() transfers a request from a servlet to another resource on the server

Capgemini

 21

Add the notes here.

Review Question

- Question 3: How is the data shared between servlets ?
  - Option 1: Using put/get value methods of Servlet Context
  - Option 2: Using set/get Attribute methods of ServletContext
  - Option 3: Using value bound/unbound methods
  - Option 4: Using set/get attribute methods of ServletResponse

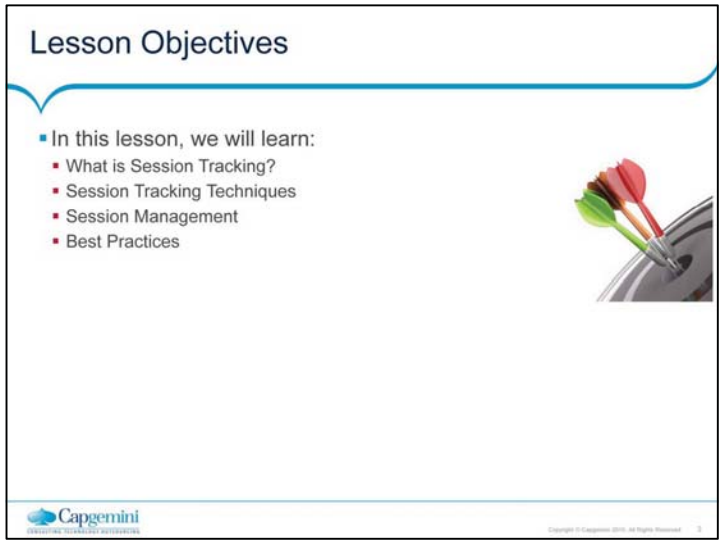
Capgemini

 22

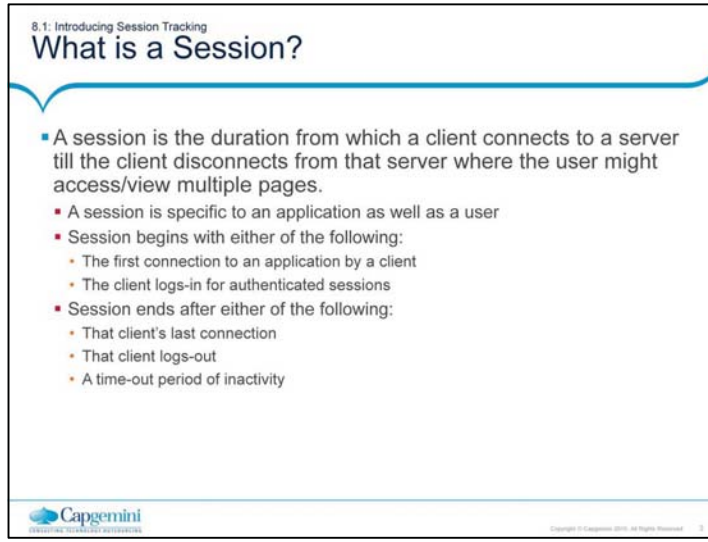
Add the notes here.

Java Servlets 3.0

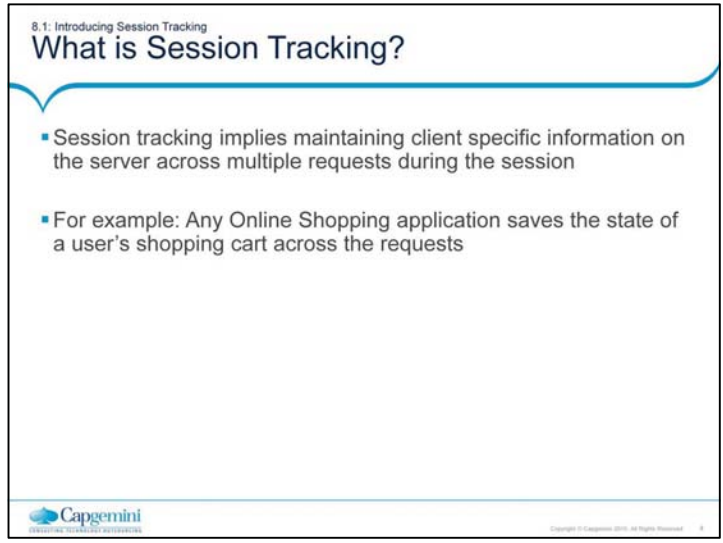
Lesson 8: Session Tracking



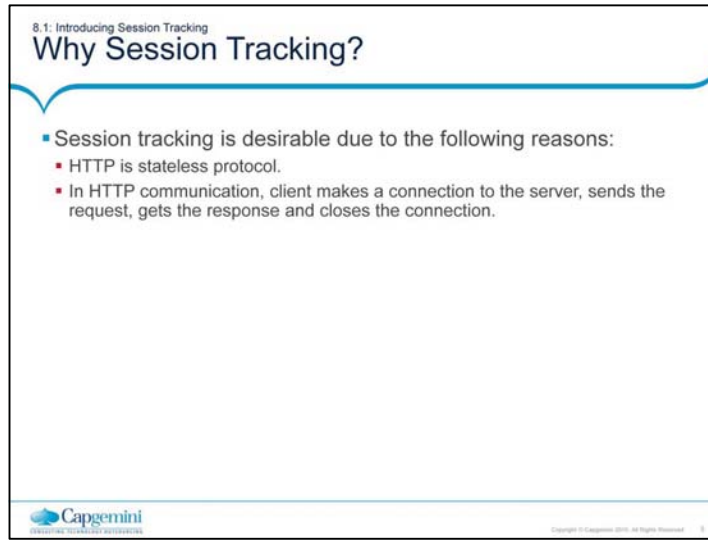
Lesson Objectives:
This lesson introduces Session Management. The lesson contents are:
Lesson 08: Session Management
- 8.1: Introducing Session Tracking
- 8.2: Ways of Session Tracking in JEE
- 8.3: Session Management
- 8.4: Best Practices



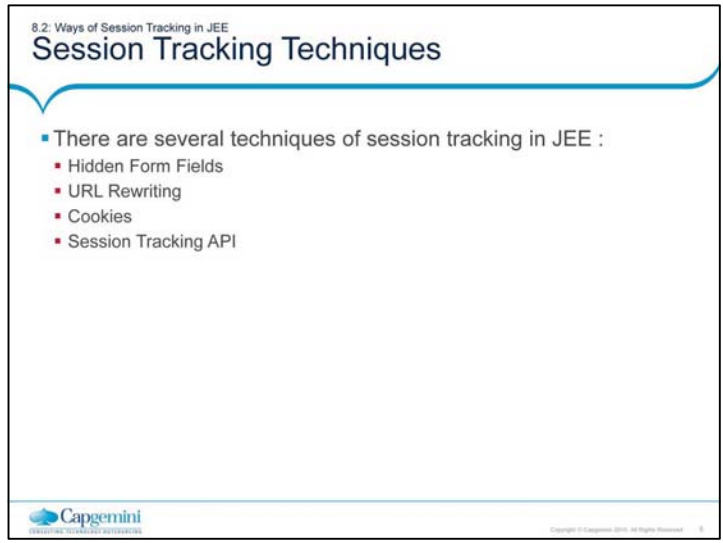
Introducing Session Tracking:
A session refers to all the connections that a single client might make to a server in the course of viewing any pages associated with a given application.
Sessions are specific to both the individual user and the application. Therefore every user of an application has a separate session. The user has access to a separate set of session variables.
Logically a session begins with the first connection to an application by a client and ends after that client's last connection.
Session can be an authenticated session also where a user has to log-in. In this case, sessions begin when the user logs in and ends when the user log-out.
Session also ends after a time-out period of inactivity.



Introducing Session Tracking:
What is Session Tracking?
In Session tracking, client first makes a request for any servlet or any page, the container receives the request.
The container then generates a unique identification, called Session ID, for that client and gives it back to the client along with the response. This ID gets stores on the client machine. Thereafter when the client sends a request again to the server, it also sends the session Id with the request. The container sees the Id and sends back the response. While seeing the ID the container recognizes that it is the same previous client making a request.
Thus container identifies the client.



Introducing Session Tracking:
Why Session Tracking?
HTTP is a stateless protocol, that is it cannot persist the information.
HTTP always treats each request as a new request. Each request made by a Web browser (client), for an image, an HTML page, or other Web object, is made via a new connection. In other words, in HTTP, the client makes a connection to the server, sends the request, gets the response, and closes the connection. Hence to maintain the state across pages we need some sort of session tracking mechanism.



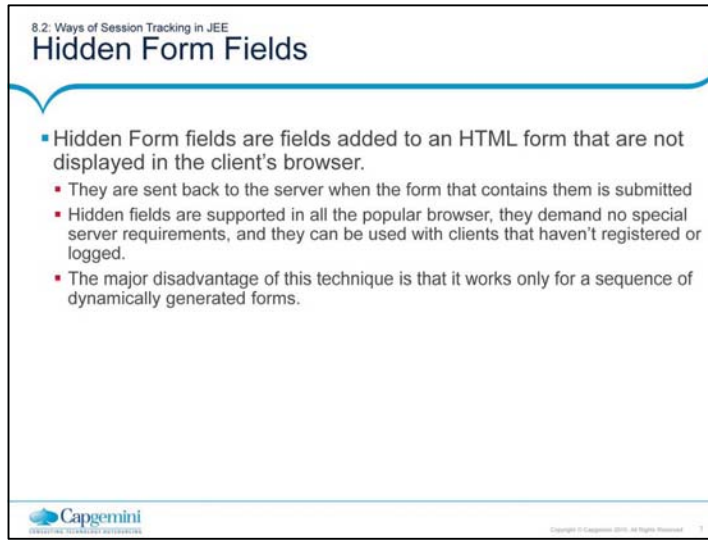
Ways of Session Tracking in JEE:
Session Tracking Techniques:
From the last slide we understand that since HTTP is stateless protocol it becomes developer's job to work around saving client specific information on server in a web application.
There are several ways in which session tracking can be done in JEE.

Hidden Form Fields: It is the simplest way of saving user-specific information where hidden parameters are encoded inside an HTML form, such as the username and type of transaction being made

URL Rewriting: URL rewriting involves placing a session id in the URL.

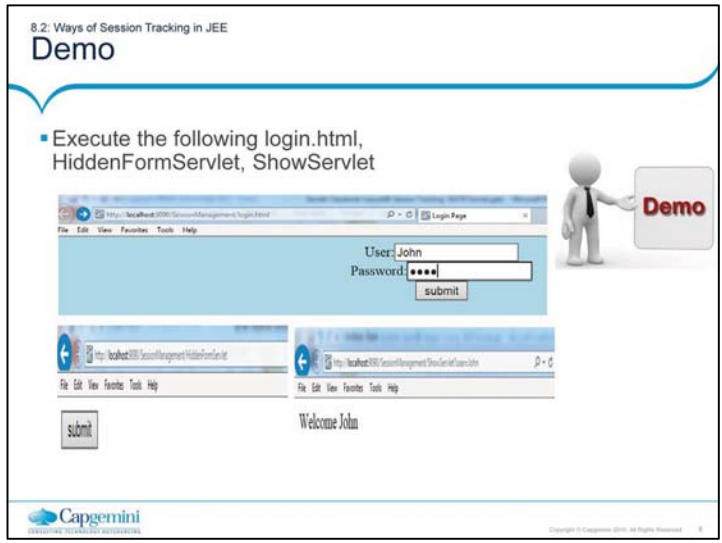
Cookies: It is a small text file storing client information, cookies are created by the server and saved on client's machine.

Session Tracking API: Servlets API provides Session Tracking API



Ways of Session Tracking in JEE:
Hidden Form Fields:
In this technique the fields are added to an HTML form which are not displayed in the client's request. The hidden form fields are sent back to the server when the form is submitted. In hidden form fields, the HTML entry will be as shown below:

This means that when form is submitted ,the specified name and value will be get included in get or post method.



Execute the following servlets:
http://localhost:9090/SessionManagement/login.html

This would lead to servlet:

http://localhost:9090/SessionManagement/HiddenFormServlet

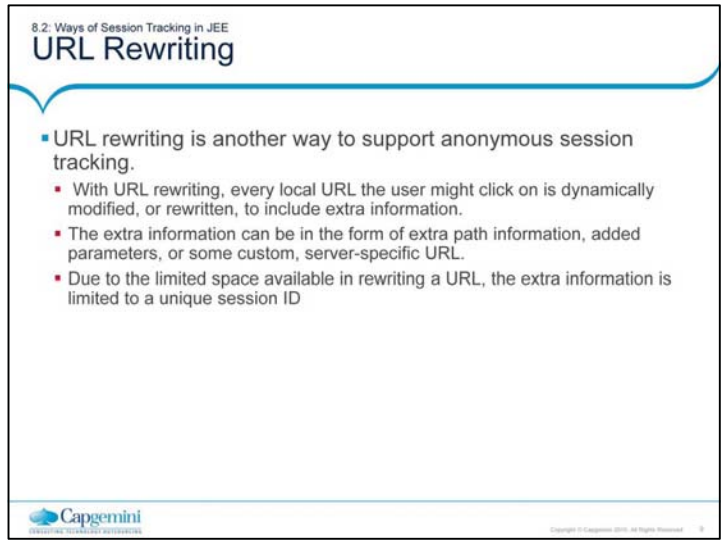
In HiddenFormServlet, username is added as Hidden Field in html form as shown below in partial listing.

```
out.println("<form action='ShowServlet'>");
out.println("<input type='hidden' name='user' value='" + user + "'>");
out.println("<input type='submit' value='submit' >");
```

Then username is retrieved in ShowServlet.

http://localhost:9090/SessionManagement/ShowServlet?user=John.

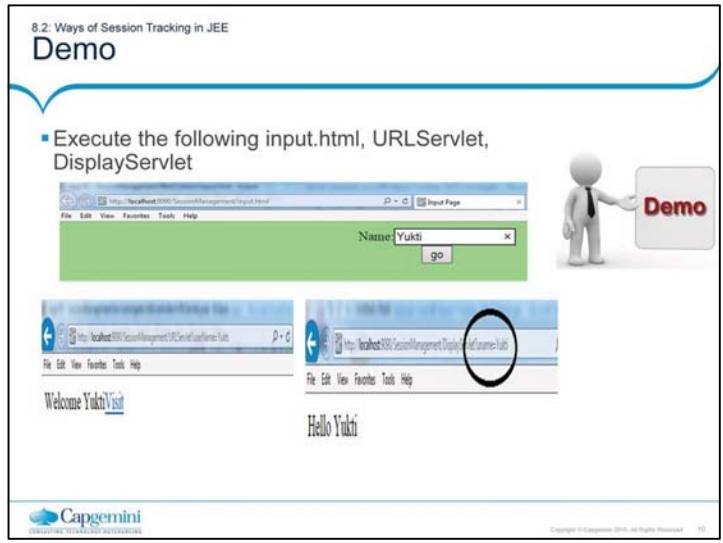
Query parameters get appended due to the GET method.



Ways of Session Tracking in JEE:
URL Rewriting:
The explanation for session Id would be covered in the HttpSession section.
URL rewriting is another way to support anonymous session tracking. With URL rewriting, every local URL the user might click on is dynamically modified, or rewritten, to include extra information. The extra information can be in the form of extra path information, added parameters, or some custom, server-specific URL. Due to the limited space available in rewriting a URL, the extra information is limited to a unique session ID. For example, the following URLs have been rewritten to pass session id 123:

Original URL :
http://server:port/servlet/Rewritten
Extra path information :
http://server:port/servlet/Rewritten/123
Added parameter :
http://server:port/servlet/Rewritten?sessionid=123
Added parameter change :
http://server:port/servlet/Rewritten;$sessionid$123

Each rewriting technique has its advantages and disadvantages. Using extra path information works on all servers, and it works as a target for forms that use both the GET and POST methods. It doesn't work well if a servlet has to use the extra path information as true path information.
An added parameter works on all servers too, but fails as a target for forms those uses the POST method, and it causes parameter-naming collisions.
Using a custom, server-specific change works under all conditions for servers that support the change. Unfortunately, it doesn't work at all servers that don't support the change.



Execute the following servlets:
http://localhost:9090/SessionManagement/input.html

This would lead to servlet:

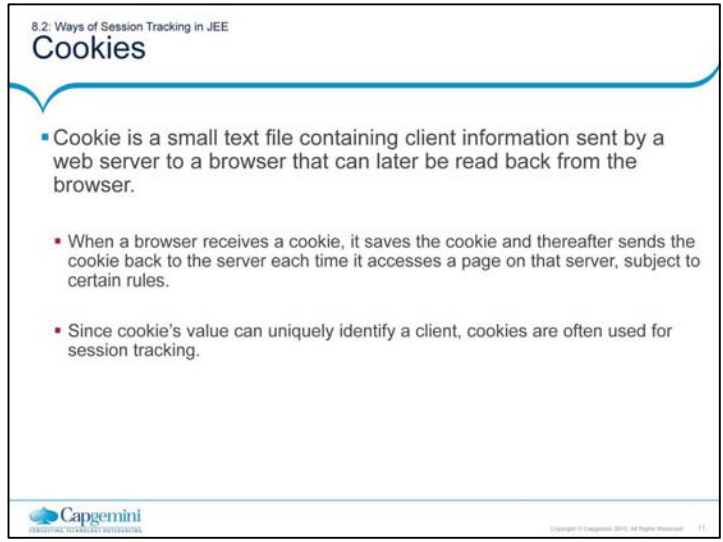
http://localhost:9090/SessionManagement/URLServlet

In URLServlet, username is appended as query string as shown below in partial listing.

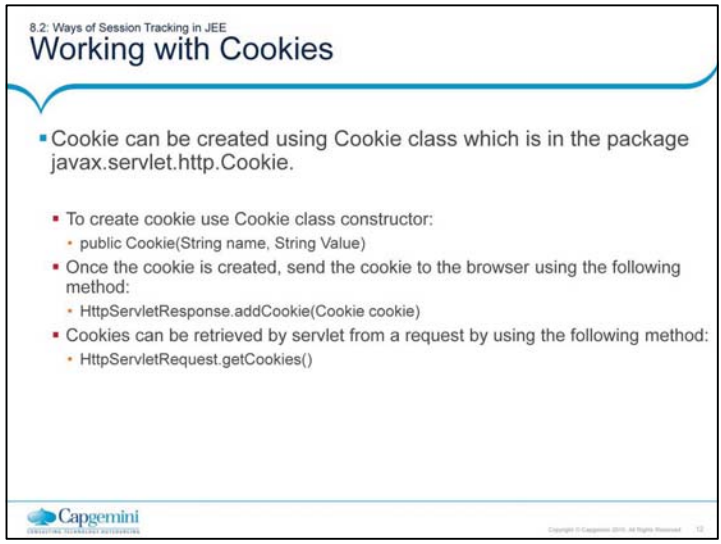
out.print("visit");

Then username is retrieved in DisplayServlet.

http://localhost:9090/SessionManagement/DisplayServlet?user=Yukti.



Ways of Session Tracking in JEE:
Cookies:
Cookie is a bit of information stored in a small text file sent by a web server to a browser that can later be read back from the browser. When a browser receives a cookie, it saves the cookie and there-after sends the cookie back to the server each time it accesses a page on that server, subject to certain rules. Since cookie's value can uniquely identify a client, cookies are often used for session tracking.
The browser is expected to support 20 cookies for each Web server, 300 cookies total, and may limit cookie size to 4 KB each.



Ways of Session Tracking in JEE:
Working with Cookies
Create a cookie with the Cookie( ) constructor:

public Cookie(String name, String value)

This creates a new cookie with an initial name and value. A cookie has a name, a single value, and optional attributes such as a comment, path and domain qualifiers, a maximum age, and a version number.
The name of the cookie must be an HTTP/1.1 token. Tokens are strings that contain none of the special characters listed in. (Alphanumeric strings qualify as tokens.)
The value of the cookie can be any string, though null values are not guaranteed to work the same way on all browsers. In addition, if a cookie is sent that complies with Netscape's original cookie specification, do not use whitespace or any of these characters:

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

A servlet can send a cookie to client by passing a Cookie object to the addCookie( ) method of HttpServletResponse:

public void HttpServletResponse.addCookie(Cookie cookie)

The method adds the specified cookie to the response. Additional cookies can be added with subsequent calls to addCookie( ).

Ways of Session Tracking in JEE:

Working with Cookies:

- Since cookies are sent using HTTP headers, they should be added to response before any content is sent.
- The code to set a cookie looks like this:

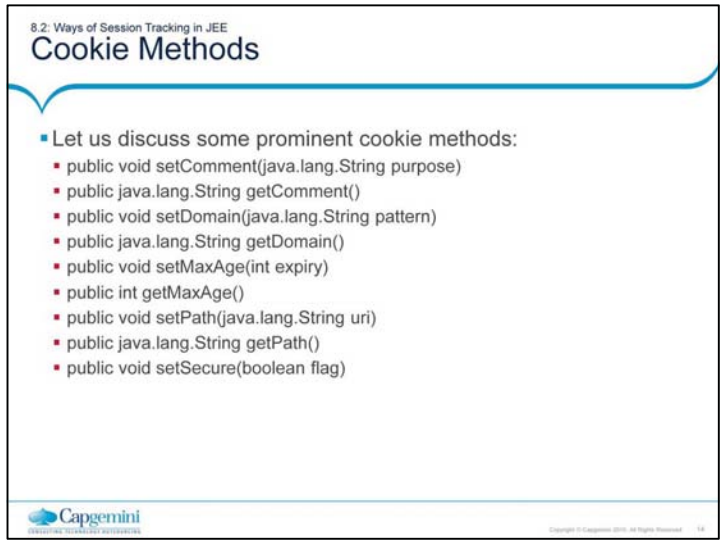
```
Cookie cookie=new Cookie("ID" , "123" );
res.addCookie(cookie);
```

- A servlet retrieves cookies by calling the getCookies( ) method of HttpServletRequest.

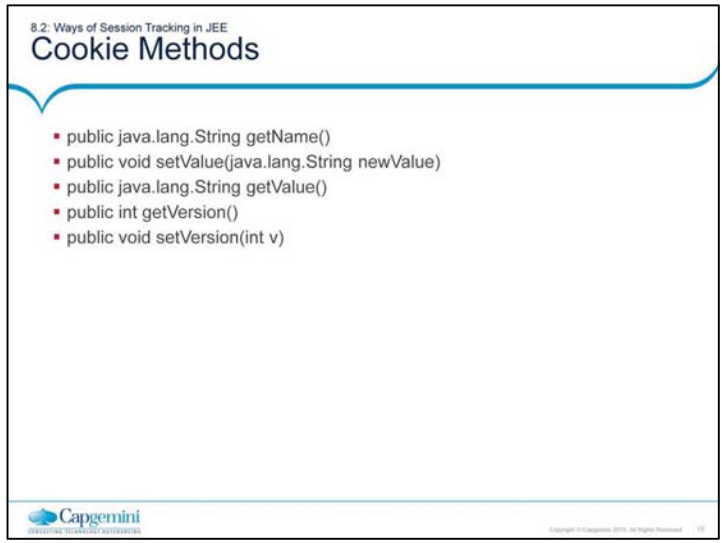
```
public Cookies[ ] HttpServletRequest.getCookies( )
```

- This method returns an array of Cookie objects that contains all the cookies sent by the browser as part of the request or null if no cookies were sent. Several cookies might have the same name but different path attributes. The code to fetch the cookies looks like this:

```
Cookie[] cookies = req.getCookies();
  if (cookies != null) {
    for (int i = 0; i < cookies.length; i++) {
      if (cookies[i].getName().equals("sessionid")) {
            String name =cookies[i].getName();
            String value =cookies[i].getValue();
      }
    }
}
```



Ways of Session Tracking in JEE:
Cookie Methods:
public void setComment(java.lang.String purpose) :
It specifies a comment that describes a cookie's purpose.
public java.lang.String getComment() :
It returns the comment describing the purpose of this cookie, or null if the cookie has no comment.
public void setDomain(java.lang.String pattern) :
It specifies the domain within which this cookie should be presented.
public java.lang.String getDomain() :
It returns the domain name set for this cookie.
public void setMaxAge(int expiry) :
It sets the maximum age of the cookie in seconds. A positive value indicates that the cookie will expire after that many seconds have passed. Note that the value is the maximum age when the cookie will expire, not the cookie's current age. A negative value means that the cookie is not stored persistently and will be deleted when the Web browser exits. A zero value causes the cookie to be deleted.
public int getMaxAge() :
It returns the maximum age of the cookie, specified in seconds. If getMaxAge returns a negative value, the cookie was not stored persistently. This method does not return a zero value, because if a cookie's age was set to zero with setMaxAge, the cookie was deleted.



Ways of Session Tracking in JEE:
Cookie Methods:
public void setPath(java.lang.String uri)

It specifies a path for the cookie, which is the set of URIs to which the client should return the cookie. The cookie is visible to all the pages in the directory being specified, and all the pages in that directory's subdirectories. A cookie's path must include the servlet that set the cookie, for example, servlet/dir1, which makes the cookie visible to all directories on the server under dir1.

public java.lang.String getPath()

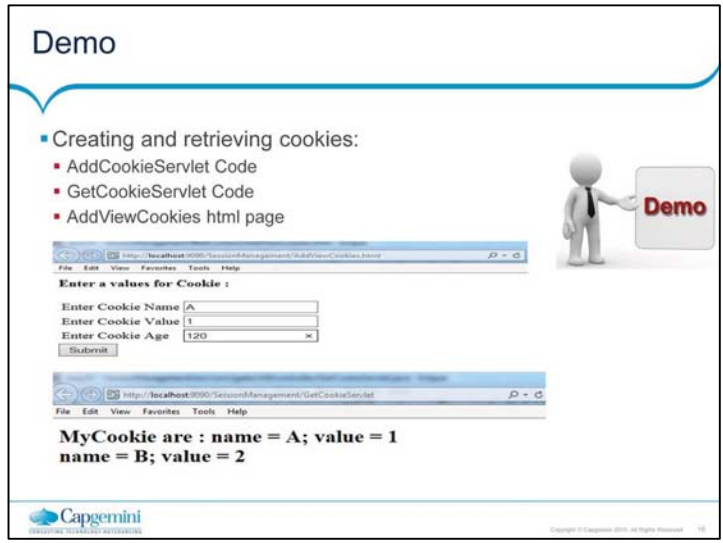
It returns the paths (that is, URIs) on the server to which the browser returns this cookie. The cookie is visible to all subdirectories within the specified path on the server.

public void setSecure(boolean flag)

It indicates to the browser whether the cookie should only be sent using a secure protocol, such as HTTPS or SSL. This method should only be used when the cookie's originating server used a secure protocol to set the cookie's value. The default value is false.

public boolean getSecure()

It returns true if the browser is sending cookies only over a secure protocol, or false if the browser can use a standard protocol.

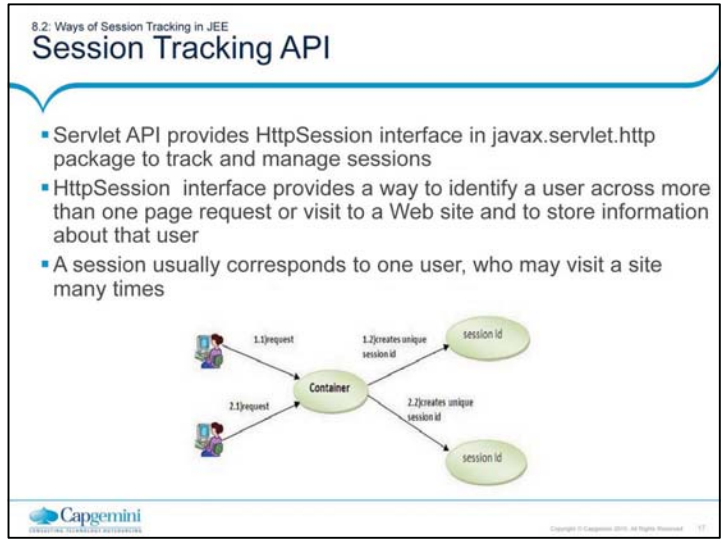


Note:
Refer to the com.igate.ch8.controller.AddCookieServlet.java code, partial listing of doGet() is given here:

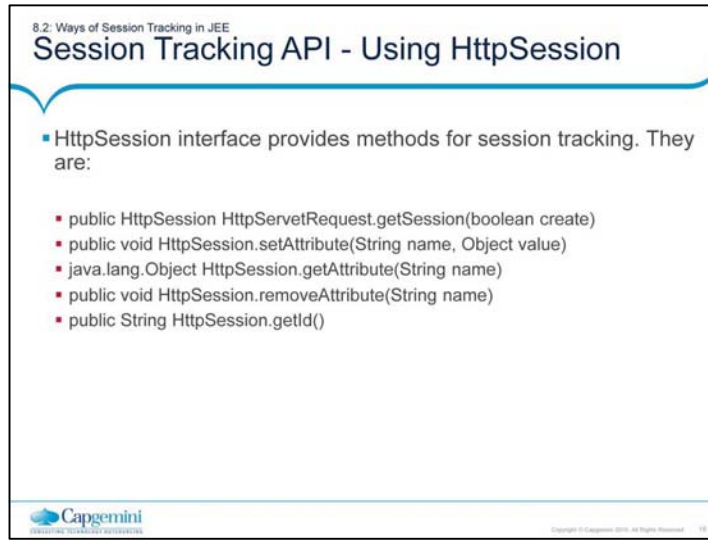
```
Cookie cookie =  new Cookie(name,data);
            if ((age==null) || (age.equals(""))) {
                  res.addCookie(cookie);
                  out.println("Set Cookie --- Name= " + name + " Value = " + data);}
            else {  try { int intage=Integer.parseInt(age);
 cookie.setMaxAge(intage);
 res.addCookie(cookie);
 out.println("Set Cookie --- Name= " + name + " Value = " +  data + " age= "+age );
            }  catch(Exception e) {  out.println("Exception " +e; }
             }
```

Partial code for com.igate.ch8.controller.GetCookieServlet is here :

```
        Cookie[] cookies =  req.getCookies();
        out.println("<B>MyCookie are  : ");
        for(int i=0;i < cookies.length;i++)  {
              String name = cookies[i].getName();
              String value = cookies[i].getValue();
              out.println("name =  " + name + "; value = " + value +"<BR>"); }
```



Ways of Session Tracking in JEE:
Session Tracking API:
One of the ways to track session is by using Session Tracking interfaces provided by Servlet API.
The API provides HttpSession interface to track and manage sessions.
The servlet container uses this interface to create a session between an HTTP client and an HTTP server creating an unique identifier called Session ID. The server can maintain a session in many ways such as using cookies or rewriting URLs. This interface allows servlets to view and manipulate information about a session, such as the session identifier, creation time, and last accessed time.
The explanation about Session Id would be explained in the next subsequent slide.



Ways of Session Tracking in JEE:
Session Tracking API - Using HttpSession:

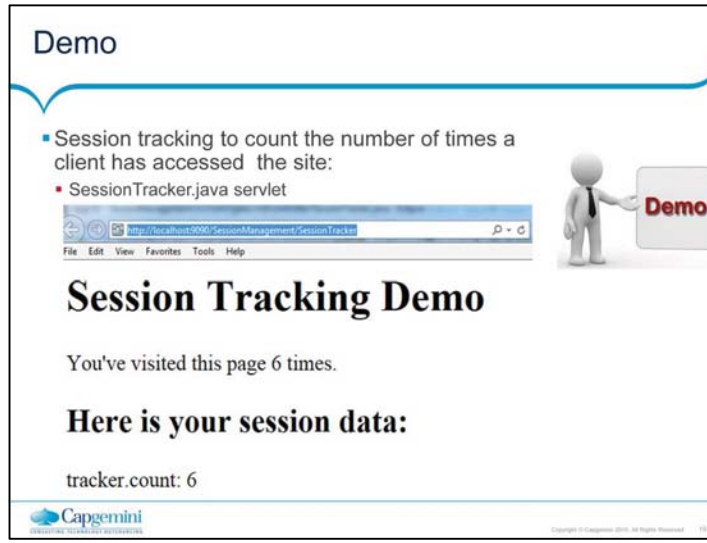
public HttpSession HttpServetRequest.getSession(boolean create) : retrieve the current HttpSession object. This method returns the current HttpSession object associated with the request or, if necessary, create a new session for the request. Use true to create a new session if none exists.

If create is false, and the request has no valid session then this method returns null else returns the existing session. To make sure the session is properly maintained, need to call this method at least once before writing any output to the response.

public void HttpSession.setAttribute(String name, Object value) :add information in form of an object to session object. This method binds an object to this session, using the name specified. If an object of the same name is already bound to the session, the object is replaced. The putValue() method is sometimes used instead of setAttribute() however it is deprecated now.

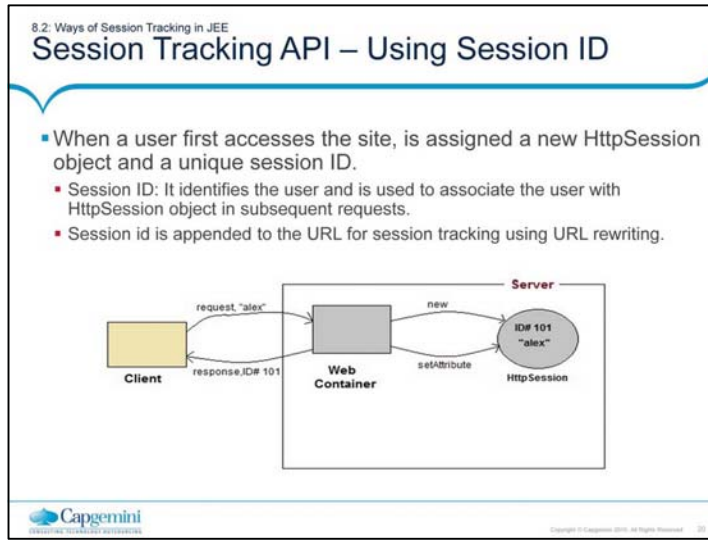
public Object HttpSession.getAttribute(String name) : retrieve information object stored by setAttribute() method in form of an object from the session. Returns null if no object of that name exists.

public java.lang.String getId() : This method returns a string containing the unique identifier assigned to this session. The identifier is assigned by the servlet engine and is implementation dependent.

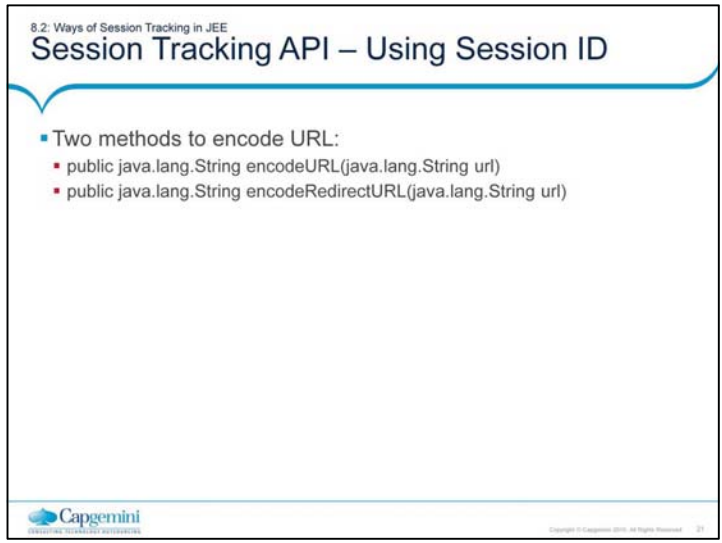


Note:
Refer the com.igate.ch8.controller.SessionTracker.java code; partial listing of doGet() method is given:

```
// Get the current session object, create one if necessary
    HttpSession session = req.getSession(true);
     // Increment the hit count for this page.  The value is saved
    // in this client's session under the name "tracker.count".
Integer count = (Integer)session.getAttribute("tracker.count");
    if (count == null) {
      count = new Integer(1); }
    else {
      count = new Integer(count.intValue() + 1); }
     session.setAttribute("tracker.count", count);
     out.println("<HTML><HEAD><TITLE>SessionTracker</TITLE></HEAD>");
    out.println("<BODY><H1>Session Tracking Demo</H1>");
    // Display the hit count for this page
    out.println("You've visited this page " + count +
      ((count.intValue() == 1) ? " time." : " times."));
     out.println("<P>");
     out.println("<H2>Here is your session data:</H2>");
    String[] names = session.getAttributeNames();
    for (int i = 0; i < names.length; i++) {
      out.println(names[i] + ": " + session.getAttribute(names[i]) + "<BR>"); }
    out.println("</BODY></HTML>");
```

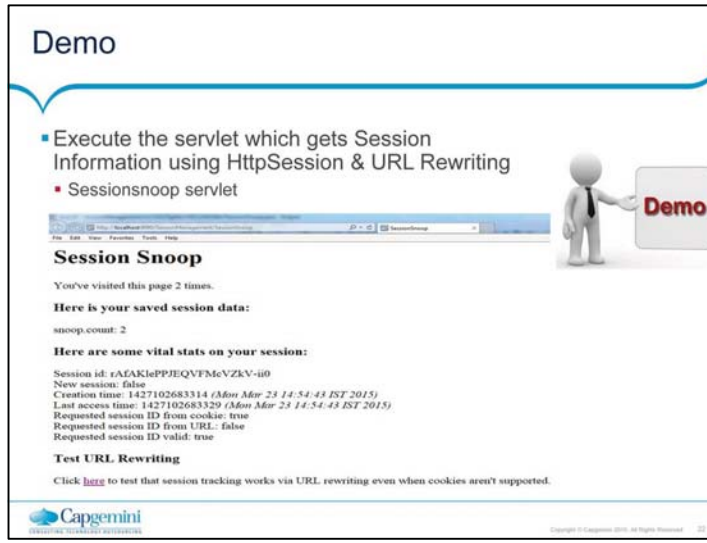


Ways of Session Tracking in JEE:
Session Tracking API – Using Session ID:
When a user first accesses the site, that user is assigned a new HttpSession object and a unique session ID. The Session ID identifies the user and is used to the user with HttpSession object in subsequent requests. Behind the scenes, the session ID is usually saved on the client in a cookie or sent as part of a rewritten URL.
A server can discover session's ID with the getID() method seen earlier.
It returns a string containing the unique identifier assigned to this session. The identifier is assigned by the servlet engine and is implementation dependent. This method throws an IllegalStateException if session is invalid.



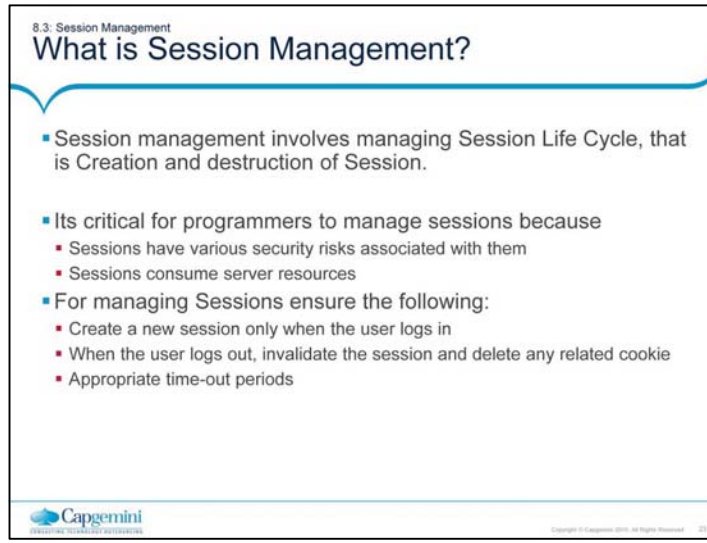
Session Tracking API:

- Session Id is stored as part of URL to support session tracking via URL rewriting. Servlet rewrites every local URL before sending it to the client. The Servlet API's HttpServletRequest interface provides two methods to perform this encoding:
  public java.lang.String **encodeURL(java.lang.String url)**
- It encodes the specified URL by including the session ID in it, or, if encoding is not needed, returns the URL unchanged. The implementation of this method should include the logic to determine whether the session ID needs to be encoded in the URL. For example, if the browser supports cookies, or session tracking is turned off, URL encoding is unnecessary.
- All URLs emitted by a Servlet should be run through this method. Otherwise, URL rewriting cannot be used with browsers, which do not support cookies.



Note:
Refer to com.igate.ch8.controller.SessionSnoop.java servlet code; partial code in doGet() method is here :

```
out.println("<H3>Here are some vital stats on your session:</H3>");
out.println("Session id: " + session.getId() + "<BR>");
out.println("New session: " + session.isNew() + "<BR>");
out.println("Creation time: " + session.getCreationTime());
out.println("<I>(" + new Date(session.getCreationTime()) + ")</I><BR>");
out.println("Last access time: " + session.getLastAccessedTime());
out.println("<I>(" + new Date(session.getLastAccessedTime()) +
")</I><BR>");
out.println("Requested session ID from cookie: "
+req.isRequestedSessionIdFromCookie() + "<BR>");
out.println("Requested session ID from URL: " +
req.isRequestedSessionIdFromURL() + "<BR>");
out.println("Requested session ID valid: " +
req.isRequestedSessionIdValid() + "<BR>");
out.println("<H3>Test URL Rewriting</H3>");
out.println("Click <A HREF=\"" +
      res.encodeURL(req.getRequestURI()) + "\">here</A>");
out.println("to test that session tracking works via URL");
out.println("rewriting even when cookies aren't supported.");
out.println("</BODY></HTML>");
```

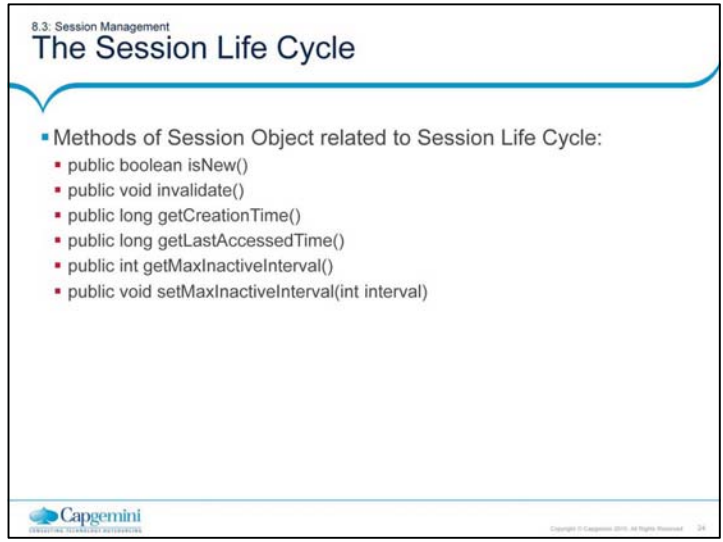


Session Management:
What is Session Management?
As we have seen in the first slide of this lesson, logically a session begins with the first connection to an application by a client and ends after that client's last connection. Session can be an authenticated session also, where a user has to log-in. In this case sessions begins when the user logs in and ends when the user logs-out. Session also ends after a time-out period of inactivity. We as programmers have to manage this life cycle of session, i.e. Creation and destruction of session.
Managing the life cycle of session is very important due to following reasons:
The Servlet API is rather liberal in creating sessions. Various tools have default behaviours which can implicitly create sessions in the background. It's very easy for an application to "accidentally" create a session, even when one was not explicitly requested.
Sessions have various security risks associated with them.
Sessions consume server resources, and should likely be avoided if possible.
Ensure the following for session management :

Create a new session only when the user logs in
Invalidate the session and delete any related cookie, session data
Set appropriate time-out period in web.xml like follows :

```
<session-config>
        <session-timeout>30</session-timeout>
</session-config>
```

Here time-out is set to 30 minutes
In Servlet 3.0 this is a global way of setting the Session time out period

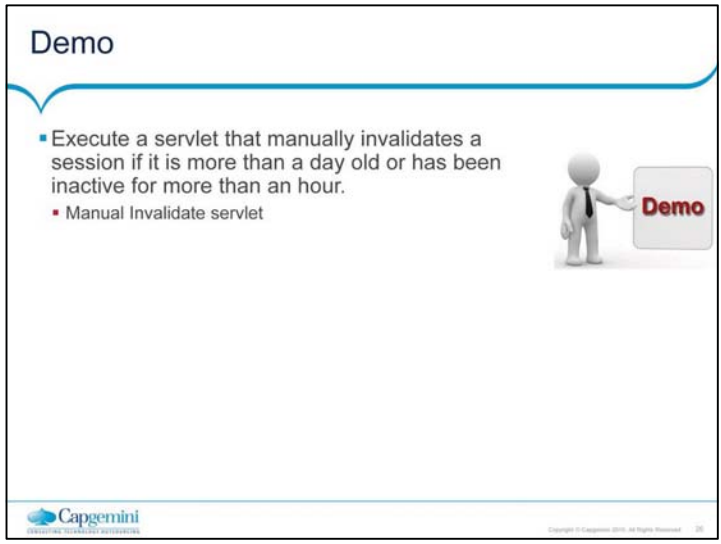


Session Management:
The Session Life Cycle:
Since there is no way for an HTTP client to signal that it no longer needs a session, each session has an associated timeout so that its resources can be reclaimed. The timeout period can be accessed by using a session's [get|set]MaxInactiveInterval methods.
Methods of Session Object related to Session Life Cycle :
public boolean isNew()
It returns true if the Web server has created a session but the client has not yet joined. For example, if the server used only cookie-based sessions, and the client had disabled the use of cookies, then a session would be new.
public void invalidate()
It invalidates this session and unbinds any objects bound to it.
public long getCreationTime()
It returns the time in long integer specifying when this session was created, measured in milliseconds since midnight January 1, 1970 GMT.

Session Management:

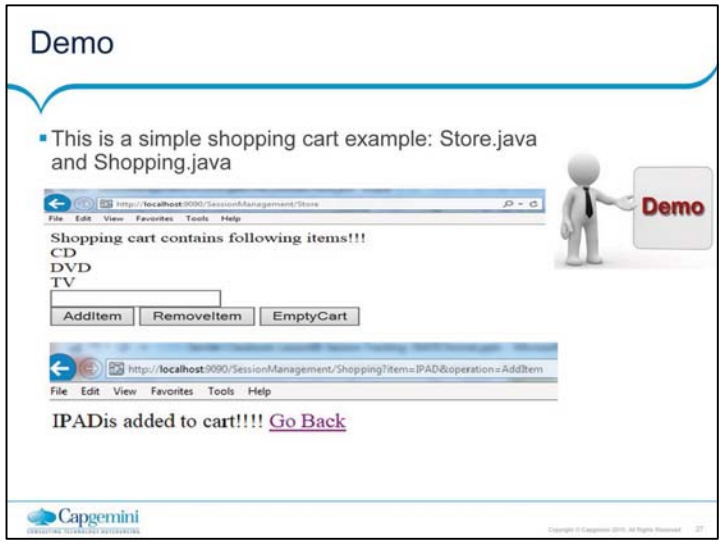
The Session Life Cycle:

- **public long getLastAccessedTime()**
  It returns the last time the client sent a request associated with this session, as the number of milliseconds since midnight January 1, 1970 GMT.
  Actions that the application takes, such as getting or setting a value associated with the session do not affect the access time.
  The last accessed time can help to manage sessions. For example, the sessions can be sorted according to age to optimize some task.
- **public int getMaxInactiveInterval()**
  It returns the maximum time interval, in seconds, as integer that the servlet engine will keep this session open between client requests. It is recommended to set the maximum time interval with the setMaxInactiveInterval method.
- **public void setMaxInactiveInterval(int interval)**
  It specifies the maximum length of time, in seconds, that the servlet engine keeps this session if no user requests have been made of the session and takes an integer as a parameter specifying the number of seconds.



Note:
Refer to com.igate.ch8.controller.ManualInvalidate.java servlet code; partial code of doGet() method is given here:

```
// Get the current session object, create one if necessary
    HttpSession session = req.getSession(true);
    // Invalidate the session if it's more than a day old or has been
    // inactive for more than an hour.
    if (!session.isNew()) {  // skip new sessions
      Date dayAgo = new Date(System.currentTimeMillis() - 24*60*60*1000);
      Date hourAgo = new Date(System.currentTimeMillis() - 60*60*1000);
      Date created = new Date(session.getCreationTime());
      Date accessed = new Date(session.getLastAccessedTime());
      if (created.before(dayAgo) || accessed.before(hourAgo)) {
        session.invalidate();
        session = req.getSession(true);  // get a new session
      }
```

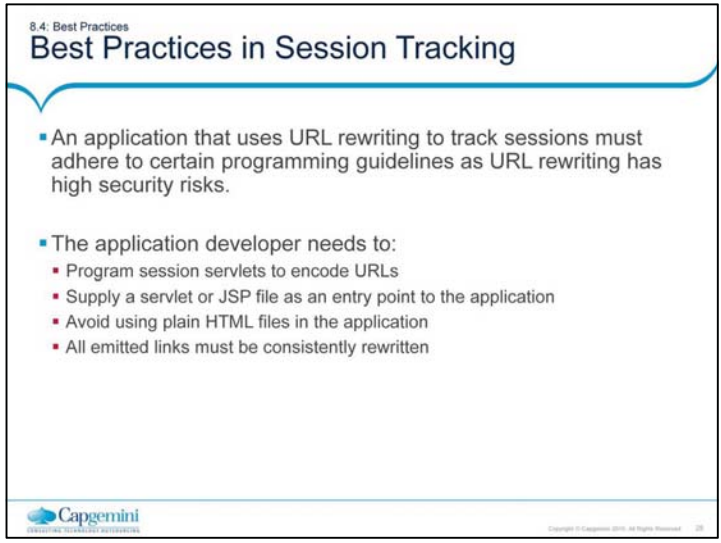


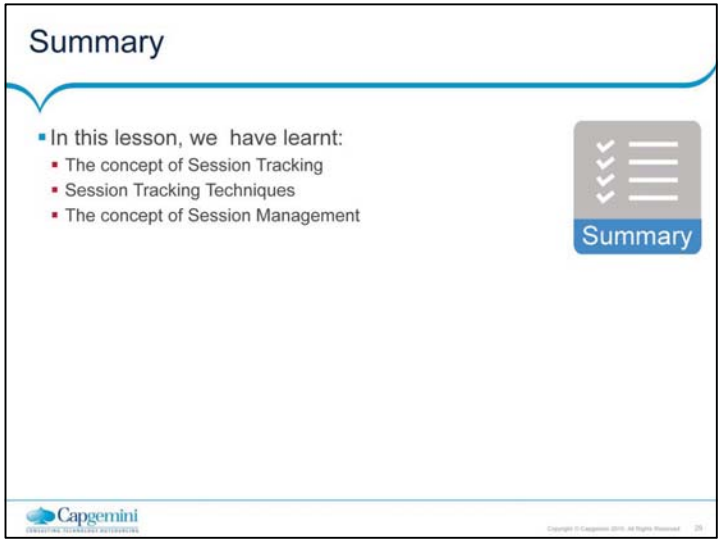
Note:
Execute Servlets:

http://localhost:9090/SessionManagement/Store

This servlet maintains a list of shopping items

http://localhost:9090/SessionManagement/Shopping?item=CD&operation=AddItem
This servlet does different operations like adding item to cart, removing item from cart and making the cart as empty.


8.4: Best Practices
Best Practices in Session Tracking
An application that uses URL rewriting to track sessions must adhere to certain programming guidelines as URL rewriting has high security risks.
The application developer needs to:
Program session servlets to encode URLs
Supply a servlet or JSP file as an entry point to the application
Avoid using plain HTML files in the application
All emitted links must be consistently rewritten
Capgemini


Summary
In this lesson, we have learnt:
The concept of Session Tracking
Session Tracking Techniques
The concept of Session Management
Summary
Capgemini

Review Questions

- Question 1: Web Application developer needs to maintain client's state on server because:
  - Option 1: To remind client of his information
  - Option 2: Http is a stateless protocol
  - Option 3: Http is unreliable protocol
  - Option 4: No need as its taken care of by the Http Server

- Question 2: Browsers required to accept ___ cookies per site:
  - Option 1: 1
  - Option 2: 40
  - Option 3: 20
  - Option 4: unlimited

Capgemini

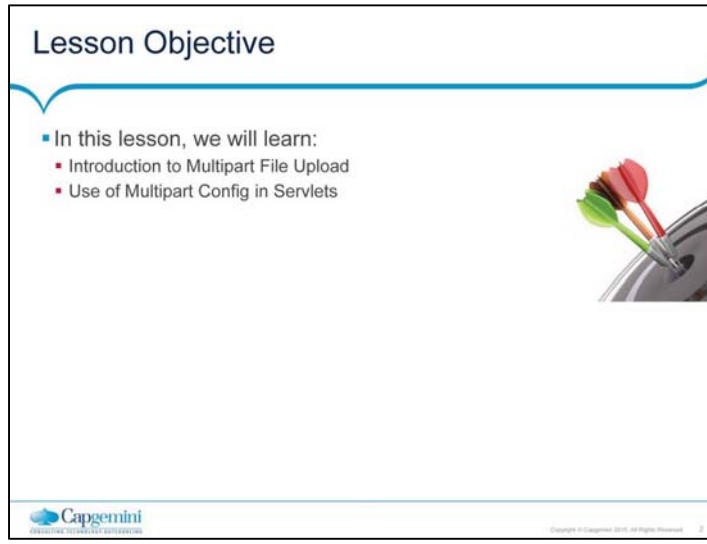
Review Questions

- Question 3: ___ method is used to end the session
  - Option 1: invalidate()
  - Option 2: logout()
  - Option 3: invalidSession()

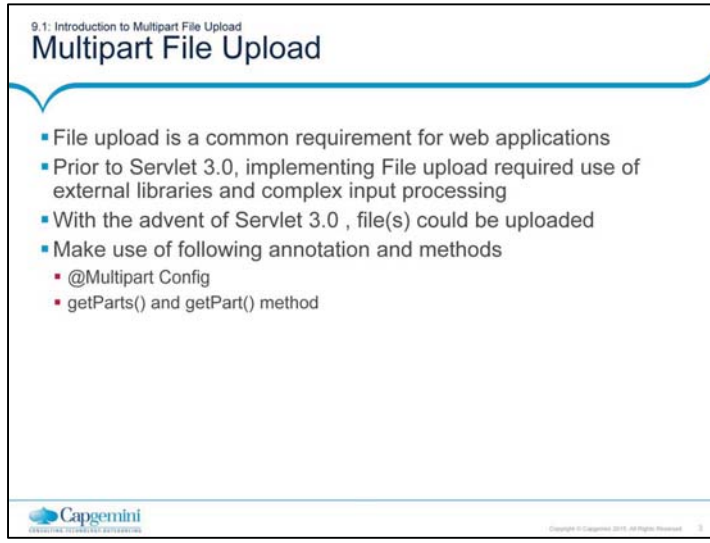
- Question 4: Global timeout for all the sessions is maintained by.
  - Option 1: HttpSession.setMaxInactiveInterval(600)
  - Option 2: iHttpSession.logout(600)
  - Option 3: mapping in web.xml
  - Option 4: All Options are true

Capgemini


Java Servlets 3.0
Lesson 9: Multipart File Upload



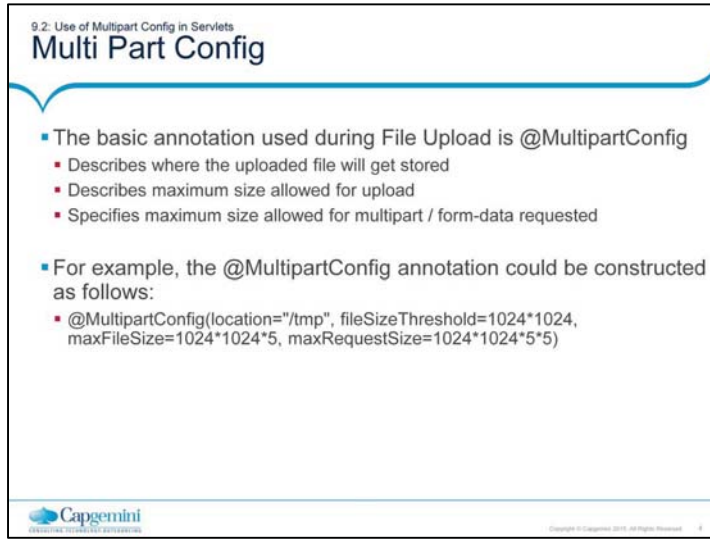
Lesson Objectives:
This lesson introduces Multipart File Upload. The lesson contents are:
Lesson 09: Multipart File Upload
9.1: Introduction to Multipart File Upload
9.2: Use of Multipart Config in Servlets



Multipart File Upload:

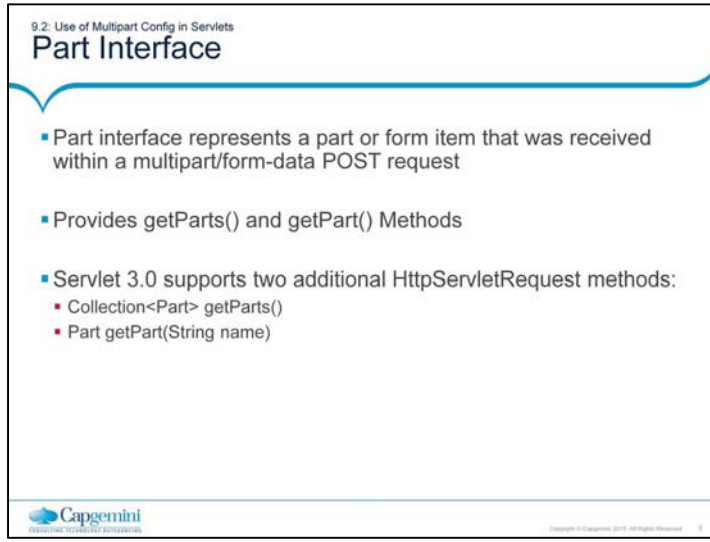
File uploading is now a common requirement for web applications.

It has now become easier since Servlet 3.0



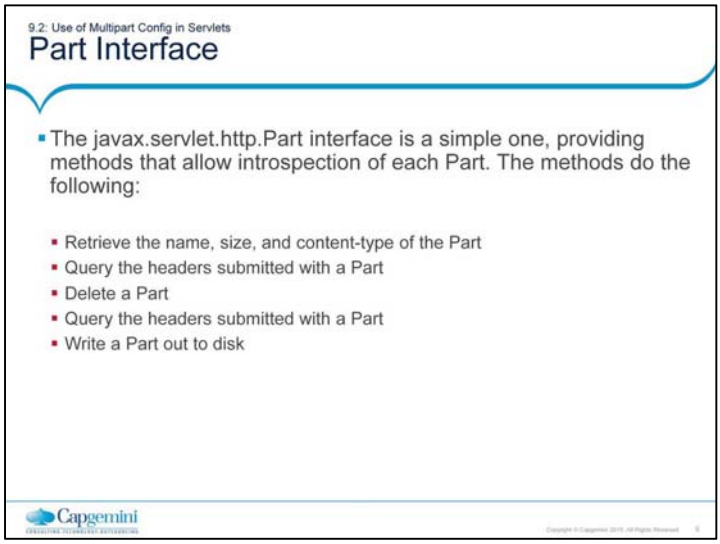
The @MultipartConfig annotation supports the following optional attributes:

location: An absolute path to a directory on the file system. The location attribute does not support a path relative to the application context. This location is used to store files temporarily while the parts are processed or when the size of the file exceeds the specifiedfileSizeThreshold setting. The default location is "".
fileSizeThreshold: The file size in bytes after which the file will be temporarily stored on disk. The default size is 0 bytes.
MaxFileSize: The maximum size allowed for uploaded files, in bytes. If the size of any uploaded file is greater than this size, the web container will throw an exception (IllegalStateException). The default size is unlimited.
maxRequestSize: The maximum size allowed for a multipart/form-data request, in bytes. The web container will throw an exception if the overall size of all uploaded files exceeds this threshold. The default size is unlimited.

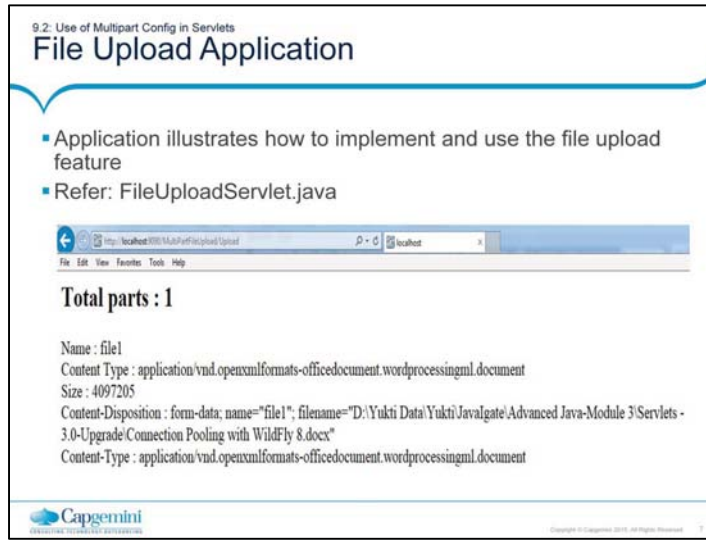


Part interface is now a part of javax.servlet.http package, and used to represent data received within a multipart / form – data.
Some important methods are getInputStream(), write(String fileName) that we can use to read and write file.

The request.getParts() method returns collections of all Part objects. If there are more than one input of type file, multiple Part objects are returned. Since Part objects are named, the getPart(String name) method can be used to access a particular Part. Alternatively, the getParts() method, which returns an Iterable<Part>, can be used to get an Iterator over all the Part objects.

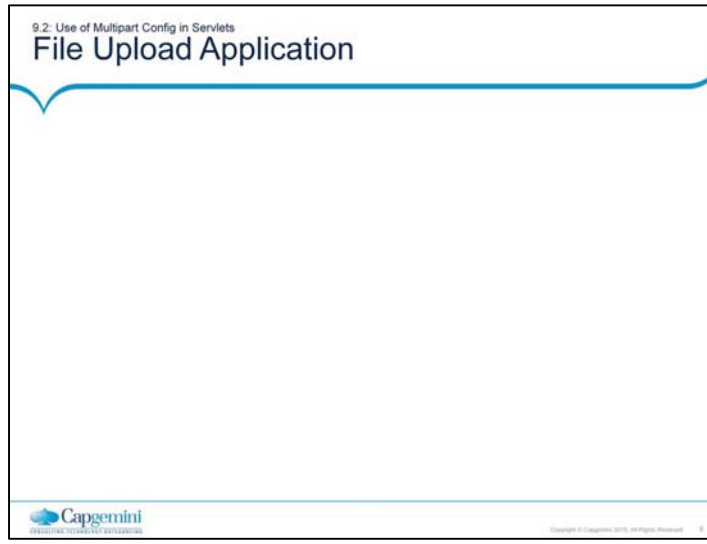


For example, the Part interface provides the write(String filename) method to write the file with the specified name. The file can then be saved in the directory specified with the location attribute of the @MultipartConfig annotation or, in the case of the file-upload example, in the location specified by the Destination field in the form.



The input type as file, enables a user to browse the local file system to select the file. When the file is selected, it is sent to the server as a part of a POST request. During this process two mandatory restrictions are applied to the form with input type file:
The enctype attribute must be set to a value of multipart/form-data.
Its method must be POST.

When the form is specified in this manner, the entire request is sent to the server in encoded form. The servlet then handles the request to process the incoming file data and to extract a file from the stream. The destination is the path to the location where the file will be saved on your computer. Pressing the Upload button at the bottom of the form posts the data to the servlet, which saves the file in the specified destination.



A POST request method is used when the client needs to send data to the server as part of the request, such as when uploading a file or submitting a completed form. In contrast, a GET request method sends a URL and headers only to the server, whereas POST requests also include a message body. This allows arbitrary-length data of any type to be sent to the server. A header field in the POST request usually indicates the message body's Internet media type.
When submitting a form, the browser streams the content in, combining all parts, with each part representing a field of a form. Parts are named after the input elements and are separated from each other with string delimiters named boundary.
This is what submitted data from the file-upload form looks like, after selecting sample.txt as the file that will be uploaded to the tmp-directory on the local file system:

Add the notes here.

Review Question

- Question 1: Which of the following attribute can be used to specify maximum size allowed for uploaded files, in bytes?
  - Option 1: fileSizeThreshold
  - Option 2: MaxFileSize
  - Option 3: maxRequestSize
  - Option 4: locationsize

Capgemini

Add the notes here.

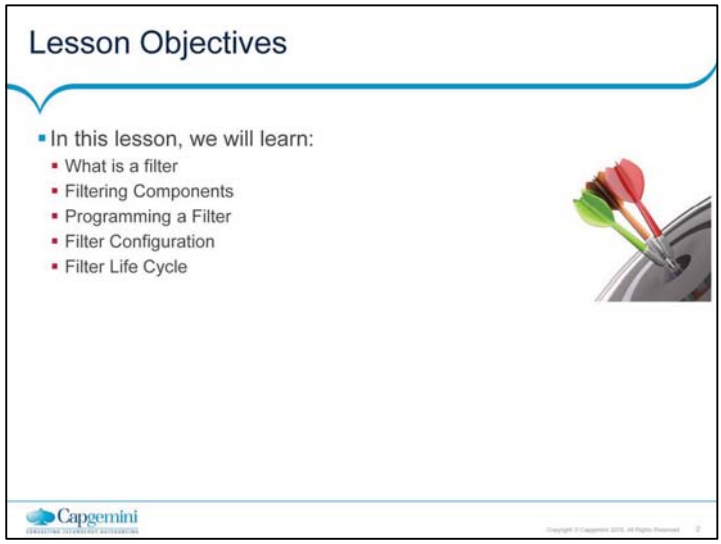
Review Question

- Question 2: Where is the file stored after uploading?
  - Option 1: Stored in current working directory
  - Option 2: Directory as that of where server is installed
  - Option 3: Path specified given in location attribute of @MultipartConfig annotation
  - Option 4: Can specify any arbitrary location on server

Capgemini

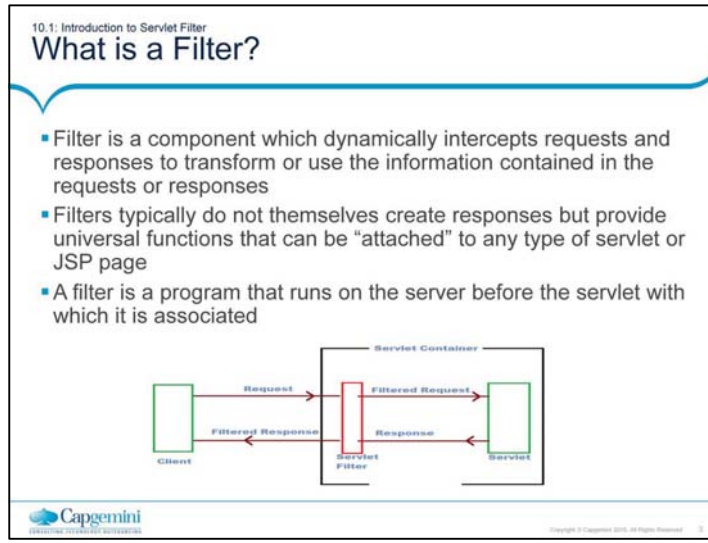
Add the notes here.


Java Servlets 3.0
Lesson 10: Servlet Filter



Lesson Objectives:
This lesson introduces Servlets Filter. The lesson contents are:
Lesson 10: Servlet Filter

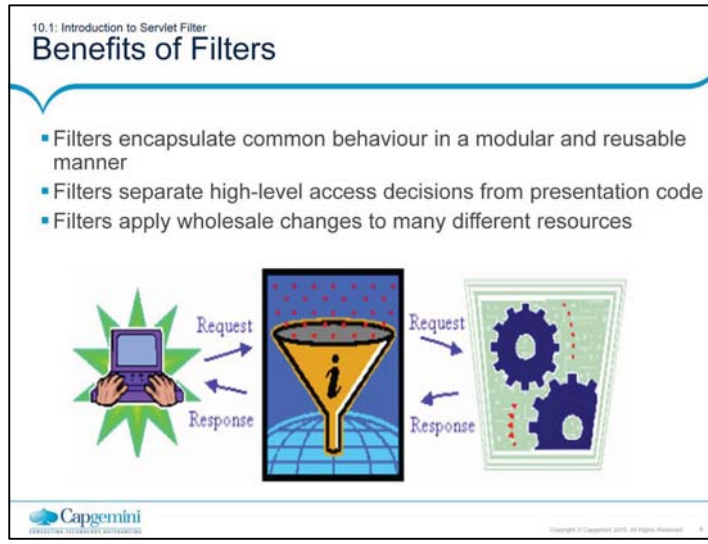
- 10.1: Introduction to Servlet Filter
- 10.2: Filtering Components
- 10.3: Programming a Filter
- 10.4: Filter Configuration
- 10.5: Filter Life Cycle



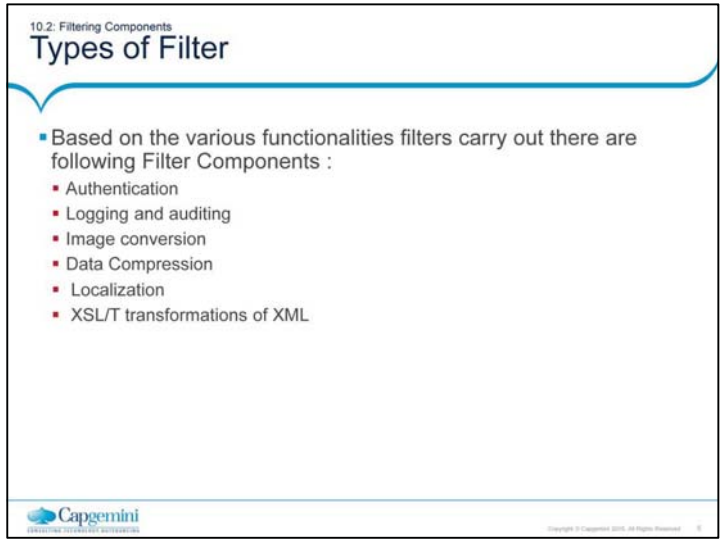
Introduction to Servlet Filter:
What is a Filter?
A filter is a reusable piece of code that transforms either the content of an HTTP request or response and can also modify header information. Filters typically do not themselves create responses, but instead provide universal functions that can be "attached" to any type of servlet or JSP page
A filter is a program that runs on the server before the servlet or JSP page with which it is associated. A filter can be attached to one or more servlets or JSP pages and can examine the request information going into these resources. After doing so, it can choose among the following options :

Invoke the resource (i.e., the servlet or JSP page) in the normal manner
Invoke the resource with modified request information
Invoke the resource but modify the response before sending it to the client
Prevent the resource from being invoked and instead redirect to a different resource, return a particular status code, or generate replacement output

Filters are important because they provide the ability to encapsulate recurring tasks in reusable units



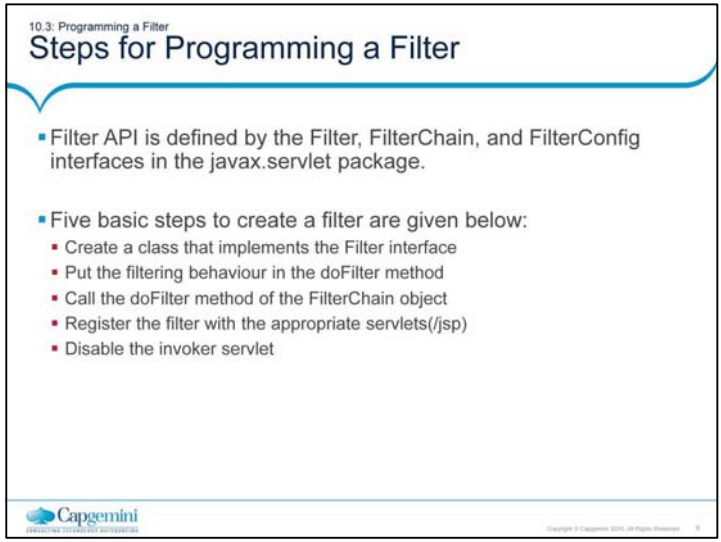
Introduction to Servlet Filter:
Benefits of Filters:
Filters allow to encapsulate common behaviour in a modular and reusable manner.
Modular code is more manageable and documentable, is easier to debug, and if done well, can be reused in another setting.
If we have 30 different servlets that need to compress their content to decrease download time? No problem, make a single compression filter and apply it to all 30 resources.
Filters separate high-level access decisions from presentation code
Filters allow to apply wholesale changes to many different resources
For example: If there a bunch of existing resources that should remain unchanged except that the company name should be changed? No problem: make a string replacement filter and apply it wherever appropriate.



Filtering Components:
Types of Filter:
Filters can perform many different types of functions. Based on their functionalities following are the filters :

Authentication-Blocking requests based on user identity
Logging and auditing-Tracking users of a web application
Image conversion-Scaling maps, and so on
Data compression-Making downloads smaller
Localization-Targeting the request and response to a particular locale
XSL/T transformations of XML content-Targeting web application responses to more that one type of client.

These are just a few of the applications of filters. There are many more, such as encryption, tokenizing, triggering resource access events, mime-type chaining, and caching.

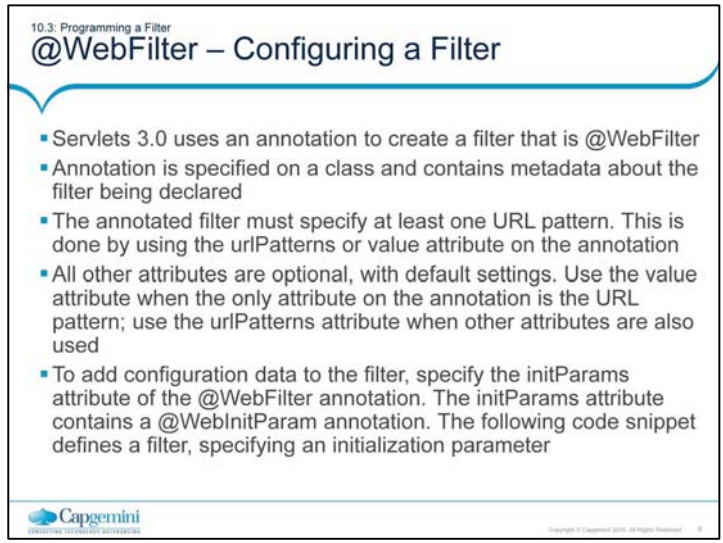

10.3: Programming a Filter
Steps for Programming a Filter
Filter API is defined by the Filter, FilterChain, and FilterConfig interfaces in the javax.servlet package.
Five basic steps to create a filter are given below:
Create a class that implements the Filter interface
Put the filtering behaviour in the doFilter method
Call the doFilter method of the FilterChain object
Register the filter with the appropriate servlets(/jsp)
Disable the invoker servlet
Capgemini

Programming a Filter:
Creating a filter involves five basic steps:
Create a class that implements the Filter interface.
The class will need three methods: doFilter, init, and destroy. The doFilter method contains the main filtering code (see Step 2), the init method performs setup operations, and the destroy method does cleanup.
Put the filtering behaviour in the doFilter method.
The first argument to the doFilter method is a ServletRequest object. This object gives the filter full access to the incoming information, including form data, cookies, and HTTP request headers. The second argument is a ServletResponse; it is mostly ignored in simple filters. The final argument is a FilterChain; it is used to invoke the servlet or JSP page or the next filter as described in the next step.
Call the doFilter method of the FilterChain object.
The doFilter method of the Filter interface takes a FilterChain object as one of its arguments. When doFilter method of that object is invoked, the next associated filter is invoked. If no other filter is associated with the servlet or JSP page, then the servlet or page itself is invoked.
Register the filter with the appropriate servlets and JSP pages.
Use the filter and filter-mapping annotations above the class.
Disable the invoker servlet. Prevent users from bypassing filter settings by using default servlet URLs.

The most important method in the Filter interface is doFilter, which is passed request, response, and filter chain objects. This method can perform the following actions:

Examine the request headers.
Customize the request object if the filter wishes to modify request headers or data.
Customize the response object if the filter wishes to modify response headers or data.
Invoke the next entity in the filter chain. If the current filter is the last filter in the chain that ends with the target web component or static resource, the next entity is the resource at the end of the chain; otherwise, it is the next filter that was configured in the WAR. The filter invokes the next entity by calling the doFilter method on the chain object, passing in the request and response it was called with or the wrapped versions it may have created.

Alternatively, the filter can choose to
block the request by not making the call to invoke the next entity. In the latter case, the filter is responsible for filling out the response.
Examine response headers after invoking the next filter in the chain.
Throw an exception to indicate an error in processing.
In addition to doFilter, the init and destroy methods must be implemented. The init method is called by the container when the filter is instantiated. If need to pass initialization parameters to the filter, can be retrieved the FilterConfig object passed to init method

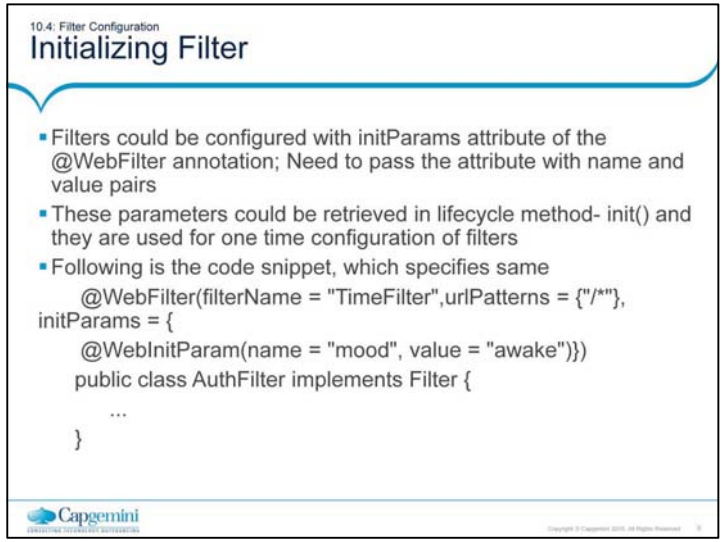


Classes annotated with the @WebFilter annotation must implement the javax.servlet.Filter interface.

@WebFilter is used to create a filter in Servlet 3.0.

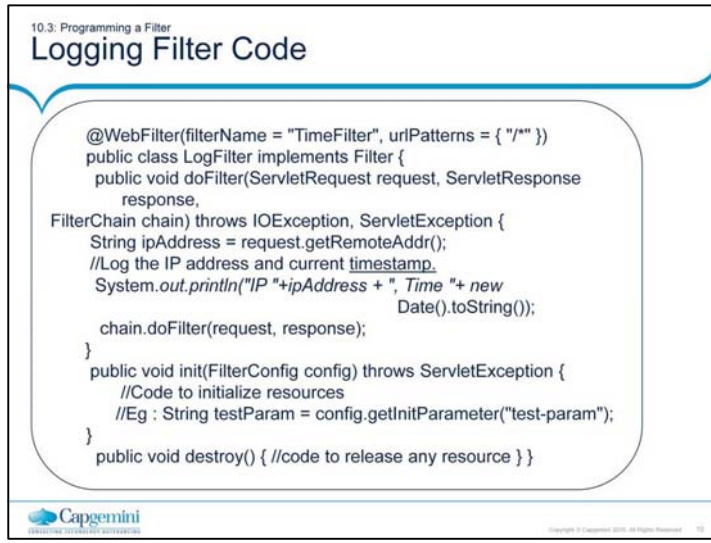
Configuring Filter with annotations:
Next is registering a filter with appropriate servlet. To map a filter to a servlet make note of following:
Declare the filter using the annotation @WebFilter above class name. This annotation will inform the container to create a filter. This filter would then be invoked before the actual web resource. Also initialization parameters could be used for the filter, by making use of annotation @WebInitParam.

Map the filter to a servlet by defining an attribute urlPatterns, this URL pattern could be of any web resource.
If all web resources need to be matched to particular filter then, use /* as the URL Pattern

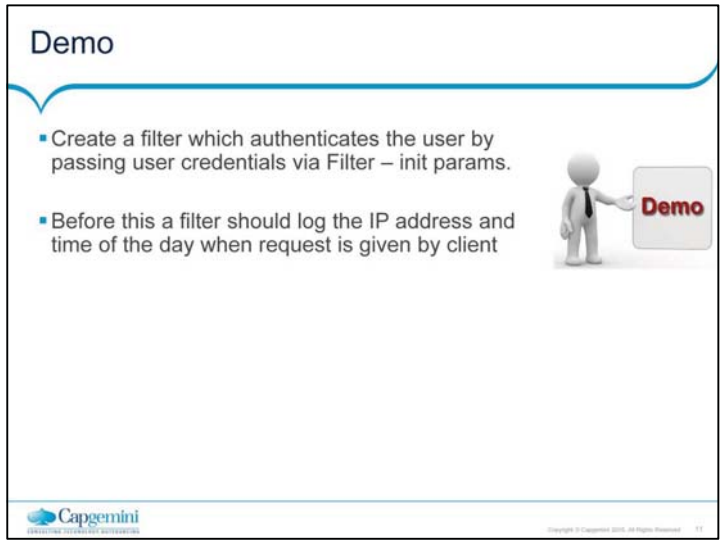


In above code snippet, we have used attribute initParams to pass the configuration information to filters.

These parameters could be retrieved in init() - method of filter



Programming a Filter:
Logging Filter Code:
The above filter creates a Servlet Filter that just prints the clients IP address and date time. (This is just to log users who are accessing the application). We have implemented the javax.servlet.Filter interface and implement its methods – init(), doFilter() and destroy().
The init() method is used to initialize any code that is used by Filter. Note that init() method will get an object of FilterConfig which contains different Filter level information as well as init parameters which is passed from web.xml.
The doFilter() method is where the actual filtering process happens. User implementation would add code which can modify request / session / response, add any attribute in request etc.
The destroy() method is called by the container when it wants to garbage collect the filter.
The above code involves first three steps of programming a filter.
A set of initialization parameters can be associated with a filter using the init-params attribute of @WebFilter annotation. The names and values of these parameters are available to the Filter at runtime via the getInitParameter() and getInitParameterNames() methods on the filter's FilterConfig. Additionally, the FilterConfig affords access to the ServletContext of the web application for the loading of resources, for logging functionality or for storage of state in the ServletContext's attribute list
See the next subsequent slides for examples



Note:
Refer the LogFilter and AuthFilter.java. AuthFilter authenticates the user by passing user credentials via init-params.

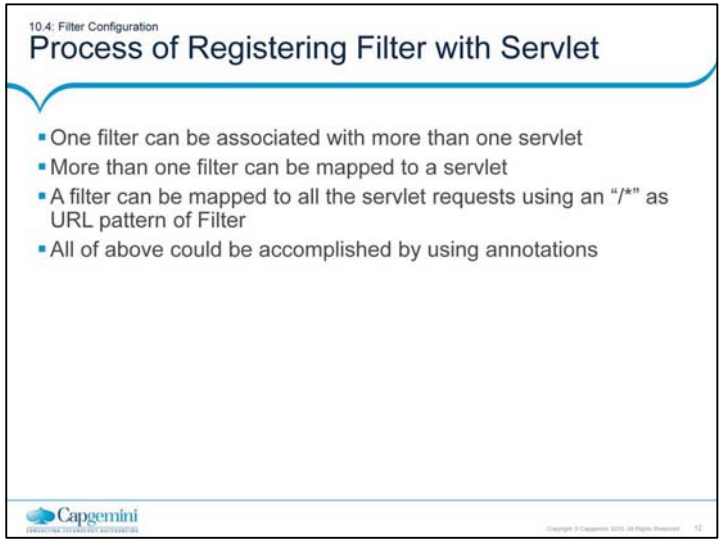
Also we need to log the IP address and time of day when request was raised.

Already a filter is created called LogFilter for this purpose.

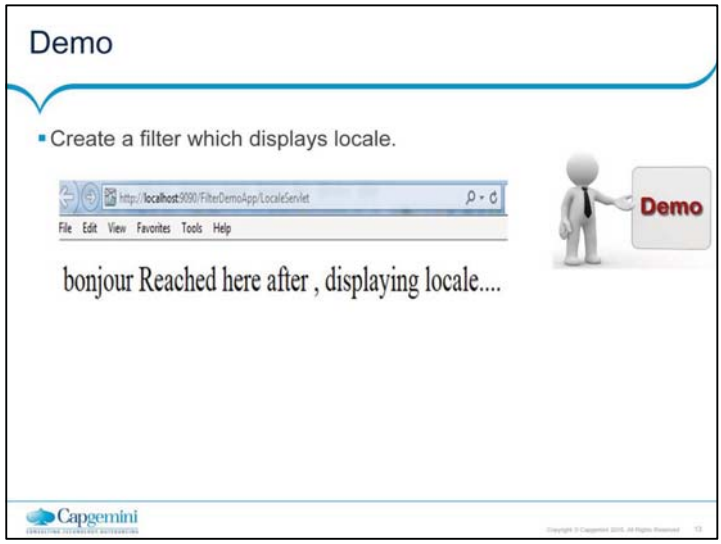
Thus we use LogFilter and AuthFilter together before the request comes to web resource – HelloServlet

If the user credentials do not match, the filter will generate a response stating that user is invalid displaying the incorrect user name

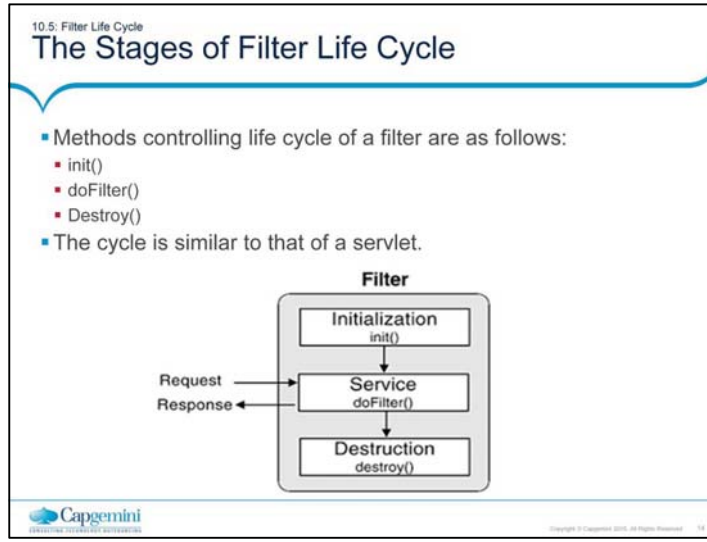
Observe that we are mapping more than one filter with Servlet


10.4: Filter Configuration
Process of Registering Filter with Servlet
One filter can be associated with more than one servlet
More than one filter can be mapped to a servlet
A filter can be mapped to all the servlet requests using an "/*" as URL pattern of Filter
All of above could be accomplished by using annotations
Capgemini

Refer:



Refer LocaleFilter that displays locale information depending upon the language selected in browser



Filter Life Cycle:
Three methods of Filter interface makes up the life cycle of a filter:
public void init(FilterConfig config) throws ServletException
The init method is executed only once, when the filter is first initialized. It is not executed each time the filter is invoked.
public void doFilter(ServletRequest request,ServletResponse response, FilterChain chain) throws ServletException, IOException
The doFilter method is executed each time a filter is invoked (i.e., once for each request for a servlet or JSP page with which the filter is associated). It is this method that contains the bulk of the filtering logic.
public void destroy()
This method is called when a server is permanently finished with a given filter object (e.g., when the server is being shut down). Most filters simply provide an empty body for this method, but it can be used for cleanup tasks like closing files or database connection pools that are used by the filter.
Detailed Lifecycle of Filter:
After the time when the web application containing filters is deployed, and before an incoming request for a resource in the web application causes a the container to access the resource and serve it back, the container must look through the list of filter mappings to locate the list of filters that must be applied to the resource. How this list is built, is described next.

Filter Life Cycle:

•The container must ensure at some point in this time that, for each filter instance that is to be applied, it has instantiated a filter of the appropriate class, and called setFilterConfig(FilterConfig config) on each filter instance in the list.

•The container ensures that the javax.servlet.FilterConfig instance that is passed in to this call has been initialized with the filter name as declared in the deployment descriptor for that filter, with the reference to the ServletContext for this web application and with the set of initialization parameters declared for the filter in the deployment descriptor.

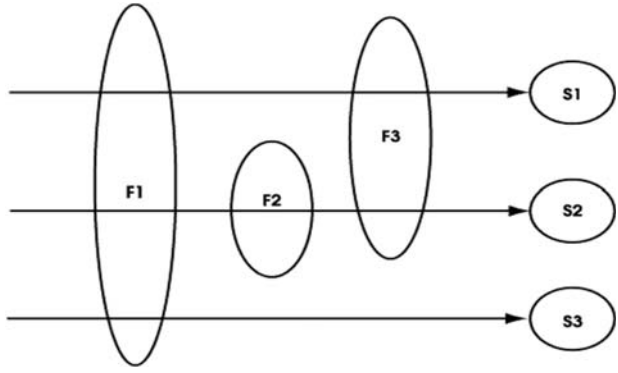
•When the container receives the incoming request, it takes the first filter instance in the list and calls its doFilter() method, passing in the ServletRequest and ServletResponse, and a reference to the FilterChain object it will use.

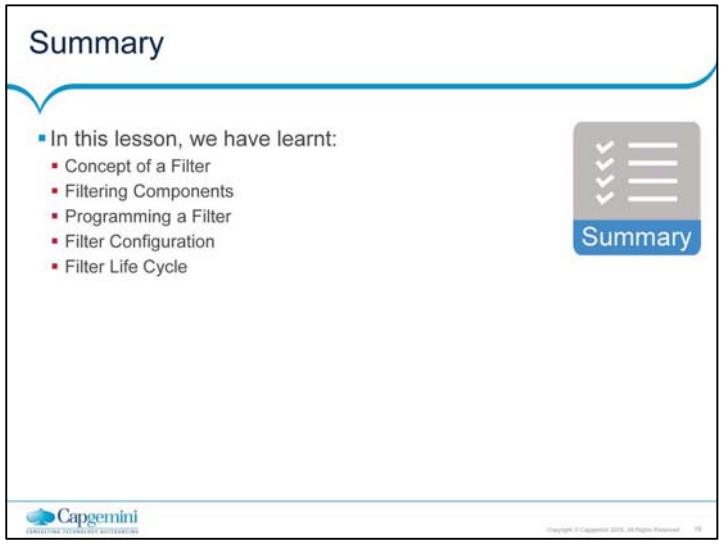
FilterConfig Interface:

•A filter configuration object used by a servlet container to pass information to a filter during initialization.

FilterChain Interface :

A FilterChain is an object provided by the servlet container giving a view into the invocation chain of a filtered request for a resource. Filters use the FilterChain to invoke the next filter in the chain, or if the calling filter is the last filter in the chain, to invoke the resource at the end of the chain. For eg, the below figure shows how Filters F1 and F3 intercept invocations to Servlet S1.





Add the notes here.

Review Question

- Question 1: Which method of Filter interface contains all the logic ?
  - Option 1: init()
  - Option 2: doFilter()
  - Option 3: destroy()
- Question 2: We can use an asterisk in a url Pattern which will represent all servlets as well as JSP referring to same filter
  - True/False
- Question 3: We can not have multiple url Pattern for a filter?
  - True/False

Capgemini

Add the notes here.


Java Servlets 3.0
Appendices

Appendix A: XML Mappings in Servlets 3.0

Basic Servlet Mappings:

If servlets need to be created without annotations, XML mappings can be used as below:
This mapping would create a servlet with name as Hello—servlet name is a logical name (label). Servlet class is HelloServlet—in package com.igate. Thus a container can create a servlet.

```
<web-app>  : Root tag; There would be XSD mappings going with the root tag
<servlet>
    <servlet-name>Hello</servlet-name>
    <servlet-class>com.igate.HelloServlet</servlet-class>
</servlet>
</web-app>
```

This mapping would bind the servlet to a particular URL pattern. Using this URL Pattern request would be given in the browser: URL pattern is preceded by '/'

```
<servlet-mapping>
    <servlet-name>Hello</servlet-name>
    <url-pattern>/Hello</url-pattern>
</servlet-mapping>
```

Servlets with Init Parameters:

Servlet can also have initialization parameters and contextual parameters which can be fetched via ServletConfig and ServletContext objects respectively.

Servlet initialization parameters are a part of servlet and cannot be shared among other servlet; whereas contextual parameters are shared by entire web application:

A servlet can have multiple initialization parameters.

```
<servlet>
    <servlet-name>Hello</servlet-name>
    <servlet-class>com.igate.HelloServlet</servlet-class>
    <init-param>
                <param-name>flower</param-name>
                <param-value>rose</param-value>
    </init-param>
</servlet>
```

Context-parameter can be done as follows: This tag should be outside <servlet-mapping> and <servlet> tags

Multiple context-parameters can be set.

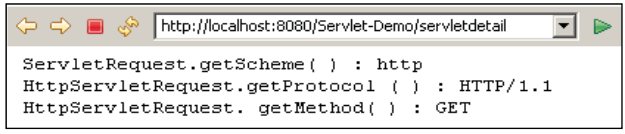
```
<context-param>
                <param-name>email</param-name>
                <param-value>info@test.com</param-value>
</context-param>
```

Mappings for welcome files:

If we deploy the web application, the welcome file is expected to execute. (i.e. the first page to execute when the application runs). This can be set through welcome file list tag. It checks for first file in listing -> index.html if not found then it will search for index.htm and so on. If no welcome file is found then HTTP 404 error is raised. See below listing for same.

```
<welcome-file-list>
   <welcome-file>index.html</welcome-file>
   <welcome-file>index.htm</welcome-file>
   <welcome-file>index.jsp</welcome-file>
   <welcome-file>default.html</welcome-file>
   <welcome-file>default.htm</welcome-file>
   <welcome-file>default.jsp</welcome-file>
 </welcome-file-list>
```

Request URL	Its URI path	Its Servlet path
http://server:port/context/classname	context/classname	context/classname
http://server:port/context/classname?var=val	context/classname	context/classname
http://server:port/context/classname/pathInfo	context/classname/pathInfo	context/classname



Appendix B : Request Object

Idempotent Methods:

GET is idempotent method as GET is NOT supposed to update data, but to get data, although we can(and we do) implement GET to update data, by posting data using query string. So, GET may sometimes behave non – idempotent manner.

POST method is considered non-idempotent. The reason is that the POST is supposed to post data to the server, which means it may be used to save the data in persistent location OR to do some action based on that posted data.

Determining What is Requested:

A Servlet can find out what field or servlet was requested using several methods. There exists no single method to find the original URL used by client to make the request.

- **public static StringBuffer HttpUtils.getRequestURL(HttpServletRequest req):** This method reconstructs the request URL based on information available in the HttpservletRequest object. The StringBuffer result contains the scheme, server name, server port, and extra path information.
- **public String HttpServletRequest.getRequestURI():** This method returns the URI (Universal Resource Identifier). This is the context path plus the servlet path plus any extra path information. A URI can be thought of as a URL minus the scheme, host, port, and query string, but including extra path information.
- **public String HttpServletRequest.getServletPath( ):** This method returns the part of the URI that refers to the servlet being invoked or null if the URI does not directly point to a servlet. The servlet path does not include extra path information.

For some examples, see the table given below.

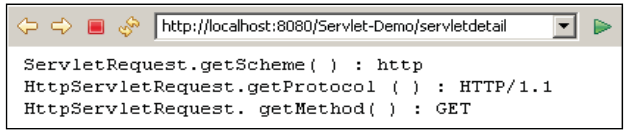
Request URL	Its URI path	Its Servlet path
http://server:port/context/classname	context/classname	context/classname
http://server:port/context/classname?var=val	context/classname	context/classname
http://server:port/context/classname/pathInfo	context/classname/ pathInfo	context/classname

Determining How it was Requested:

Apart from knowing what was requested, a servlet can also find out how it was requested. The methods below provide this information.

- public String ServletRequest.getScheme( ) : It returns the name of the scheme used to make this request, for example, http, https, or ftp.
- public String HttpServletRequest.getProtocol ( ) : It returns the name and version of the protocol the request uses.
- public String HttpServletRequest. getMethod( ) : It returns the name of the HTTP method with which this request was made like GET, POST, HEAD, etc.

For an example, see the figure below.



Appendix B : Request Object

Methods for Using Input Stream:

We have seen in our examples how every **response** has a **output stream** associated with it. Similarly, each **request** handled by a servlet has an **input stream** associated with it. The servlet can read from the **Reader** or **InputStream** associated with its request object. The data read from the input stream can be of any content type and of any length. The input stream has three purposes:

- To pass the response body from one servlet to another in a servlet chain
- To pass an HTTP servlet the content associated with a POST request
- To pass a non-HTTP servlet raw data by the client

There are several methods that the request object uses to read data:

- **BufferedReader ServletRequest.getReader( ):** It retrieves the body of the request as character data using a BufferedReader. This method throws an IllegalStateException if **getInputStream( )** has been called before on the same request and an UnsupportedEncodingException if the character encoding of the input is unsupported or unknown.

- **ServletInputStream ServletRequest.getInputStream( ):** It retrieves the body of the request as binary data using a **ServletInputStream**. The method throws an IllegalStateException if **getReader( )** has been called before on this same request. Once you have the ServletInputStream, you can read a line from it using **readLine()**.

- **String ServletRequest.getContentType( ) and int ServletRequest.getContentLength( ):** It checks the content type and the length of the data being sent via the input stream.

Appendix C : Response Object

Sending Compressed Data Using the "Accept-Encoding" Header :

Browsers that support content encoding feature, indicate that they do so by setting the 'Accept-Encoding' request header.

Servlets can check this header and send compressed response to clients that support encoding and send a regular web page to those that don't. For example, let us assume that client accepts **gzip** encoding. Implementing compression is straightforward since gzip is built into the Java programming language via **java.util.zip**.

The servlet first checks the **Accept-Encoding** header to see if it contains an entry for **gzip**.

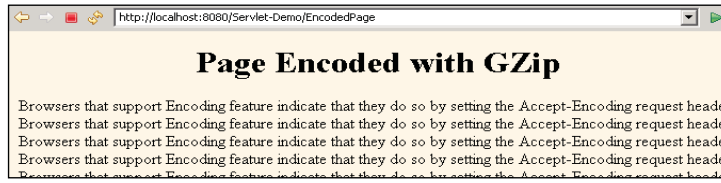
•If so, it uses a **GZIPOutputStream** to generate the page, specifying **gzip** as the value of the **Content-Encoding** header. You must explicitly call **close** when using a **GZIPOutputStream**.

•If **gzip** is not supported, then the servlet uses the normal PrintWriter to send the page.

Note: GZIP is often used while loading JavaScript files to reduce the time for loading. Refer com.igate.ch4.EncodedPage.java class. A partial listing is shown below:

```
public void doGet(HttpServletRequest req, HttpServletResponse resp){
  //check which coding style browser accepts
  String encode = req.getHeader("Accept-Encoding");
  PrintWriter out; String title="Page Encoded with Gzip";
   if ((encode != null) && (encode.indexOf("gzip") != -1)){
       OutputStream out1 = resp.getOutputStream();
       out = new PrintWriter(new GZIPOutputStream(out1), false);
       resp.setHeader("Content-Encoding", "gzip");
    } else {out = resp.getWriter(); }
     out.println("<HTML><BODY><H1 ALIGN=CENTER>" + title +
"</H1>\n");
     String line = "Browsers that support Encoding feature indicate that "
              + "they do so by setting the Accept-Encoding request header";
    for (int i = 0; i < 10000; i++)  out.println(line);
     out.println("</BODY></HTML>"); out.close();
}
```

Output:



Appendix C : Response Object

Restricting Access to Web Pages:

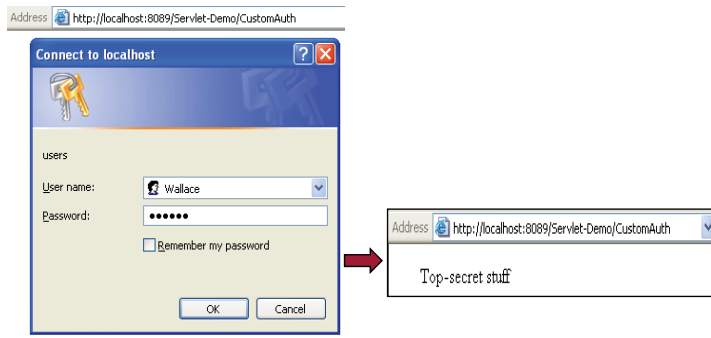
- Many web servers support standard mechanisms for limiting access to designated Web pages. These mechanisms can apply to static pages as well as those generated by servlets, so many authors use their server specific mechanism for restricting access to servlets. Furthermore, most users at e-commerce sites prefer to use regular HTML forms to provide authorization information since these forms are more familiar, can provide more explanatory information, and can ask for additional information beyond just a user name and password. Once a servlet that uses form-based access grants initial access to a user, it would use session tracking to give the user access to other pages that require the same level of authorization.
- Nevertheless, form-based access control requires more effort on the part of the servlet developer, and HTTP-based authorization is sufficient for many simple applications. Here, the servlet uses status codes and HTTP headers to manage its own basic authentication policy. It receives encoded user credentials in Authorization header. If it chooses to deny those credentials, it does so by sending SC_UNAUTHORIZED status code and a WWW_Authenticate header that describes the desired credentials. The Authorization header, if sent by the client, contains the client's username and password ("username:password") encoded in Base64. It also tells the client the authorization scheme and the realm against which the users will be verified. *(A realm is just a collection of user accounts and protected resources)*. For example: To tell the client to use basic authentication for the realm Admin, the WWW_Authenticate header is:
  - WWW_Authenticate: BASIC realm="Admin"

```
public void init(ServletConfig config) {
      users.put("Wallace:cheese", "allowed"); //Hashtable entry
          //other hardcoded entries … … }
public void doGet(HttpServletRequest req, HttpServletResponse res) {
    String auth = req.getHeader("Authorization");    // Get Authorization
header
    if (!allowUser(auth)) {
        res.setHeader("WWW-Authenticate", "BASIC realm=\"users\"");
        res.sendError(res.SC_UNAUTHORIZED);
     }  else out.println("Top-secret stuff");
 }
protected boolean allowUser(String auth) throws IOException {
    if (auth == null) return false;  // no auth
    if (!auth.toUpperCase().startsWith("BASIC ")) return false;
    String userpassEncoded = auth.substring(6);
    sun.misc.BASE64Decoder dec = new sun.misc.BASE64Decoder();
    String userpassDecoded = new
String(dec.decodeBuffer(userpassEncoded));
        if ("allowed".equals(users.get(userpassDecoded))) return true;
        else return false;
  }
```

Appendix C : Response Object

Restricting Access to Web Pages:

- The previous example shows a servlet that performs custom authorization, receiving an Authorization header and sending the SC_UNAUTHORIZED status code and WWW-Authenticate header when necessary. The servlet restricts access to its "top-secret stuff" to those users (and passwords) it recognizes in its user list. For this example, the list is kept in a simple **Hashtable** and its contents are hard-coded. Of course, for a production servlet data will be retrieved from an external media like database.
- To retrieve the Base64-encoded username and password, the servlet needs to use a Base64 decoder. Fortunately, there are several freely available decoders. For this servlet, we have chosen to use the sun.misc.BASE64Decoder class that accompanies the JDK, so it is probably already on your system.
- Although the web server is asked to grant any client access to this servlet, the servlet sends its "top-secret" output only to those users it recognizes.
- The **doGet()** method checks if there is an Authorization header. If there is no such header, it returns a 401 (Unauthorized) response code and a header of the following form: WWW-Authenticate: BASIC realm="name"
- With this response, the browser pops up a dialog box telling the user to enter a name and password for *name*, then to reconnect with that username and password embedded in a single base64 string inside the Authorization header.
- If there is a Authorization header skip over the word "basic" and reverse the base64 encoding of the remaining part. This results in a string of the form username:password. Check the username and password, against some stored set. If it matches, return the page.
- Refer the com.igatepatni.ch4.CustomAuth.java class. Invoke this servlet as follows:
  - http://localhost:8080/Servlet-Demo/CustomAuth
- When you access this servlet it asks for username and password as shown in the first figure above. Type username as "Wallace" and password as "cheese" to display the servlet output as shown in the second figure in the above slide.



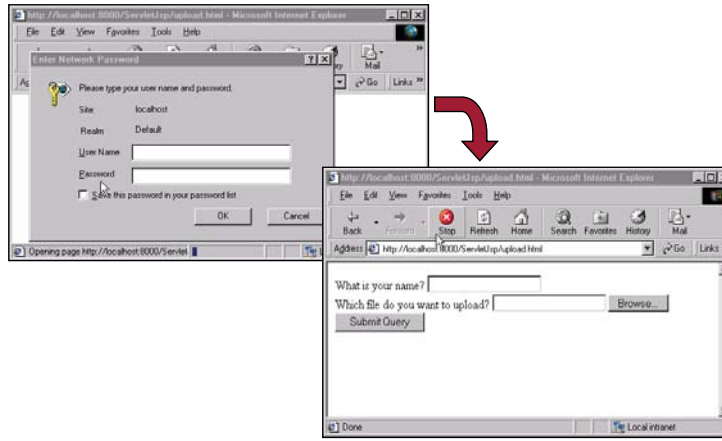
Appendix C : Response Object

Protecting Pages by configuring Server:

•You can protect pages by configuring server. Please refer to server manual at your location to understand how new users may be added to an existing list of users.

•Invoke the protected page as shown in the first figure below. The browser asks the user to enter username and password. This page is valid only for users who have entry in the realm. In this example, assuming upload.html is protected for authorized users, it is shown only after providing valid username and password.

We shall see in the SSL session how server can be configured to restrict access to pages.



Appendix D :More on HttpSession API

Session Binding Events

Some objects may wish to perform an action when they are bound or unbound from a session. For example, a database connection may begin a transaction when bound to a session and end the transaction when unbound. Any object that implements the javax.servlet.http. HttpSessionBindingListener interface is notified when it is bound or unbound from a session. The interface is:

```
public abstract interface HttpSessionBindingListener
extends java.util.EventListener {
  public void valueBound(HttpSessionBindingEvent event);
  public void valueUnbound(HttpSessionBindingEvent event);
}
```

The *valueBound()* method is called when the listener is bound into a session, and *valueUnbound()* is called when the listener is unbound from a session.

The HttpSessionBindingEvent constructor argument provide access to the name under which the object is being bound(or unbound) with *getName()* method. *getSession()* method returns the session object.

public HttpSessionBindingEvent(HttpSession session,String name) : It constructs an event that notifies an object that it has been bound to or unbound from a session. To receive the event, the object must implement HttpSessionBindingListener.

public java.lang.String getName() : It returns the name with which the object is bound to or unbound from the session.

Refer to SessionBindings servlet. This servlet demonstrate the use of HttpSessionBindingListener and HttpSessionBindingEvent with listener that logs when it is bound and unbound from a session.

```
public class SessionBindings extends HttpServlet {
   public void doGet(HttpServletRequest req,
                     HttpServletResponse res)
                      throws ServletException,
IOException {
      res.setContentType("text/plain");
      PrintWriter out = res.getWriter();
      // Get the current session object, create one if
necessary
      HttpSession session = req.getSession(true);
     // Add a CustomBindingListener
      session.setAttribute("bindings.listener",
          new
com.igatepatni.ch5.CustomBindingListener(

getServletContext()));
      out.println("This page intentionally left blank");
   }
}
```

Appendix D :More on HttpSession API
Session Binding Events

```
class CustomBindingListener implements
                                    HttpSessionBindingListener {
    // Save a ServletContext to be used for its log() method
    ServletContext context;
    public CustomBindingListener(ServletContext context) {
        this.context = context;
    }
    public void valueBound(HttpSessionBindingEvent event) {
        context.log("BOUND as " + event.getName() +
                                    " to " + event.getSession().getId());
    }
    public void valueUnbound(HttpSessionBindingEvent event) {
        context.log("UNBOUND as " + event.getName() +
                                    " from " + event.getSession().getId());
    }
}
```

•Each time a CustomBindingListener object is bound to a session, its valueBound() method is called and the event is logged. Each time it is unbound from a session, its valueUnbound() method is called so that event too is logged. We can observe the sequence of events by looking at the server's event log.
•Let us assume that this servlet is called once, reloaded 30 seconds later, and not called again for at least a half hour. The event log would look something like this:

```
[Tue Jan 27 01:46:48 PST 1998]
  BOUND as bindings.listener to INWBUJIAAAAAHQDGPM5QAAA
[Tue Jan 27 01:47:18 PST 1998]
  UNBOUND as bindings.listener from INWBUJIAAAAAHQDGPM5QAAA
[Tue Jan 27 01:47:18 PST 1998]
  BOUND as bindings.listener to INWBUJIAAAAAHQDGPM5QAAA
[Tue Jan 27 02:17:18 PST 1998]
  UNBOUND as bindings.listener from INWBUJIAAAAAHQDGPM5QAAA
```

Appendix E : Inter-Servlet Communication:

Additional Request-scoped Attributes:

While doing include or forward by default some attributes will get populated. They are:

javax.servlet.include.request_uri : It is the part of this request's URL from the protocol name up to the query string. For example: /cart/products.html.

javax.servlet.include.servlet_path : It is the part of this request's URL that calls the servlet. This path starts with a "/" character and includes either the servlet name or a path to the servlet, but does not include any extra path information or a query string.

javax.servlet.include.context_path : It is the portion of the request URI that indicates the context of the request. The context path always comes first in a request URI. The path starts with a "/" character but does not end with a "/".

javax.servlet.include.path_info : It is any extra path information associated with the URL the client sent when it made this request. The extra path information follows the servlet path but precedes the query string and will start with a "/" character.

javax.servlet.include.query_string : It is the query string that is contained in the request URL after the path.

A servlet that has been invoked by another servlet using the **include()** method of RequestDispatcher has access to the path by which it was invoked. The request attributes discussed above must be set. These attributes are accessible from the included servlet via the getAttribute() method on the request object and their values must be equal to the request URI, context path, servlet path, path info, and query string of the INCLUDED servlet, respectively. If the request is subsequently included, these attributes are replaced for that include. Eg:

request.getAttribute("javax.servlet.include.request_uri")

On the other hand, a servlet that has been invoked by another servlet using the **forward()** method of RequestDispatcher has access to the path of the ORIGINAL request. The request attributes shown below must be set.

- javax.servlet.forward.request_uri
- javax.servlet.forward.servlet_path
- javax.servlet.forward.path_info
- javax.servlet.forward.query_string

These attributes are accessible from the forwarded servlet via the getAttribute method on the request object. Note that these attributes must always reflect the information in the original request even under the situation that multiple forwards and subsequent includes are called.

If the included or forwarded servlet was obtained by using the getNamedDispatcher() method, these attributes MUST NOT be set.

Appendix E : Inter-Servlet Communication:

Accessing Passive Server resources:

Passive server resources (for example: static HTML pages which are stored in local files) cannot be accessed with RequestDispatcher objects. The ServletContext method getResource(String path) returns a URL object for a resource specified by a local URI (for example: "/" for the server's document root) which can be used to examine the resource.

If you only want to read the resource's body you can directly ask the ServletContext for an InputStream with the getResourceAsStream () method.

A Servlet may need to access additional resources like configuration files whose locations should not need to be specified in init parameters. Those resources can be accessed with the methods getResource(String name) and getResourceAsStream(String name) of the java.lang.Class object which represents the Servlet's class.

Example. The following code gets an InputStream for a configuration file named mycfg.cfg which resides in the same directory as the class in which the code is executed:

Note that the servlet class loader must implement the getResource() and getResourceAsStream() methods in order for this to work. This may not be the case with all Servlet engines.

```
InputStream is = getClass().getResourceAsStream("mycfg.cfg");
```

Appendix F : Application Lifecycle Events

- One of the most significant change in Servlet API 2.3 was the addition of application lifecycle events, which let "listener" objects be notified when servlet contexts and sessions are initialized and destroyed, as well as when attributes are added or removed from a context or session.
  Servlet lifecycle : These events work like Swing events.
  ServletContext lifecycle : Any listener interested in observing the ServletContext lifecycle can implement the ServletContextListener interface. The interface has two methods:

- **void contextInitialized(ServletContextEvent e):** Called when a Web application is first ready to process requests (i.e. on Web server startup and when a context is added or reloaded). Requests will not be handled until this method returns.
- **void contextDestroyed(ServletContextEvent e):** Called when a Web application is about to be shut down (i.e. on Web server shutdown or when a context is removed or reloaded). Request handling will be stopped before this method is called.

The ServletContextEvent class passed to those methods has only a getServletContext() method that returns the context being initialized or destroyed.

ServletContext attribute lifecycle : A listener interested in observing the ServletContext attribute lifecycle can implement the ServletContextAttributesListener interface, which has three methods:

- void attributeAdded(ServletContextAttributeEvent e): Called when an attribute is added to a servlet context
- void attributeRemoved(ServletContextAttributeEvent e): Called when an attribute is removed from a servlet context
- void attributeReplaced(ServletContextAttributeEvent e): Called when an attribute is replaced by another attribute in a servlet context

The ServletContextAttributeEvent class extends ServletContextEvent, and adds getName() and getValue() methods so the listener can learn about the attribute being changed. That is useful because Web applications that need to synchronize application state (context attributes) with something like a database can now do it in one place.

Session lifecycle: The session listener model is similar to the context listener model. In the session model, there's an **HttpSessionListener interface** with two methods:

- **void sessionCreated(HttpSessionEvent e)**: Called when a session is created
- **void sessionDestroyed(HttpSessionEvent e)**: Called when a session is destroyed (invalidated)

Appendix F : Application Lifecycle Events (contd.)

- The methods accept an HttpSessionEvent instance with a getSession() method to return the session being created or destroyed. You can use all these methods when implementing an admin interface that keeps track of all active users in a Web application.

Session Attributes lifecycle: The session model also has an HttpSessionAttributesListener interface with three methods. Those methods tell the listener when attributes change, and could be used, for example, by an application that synchronizes profile data held in sessions into a database:

- **void attributeAdded(HttpSessionBindingEvent e):** Called when an attribute is added to a session.
- **void attributeRemoved(HttpSessionBindingEvent e):** Called when an attribute is removed from a session.
- **void attributeReplaced(HttpSessionBindingEvent e):** Called when an attribute replaces another attribute in a session.

As you might expect, the HttpSessionBindingEvent class extends HttpSessionEvent and adds getName() and getValue() methods. The only somewhat abnormal thing is that the event class is named HttpSessionBindingEvent, not HttpSessionAttributeEvent. That's for legacy reasons; the API already had an HttpSessionBindingEvent class, so it was reused.

A possible practical use of lifecycle events is a shared database connection managed by a context listener. The listener is declared in the web.xml as:

This mapping was prior to Servlets 3.0:

```
<listener>
        <listener-
class>com.igate.listener.MyConnectionManager</listener-class>
</listener>
```

The server creates an instance of the listener class to receive events and uses introspection to determine what listener interface (or interfaces) the class implements. Bear in mind that because the listener is configured using annotations, you can add new listeners but there would be code change.

You could write the listener itself as something like this:

```
@WebListener
public class MyConnectionManager implements ServletContextListener {
   public void contextInitialized(ServletContextEvent e) {
   Connection con = // create connection
   e.getServletContext().setAttribute("con", con);
 }
  public void contextDestroyed(ServletContextEvent e) {
     Connection con = (Connection) e.getServletContext().getAttribute("con");
 try { con.close(); } catch (SQLException ignored) { } // close connection }t there
}
```

This listener ensures that a database connection is available in every new servlet context, and that all connections are closed when the context shuts down.

Appendix F : Application Lifecycle Events (contd.):

- The **HttpSessionActivationListener** interface is designed to handle sessions that migrate from one server to another. A listener implementing HttpSessionActivationListener is notified when any session is about to *passivate* (move) and when the session is about to *activate* (become live) on the second host. These methods give an application the chance to persist nonserializable data across JVMs, or to glue or unglue serialized objects back into some kind of object model before or after migration. The interface has two methods:
  - **void sessionWillPassivate(HttpSessionEvent e):** The session is about to passivate. The session will already be out of service when this call is made.
  - **void sessionDidActivate(HttpSessionEvent e):** The session has been activated. The session will not yet be in service when this call is made.
- You register this listener just like the others. However, unlike the others, the passivate and activate calls here will most likely occur on two different servers!
- You can monitor and react to events in a servlet's life cycle by defining listener objects whose methods get invoked when life cycle events occur. To use these listener objects you must 1) Define the listener class and 2) Specify the listener class

Defining The Listener Class

- You define a listener class as an implementation of a listener interface. The table below lists the events that can be monitored and the corresponding interface that must be implemented. When a listener method is invoked it is passed an event that contains information appropriate to the event. For example, the methods in the HttpSessionListener interface are passed an HttpSessionEvent, which contains an HttpSession.

Servlet Life Cycle Events		
Object	**Event**	**Listener Interface and Event Class**
Web context	Initialization and destruction	javax.servlet. ServletContextListener & ServletContextEvent
	Attribute added, removed, or replaced	javax.servlet. ServletContextAttributesListener & ServletContextAttributeEvent
Session	Creation, invalidation, and timeout	javax.servlet.http. HttpSessionListener & HttpSessionEvent
	Attribute added, removed, or replaced	javax.servlet.http. HttpSessionAttributesListener & HttpSessionBindingEvent

Appendix F : Application Lifecycle Events (contd.):

The SessionListener class shown in the example below implements ServletContextListener, HttpSessionAttributesListener, HttpSessionListener interfaces. This listener merely documents the occurrence of events in the application log associated with our servlet context.

```
import javax.servlet.*;
import javax.servlet.http.*;
public final class SessionListener implements ServletContextListener,
                    HttpSessionAttributesListener, HttpSessionListener {
  private ServletContext context = null;
   public void attributeAdded(HttpSessionBindingEvent event) {
        log("attributeAdded('" + event.getSession().getId() + "', '" +
                event.getName() + "', '" + event.getValue() + "')");
   }
   public void attributeRemoved(HttpSessionBindingEvent event) {
        log("attributeRemoved('" + event.getSession().getId() + "', '" +
                event.getName() + "', '" + event.getValue() + "')");
   }
   public void attributeReplaced(HttpSessionBindingEvent event) {
        log("attributeReplaced('" + event.getSession().getId() + "', '" +
                event.getName() + "', '" + event.getValue() + "')");
   }
  public void contextDestroyed(ServletContextEvent event) {
       log("contextDestroyed()"); this.context = null;
   }

 public void contextInitialized(ServletContextEvent event) {
       this.context = event.getServletContext(); log("contextInitialized()");
   }
   public void sessionCreated(HttpSessionEvent event) {
        log("sessionCreated('" + event.getSession().getId() + "')");
   }
   public void sessionDestroyed(HttpSessionEvent event) {
       log("sessionDestroyed('" + event.getSession().getId() + "')");
   }
  private void log(String message) {
     if (context != null) context.log("SessionListener: " + message);
     else System.out.println("SessionListener: " + message);
   }
   private void log(String message, Throwable throwable) {
      if (context != null)context.log("SessionListener: " + message, throwable);
      else { System.out.println("SessionListener: " + message);
             throwable.printStackTrace(System.out);
    }
   }
```

Appendix F : Application Lifecycle Events (contd.):
Add SessionListener class .

The server creates an instance of the listener class to receive events and uses introspection to determine what listener interface (or interfaces) the class implements. Invoke the SessionSnoop servlet as shown in figure A-1. The SessionListener class log information in the context log file is stored in the server's logs\client_machine\web directory. The contents of this file are shown in figure A-2.

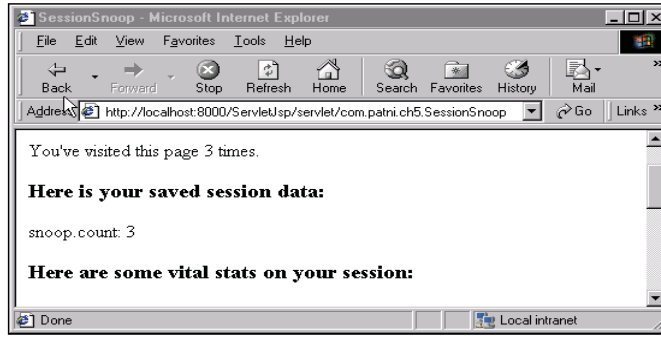


Figure A-1 Invoking SessionSnoop Servlet

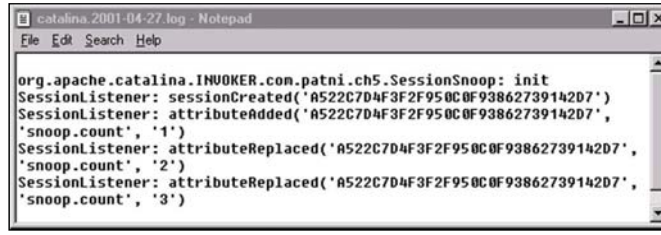


Figure A-2 Contents of log file after invoking SessionSnoop

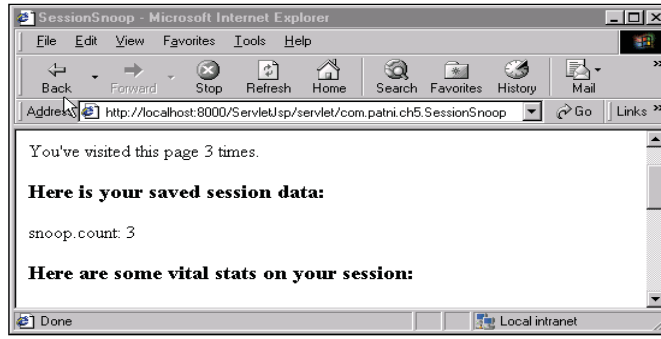


Figure A-1 Invoking SessionSnoop Servlet

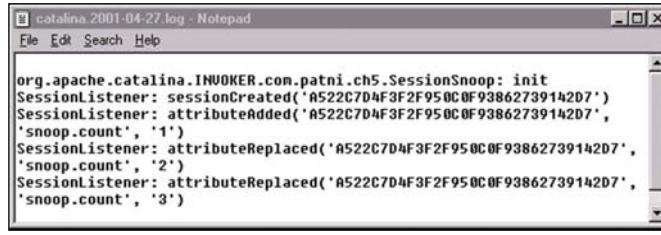


Figure A-2 Contents of log file after invoking SessionSnoop

Appendix G : Asynchronous Servlet:

Asynchronous servlet:

Using Asynchronous processing avoids blocking requests by allowing the thread to perform some other operation, meanwhile the input is returned to the client.

Asynchronous Servlet Implementation

First of all the servlet where we want to provide async support should have @WebServlet annotation with asyncSupported value as true.

Since the actual work is to be delegated to another thread, we should have a thread pool implementation. We can create thread pool using Executors framework and use servlet context listener to initiate the thread pool.

We need to get instance of AsyncContext through ServletRequest.startAsync() method. AsyncContext provides methods to get the ServletRequest and ServletResponse object references. It also provides method to forward the request to another resource using dispatch() method.

We should have a Runnable Implementation where we will do the heavy processing and then use AsyncContext object to either dispatch the request to another resource or write response using ServletResponse object. Once the processing is finished, we should call AsyncContext.complete() method to let container know that async processing is finished.

We can add AsyncListener implementation to the AsyncContext object to implement callback methods – we can use this to provide error response to client incase of error or timeout while async thread processing. We can also do some cleanup activity here.

Consider servlet with following code:

```
protected void doGet(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException {
    long startTime = System.currentTimeMillis();
    System.out.println("LongRunningServlet Start::Name="
        + Thread.currentThread().getName() + "::ID="
        + Thread.currentThread().getId());
```

```
String time = request.getParameter("time");
      int secs = Integer.valueOf(time);
      // max 10 seconds
      if (secs > 10000)
         secs = 10000;

      longProcessing(secs);

      PrintWriter out = response.getWriter();
      long endTime = System.currentTimeMillis();
      out.write("Processing done for " + secs + " milliseconds!!");
      System.out.println("LongRunningServlet Start::Name="
            + Thread.currentThread().getName() + "::ID="
            + Thread.currentThread().getId() + "::Time Taken="
            + (endTime - startTime) + " ms.");
   }

   private void longProcessing(int secs) {
      // wait for given time before finishing
      try {
         Thread.sleep(secs);
      } catch (InterruptedException e) {
         e.printStackTrace();
      }
   }
```

If we hit above servlet through browser with URL as http://localhost:9090/AsyncServletExample/LongRunningServlet?time=8000, we get response as “Processing done for 8000 milliseconds!!” after 8 seconds. Now if you will look into server logs, you will get following log:

LongRunningServlet Start::Name=http-bio-8080-exec-34::ID=103
LongRunningServlet Start::Name=http-bio-8080-exec-34::ID=103::Time Taken=8002 ms.

This can leads to Thread Starvation – Since our servlet thread is blocked until all the processing is done, if server gets a lot of requests to process, it will hit the maximum servlet thread limit and further requests will get Connection Refused errors.

Prior to Servlet 3.0, there were container specific solution for these long running threads where we can spawn a separate worker thread to do the heavy task and then return the response to client. The servlet thread returns to the servlet pool after starting the worker thread. Tomcat’s Comet, WebLogic’s FutureResponseServlet and WebSphere’s Asynchronous Request Dispatcher are some of the example of implementation of asynchronous processing.
The problem with container specific solution is that we can’t move to other servlet container without changing our application code, that’s why Async Servlet support was added in Servlet 3.0 to provide standard way for asynchronous processing in servlets.

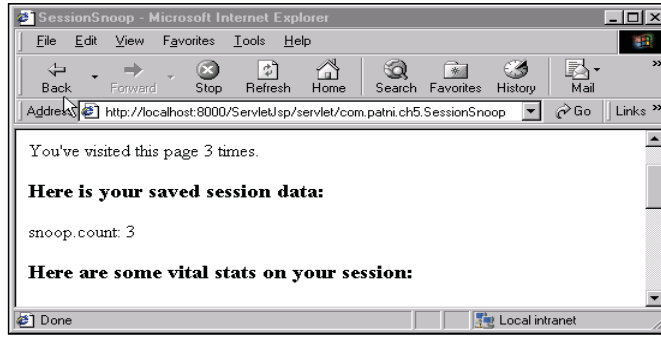


Figure A-1 Invoking SessionSnoop Servlet

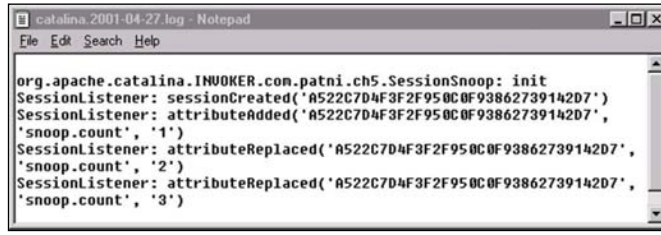


Figure A-2 Contents of log file after invoking SessionSnoop

Appendix H : SSL and WildFly:

Application security can be applied in three areas:-

Authentication – A process by which system verifies the identity of the user
Authorization – A process of verifying that the user has the required credentials to access the applications resources
Secure Communication- To ensure Confidentiality, Data integrity and Source Integrity of the Data. The Secure Protocols are:-

Transport Layer Security (TLS)
Secure Socket Layer (SSL)

Understanding Application Security:

Security is an important aspect of most enterprise applications because loss of sensitive data and system susceptibilities can be costly. Unauthorized users may access application data or someone may intercept a message being transmitted or hack into your application and run malicious code, as we see Security can be compromised in multiple ways.

Application Security can be applied in three areas. Let us see each one.

Authentication: It is a process by which system verifies the identity of the user.
Authorization: It is a process of verifying that the user has the required credentials/privileges to access applications resources.

To understand this let us consider an simple example, in an application the user is prompted to enter username & password. This username & password is verified against some predefined values either in the database or some file. This is termed as authenticating the user. Once it is determined that user is valid, the application can then check for privileges that user has based on components requested by the user.

Apart from these two methods application security can also be applied by secure communication.

Secure Communication: Data is often sent over open network. In such cases we should be aware of confidentiality, data integrity and source integrity.

Confidentiality protects a message from being read by any third person other then the intended recipient. Often the message in encrypted so that the message contents are not manipulated, this is Data Integrity. Source integrity guarantees the identity of the sender, which means the message was indeed sent by the sender.

Source integrity can be protected by using a reliable certificate issuing authority. A person who wants to be trusted by other people can obtain a certificate and send it to other parties who can verify the certificate and send it to other parties who can verify the certificate owner's identity by asking the trusted third party which is normally the certificate issuing authority

Secure protocols such as Transport Layer Security (TLS) or Secure Socket Layer (SSL) can be used to provide the three characteristics of secure communication. These protocols use a combination of public and private key cryptography and digital certificates.

Appendix H : SSL and WildFly:

Configuring Security in WildFly:
Application Server's security implementation is called as JBoss SX
It is built on top of the JAAS (Java Authentication and Authorization Service).
Each JEE component or resource has a different mechanism for defining security. What method to be used and what should be secured is defined in each component's standard deployment descriptor.
JEE specification does not specify the security implementation, also it does not describe where the security data should be kept or how it should be retrieved or how the validation should happen. Every application server vendor has to create its own security implementation and allow programmers to configure and use it through vendor-specific deployment descriptors.
When deploying applications on WildFly most of the time it is likely that we would be deploying a web application and just require a security realm to be defined with certificates
The security subsystem is enabled by default by the addition of the following extension: -
<extension module="org.jboss.as.security"/>
The namespace used for the configuration of the security subsystem is urn:jboss:domain:security:1.0, the configuration is defined within the <subsystem> element from this namespace.

Within WildFly 8 we make use of security realms to secure access to the management interfaces, these same realms are used to secure inbound access as exposed by JBoss Remoting such as remote JNDI , the realms are also used to define an identity for the server - this identity can be used for both inbound connections to the server and outbound connections being established by the server.

The general structure of a management realm definition is: -

```
<security-realm name="ManagementRealm">
  <plug-ins></plug-ins>
  <server-identities></server-identities>
  <authentication></authentication>
  <authorization></authorization>
</security-realm>
```

plug-ins - This is an optional element that is used to define modules what will be searched for security realm Plug-In Providers to extend the capabilities of the security realms.
server-identities - An optional element to define the identity of the server as visible to the outside world, this applies to both inbound connection to a resource secured by the realm and to outbound connections also associated with the realm.
authentication - This is probably the most important element that will be used within a security realm definition and mostly applies to inbound connections to the server, this element defines which backing stores will be used to provide the verification of the inbound connection.
authorization - This is the final optional element and is used to define how roles are loaded for an authenticated identity. At the moment this is more applicable for realms used for access to EE deployments such as web applications but this will also become relevant as we add role based authorization checks to the management model.

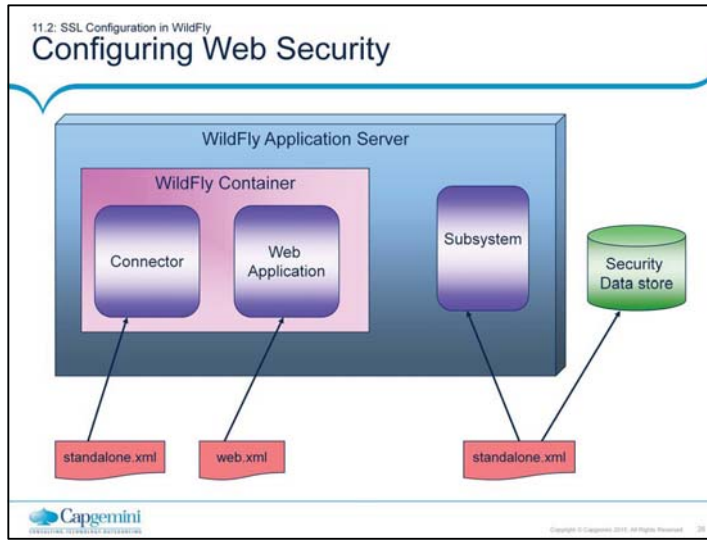
Consider a sample mapping of the standalone.xml file:

```
<security-realm name="UndertowRealm">
                <server-identities>
                <ssl>
                                <keystore path="clientkeystore.jks" relative-
to="jboss.server.config.dir" keystore-password=“igate123"/>
                </ssl>
                </server-identities>
</security-realm>
```

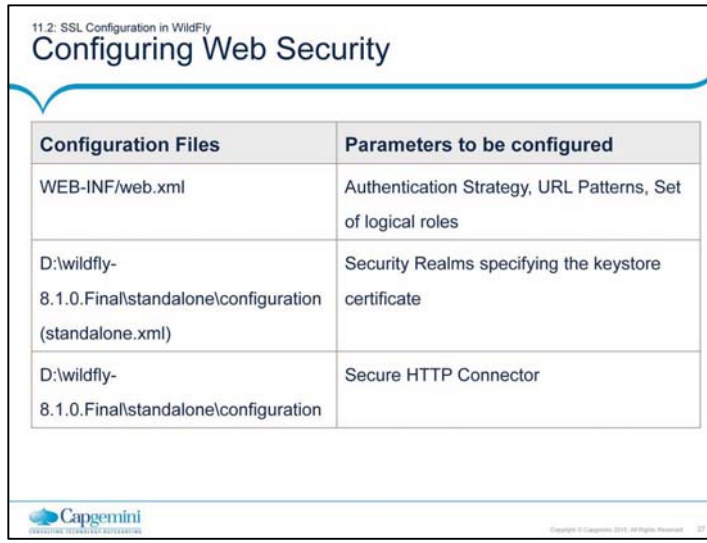
We need to generate a keystore certificate. Save the certificate with .jks extension – Java Key store. Generating certificates would be a part of Module 2

Need to save keystore in D:\wildfly-8.1.0.Final\standalone\configuration folder

The password while generating the keystore for the above certificate is “igate123”; The password could be changed.
Configuring Security for Web Applications is quite simple. Simply configure the SSL through the HTTP connector. But for authenticating clients based on certificate information we need to define SSL aware sub systems



SSL Configuration in WildFly:
Configuring Web Security:
By default, web applications are not secured. The URL of your web application will be accessible publicly.
The WildFly Server is also not secure by default since the Secure HTTP connector is not enabled. The figure on the slide shows which configuration files are used to configure security within the server and for applications. Every file include here has a specific purpose.



SSL Configuration in WildFly:
Configuring Web Security:
The table on the slide mentions about the primary configuration files which are required to configure security for your web applications.

web.xml: This is the deployment descriptor for your web application. This file configures application security by defining the authentication strategy (BASIC, FORM, DIGEST, CLIENT-CERT). Specifies which URLs of the application are secured and any logical roles

standalone.xml: This is the configuration file for WildFly Server and is used to define the secure HTTP Connector and set the security certificate generated Also we need to set the HTTP port to 8443. See below listing to update the change

```
<socket-binding-group name="standard-sockets" default-
interface="public" port-offset="${jboss.socket.binding.port-offset:0}">
        <socket-binding name="https"
        port="${jboss.https.port:8443}"/>
</socket-binding-group>
```

Appendix I : Servlet Security:

There are several ways in which you can secure web applications. These include the following options:
You can define a user authentication method for an application in its deployment descriptor. Authentication verifies the identity of a user, device, or other entity in a computer system, usually as a prerequisite to allowing access to resources in a system. When a user authentication method is specified for an application, the web container activates the specified authentication mechanism when you attempt to access a protected resource.
The options for user authentication methods are discussed in Specifying an Authentication mechanism. All of the example security applications use a user authentication method.
You can define a transport guarantee for an application in its deployment descriptor. Use this method to run over an SSL-protected session and ensure that all message content is protected for confidentiality or integrity. The options for transport guarantees are discussed in Specifying a secure connection.
When running over an SSL-protected session, the server and client can authenticate one another and negotiate an encryption algorithm and cryptographic keys before the application protocol transmits or receives its first byte of data.
SSL technology allows web browsers and web servers to communicate over a secure connection. In this secure connection, the data is encrypted before being sent, and then is decrypted upon receipt and before processing. Both the browser and the server encrypt all traffic before sending any data. For more information, see Establishing secure connection using SSL.
Digital certificates are necessary when running HTTP over SSL (HTTPS). The HTTPS service of most web servers will not run unless a digital certificate has been installed. Digital certificates have already been created for the Application Server.

We shall see how to secure the users with specific roles

@ServletSecurity

The @ServletSecurity annotation provides an alternative mechanism for defining access control constraints equivalent to those that could otherwise have been expressed declaratively via security-constraint elements in the portable deployment descriptor.

Using the @ServletSecurity annotation one can now have a descriptor free secure WebApplication.

Prior to Servlet 3.0

Web.xml

```
<security-constraint>

<web-resource-collection>

<url-pattern>/TutorialServletBASIC</url-pattern>

</web-resource-collection>

<auth-constraint>

<security-role-name>TutorialUser</security-role-name>

<security-role-name>guest</security-role-name>

</auth-constraint>

</security-constraint>
```

Now:

```
//for all HTTP methods, auth-constraint requiring
membership in Role TutorialUser or guest

@ServletSecurity(@HttpConstraint(rolesAllowed =
{“MyUser", "guest"}))
//for particular methods:
        @RolesAllowed("TutorialUser")

   public rettype meth() { … }
```

Appendix J : Web Fragment

Pluggability

Enable use of 3rd party framework without boiler plate configuration in deployment descriptors. Modularize web.xml to allow frameworks to be self-contained within their own JAR file and Programmatic configuration APIs Use of annotations.

The motivation behind pluggability is to decrease the complications of putting all configuration in one single sile (web.xml) if a specific framework is needed. For example, if a developer needs Spring support for an existing Web Application – the 1st thing to do is put in the servlet/ listener for Spring as part of it support – it can get ugly when in the future, a new EE technology arises as you need to put everything again on the web.xml file.

Now with JEE7 – you can create a sub-project with a web-fragment.xml that will mimic a section on the main web.xml – this allows further improvements (or pluggable functional requirements) to be independently created and isolated. Below is an example of a web-fragment.xml

```
<web-fragment>
  <servlet>
    <servlet-name>welcome</servlet-name>
    <servlet-class>com.mycom.WelcomeServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>welcome</servlet-name>
      <url-pattern>/Welcome</url-pattern>
  </servlet-mapping>
  ...
</web-fragment>
```

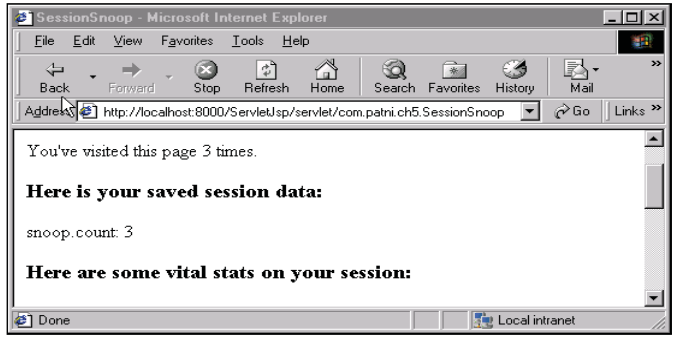


Figure A-1 Invoking SessionSnoop Servlet

Figure A-2 Contents of log file after invoking SessionSnoop

Java Servlets 3.0
Lab Book

Document Revision History

Date	Revision No.	Author	Summary of Changes
May 2015	4.0	Yukti A Valecha	Revamped from Servlets 2.5 to Servlets 3.0
May 2016	4.1	Yukti A Valecha /Anjulata	Revamped as per revised course contents

Table of Contents

Getting Started

Overview

This lab book is a guided tour for learning Servlets 3.0. It comprises scenario based applications and 'To Do' assignments. Flow diagrams and screen snap shots are provided where necessary.

Setup Checklist for Servlets 3.0

Here is what is expected on your machine in order for the lab to work.

Minimum System Requirements

- Intel Pentium 90 or higher (P166 recommended)
- Microsoft Windows 95, 98, or NT 4.0, 2k, XP.
- Memory: 32MB of RAM (64MB or more recommended)
- Java SE version 8
- Internet Explorer 6.0 or higher
- Connectivity to Oracle database
- WildFly server Version 8.1.0

Please ensure that the following is done:

- Eclipse_JEE_Luna (4.x) is installed.

Instructions

- All lab assignments should refer coding standards.
- Create a directory by your name in drive <drive>. In this directory, create a subdirectory servlet_assgn. For each lab exercise create a directory as lab <lab number>.
  You may also look up the on-line help provided in JEE documentation

Lab 1. Servlet Basics

Goals	• Understanding Servlet Basics and Invocation
Time	0.5 Hour

1.1 Create a Servlet program that will print the system date and time.

Note: The Servlet should be invoked in the following ways**:**

- On clicking of Hyper Link in html page
- On clicking of button in html page
- On typing the servlet URL pattern directly in Address bar of browser

Lab 2. Request and Response Objects

Goals	• Understanding the request and response objects
Time	2 Hours

2.1: Design a login Page accepting username and password in HTML. The credentials need to be authenticated in Servlet.
The credentials could be validated by using hard coded values for username and password.
If user is valid then display HTML page as “Success” and display HTML page as “Failure” if user is invalid.

Note:

- Checking could be done by using private method inside servlet.
- The redirection to HTML pages is done using response.sendRedirect(String URLPattern) method

Refer below figure for HTML Login Page:

Figure 1

In extension to the previous assignment print the following details extracted from the ‘Request’ object….

1. MIME type accepted by the client
2. The locale setting
3. Data transferred in bytes

2.2: Consider a webpage which is displaying live stock market status. For such type of page, you would need to refresh your web page regularly using refresh or reload button with your browser.
Java Servlet makes this job easy by providing you a mechanism where you can make a webpage in such a way that it would refresh automatically after a given interval.
Consider the servlet code snippet below and complete the blank space left for automatic refresh to happen and auto-load page after 5 seconds.

Refer below figure:


```
package com.cg.lab2.controller;
import java.io.IOException;
/**
 * Servlet implementation class Refresh
 */
@WebServlet("/Refresh")
public class Refresh extends HttpServlet {
    protected void doGet(HttpServletRequest request,
            HttpServletResponse response) throws ServletException, IOException {
        // TODO Auto-generated method stub
        // Set refresh, autoload time as 5 seconds

    }

}
```

Figure 2

Create a servlet that will redirect the current web application to ISPACE.

See below figure for same:

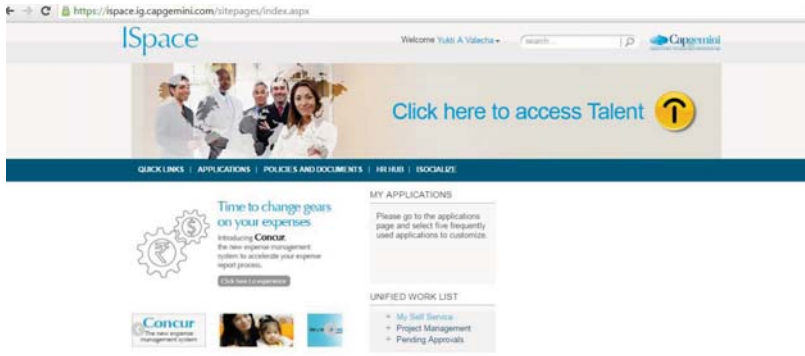


Figure 3

Lab 3. Database Connectivity

Goals	• Creating DB connection pool with WildFly
Time	2 Hours

3.1: Steps to configure Database connection pool in WildFly:

Need to add Management User, Run D:\wildfly-8.1.0.Final\bin>add-user.bat
Create a Management User. Select Option (a)

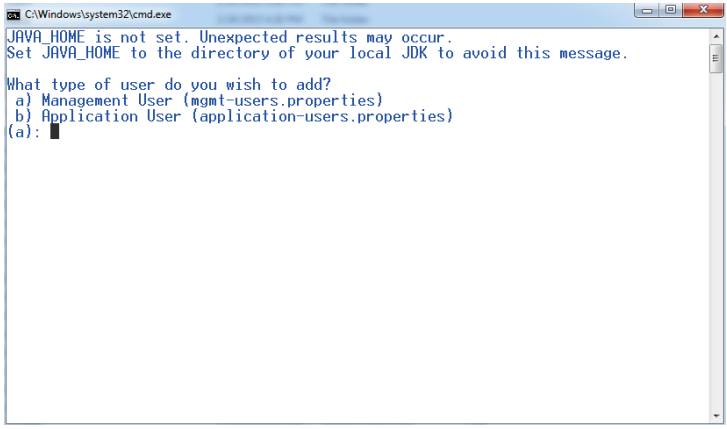


Figure 4

Enter Username / Password

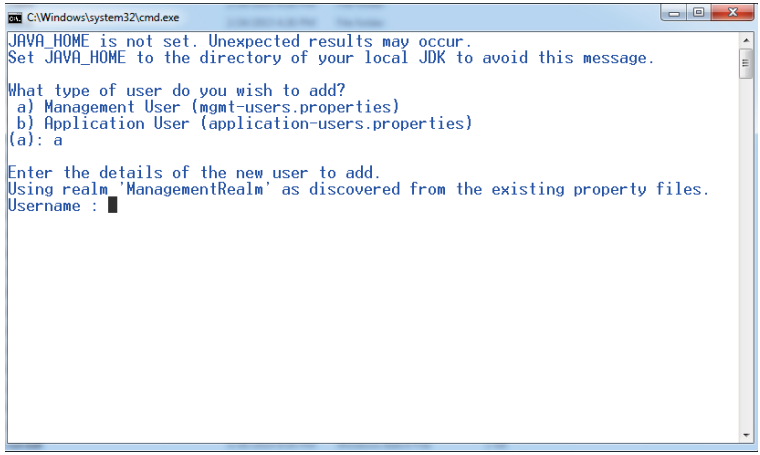


Figure 5

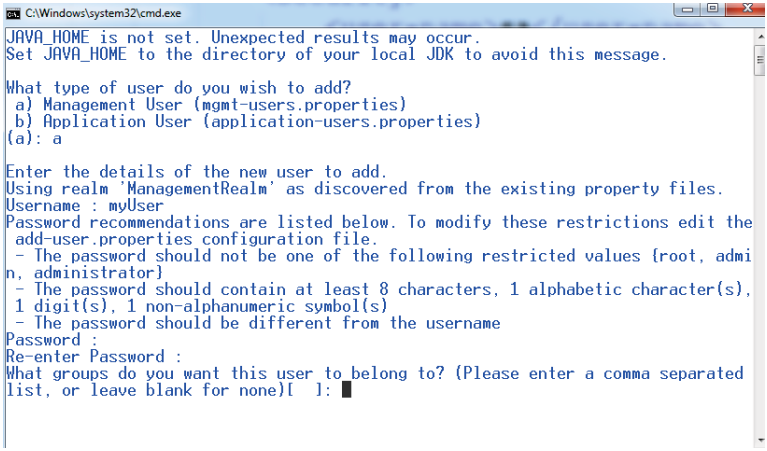


Figure 6

What Group this User should belong to – Leave as blank, Add the user to Management Realm with Option Yes.

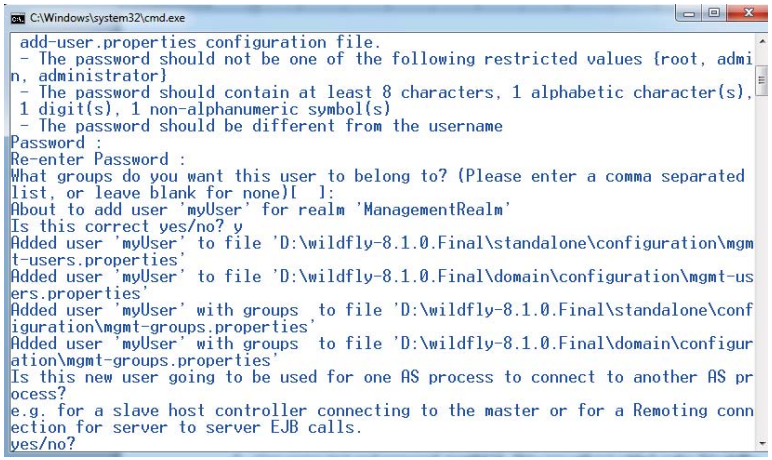


Figure 7

No need to add the User for server EJB Calls.

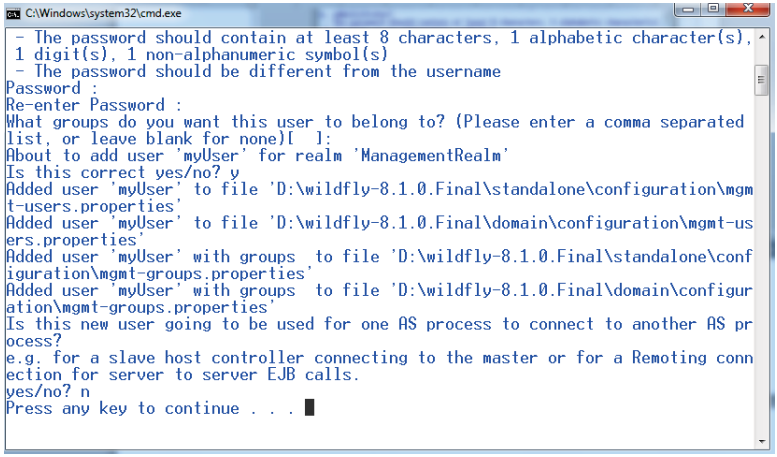


Figure 8

User name: myUser and password: myUser@12.This user will get added under D:\wildfly-8.1.0.Final\standalone\configuration\mgmt-users.properties

Start the WildFly server.
Open the link: http://localhost:9090/console/.Enter the credentials to log in (Which you have created under Management Realm).

Note: Default port number is 8080. But due to port number conflict, it has been changed to 9090.

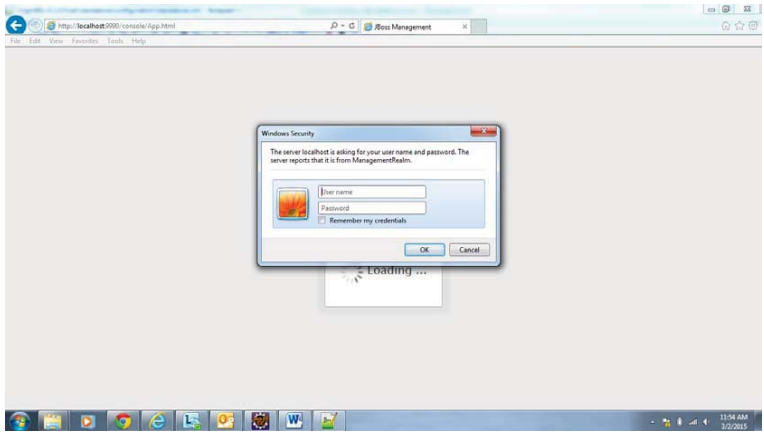


Figure 9

Following is figure of Home Page

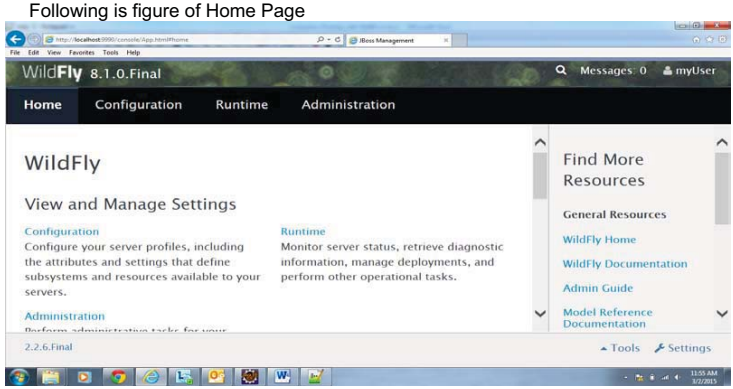


Figure 10

Before creating a datasource we need to deploy the JDBC driver class file to use for connecting to DB.

a. Thus we need to use ojdbc6_g.jar (which is JDBC 4 complaint)
b. Any JDBC4-compliant driver is automatically recognized by WildFly and made available for new datasources.
c. Go to Runtime > Server > Manage Deployments and click on Add to deploy the Oracle complaint Jar file

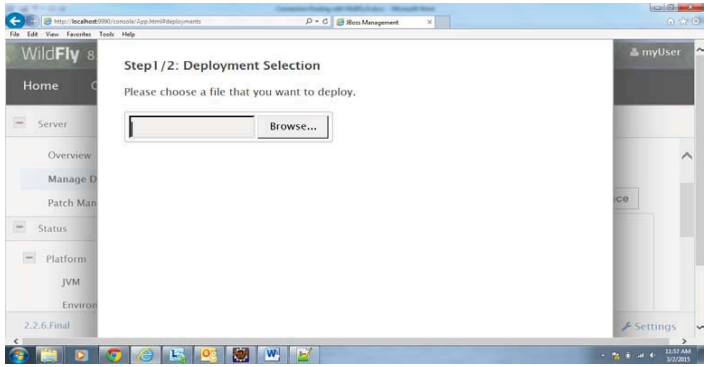


Figure 11

Give the path from your local machine (where ojdbc6_g.jar) is present

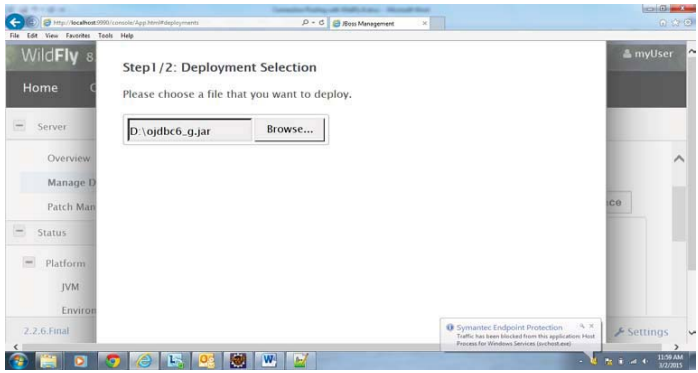


Figure 12

d. Click Next. Supply the Name and Runtime Name with which you want to identify Oracle jar file

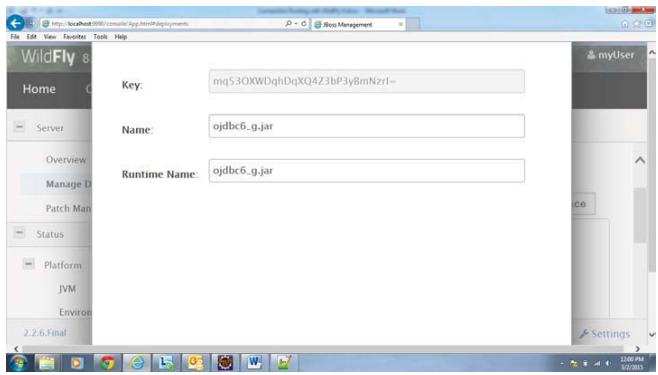


Figure 13

Click on Save

e. After Oracle jar file is deployed, click on Enable/Disable to make it available for Datasources

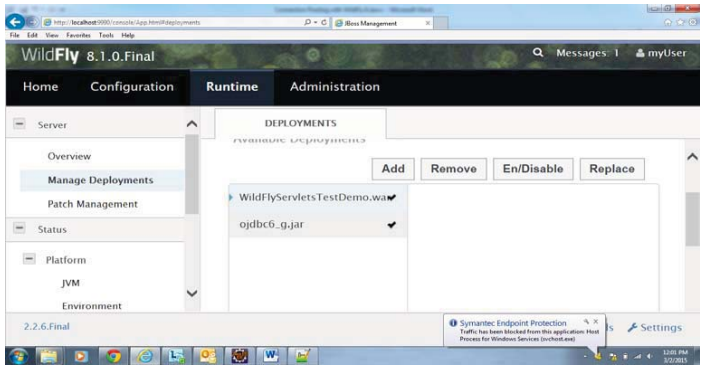


Figure 14

Go to Home > Create a datasource > Datasources. Click on Add to add a Datasource

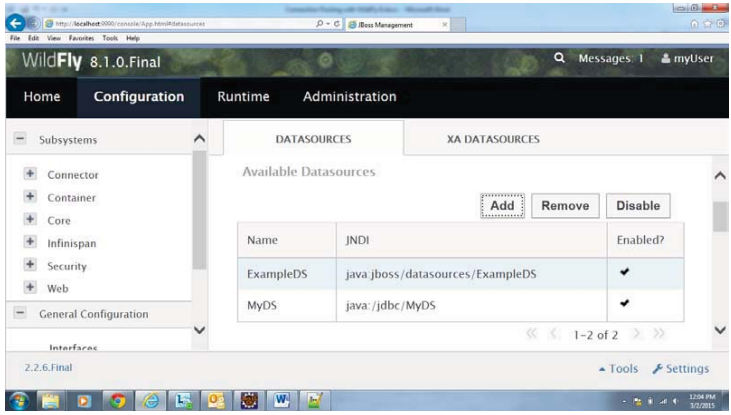


Figure 15

Supply Connection Pool Name (example: MyDS): as well as JNDI Name: (example: java:/jdbc/MyDS)

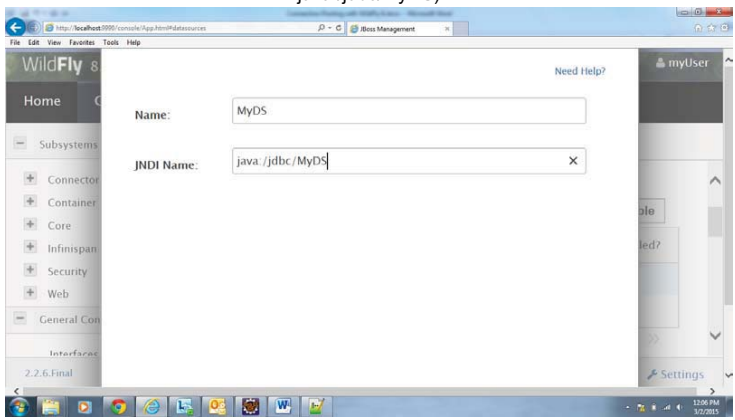


Figure 16

Click Next > Select the Detected Driver > ojdbc6_g.jar

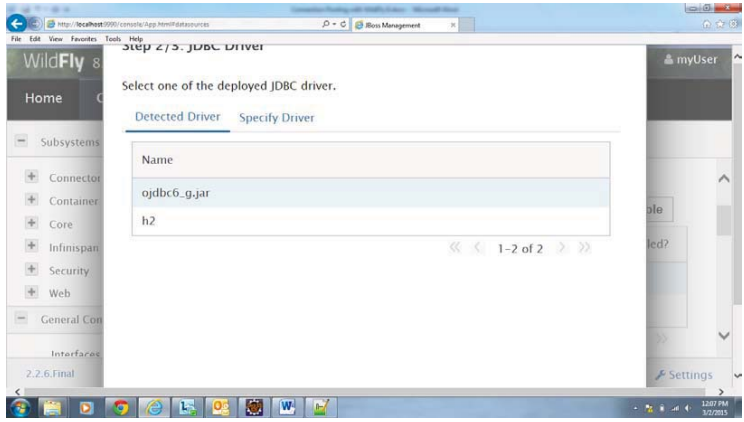


Figure 17

Click Next > Supply Connection URL (jdbc:oracle:thin:@localhost:1522:XE), Username (system), Password (yukti) . These details are with respect to Oracle 11g installed on my machine. These details have to be filled in with respect to the Database you want to connect to.

Figure 18

Click on Test Connection. It should show Connection tested successfully

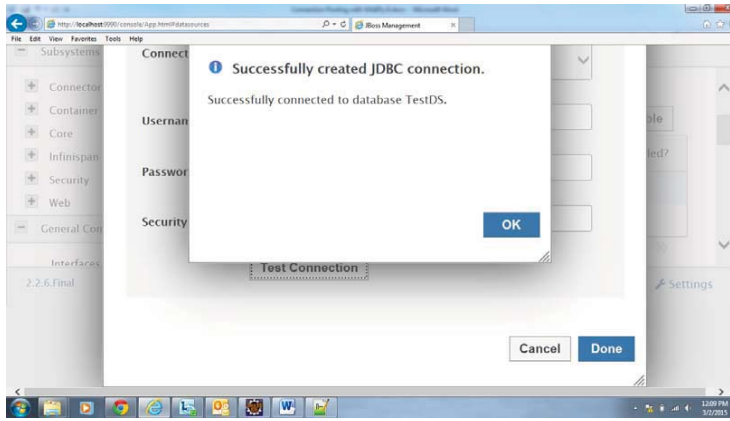


Figure 19

Click on Done to create Datasource.
Select the Datasource and Click on Enable button to make it available for Connection Pooling.

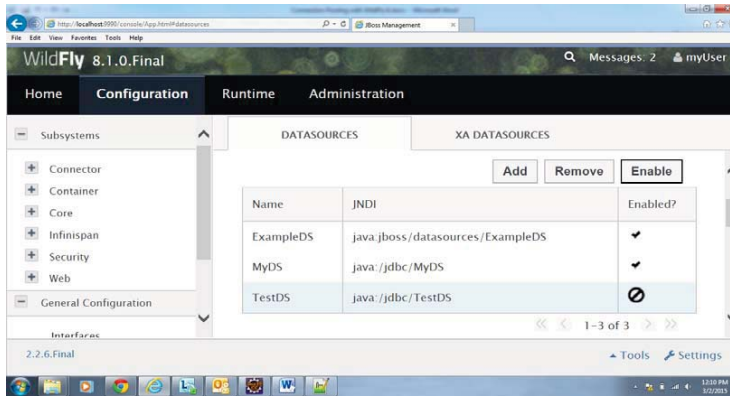


Figure 20

Click on Confirm to Enable Datasource.

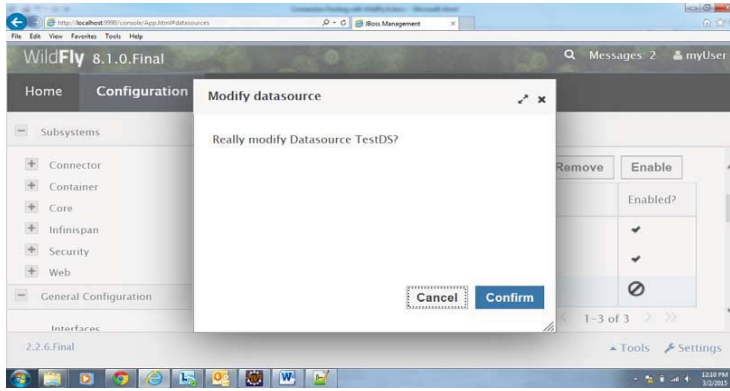


Figure 21

A check mark now appears on Enabled Datasource

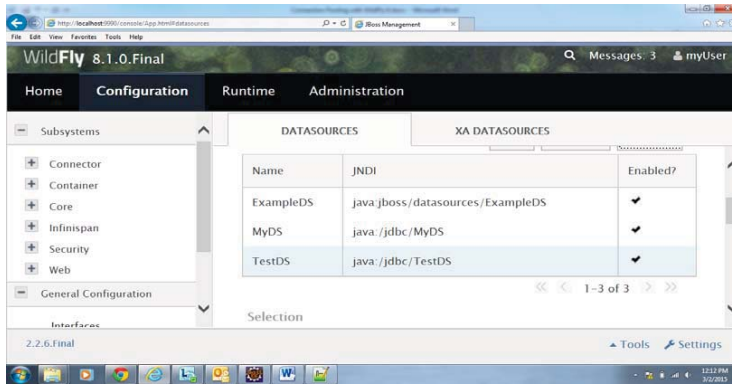


Figure 22

It should now be available on the page of JDBC Datasources
These Datasource mappings would be reflected in standalone.xml file (D:\wildfly-8.1.0.Final\standalone\configuration\standalone.xml). Check Datasource tag in this file.

Write Servlet code to test the connection Pool.

3.2: Design a registration page in HTML. Refer below figure.

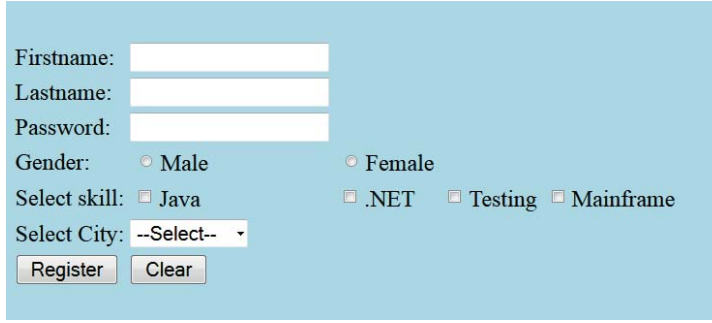


Figure 23

All the details entered in HTML page should be accepted by the Servlet.
The Servlet should connect via Connection Pool to database and store the user registration details in database table.

After the user details are inserted successfully in table (RegisteredUsers), display the message "Registration done!!!" in HTML page . Display "Registration failed!!!" in case of any exception .

Note:

- The redirection to HTML page is done using response.sendRedirect (String URLPattern) method
- The application needs to be developed using Layered architecture approach
- User can select more than one skillset, values of which should be concatenated and stored in skillset column

Database Table used: RegisteredUsers

CREATE TABLE RegisteredUsers (firstname VARCHAR (20), lastname VARCHAR (30), password VARCHAR (12) UNIQUE, gender CHAR, skillset VARCHAR (40), city VARCHAR (12));

Refer below figure for layered architecture:

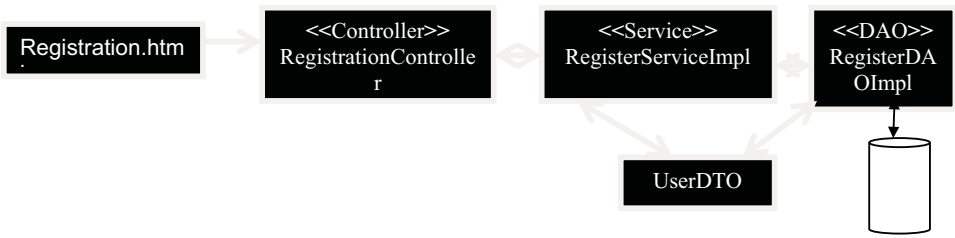


Figure 24

Lab 4. Inter-Servlet Communication

Goals	• Understanding Inter-Servlet Communication and data transfer between servlets
Time	1 Hour

4.1: Modify the assignment in lab 3.2 and print all the users' registered details on next page.

Note:

- Make use of RequestDispatcher.forward(String URLPattern) method to navigate to next page
- To store all the users' details make use of Request Scope

Lab 5. Session Management

Goals	• Understanding Session Management
Time	3 Hours

5.1 Rima is working in State electricity board project (eBill application) and is being given the requirement to accept the details online and calculate the net electricity bill amount and persist the details in database.

Refer the below script:

```
CREATE TABLE Consumers(
        consumer_num NUMBER(6) PRIMARY KEY,
        consumer_name VARCHAR2(20) NOT NULL,
        address VARCHAR2(30)
);

INSERT INTO Consumers VALUES(100001,'Sumeet','Shivaji Nagar, Pune');
INSERT INTO Consumers VALUES(100002,'Meenal','M G Colony Panvel,
Mumbai');
INSERT INTO Consumers VALUES(100003,'Neeraj','Whitefield, Bangalore');
INSERT INTO Consumers VALUES(100004,'Arul','Karapakkam, Chennai');

CREATE TABLE BillDetails(
        bill_num NUMBER(6) PRIMARY KEY,
        consumer_num NUMBER(6) REFERENCES Consumers(consumer_num),
        cur_reading NUMBER(5,2),
        unitConsumed NUMBER(5,2),
        netAmount NUMBER(5,2),
        bill_date DATE DEFAULT SYSDATE);

CREATE SEQUENCE seq_bill_num START WITH 100;
```


After login the admin user should be able to view the following UI:

[Note : Use login functionality implemented in Lab 2.1]

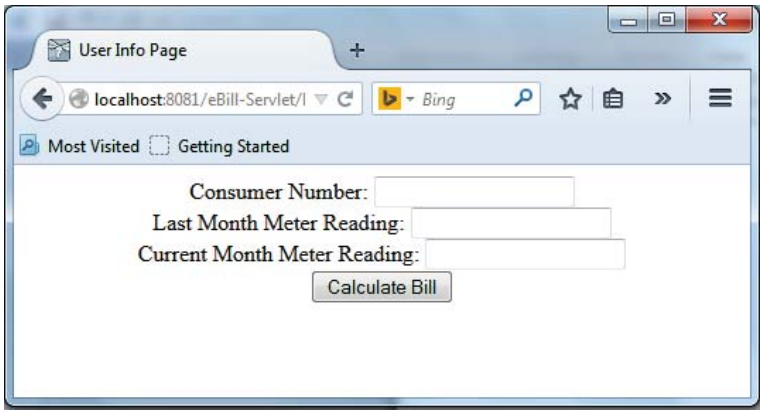


Figure 25

Following Client side validations need to be performed:

- All fields are mandatory
- Consumer number can contain only numbers i.e. 6 digit , Consumer number cannot start with 0
- Last Month and Current Month meter reading can take decimal values, maximum 2 digits after decimal point and should not be negative
- Current Month meter reading cannot be less than Last Month meter reading

Refer below figure with valid details:

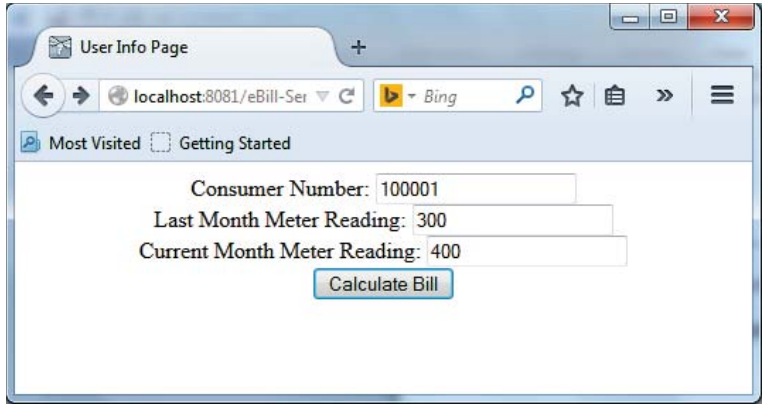


Figure 26

After user enters the valid details Units consumed and Net bill amount need to be calculated in servlet:

Refer below calculations:

Units consumed = (Last month meter reading) – (Current month meter reading)
Net Amount = Unit consumed * 1.15 + Fixed Charge

Assume Fixed Charge is always Rs.100.

After calculating the electricity bill, bill details needs to be inserted into the database table **billdetails.**

Note:
1. Bill_id should be auto generated by sequence.
2. Bill_date should be current date.

Print the following details for the Consumer. Refer below figure.

Figure 27

Following figure displayed the error in case of invalid consumer number entered

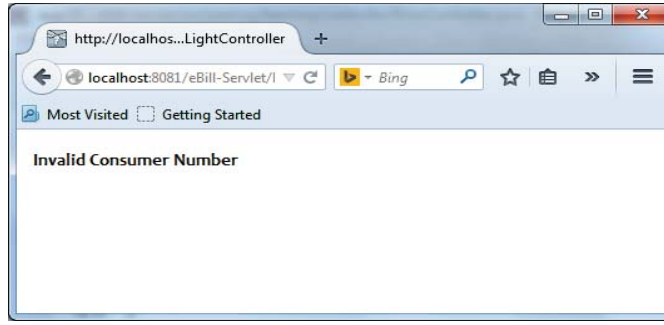


Figure 28

Layered architecture to be followed:

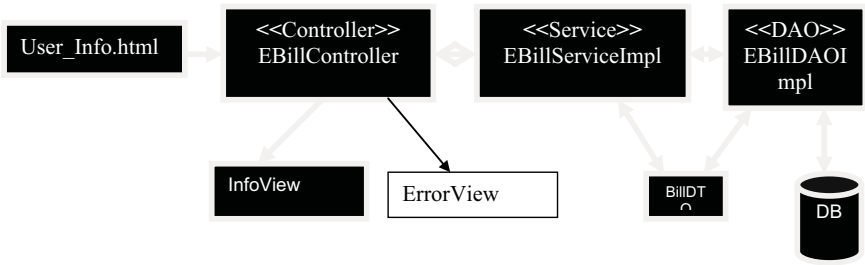


Figure 29

Lab 6. Filters

Goals	• Understanding the use of filters
Time	0.5 Hour

6.1: Consider below filter code snippet, many servlet requests are passing through this Filter. If servlets are invoked then predict the output.

```
public void doFilter(ServletRequest request, ServletResponse response,
      FilterChain chain) throws IOException, ServletException {
   |
   long before = System.currentTimeMillis();
   chain.doFilter(request, response);
   long after = System.currentTimeMillis();
   String name = "";
   if (request instanceof HttpServletRequest) {
     name = ((HttpServletRequest)request).getRequestURI();
   }
   config.getServletContext().log(name + ": " + (after - before) + "ms");
}
```

Figure 30

Additional Exercise: <<To Do>>

Understanding Session Management with Cookies:

Implementing a **Remember Me** assignment with **cookies**. A user wants to login to an application, but first is being presented with a welcome page. The welcome page has a “remember me” check box. When user clicks on checkbox and says “remember me” the next time when he access the application he / she should be shown the Login Page directly and not Welcome Page. If the user does not click on “remember me” checkbox then next time when he access the application he / she should be shown the Welcome Page and on click of contine button display the Login Page.

See below code snippets and fill in the blank spaces left with cookie / servlet related code.

Welcome.html looks as below

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4  <meta charset="ISO-8859-1">
5  <title>Welcome Page</title>
6  </head>
7  <body bgcolor="gold" style="text-align: center">
8  <form action="NextPage" method="get">
9     <input type="submit" value="Continue"/><BR>
10    <input type="checkbox" name="remember"/>Don't show this page again
11 </form>
12 </body>
13 </html>
```

Figure 31

Note: This servlet has to be the first page to execute to check for the availability of cookies. CheckServlet.java code snippet as below: Here from doGet (request, response) method doPost (request, response) is invoked

```
protected void doPost(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException {
    // code to obtain the existing cookies

    //checking if cookies are present)
    {
        // if cookies - not present then redirecting to welcome page
        response.sendRedirect("welcome.html");
    }
    // if cookies present, then
        // code to iterate over cookies

        // code to obtain the cookie name and value pair

        // code to check if cookie name = remember and cookie value is yes i.e check box checked ir

        // if cookie value is remember me then code to redirect to login page
```

Figure 32

Refer NextPage Servlet below to create a cookie.
Here from doPost (request, response) method doGet (request, response) is invoked.

```
protected void doGet(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException {
    // checking if check box is selected
    String status = request.getParameter("remember");
    if (status != null) {
        // code to create a cookie, and setting age of cookie to 160 seconds
        // and adding cookie to response



    }
    // redirecting to login page
    response.sendRedirect("login.html");
}
```

Figure 33

Below is login.html page.

```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body bgcolor="gold" style="text-align: center">
<form>
User Name: <input type="text" name="name"/><BR>
Password: <input type="password" name="pwd" /><BR>
<input type="submit" value="Login"/>
</form>
</body>
</html>
```

Figure 34

Appendix A: Table of Figures

