Enterprise Web Applications

Preface:

This will be the fifth entry in the Java collection of notes. Up until now, we have learned a lot about Java Standard Edition (SE). You are likely aware that another version of Java exists, known as Java Enterprise Edition (EE). It is the latter that we will be directing our focus on in this document. The licensing for Java EE has changed, thus it is now refered to as Jakarta EE and is maintained by the Eclipse foundation (the creators and maintainers of Eclipse IDE). Jakarta EE provides the tools and resources that are necessary for large-scale web-based development. Java SE is good for personal projects or isolated applications, but does not provide the best way of interacting with the web. In this document, we will learn more about server-side Java (Jakarta EE), Netbeans IDE (even though I hate NetBeans), Three Tiered Architecture, Persistence, the RESTful API, Java Server Faces Presentation Layer, Securing with Authentication/Authorization, and Testing.

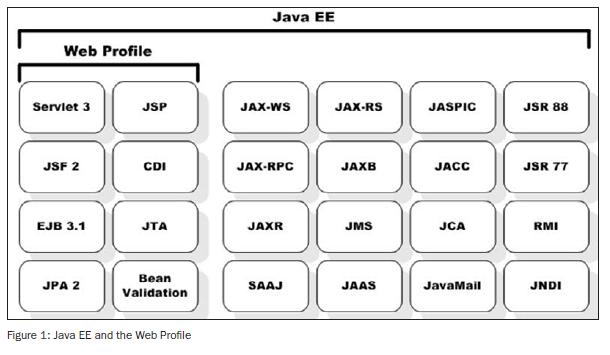
Intro:

Thus far, we have put a lot of focus on smaller, more individual projects to get a sense of the syntax and concepts of programming. In real life, however, whether we like it or not, most of the code that we interact with uses what is known as Enterprise Architecture. Fundamentally, this is where we have a client/server (C/S) architecture where the server hosts the application and the client is able to access it. A database is usually connected at the back-end, and the server provides a front end to its clients (the end users) upon request (e.g. a web page that uses HTML and CSS files to display content). In reality, the architecture is often more complex than just that. We may have multiple servers that provide different functionality. For example, there can be a client layer, security layer, web services layer, content management layer, business logic later, and data storage layer. The nice thing about JakartaEE is that it provides a lot of this layers for us.

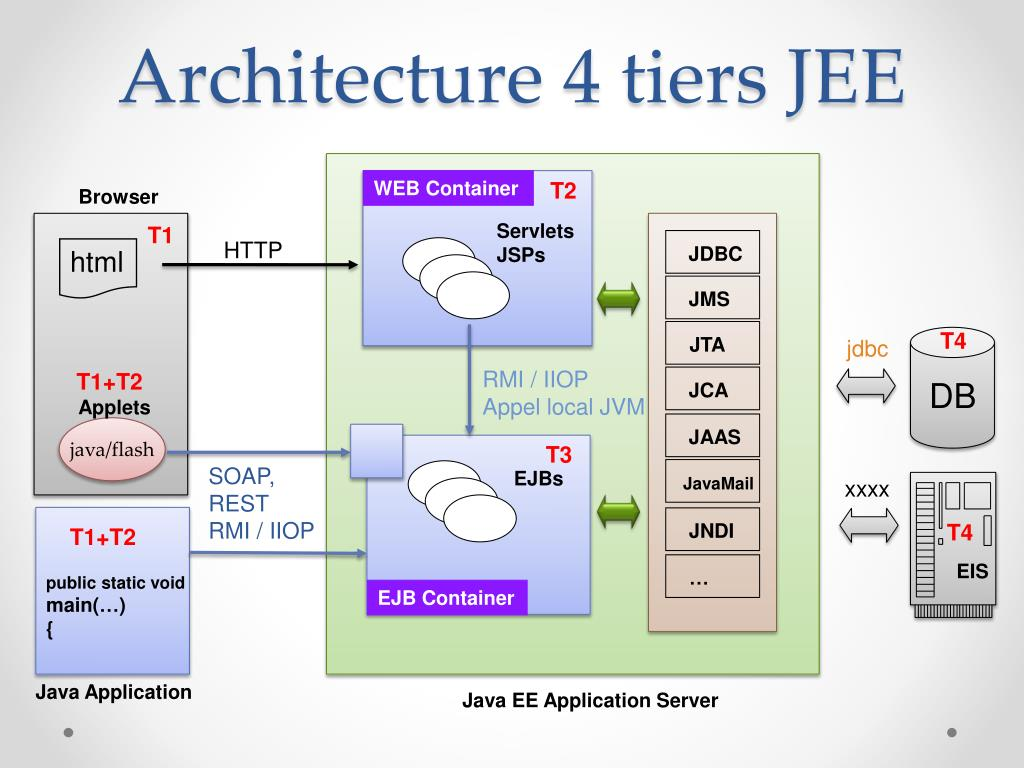
There also exist many problems that we must deal with. For example, the potentially large number of users means that we require a lot of efficiency on our servers. One of many solutions to this might be multithreaded applications. The end user may need to request that the server perform some action; for instance, when an event such as a button event is triggered. One solution to this is called Remote Method Invocation (RMI) which is where the client is able to invoke methods that are exposed on the server from the client-side remotely. One final issue that I will mention is the issue of persistent data. When I say persistent data, I do not mean semi-persistent data (e.g. cookies), but rather, fully persistent data such as client info stored in a database. In terms of Java, we will look at Object Relational Mapping (ORM) as the method of storing data on our database.

Jakarta EE:

I briefly touched on Jakarta EE earlier. I want to make clear a few points. First is that Jakarta EE includes Java SE, so all of the functionality of Java SE exists in Jakarta EE. Jakarta EE is the specification for creating web applications, not the implementation. There exist many application servers that implement Jakarta EE such as Jboss, Glassfish, Weblogic, Websphere, etc. We will be looking at Glassfish for no other reason than this is the one I was taught in school and it works well with NetBeans, which provides us with useful features for developing JakartaEE applications. JakartaEE relies heavily on the use of annotations, which are tags that we decorate classes, fields, and methods with. JakartaEE parses our code for these annotations and determines how to structure the code on the server (this will hopefully make a bit more sense in a bit).

 In the figure above, we can see the full scope of Jakarta EE. We will be covering many of these concepts throughout this document to some degree of detail. Note that an application server such as Glassfish provides us with all of these services, whereas a *web profile* such as Apache Tomcat, only provides us with the services that you see under the web profile category. We want to use some of the more feature rich aspects of JakartaEE such as JAX-RS for RESTful APIs, Java Authentication and Authorization Services (JAAS), Java Native Directory Interface (JNDI), and Remote Method Invocation (RMI), so a web profile is not sufficient (although it may be for your project).

Tiered Applications:

A tiered application is comprised of three distinct parts (which we’ve already seen prior, but now I’m giving more official terminology). The three parts are known as the Presentation Layer (which is segmented into the client-side presentation and server-side presentation), the Business Layer, and the Data Layer. As you may be able to deduce ahead of time, the Presentation Layer is essentially the front end, the Business Layer is the web application running on the server, and the Data Layer (sometimes also called the Persistence layer) is our backend/DB. We can see a few things in this figure of noteworthiness. As expected, HTML pages are being served via the HTTP(S) protocol by our application server. The client views this through their browser. The browser’s web engine is responsible for parsing the HTML/CSS/JS files to produce the viewable/interactable page.

Unfortunately, this diagram is not that clear, but it is important that we learn about JakartaEE containers, of which there are 4 kinds: Web Containers, EJB Containers, Application-client Containers, and Applet Containers. If you’re not familiar with the concept of containers in computing, as a basic description, they are isolated programs which can be ran on a server. Using Docker as an example, Docker packages individual applications and all of their dependencies in a docker image. A Docker image then becomes a container at runtime. A server can then run Docker as a daemon and have any application written in any language run on it. This is where those annotations I mentioned come into play. Annotations are used to group our classes into containers on the server.

Let’s first look at the Application-client container. This would be the blue box on the bottom left of the diagram. An Application-client container contains additional components that allows Java SE apps to interact with our server (e.g. though lookup or RMI).

Next, we’ll look at Applet containers (the small blue box which is on top of the EJB container). Java Applet containers manage the execution of components (Applets) that run in a Browser’s Java plugin. Java Applets are essentially not used any longer due to security flaws. An Applet is a small application that performs a specific task that runs within the scope of a dedicated widget engine e.g. a browser plug-in (even though it does not use Java Applets specifically, Adobe Flash is another example of a platform that was used to serve applets which reached its EOL due to security concerns as well). A Java Applet, specifically, refers to an applet written in Java, often designed to be placed on a web page. An common way of using Java applets would be to capture mouse input and provide control such as buttons or check boxes. The applet would then change the graphic content based on this feedback.

A Java Web Container is a Java application that controls a servlet. Servlets are just Java programs which do not have a main() method, therefore requiring a container to load them. So, we can think of the Web Container as the place where servlets get deployed. Note that in the image we refer to something known as JSPs. JSP stands for Java Server Pages. A JSP can be thought of as an extension to servlets because they provide more functionality than a standard servlet. We will look at JSP more in-depth later.

Let’s also look at the Enterprise Java Bean (EJB) Container. As you might guess, the EJB Container controls EJBs. EJBs must always be used when an application requires remote access via RMI. EJBs also provide scalability since they support load balancing, clustering, and fail-over (we will discuss these eventually). Most important of all is that EJBs encapsulate the business logic of our application, which is the primary reason that we will be using them.

Java Beans:

You may be asking yourself what a Java Bean is? This will come up often when discussing Jakarta EE, so we must understand the concept. Essentially, a Java Bean is a Plain Old Java Object (POJO) class which must provide public getters and setters for the object’s properties (which must all be private). When a private field has both a public getter and/or setter, this is referred to as a *property* of the object. Typically (although not mandatory,) Java beans are serializable, as this is useful for persistence or transferring them to hard disk. In case you are not familiar with what a POJO is, it is a Java object that is not bound by any special restrictions other than those forced by the Java Language Specification, and that does not depend upon any sort of classpath. A POJO should not extend prespecified classes, prespecified interfaces, or contain prespecified annotations. These restrictions exist to make POJOs portable across any device running Java.

So now we understand that a Java bean is just a POJO class with private instance/member variables that have public getters and setters (properties). An *Enterprise* Java Bean can either be a *session bean* or a *message bean* (this gets quite confusing, I apologize). Message beans are for asynchronous messaging with the web server. We will not concern ourselves with those for now. The more important one is the session bean, which comes in three types:

**@stateful:** Stateful EJBs preserve state in a DB

**@stateless:** Stateless EJBs do not preserve state

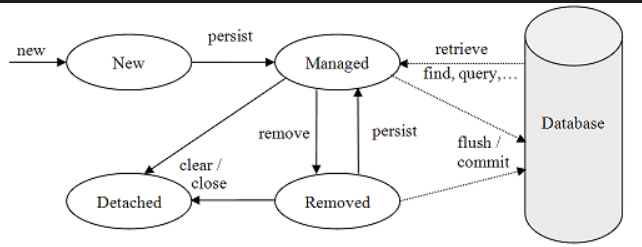
**@singleton:** Singleton EJBs can only have one instance and do not preserve state

A session bean is an EJB that is used to store session data. In our application, a session bean is usually appended with the word “Facade”. So if we had a Person class and wanted to manage it and have its members be persistent in our database, then the session bean that manages the Person entity would be called PersonFacade.

A *Named* bean is *not* an EJB, however, it *is* still a Java bean. Rather than using the @stateful, @stateless, or @singleton annotations, it always uses the @Named annotation. Named beans do not manage entities like session beans do. You might hear other terms for Named beans because the naming is always changing (very confusing, I know). Other names for Named beans are “Managed beans” and “Backing bean”. From now on I will be calling these backing beans, because a backing bean is an *implementation* of a Named bean. Backing beans control a set of JavaServer Face pages (JSF). I don’t think I covered JSF yet, but essentially, JSF is the standard for the presentation layer in Java web applications. It is implemented behind the scenes by servlets and JSPs, which I’ve mentioned already. Backing beans are appended with “Controller”. In our Person class example, the backing bean in our application would be called PersonController.

So to recap, session beans (a form of EJB) manage the entities by providing functions that act on the entity manager using JPA. They are the mediator between the server and the DB and they end with the word ‘Facade’. Backing beans are the mediator between our JSF pages (which are the presentation layer data that gets sent to our client) and the server and they end with ‘Controller’.

Entities:

An entity in Jakarta EE is essentially a class which creates instances of itself to be stored or loaded from the DB. Entities are closely tied to tables in the DB. They are POJOs which have the @Entity annotation before the class declaration. Each property of the class represents a column in the DB, and each instance of the class represents a new row/entry in the DB. An entity is managed by a session bean (a class ending with “Facade”), as we’ve covered. The *Java Persistence API* (JPA) is the API that allows us to do all of this.   
Below, you can see the JPA entity lifecycle. First, a new entity is created. Through the JPA, we can tell the session bean that we want to have this entity persist. Once we do this, it is in a “managed” state, meaning that the session bean is keeping track of it. We can clear/close the entity to delete it permanently, or temporarily remove it, at which point it can either be detached, or move back to being persistent. From either the managed state, or the removed state, we can have our entity stored in the DB. If we want to retrieve data *from* the DB, and store it in our entity, then the entity will implicitly be managed.

Entity Class Example:

Here is an example of what an entity class may look like:

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

import javax.persistence.Table;

@Entity

@Table(name=”acme\_employees”)

public class Employee {

private int empId;

private String empName;

public string getName() {

return empName;

}

public void setName(String name) {

this.empName = name;

}

@Id

@GeneratedValue(strategy=GenerationType.IDENTITY)

public int getId() {

return empId;

}

}

A few things to note in this class. First, notice that it is a POJO class. It only uses imports from the Java Class Library. If you read my notes on Java application programming, you may recognize the javax package. The javax package was originally created for extension classes which would sometimes be migrated to the java package. Rather than using javax.swing, we will mostly use javax.persistence (although we’ll look at others like javax.security). We require the persistence.Entity package as it provides us with the @Entity annotation, and likewise, we require persistence.Id for the @Id annotation and persistence.Table for the @Table annotation. As I explained earlier, @Entity marks this class as an entity for use with JPA. @Table, as you probably guessed, renames the table representing the entity class in the database. Without the @Table annotation, the table will be named after the class name by default. @Id marks a field as the Primary Key (PK) in the database. Note that we also import GeneratedValue and GeneratedType. @Generated value is another annotation that accepts a GeneratedType enum for its “strategy” parameter. The available members of the GeneratedType enum are TABLE, SEQUENCE, IDENTITY, and AUTO. TABLE indicates that the persistence provider must assign primary keys for the entity using an underlying database table to ensure unique IDs are created. SEQUENCE indicates that the persistence provider must assign primary keys for the entity using a database sequence (e.g. incrementing numbers). IDENTITY indicates that the persistence provider must assign PKs for the entity using a database’s *identity* column, which must be added manually. Finally, AUTO decides which of the three previous methods is best given the context of the Id field.

Java Naming and Directory Interface (JNDI):

Our applications use JNDI to look up data sources. JNDI URIs are composed of three parts: The root name, a subcontext, and the path. Here is an example of a JNDI name: Java:comp/DefaultDataSource. Here, Java is the root name (sort of like a namespace), comp is the subcontext (specifies the *type* of resource), and DefaultDataSource is the path. The Java root name is almost always used, so we will skip that for now. Here are the different kinds of subcontext and what each of them are used for:

**- global:** Means that the resource can be found on another server

**- module:** Means that the resource exists in the same module or war file

**- comp:** Component reserved for component-related bindings

**- app:** Used if reusing the same EJB or same application

Packaging JakartaEE Applications:

A web-module is the smallest deployable/usable unit of resoruces. It gets packaged into a Web Archive (.war) file. WAR files are very similar to JAR files, with a slightly different directory layout. The wwwroot file contains static files such as .xhtml, .css, and image files. A subdirectory in wwwroot called WEB-INF contains a classes directory which contains servlets, EJBs, and other EE components. It also contains a lib folder containing third-party libraries in the form of JAR files. A file called persistence.xml manages our DB connection and all of our existing entities. It must exist in either WEB-INF/classes/META-INF/ or WEB-INF/lib/MyEntities.jar/META-INF/.

An Enterprise Archive (.ear file), is the recommended packing for larger applications that contain multiple web-modules. An EAR file can contain one or more web-modules, each packaged as WAR files, one or more EJB modules packages as JAR files, one or more Resource Adapter modules packaged as RAR files, one or more library files packaged as JAR files, and the JakartaEE application deployment descriptor application.xml file inside WEB-INF.

ORM vs JDBC:

Java Database Connectivity (JDBC) is a low level API that was used once upon a time. Programmers would have to manually create a query string and send it to the database in order to fetch data e.g. dataBase.Query(“select retailer\_name, curr\_inv\_level from regional\_inventory where region = ?”); The issue with this approach is that if we change the DB in any meaningful way then we would be forced to rewrite all of our affected query strings. This is why JDBC is not considered to be a persistence framework.

The solution to this, of which you may be familiar, is Object Relational Mapping (ORM). This is the process of mapping our classes (entities) to our DB using the JPA, as we’ve discussed prior. Using JPA, we can either create our entity classes automatically based on our database, or we can create our database based on our entity classes. These two approaches are called database-first and code-first, respectively. Rather than using String literals to query, we can use our entity properties an other methods to maintain a relationship between the DB and our entity.

In order to make a query using JPA, we first create a CriteriaQuery object. javax.persistence.criteria.CriteriaQuery must be imported to use this class. Once imported, we *create the object like so:*

public List<T> findRange(int firstIndex, int lastIndex) {

CriteriaQuery cq = getEntityManager().getCriteriaBuilder().createQuery();

cq.select(cq.from(Entity.getClass())); // Select the entity class to operate on

Query query = getEntityManager().createQuery(cq);

query.setMaxResults(lastIndex – firstIndex + 1); // Maximum number of entities to return

query.setFirstResult(range[0]); // Starting position of the first entity

return query.getResultList();

}

RESTful API:

The RESTful API is a very powerful and common API used on the web to perform CRUD operations (create, read, update, delete). “REST” stands for *REpresentational State Transfer*. It was introduced with HTTP1.1. Resources are retrieved and manipulated through URIs using the GET, PUT, POST, and DELETE HTTP verbs. With RESTful APIs, data is typically sent through XML and/or JSON formats. In JakartaEE, REST is implemented using JAX-RS, which stands for Java API for RESTful (Web) Services.

A RESTful interface must satisfy the following constraints in order to truly be a RESTful API:

- Client-Server

- Stateless

- Cacheable

- Uniform Interface

- Layered System

- Code-on-Demand (optional)

An example of accessing a resource through a RESTful API would be to search for a URI in your search bar e.g. [http://www.myserver.com/resource](http://www.myserver.com/myresource)s/foo. Note that we are not trying to access a website page or anything like that. Under normal circumstances, if a resource such as a web page could not be found at this location, we would recieve a 404 error. However, we can setup our web application’s controller to listen for GET requests for “/resources/foo” and then return some XML/JSON data that represent the resource being accessed instead. So if we entered that URL, the server might respond with something like the following:

**JSON format:**

{{name: “Todd”}}

**XML format:**

<element\_list>

<some\_element>

<name>Todd</name>

</some\_element>

</element\_list>

Whether the server returns JSON or XML is up to the developer writing the web application. You can allow requests to return strictly JSON, strictly XML, or both.

Idempotency:

For a long while I wasn’t aware about the differences between PUT and POST since school taught me to just always use POST. Well there *is*, in fact, a difference. PUT is intended to be idempotent, meaning that calling it once or many times successively has the same effect. For example, clicking the buy button multiple times on an online store will not make multiple purchases if the website stalls for whatever reason. It will recognize that you’ve pressed it, so subsequent PUT requests will not have any effect. A POST request, on the other hand, is not idempotent. This would be desired if, for example, you had a button that incremented the number of a certain product that you wanted to purchase.

HTTP Requests and Responses:

HTTP requests and responses are structured as follows:

1. A status line which contains the status code for the request/response and a brief description of what the status code indicates. It may include a bit of extra information depending on whether or not the HTTP message type is a request or a response

2. HTTP headers which specify the request or describe the body are optionally included

3. A blank line

4. The body, which contains data associated with the message (HTML, JSON, XML). The bodies contents and size are specified by the status line and HTTP headers.

For an HTTP *request*, the status line would include an HTTP verb, such as GET, PUT, POST, or DELETE. It would also include the request target (URL) and the HTTP version. The headers would include the following:

**- General headers:** Apply to the entire message e.g. a *Via-Type* header

**- Request headers:** Modify the request by specifying it further (e.g. Accept-Language), providing additional context (e.g. Referer), or by conditionals (e.g. If-None). Examples are *User-Agent* and *Accept-Type* headers.

**- Entity headers:** Apply specifically to the body of the request e.g. Content-Type and Content-Length. This header is not included if there is no body in the request.

The body (if it exists) can either be a single-resource body defined by Content-Type and Content-Length, or multiple-resource bodies which have multiple parts (e.g. HTML forms).

For an HTTP response, the status line includes the protocol version (usually HTTP/1.1), status code indicating success or failure, and some status text which provides a brief textual description of the status code. An example would be “HTTP/1.1 404 Not Found”. The headers for a response are the same as they are for a request. The body (if it exists) can be a single-resource body defined by Content-Type and Content-Length. Unlike HTTP requests, single-resource bodies in the response can have an unknown length as well. Multiple-resource bodies are rare in responses.

RESTful APIs With JAX-RS:

In order to use JAX-RS in our project, our class must extend javax.ws.rs.core.Application. It also requires the use of the @ApplicationPath annotation. The application path specifies the initial match that needs to be made in the URL to access our RESTful services. So for example, if we wanted to access the foo resource, we would search for <https://www.website.com/resources/foo>. “resources” is the default, but this can be changed. Here is an example:

import javax.ws.rs.ApplicationPath;

imoprt javax.ws.rs.core.Application;

@ApplicationPath(“resources”)

public class JAXRSConfiguration extends Application {

...

}

Depending on what OS you’re running, the class may be called something other than JAXRSConfiguration, such as JakartaRestConfiguration.

In order to create a resource to be accessed through the RESTful API, we must create what is called a resource class or resource method. A resource is just a class or method, but they are distinguished through annotations. Resource classes/methods must have a @Path annotation which describes the name of the resource. From the client’s perspective, the resource that they are trying to access is also called an endpoint i.e. the last portion of the URL. In a previous example (<http://www.website.com/resources/foo>), “foo” would be the endpoint since it is the last portion of the URL. Therefore, the @Path annotation would look like: @Path(“foo”). Next, the resource classes/methods require an HTTP method annotion which can be one or more of @GET, @POST, @PUT, @DELETE. Here is an example of a class and method resource:

@Path(“sample”)

public class SampleResource {

@GET

public Response message() {

return Response.ok(“Hello World”).build();

}

}

The path annotation creates a resource class called sample. If the end user tries to access the sample endpoint with an HTTP GET request message, the HTTP response will be OK (status code 200) and it will contain the message “Hello World” in the body.

Netbeans is capable of creating a default template for RESTful services, either from an existing database, or from our entity classes. The easiest way to do this is to right click on the project and click new > RESTful Web Services from [Entity Classes]/[Database].

Working With XML and JSON:

Since our RESTful services are capable of recieving either XML or JSON data in the HTTP request headers, it is important that we know how to parse these formats. Parsing/deserializing XML and JSON is sometimes referred to as *unmarshaling* and generating/serializing XML or JSON through code is sometimes referred to as *marshaling*.

To marshal XML, we make use of some more annotations (each come from the javax.xml.bind.annotation package). The first annotation is @XmlRootElement. This goes over a class typically and creates the root element of the XML document (I’ll show examples in a moment). The next annotation is @XmlAccessorType and the XmlAccessType enum class which comes from the java.lang.Enum package. The enum members are PROPERTY, FIELD, PUBLIC\_MEMBER, and NONE. These describe which class members should get translated into XML elements by default. PROPERTY would be private fields with getters/setters, FIELD would be private fields, PUBLIC would be public fields, and NONE would not convert anything into XML. Finally, the @XmlElement annotation marks individual members as XML elements which are not already included by @XmlAcessorType. In order to set element attributes, we can fill in the “name” parameter for @XmlElement or @XmlRootElement, or simply use @XmlAttribute to mark a specific member an attribute. Here is an example of marshalling a class to XML:

@XmlRootElement

@XmlAccessorType(XmlAccessType.FIELD)

public class Sprites {

@XmlElement(name=”sprite”)

private List<Sprite> sprites;

public List<Sprite> getSprites() {

return sprites;

}

}

Just because we have the annotations, doesn’t mean that the XML will automatically generate itself for us. We still need to call the marshaller manually:

try {

JAXBContext jContext = JAXBContext.newInstance(Student.class);

Marshaller marshallObj = jContext.createMarshaller();

marshallObj.setProperty(Marshaller.JAXB\_FORMATTED\_OUTPUT, true);

Sprites sprites = new Sprites();

marshallObj.marshal(sprites, new FileOutputStream(“/path/to/output.xml”));

}

catch(Exception e) {

e.printStackTrace();

}

The resulting XML will look something like:

<?xml version=”1.0” encoding=”UTF-8” standalone=”yes”?>

<sprites>

<xs:sequence>

<xs:element name=”sprite”>

<color>Red</color>

<height>100</height>

<width>200</width>

</xs:element>

</xs:sequence>

<xs:sequence>

...

</xs:sequence>

</sprites>

Converters, Listeners, and Validators:

Converters, Listeners, and Validators are used in JakartaEE web applications to help us interact with the client. Here is a brief description of each:

**- Converters:** Converters are used to convert form data (almost always string data) into more complex types. In order to convert between more complex types such as class instances like Color, we need custom converters. JakartaEE has built-in converters for conversions to more basic types e.g. IntegerConverter, LongConverter, ShortConverter, FloatConverter, DoubleConverter, DateTimeConverter, CharacterConverter, ByteConverter, BooleanConverter, BigIntegerConverter, etc.

**- Listeners:** Listeners are used to listen for events happening on the page and handle them with event handlers.

**- Validators:** Validators are used to validate data which is received from input components. Validators allow an application to express constraints on form data to ensure that the necessary requirements are met before data is processed. Essentially this is input validation.

There are a few standard validators for JSF that I will briefly cover:

|  |  |  |
| --- | --- | --- |
| **Validator Class** | **Tag** | **Function** |
| BeanValidator | validateBean | Registers a bean validator for the component |
| DoubleRangeValidator | validateDoubleRange | Checks whether the local value of a component is within a certain range. The value must be a floating-point or be convertable to a float |
| LengthValidator | validateLength | Checks whether the length of a component’s local value is within a certain range. The value must be a java.lang.String |
| LongRangeValidator | validateLongRange | Checks whether the local value of a component is within a certain range. The value must be any numberic type or String that can be converted to a long |
| RegexValidator | validateRegex | Checks whether the local value of a component is matched against a regex from the java.util.regex package |
| RequiredValidator | validateRequired | Ensures that the local value is not empty on an EditableValueHolder component |

Expression Language (EL):

Expression Language is an important mechanism for enabling the presentation layer (web pages) to communicate with the applications business logic (managed/named beans). EL is used by several JakartaEE technologies such as JSF, JSP, and Contexts and Dependency Injection (CDI).

JSF pages uses tags similar to HTML which are rendered on the client-side upon request of the page. For instance, rather than using an <a> tag for hyperlinks, as we would in regular HTML, EL would have something like: <h:commandLink action=”#{contactController.create}” value=”#{bundle.CreateContactSaveLink}” />. Note here that the text enclosed within “#{}” is all EL code. This particular JSF component would invoke the create() method in ContactController.java. The create() method would, pressumably, call getFacade().create(current); which in turn would call getEntityManager().persist(entity); and put the new entity in the JPA lifecycle in the persist state (i.e. put it in the database). The bundle class is something that we’ll look at a bit later. It pertains to the text of the hyperlink. Bundles allow us to dynamically load text based on the user’s locale. We can also validate user input in an HTML form using validators. Here is an example:

<h:inputText id=”quantity” size=”4” value=”#{item.quantity}” validatorMessage=”minimum 1 please”>

<f:validateLongRange minimum=”1” />

</h:inputText>

<h:message for=”quantity” />

This code would provide the user with a text box on the page. The value entered by the user would be processed and validated and then, if successful, bound to the “quantity” field in the class associated with the JSF page. The size attribute specifies the number of characters in the string which are permissible. Value would set the default text in the text box to be whatever item.quantity is. Finally, the validatorMessage is the error message that is thrown if quantity could not be validated. Note that we can define our own custom validators as well. The letter prepending the JSF component represents the namespace that the particular component is derived from. The letters are arbitrarily mapped to their respective namespaces in some XML file located within the project. The common mappings are as follows:

h http://java.sun.com/jsf/html

f http://java.sun.com/jsf/core

c http://java.sun.com/jsp/jstl/core

fn http://java.sun.com/jsp/jstl/functions

ui http://java.sun.com/jsf/facelets

As mentioned in the previous section, JakartaEE provides built-in converters for translating user input via form fields (string data) into common data types. A built in converter would use a JSF component such as <f:convertDateTime> or <f:convertNumber>. Custom converters are much more complex. They require us to create a class which has the @FacesConverter annotation. The class must implement the Converter interface which provides definitions for

- public Object getAsObject(FacesContext facesContext, UIComponent component, String value)

- public String getAsString(FacesContext facesContext, UIComponent component, Object object)

These two methods must be overriden in the converter class. getAsObject() returns the object representation of the conversion to the server. For example, if you were converting the string to a Color object, getAsObject() would return a Color object. getAsString() returns a string representation of the object to the server.

In JakartaEE web applications, EL is typically embedded in an Extended Hyper Markup Language (XHTML) file. XHTML is not the same thing as EL. XHTML is part of the family of XML markup languages and was meant to extend the features of HTML. EL was written to be compatible with the XHTML format, however, XHTML files do not necessarily always contain EL code.

Java Server Faces (JSF) in Detail:

Until now, I’ve been a little vague on JSF because we’ve been focusing more on the backend. In short, JSF is the Java standard technology for building component-based, event-oriented web interfaces. As you know, JSF is implemented behind the scenes using servlets and JSPs. Like JSP, JSF allows access to server-side data and logic. Unlike JSP, which is basically HTML page embedded with server-side capabilities, JSF is an XML document that represents formal components in a logical tree (a schema). JSF components are backed by Java objects, which are independent of the HTML, and have the full range of Java abilities, including accessing remote APIs and databases. JSF components can be backed by the following: backing beans, validators, converters, and other server-side objects.

JSF provides the following: a core library, a set of UI components (standard HTML input elements), extension of the base UI components to create additional UI component libraries or extend existing components, and multiple rendering capabilities that enable JSF UI components to render themselves differently depending on the client types.

A JSF page’s lifecycle begins when a new page is requested by the client. It ends when the HTML page has been sent as a response. The tree of components on the page is called the View. Upon an initial request from a client, we “restore” the view i.e. create an empty view. We then render the page by populating the empty view with components that are referenced by the EL tags embedded in the page. When the user submits a form a few things happen: First, the view is restored by using the state information saved on either the client’s browser cache or the server. Each component in the component tree then extracts its new value from the request parameters. The validator processes each of the components parameters to ensure they match our criteria. Next, we traverse the component tree and assign our entity’s properties to the values returned on the form. In the case that we cannot assign our form data to one or more of our entities properties because either the data could not be validated or because the data could not be converted to the appropriate Java type, then an error handler method is invoked which is defined within the EL code. Finally, we render our response. If you are familiar with model binding in ASP.NET MVC, then these concepts may be more familiar to you.

Backing Beans in More Detail:

If you recall, backing beans (a.k.a. named beans) are lightweight container-managed objects (POJOs, specifically) that have the @Named annotation and their class name ends with ‘Controller’. If the @Named annotation is used in unison with the @SessionScoped annotation, then the backed bean is automatically registered as a resource with JSF that JSF components can back themselves with. E.g.:

@Named(“cart”)

@SessionScoped

public class ShoppingCart ... {

...

}

There are three backing bean scopes. These alter a user’s access to resources located on the server-side:

**Application (javax.enterprise.context.ApplicationScoped):** The application scope persists across all users’ interactions with a web app.

**Session (javax.enterprise.context.SessionScoped):** Session scope persists across multiple HTTP requests in a web app.

**Request (javax.enterprise.context.RequestScoped):** Request scope persists during a single HTTP request in a web app.

Let’s also talk about dependency injection, which is something that comes up a lot, especially in backing beans. I’ve already explained dependency injection in my .NET notes, but I suppose I will have to explain the concept once again. The issue arises when

The simple solution to this in JakartaEE is to use either the @EJB annotation or the @Inject annotation. @Inject is preferred over @EJB for a multitude of reasons (more typesafe, supports @Alternatives, is aware of the scope of the injected object) but this will break code under certain circumstances, in which case, you will need to revert to using @EJB. Dependency injection is handled through a system known as Contexts and Dependency Injection (CDI). @Inject should be placed over the method being injected and over the injection point. A common use for @Inject is for constructors. Here is a simple example:

@WebServlet("/cdiservlet")

public class NewServlet extends HttpServlet {

@Inject private Message message;

@Override

public void doGet(HttpServletRequest request, HttpServletResponse response)

throws IOException {

response.getWriter().write(message.get());

}

}

Business to Business (B2B):

B2B is when a third-party/another company makes use of our service. They may have their own users using their own presentation layer, but they require our services. Rather than the third-party business connecting to our database directly (which should never be the case), we can serve them data through RESTful APIs and XML/JSON data. This way our business layer is directly interacting with their business layer.

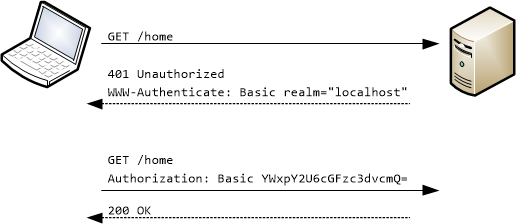
Criteria API and Java Persistence Query Language (JPQL):

Internationalization:

Security:

There are three pillars of security: identification, authentication, and authorization. Identification is the method of distinguishing between users. For example, entering an email or username would be a form of identification. The issue is that anyone can pretend to be that user, so we must authenticate them. Authentication checks for a method of proving the user’s identity, such as a password. Authorization deals with the level of access that we grant to a user. For instance, checking if a user has administrative rights.

The HTTP protocol has built-in support for something called basic authentication.



Basic authentication can be used on any website or webapp. Basic authentication must take place over an HTTPS connection. First, the user sends an HTTP request to the server. The server replies with an HTTP 401 status (unauthorized) and it provides information on how to authorize with a WWW-Authenticate response header containing at least one challenge. A client that wants to authenticate itself with the server can then do so by including an Authorization request header with the appropriate credentials. This is when the client (usually the browser) would present the password prompt to the user and issues the Authorization request containing the correct Authorization header.

The Glassfish web server uses port 8181 for HTTPS traffic by default. For the purposes of development, we work off of port 8080 (HTTP traffic). This is because Glassfish signs its own TLS/SSL certificates when using HTTPS on port 8181. Modern browsers will not accept these self-signed certificates, and so, we must pay money for a certificate (there are exceptions to this, but regardless, using HTTP on localhost is the easiest solution for testing).

Basic authentication does not actually encrypt passwords. It only encodes them through base64, which is easily reversible. Therefore, you should never use basic authentication in production. Once again, it is more for testing purposes. We’ll look into proper form authentication momentarily. As an aside, you may recall that I briefly mentioned message beans which were used for message sending. Message layer security is handled through a network protocol called SOAP, whereas HTTPS requests are handled through TLS/SSL.

JakartaEE Security API:

JakartaEE 8 provides a security API which includes various annotations, methods, and interfaces for handling security. The API separates login security into 4 categories: realms/identity stores, users, groups, and roles. The users category is essentially the actual user themselves i.e. the clients that we are keeping track of. Groups work similar to how they work on a system like Linux, where multiple users can be added to a group and share the same access rights for resources owned by the group. Roles are users + groups. By default, each group is also a role e.g. an admin group would create an admin role, which would contain the admin group and all of the users of that group. The fourth category (realms/identity stores) represent the username and password database. The old method of doing this was to use a JDBC Realm (a database) and specifying where the system could retrieve usernames, passwords, and groups. The new way of doing this to create an IdentityStore (also just a database) using the @DatabaseIdentityStoreDefinition annotation. The Glassfish implementation for indentity stores is called Soteria.

Security constraints need to be set in the web.xml file by specifying the URL path pattern. For example, we would specify that a user needs to be in the admin role to access URLs that begin with /admin/\*. RESTful services require their own roles to be set for any resources accessed through /resources/\*.

Realm vs. IdentityStore:

I’ll focus mostly on how realms work, since IdentityStores are simple in comparison. A realm is a complete database of users and groups identified as valid users of one or more applications and controlled by the same authentication policy. The JEE server authentication service can keep track of multiple realms at once.

Internationalization:

React Pages:

Junit and Selenium: