# JPA Joins & Attribute Conversion

What good would JPA be if we can't use it with a real-life database? JPA supports the use of table joins both in entity annotation and query creation. For our purposes, we are going to focus on entity annotation. The representation of relationships between entities is one of the most powerful features of JPA. It makes it very easy to navigate from one entity to one or more related entities in your Java code, like from a Person entity to the corresponding Address entity or Phone Number entity.

## Database Modification

Currently, our shopping database has only one table. But we aren't going to create the tables in MySQL – let EclipseLink do the work for us. Open up your ShoppingList project and go into the persistence.xml file. Modify the schema generation to "Drop and Create Tables". You'll lose all your data in it thus far.

## Shopper Class

Inside our model package, create a new Shopper class and annotate it to commit to the Shopper table.

```
// package and import statements
@Entity
@Table(name="shopper")
public class Shopper {
	@Id
	@GeneratedValue
	private int id;
	private String shopperName;

	public Shopper() {
		super();
		// TODO Auto-generated constructor stub
	}
	public Shopper(int id, String shopperName) {
		super();
		this.id = id;
		this.shopperName = shopperName;
	}

	public Shopper(String shopperName) {
		super();
		this.shopperName = shopperName;
```

```
	}
	public int getId() {
		return id;
	}
	public void setId(int id) {
		this.id = id;
	}
	public String getShopperName() {
		return shopperName;
	}
	public void setShopperName(String shopperName) {
		this.shopperName = shopperName;
	}
	@Override
	public String toString() {
		return "Shopper [id=" + id + ", shopperName=" +
shopperName + "]";
	}

}
```

Note the extra helper constructor that just takes in the name – this is so we can create a shopper with just a name to be passed into the database and let the database auto-generate the next id.

After annotating, open up persistence.xml and add it to the managed classes.

### ShopperHelper class

Inside the controller class, create a ShopperHelper class. Let's create the methods to insert and showAllShoppers.

```
//package and import statements
public class ShopperHelper {
	static EntityManagerFactory emfactory =
Persistence.createEntityManagerFactory("ShoppingList");

	public void insertShopper(Shopper s) {
		EntityManager em = emfactory.createEntityManager();
		em.getTransaction().begin();
		em.persist(s);
		em.getTransaction().commit();
		em.close();
	}
```

```
	public List<Shopper> showAllShoppers() {
		EntityManager em = emfactory.createEntityManager();
		List<Shopper> allShoppers = em.createQuery("SELECT s
FROM Shopper s").getResultList();
		return allShoppers;
	}
}
```

### Testing Persistence without a web form

We need to test our persistence. We could go ahead and create an HTML form and servlets, launch Tomcat each time we make a change and reload… Or we could cheat and create a tester class to make sure our persistence works. Let's do that.

Inside the default package, create a class called ShopperTester with a main method. Create an instance of our ShopperHelper and let's try to add a shopper and recall the list. Your class might look something like this:

```
public class ShoppingTester {

	public static void main(String[] args) {

		Shopper bill = new Shopper("Bill");

		ShopperHelper sh = new ShopperHelper();

		sh.insertShopper(bill);

		List<Shopper> allShoppers = sh.showAllShoppers();
		for(Shopper a: allShoppers) {
			System.out.println(a.toString());
		}

	}

}
```

When you run this class, you'll run it as a Java application – a bit different than running a servlet. Again, this is just a quick and easy way to test your entity. Your code should insert one shopper and you should see it console below.

```
	Shopper [id=1, shopperName=Bill]
```

If you run your class again, the tables will drop and you'll see the same thing again. To add another shopper, create another object and insert it.

```
        Shopper jim = new Shopper("Jim");
        sh.insertShopper(jim);

Shopper [id=1, shopperName=Bill]
Shopper [id=2, shopperName=Jim]
```

## Creating a Shopping List – ListDetails class

Next, let's create another class to assign a Shopper to a shopping list. In your model, let's create a ListDetails class. This class will hold all the details of the list, including the list name, the shopping date, the shopper and the items for the shopping trip.

```
public class ListDetails {
    private int id;
    private String listName;
    private LocalDate tripDate;
    private Shopper shopper;
    private List<ListItem> listOfItems;
```

We have several things we have to implement to get this entity working. Let's start with our getters and setters, a toString( ) and our constructors. In addition to our default, no-arg constructor (which you have to have with JPA), create three additional helper constructors:

```
public ListDetails(int id, String listName, LocalDate tripDate,
Shopper shopper, List<ListItem> listOfItems) {
        //omitted for space but set them in your code    }

    public ListDetails(String listName, LocalDate tripDate,
Shopper shopper, List<ListItem> listOfItems) {
        //omitted for space but set them in your code    }

    public ListDetails(String listName, LocalDate tripDate,
Shopper shopper) {
        //omitted for space but set them in your code    }
```

Let's start annotating:

```
@Entity
public class ListDetails {
    @Id
    @GeneratedValue
    private int id;
    private String listName;
    private LocalDate tripDate;
```

(Open up persistence.xml and add it to the managed classes!)

The Shopper should link to our Shopper object and the table in it. In our program, a shopper can have many lists, but only one list has one shopper. This is called a ManyToOne relationship and we have to provide it in the annotation.

```
@ManyToOne
private Shopper shopper;
```

Let's test what we have so far – we don't have the items on the list persisting yet, but we should still be able to get the details to persist. First, we need a helper class to persist for us and recite all the list details in the database table back to us. Create a ListDetailsHelper class in the controller package and add in the following methods:

```
public class ListDetailsHelper {
     static EntityManagerFactory emfactory =
Persistence.createEntityManagerFactory("ShoppingList");

     public void insertNewListDetails(ListDetails s) {
          EntityManager em = emfactory.createEntityManager();
          em.getTransaction().begin();
          em.persist(s);
          em.getTransaction().commit();
          em.close();
     }

     public List<ListDetails> getLists() {
          EntityManager em = emfactory.createEntityManager();
          List<ListDetails> allDetails = em.createQuery("SELECT
d FROM ListDetails d").getResultList();
          return allDetails;
     }
}
```

Create a new class called ListDetailsTester with a main method. Let's use it to test our List Details persistence. Add the following below the existing code:

```
Shopper cameron = new Shopper("Cameron");

ShopperHelper sh = new ShopperHelper();

sh.insertShopper(cameron);

ListDetailsHelper ldh = new ListDetailsHelper();
```

```
ListDetails cameronList = new ListDetails("Cameron's List",
LocalDate.now(), cameron);

ldh.insertNewListDetails(cameronList);

List<ListDetails> allLists = ldh.getLists();

for (ListDetails a : allLists) {
     System.out.println(a.toString());
}
```

Run it and you'll see it works. But take a look at the table by running "Select * from listdetails;" in your MySQL window. Check out the trip date field:

```
mysql> select * from listdetails;
+----+---------------+-----------------------------------------------------------------------------------------------------------------+------------+
| ID | LISTNAME      | TRIPDATE                                                                                                        | SHOPPER_ID |
+----+---------------+-----------------------------------------------------------------------------------------------------------------+------------+
|  2 | Cameron's List | 0xACED00057372000D6A6176612E74696D652E536572955D84BA1B2248B20C000078707707030000007E5090278 |          1 |
+----+---------------+-----------------------------------------------------------------------------------------------------------------+------------+
1 row in set (0.00 sec)
```

LocalDate and SQL date are two different data types. What a mess. There are a lot of workarounds for this. You can read about them here: https://thoughts-on-java.org/persist-localdate-localdatetime-jpa/ for reference. We are going to allow Java to do the conversion for us.

## Creating an Attribute Converter class

One of the easiest ways to fix this is to implement a converter. They are easy to add and you can set them to automatically apply when any type of conversion from one datatype to another. We will be creating a LocalDate/DATE converter but you can also do a LocalTime/TIME conversion. The code for the time conversion is in the Blackboard course for you to reference.

Inside your controller package, create a class called LocalDateAttributeConverter that implements the AttributeConverter<LocalDate, Date> interface.

Do you remember the deal with interfaces? They will do the work for you but you have to implement their methods. After you're finished adding in the necessary imports, be sure to allow Eclipse to insert the two unimplemented methods. Implementation for these two methods will follow in just a moment.

For this class, we need to include a very important annotation as well – the @Converter annotation. This tells our project to use this Converter in the type conversion that we have outlined for all basic attributes defined by entity classes, mapped superclasses, or embeddable classes that uses those types. What an awesome tool! Let's tell our program to automatically apply it after the annotation.

Hint: some people receive an error that they cannot import LocalDate or Date. If this happens to you, first import the javax.persistence.AttributeConverter. Then it will allow you to import LocalDate and Date.

Here is what your completed LocalDateAttributeConverter should look like.

```
package controller;

import java.sql.Date;
import java.time.LocalDate;
import javax.persistence.AttributeConverter;
import javax.persistence.Converter;

@Converter(autoApply = true)
public class LocalDateAttributeConverter implements
AttributeConverter<LocalDate, Date> {

     @Override
     public Date convertToDatabaseColumn(LocalDate attribute) {
          // TODO Auto-generated method stub
          return (attribute == null ? null:
Date.valueOf(attribute));
     }

     @Override
     public LocalDate convertToEntityAttribute(Date dbData) {
          // TODO Auto-generated method stub
          return (dbData == null ? null: dbData.toLocalDate());
     }

}
```

What are these statements? This are essentially if statements in short form. The part before the ? is the condition. The left side of the colon (:) is what happens if the condition is true. The second part is what happens if the condition is false. In this case:
If the attribute is null, put null in the database field. If it's not null, use the Date.valueOf( ) method to change it to the DATE datatype and persist that value.

Don't forget to add this to the managed classes in persistence.xml.

Now try to run your ListDetailsTester and you should see your date will persist.

```
mysql> select * from listdetails;
+----+---------------+------------+------------+
| ID | LISTNAME      | TRIPDATE   | SHOPPER_ID |
+----+---------------+------------+------------+
|  2 | Cameron's List | 2021-09-02 |          1 |
+----+---------------+------------+------------+
1 row in set (0.00 sec)
```

We still have that listOfItems to implement. Let's do that now.

## Lists – OneToMany annotations

Back in the ListDetails.java class, find the List of ListItems. This will contain the items for our shopping list.

```
private List<ListItem> listOfItems;
```

We have a table in our database that will hold the list id and the item id. We need that list of items to go into that table. Each list can hold many items. Therefore, we annotate this list with a @OneToMany annotation – one list, many items on the list.

Next, we need to define where the list is saved. We complete that by using a @JoinTable annotation. We can define our table and which Java field name is saved in which column. The annotation looks like this:

```
@OneToMany(cascade=CascadeType.PERSIST, fetch=FetchType.EAGER)
    private List<ListItem> listOfItems;
```

## Cascade Types

What is that cascade option? A cascade option determines what happens to the mapped entity whenever a relationship owner entity is saved. In this case, when a list is created, the database will first persist the ListItem entities' and then it will save the list details. Very cool! We could also do something like that with our shopper so we don't have to create a new instance of the shopper before it's persisted.

```
@ManyToOne (cascade=CascadeType.PERSIST)
     private Shopper shopper;
```

There are six different cascade types in JPA (adapted from https://howtodoinjava.com/hibernate/hibernate-jpa-cascade-types/):

CascadeType.PERSIST : save() or persist() operations cascade to related entities.
CascadeType.MERGE : related entities are merged when the owning entity is merged.
CascadeType.REFRESH : Refresh the state of the instance from the database, overwriting changes made to the entity, if any
CascadeType.REMOVE : removes all related entities association when the owning entity is deleted.
CascadeType.DETACH : detaches all related entities if a "manual detach" occurs.- many more options available
CascadeType.ALL : shorthand for all of the above cascade operations.

You can use more than one cascade type. For example, you could merge and refresh a list. Just put a comma between the different types `cascade={CascadeType.REFRESH, CascadeType.MERGE},.` This is very helpful when you are trying to edit or delete an item.

## FetchTypes

FetchTypes determine when JPA gets related entities from your database. There are two types of fetches: lazy and eager. With an eager fetch, JPA returns all elements in a relationship when selecting the main entity. In our case, when we want details on our lists, it will eagerly get the

items on the list and populate it into the list for us. FetchType.Lazy fetchs when you need it – not automatically. You can use the getter method in lazy fetches to get the list when you need it.

Let's test it. Change the code in your ListDetailsTester to create a few new items, create a list, add the list to the list details and then persist it all. Note that I took out the call to persist Susan The Shopper before using the shopper in the ListDetails constructor. Since a cascade type was added to persist the shopper, I no longer have to handle it in my code.

```
public class ShoppingTester {

     public static void main(String[] args) {

          Shopper cameron = new Shopper("Cameron");

          ListDetailsHelper ldh = new ListDetailsHelper();

          ListItem shampoo = new ListItem("Target", "Shampoo");
          ListItem brush = new ListItem("Target", "Brush");

          List<ListItem> cameronsItems = new
ArrayList<ListItem>();
          cameronsItems.add(shampoo);
          cameronsItems.add(brush);

          ListDetails cameronsList = new ListDetails("Cameron's
ShoppingList", LocalDate.now(), cameron);
          cameronsList.setListOfItems(cameronsItems);

          ldh.insertNewListDetails(cameronsList);

          List<ListDetails> allLists = ldh.getLists();
          for(ListDetails a: allLists) {
               System.out.println(a.toString());
          }

     }

}
```

If you look through the toString, you'll now see a list of the items in the listOfItems spot.

Now that we are finished with adding objects and setting the relationships, go back into persistence.xml and change the schema generation from Drop and Create Tables to Default(none).

## Viewing All Lists on the Web

Create a new servlet (in the controller class) and call it "`ViewAllListsServlet`". This servlet will gather all the lists to send it to the JSP. Inside the `doGet( )` add the following code to get the shopping lists to send to a new JSP.

```
ListDetailsHelper slh = new ListDetailsHelper();
List<ListDetails> abc = slh.getLists();
request.setAttribute("allLists", abc);

if(abc.isEmpty()){
          request.setAttribute("allLists", " ");
}

getServletContext().getRequestDispatcher("/shopping-list-
by-user.jsp").forward(request, response);
```

Update the @WebServlet annotation to use the lowercase v:

```
@WebServlet("/viewAllListsServlet")
```

Next, create a JSP called shopping-list-by-user.jsp in the webapp folder. Add the following code:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
 <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Shopping Lists</title>
</head>
<body>
<form method = "post" action = "listnavigationServlet">
<table>
<c:forEach items="${requestScope.allLists}" var="currentlist">
<tr>
   <td><input type="radio" name="id" value="${currentlist.id}"></td>
   <td><h2>${currentlist.listName}</h2></td></tr>
   <tr><td colspan="3">Trip Date: ${currentlist.tripDate}</td></tr>
   <tr><td colspan="3">Shopper:
${currentlist.shopper.shopperName}</td></tr>
   <c:forEach var = "listVal" items = "${currentlist.listOfItems}">
            <tr><td></td><td colspan="3">
                ${listVal.store}, ${listVal.item}
```

```
                </td>
            </tr>
  </c:forEach>
</c:forEach>
</table>
<input type = "submit" value = "edit" name="doThisToList">
<input type = "submit" value = "delete" name="doThisToList">
<input type="submit" value = "add" name = "doThisToList">
</form>
<a href="addItemsForListServlet">Create a new List</a>
<a href="index.html">Insert a new item</a>
</body>
</html>
```

Edit the index.html to add two links to the bottom.

```
<a href="viewAllListsServlet">View all shopping lists</a> <br />
<a href="addItemsForListServlet">Create a new list</a>
```

Open the index.html file, launch Tomcat & click the link to view all the shopping lists. You should see your lists.

## Creating a New List (with a web view)

Now, let's create the ability to add a new list. First off, let's gather all the items in the database to put on the list. Create a new servlet called `AddItemsForListServlet` with the following code in the `doGet( ):`

```
protected void doGet(HttpServletRequest request,
HttpServletResponse response) throws ServletException,
IOException {
          // TODO Auto-generated method stub
          ListItemHelper dao = new ListItemHelper();

          request.setAttribute("allItems", dao.showAllItems());

          if(dao.showAllItems().isEmpty()){
                    request.setAttribute("allItems", " ");
          }

          getServletContext().getRequestDispatcher("/new-
list.jsp").forward(request, response);

     }
```

Change the @WebServlet annotation to lowercase:

```
@WebServlet("/AddItemsForListServlet")
```

Now, create a new JSP page called new-list.jsp. Here is some code for that:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Create a new list</title>
</head>
<body>
<form action = "createNewListServlet" method="post">
List Name: <input type ="text" name = "listName"><br />
Trip date: <input type ="text" name = "month" placeholder="mm"
size="4"> <input type ="text" name = "day" placeholder="dd"
size="4">, <input type ="text" name = "year" placeholder="yyyy"
size="4">
Shopper Name: <input type = "text" name = "shopperName"><br />

Available Items:<br />
<select name="allItemsToAdd" multiple size="6">
<c:forEach items="${requestScope.allItems}" var="currentitem">
   <option value = "${currentitem.id}">${currentitem.store} |
${currentitem.item}</option>
</c:forEach>
</select>
<br />
<input type = "submit" value="Create List and Add Items">
</form>
<a href = "index.html">Go add new items instead.</a>
</body>
</html>
```

Finally, we need a way to process all of these form fields. Create a new servlet called CreateNewListServlet. Remember to change the @WebServlet annotation to a lowercase c. Add the following code in the doGet( ):

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

```
IOException {
          // TODO Auto-generated method stub
          ListItemHelper lih = new ListItemHelper();
          String listName = request.getParameter("listName");
          System.out.println("List Name: "+ listName);

          String month = request.getParameter("month");
          String day = request.getParameter("day");
          String year = request.getParameter("year");
          String shopperName =
request.getParameter("shopperName");
          LocalDate ld;
          try {
               ld = LocalDate.of(Integer.parseInt(year),
Integer.parseInt(month), Integer.parseInt(day));
          }catch(NumberFormatException ex) {
               ld = LocalDate.now();
          }

          String[] selectedItems =
request.getParameterValues("allItemsToAdd");
          List<ListItem> selectedItemsInList = new
ArrayList<ListItem>();
//make sure something was selected - otherwise we get a null
pointer exception
          if (selectedItems != null && selectedItems.length > 0)
{

          for(int i = 0; i<selectedItems.length; i++) {
               System.out.println(selectedItems[i]);
               ListItem c =
lih.searchForItemById(Integer.parseInt(selectedItems[i]));
               selectedItemsInList.add(c);

          }
}

          Shopper shopper = new Shopper(shopperName);
          ListDetails sld = new ListDetails(listName, ld,
shopper);
```

```
            sld.setListOfItems(selectedItemsInList);
            ListDetailsHelper slh = new ListDetailsHelper();
            slh.insertNewListDetails(sld);

            System.out.println("Success!");
            System.out.println(sld.toString());

        getServletContext().getRequestDispatcher("/viewAllListsServ
let").forward(request, response);


        }
```

Verify that the doPost( ) calls the doGet( ) method:

```
protected void doPost(HttpServletRequest request,
HttpServletResponse response) throws ServletException,
IOException {
            doGet(request, response);
```

Test your code. If the capitalization is correct in your servlet name and links, then you should be able to create a list but receive an error on persistence. This is because the program is persisting the list each time but the id already exists for the existing items. Inside the ListDetails class, change the cascadeType to Merge.

```
@OneToMany(cascade=CascadeType.MERGE, fetch=FetchType.EAGER)
    private List<ListItem> listOfItems;
```

This will allow existing ListItems to just merge back if there are any changes. Any new items (which is currently don't have a way to add) would be added in to the ListItem table.

## Edit A List

Create a servlet called ListNavigationServlet and have the url use what you placed in your shopping-list-by-user.jsp – listnavigationServlet. In the doPost( ), we are going to use the basic structure from our Edit a List Item servlet (NavigationServlet).

```
protected void doPost(HttpServletRequest request,
HttpServletResponse response) throws ServletException,
IOException {
    // TODO Auto-generated method stub
    ListDetailsHelper dao = new ListDetailsHelper();
    String act = request.getParameter("doThisToList");

    if (act == null) {
```

```
          // no button has been selected

     getServletContext().getRequestDispatcher("/viewAllListsServ
let").forward(request, response);

     } else if (act.equals("delete")) {
          try {
               Integer tempId =
Integer.parseInt(request.getParameter("id"));
               ListDetails listToDelete =
dao.searchForListDetailsById(tempId);
               dao.deleteList(listToDelete);

               } catch (NumberFormatException e) {
                    System.out.println("Forgot to click a
button");
               } finally {

     getServletContext().getRequestDispatcher("/viewAllListsServ
let").forward(request, response);
               }

     } else if (act.equals("edit")) {
          try {
               getServletContext().getRequestDispatcher("/edit-
list.jsp").forward(request, response);
               } catch (NumberFormatException e) {
     getServletContext().getRequestDispatcher("/viewAllListsServ
let").forward(request, response);
               }

          } else if (act.equals("add")) {
               getServletContext().getRequestDispatcher("/new-
list.html").forward(request, response);
          }

     }
```

We need to add two helper methods into our ListDetails helper in order for the delete to function:

```
public void deleteList(ListDetails toDelete) {
```

```
     // TODO Auto-generated method stub
     EntityManager em = emfactory.createEntityManager();
     em.getTransaction().begin();
     TypedQuery<ListDetails> typedQuery = em
                     .createQuery("select detail from ListDetails
detail where detail.id = :selectedId", ListDetails.class);

// Substitute parameter with actual data from the toDelete item
     typedQuery.setParameter("selectedId", toDelete.getId());

// we only want one result
     typedQuery.setMaxResults(1);

// get the result and save it into a new list item
     ListDetails result = typedQuery.getSingleResult();

// remove it
     em.remove(result);
     em.getTransaction().commit();
     em.close();

     }

public ListDetails searchForListDetailsById(Integer tempId) {
     EntityManager em = emfactory.createEntityManager();
     em.getTransaction().begin();
     ListDetails found = em.find(ListDetails.class, tempId);
     em.close();
     return found;
     }
```

This should make your deleting of lists work.

### Editing Lists

Editing Lists are more tricky, as you have to check to see if the Shopper already exists and you have to deal with the LocalDate. For simplicity, the lab will pass everything over individually.

In the ListNavigationServlet – if statement for edit, add the following code:

```
} else if (act.equals("edit")) {
try {
```

```
Integer tempId = Integer.parseInt(request.getParameter("id"));
ListDetails listToEdit = dao.searchForListDetailsById(tempId);
request.setAttribute("listToEdit", listToEdit);

request.setAttribute("month",
listToEdit.getTripDate().getMonthValue());
request.setAttribute("date",
listToEdit.getTripDate().getDayOfMonth());
request.setAttribute("year",
listToEdit.getTripDate().getYear());

ListItemHelper daoForItems = new ListItemHelper();

request.setAttribute("allItems", daoForItems.showAllItems());

if(daoForItems.showAllItems().isEmpty()){
     request.setAttribute("allItems", " ");
}

getServletContext().getRequestDispatcher("/edit-
list.jsp").forward(request, response);
     } catch (NumberFormatException e) {

     getServletContext().getRequestDispatcher("/viewAllListsServ
let").forward(request, response);
               }

          } else if (act.equals("add")) {
```

Remember to pay attention to those curly brackets. You cannot just copy and paste it in, as I've also included both of the else statements to show you where all the code goes in relation to the if statements.
You can see from this code that we need an edit-list.jsp to send our user to with all this data. Create an edit-list.jsp and add the following code:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
     <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
```

```
<title>Edit An Existing List</title>
</head>
<body>
<form action = "editListDetailsServlet" method="post">
<input type ="hidden" name = "id" value= "${listToEdit.id}">
List Name: <input type ="text" name = "listName" value=
"${listToEdit.listName}"><br />

Trip date: <input type ="text" name = "month" placeholder="mm"
size="4" value= "${month}"> <input type ="text" name = "day"
placeholder="dd" size="4" value= "${date}">, <input type ="text" name
= "year" placeholder="yyyy" size="4" value= "${year}">

Shopper Name: <input type = "text" name = "shopperName" value=
"${listToEdit.shopper.shopperName}"><br />

Available Items:<br />

<select name="allItemsToAdd" multiple size="6">
<c:forEach items="${requestScope.allItems}" var="currentitem">
   <option value = "${currentitem.id}">${currentitem.store} |
${currentitem.item}</option>
</c:forEach>
</select>
<br />
<input type = "submit" value="Edit List and Add Items">
</form>
<a href = "index.html">Go add new items instead.</a>
</body>
</html>
```

With this code, you should be able to click on a list to edit and it will populate. However, we need to implement the EditListDetailsServlet in order for it to get the data off the page and submit it. Go ahead and do that now with 'editListDetailsServlet' as your path in the WebServlet annotation: `@WebServlet("/editListDetailsServlet")`

In the doPost( ), we'll process the form, update all the fields and finally persist the updates.

```
protected void doPost(HttpServletRequest request,
HttpServletResponse response) throws ServletException,
IOException {

// TODO Auto-generated method stub
ListDetailsHelper dao = new ListDetailsHelper();
```

```
ListItemHelper lih = new ListItemHelper();
ShopperHelper sh = new ShopperHelper();


Integer tempId = Integer.parseInt(request.getParameter("id"));
ListDetails listToUpdate = dao.searchForListDetailsById(tempId);

String newListName = request.getParameter("listName");

String month = request.getParameter("month");
String day = request.getParameter("day");
String year = request.getParameter("year");

String shopperName = request.getParameter("shopperName");
//find our add the new shopper
Shopper newShopper = sh.findShopper(shopperName);

LocalDate ld;
try {
     ld = LocalDate.of(Integer.parseInt(year),
Integer.parseInt(month), Integer.parseInt(day));
} catch (NumberFormatException ex) {
     ld = LocalDate.now();
}

try {
     //items are selected in list to add
     String[] selectedItems =
request.getParameterValues("allItemsToAdd");
     List<ListItem> selectedItemsInList = new
ArrayList<ListItem>();

     for (int i = 0; i < selectedItems.length; i++) {
          System.out.println(selectedItems[i]);
     ListItem c =
lih.searchForItemById(Integer.parseInt(selectedItems[i]));
                    selectedItemsInList.add(c);

     }
          listToUpdate.setListOfItems(selectedItemsInList);
```

```
     } catch (NullPointerException ex) {
          // no items selected in list - set to an empty list
     List<ListItem> selectedItemsInList = new
ArrayList<ListItem>();
     listToUpdate.setListOfItems(selectedItemsInList);
     }

     listToUpdate.setListName(newListName);
     listToUpdate.setTripDate(ld);
     listToUpdate.setShopper(newShopper);

     dao.updateList(listToUpdate);

     getServletContext().getRequestDispatcher("/viewAllListsServ
let").forward(request, response);
     }
```

In order to make this functional, we have two methods we need to add – one in the ListDetailsHelper and one in the ShopperHelper. The ListDetailsHelper needs a method to merge changes into an existing object.

```
     public void updateList(ListDetails toEdit) {
          EntityManager em = emfactory.createEntityManager();
          em.getTransaction().begin();

          em.merge(toEdit);
          em.getTransaction().commit();
          em.close();
     }
```

The ShopperHelper needs the following method:

```
public Shopper findShopper(String nameToLookUp) {

     EntityManager em = emfactory.createEntityManager();
     em.getTransaction().begin();
     TypedQuery<Shopper> typedQuery = em.createQuery("select sh
from Shopper sh where sh.shopperName = :selectedName",
                    Shopper.class);
```

```
typedQuery.setParameter("selectedName", nameToLookUp);
typedQuery.setMaxResults(1);

Shopper foundShopper;
try {
     foundShopper = typedQuery.getSingleResult();
} catch (NoResultException ex) {
     foundShopper = new Shopper(nameToLookUp);
}
em.close();

return foundShopper;
}
```

This method will look up an existing shopper by name. If that name is found, it will return that result. Otherwise, the method will create a new shopper with the name that was passed in via the parameter and return that new shopper. We can use that new shopper (or existing shopper in our ListDetails to update the shopper from our edit form.

Now, kill the cat and reload Tomcat to test it out. The only item that does not work is the lack of the existing list being highlighted in the edit. This is one component that I have not had time to implement but plan to on down the line. If you want to mess around with it and try it, please go ahead! But finish your other work first!

You can check out my code for editing and deleting (as well as the portions in this lab) on GitHub at https://github.com/kjkleindorfer/WebShoppingListwithShoppers