Lesson 8: Functions on Collections

Created by Jacques Marais, last modified on Sep 13, 2018

> *As a farmer, I want to specify for my profile the crops I cultivate.*

Lesson Outcomes

By the end of this lesson you should:

- Know the basics of using collections and collection specific built-in functions
- Know how to create a custom submenu using a data table and non-persistent object
- Be aware of the side effects of removing multiple items from a collection using loops and the remove built-in function

New & Modified App Files

`./model/enums/Enums.mez`

`./model/roles/Farmer.mez`

`./services/UserInvite.mez`

`./utilities/TestData.mez`

`./web-app/images/Admin.png`

`./web-app/lang/en.lang`

`./web-app/presenters/user_managment/FarmerUserMgmt.mez`

`./web-app/presenters/enity_management/ShopMgmt.mez`

`./web-app/presenters/user_managment/ShopOwnerUserMgmt.mez`

`./web-app/presenters/user_managment/SystemAdminUserMgmt.mez`

`./web-app/presenters/enity_management/StockMgmt.mez`

`./web-app/views/user_management/FarmerUserDetails.vxml`

`./web-app/views/user_management/FarmerUserMgmt.vxml`

`./web-app/views/entity_management/ShopMgmt.vxml`

`./web-app/views/user_management/ShopOwnerUserMgmt.vxml`

`./web-app/views/user_management/SystemAdminUserMgmt.vxml`

`./web-app/views/entity_management/StockMgmt.vxml`

App Folder Structure Refactor

Our app is now large enough to consider revisiting the folder structure in order to clean up the project. Note that Helium requires a basic top level folder structure as highlighted in here. There is, however, no requirement on the subfolders to be used. We will take the following steps to clean up our project:

- Sort model files into separate folders for objects, roles, enums and validators
- Views and presenters wil be sorted by function

Submenu Using Non-Persistent Objects

To further clean up our project and to simplify the user interface we will add submenus for user management and entity management. Although this section only highlights the most important components of this, the full example can be seen in the source code for this lesson. Note that Helium does not natively support submenus. We use the following components to implement a custom submenu:

- A non persistent object to represent a generic menu item
- A view with a data table representing the menu
- Row actions for selecting items from the table and subsequently navigating to the desired views

The code snippets below shows these components as used for the user management submenu:

MenuItem object

```
 object MenuItem {
    string label;
    string description;
    DSL_VIEWS navigationDestination;
}
```

UserMgmtMenu presenter

```

 unit UserMgmtMenu;

// Variable to mind on from 'select' row action
MenuItem menuItem;

// Menu table collection source
MenuItem[] menuItems;

// Create the menu items for our menu
void initMenu() {
    menuItems.clear();

    // Menu item for System Admin user management
    MenuItem systemAdminMenuItem = createMenuItem(
        String:translate("menu_item.system_admin_user_
        DSL_VIEWS.SystemAdminUserMgmt,
        countSystemAdmins()
    );
    menuItems.append(systemAdminMenuItem);

    // Menu item for Shop Owner user management
    MenuItem shopOwnerMenuItem = createMenuItem(
        String:translate("menu_item.shop_owner_user_ma
        DSL_VIEWS.ShopOwnerUserMgmt,
        countShopOwners()
    );
    menuItems.append(shopOwnerMenuItem);

    // Menu item for Farmer user management
    MenuItem farmerMenuItem = createMenuItem(
        String:translate("menu_item.farmer_user_manage
        DSL_VIEWS.FarmerUserMgmt,
        countFarmers()
    );
    menuItems.append(farmerMenuItem);
}

// Navigate to the view associated with the select mer
DSL_VIEWS selectMenuItem() {
    return menuItem.navigationDestination;
}
.
.
.
```

Note the use of the `clear` and `append` built-in functions above. These operate on collections to clear the collection and append a single item to a collection respectively. The append function is demonstrated further later in this lesson.

UserMgmtMenu view

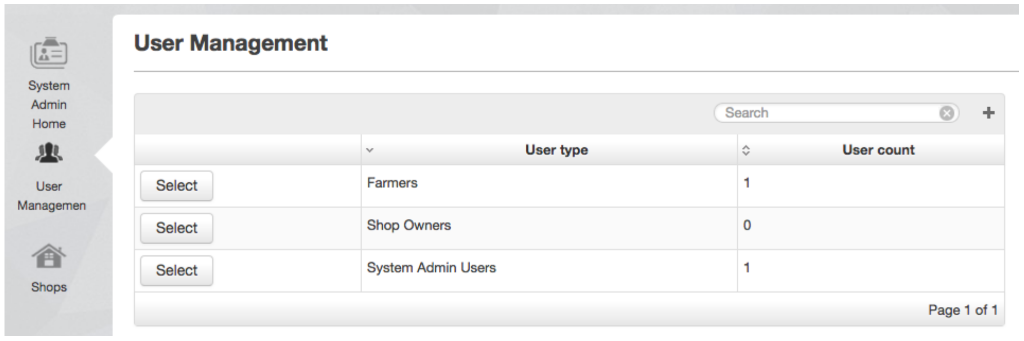
```

 <?xml version="1.0" encoding="UTF-8"?>
<ui xmlns="http://uiprogram.mezzanine.com/View"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instar
    xsi:schemaLocation="http://uiprogram.mezzanine.com
    <view label="view_heading.user_management" unit="U

        <menuitem label="menu_item.user_management" ic
            <userRole>System Admin</userRole>
        </menuitem>

        <table csvExport="disabled">
            <collectionSource variable="menuItems"/>
            <column heading="column_heading.user_type"
                <attributeName>label</attributeName>
            </column>
            <column heading="column_heading.user_count
                <attributeName>description</attributeN
            </column>

            <rowAction label="button.select" action="s
                <binding variable="menuItem"/>
            </rowAction>
        </table>
    </view>
</ui>
```



User Management

	User type	User count
Select	Farmers	1
Select	Shop Owners	0
Select	System Admin Users	1

Page 1 of 1

App Features for the Benefit of Development Velocity

In order to link farmers to crop types, we will have need for a `Stock` persistent object, meant for maintaining a list of all types of crop seeds and fertilizers available in the app.

```
persistent object Stock {
    string name;
    STOCK_TYPE stockType;
}
```

```
enum STOCK_TYPE {
    fertilizer,
    crop_seed
}
```

Look out for scenarios like this early in the development life cycle, even during the initial planning stage. Writing code that bypass manual data entry for testing purposes will likely save you time in the long run.

We will provide the system admin with the frontend functionality to add, edit and remove stock types. Seeing as this type of functionality has already been demonstrated in the tutorial, we will not discuss it again here. In brief, we will create a new view and unit, with the view containing a table of stock items already entered into the system. When later uploading a CSV containing stock updates, we will be able to link each CSV-parsed record with one of these stock items.

However, this means that each time we deploy a new sandbox of our app, the steps to manually add the different stock items will need to be repeated before this feature will work. This can be tedious and time consuming. It is common to add code to DSL apps that quickly generates the needed data for any features we might want to repeatedly test. We'll add a function in a unit under utilities which does the following:

```
void generateStockItems() {
   if (stockExists() == false) {
      string[] stockNames = getStockNames();
      for (int i = 0; i < stockNames.length(); i++) {
         Stock stock = Stock:new();
         stock.name = stockNames.get(i);
         stock.stockType = getStockType(stockNames.get(i));
         stock.save();
      }
   }
}
```

See the attached source code for the omitted functions, if necessary.

Currently we have a few options for when to execute this function. The easiest is to add the function call to our `inviteSystemAdmin()` function which has the **`@InviteUser`** annotation, meaning it will execute each time we deploy a sandbox using the HeliumDev client. Inspect the aforementioned view to check that the function executed.

Many to Many Relationship

The final model change required for this lesson is to add a relationship between the `Stock` object and the `Farmer` object. Each farmer cultivates multiple crops and crop may be cultivated by more than one farmer. For this reason we add a many to many relationship. Seeing as we will mostly reference the relationship from the `Farmer` object's side, we will add the relationship on the `Farmer` object:

```
@ManyToMany
Stock cropTypes via farmers;
```

Functions on Collections

The Helium DSL provides many functions for manipulating collections. These are listed in the Quick Reference. Although we briefly showed the append and remove functions as part of Lesson 6 we will again discuss them here and in more detail seeing as they are most frequently used.

Functions on Relationship Collections

In this lesson we discuss the use of functions on collections. This is, however, demonstrated using a collection that represents a relationship. When performing operations on collections that represent underlying relationships, the relationship is also affected even though the relationship collection is assigned to a different in memory collection. For example removing an item from such a collection also removes that particular relationship between the two object instances in question.

Performing operations of collections that are populated from underlying relationships, also affects the underlying relationship.

Collection Sources For Farmer Crops

As with setting the relationship between shop owners and shops, we will use a table that shows which crop types have been linked to the farmer, a select box that lists eligible crop types that can still be linked to the farmer and a submit button to link the result of the select box to the farmer. The components are added to a new `FarmerProfile` view and unit with a menu item on the view for the "`Farmer"` role. The code snippets below show the collection source function for the table and collection source function for the select box:

```
Stock[] getFarmerCropTypes() {
    return farmer.cropTypes;
}
```

```
Stock[] getEligibleCropTypes() {
    return Stock:diff(
        equals(stockType, STOCK_TYPE.crop_seed),
        relationshipIn(farmers, farmer)
    );
}
```

Switching to Another User Role

Now that we have a view specifically for the "`Farmer"` role, namely `FarmerProfile`, we can test the application as a farmer user to see the new view. One way to achieve this is to invite yourself to the application as a Farmer user from the app itself.

- Using the farmer management functionality introduced in Lesson 8, create a new farmer.
- Be sure to use your mobile number for the new farmer user. In other words, the mobile number that was used to sign up to Helium and the same number being used when inviting yourself to the application as an administrator user.
- After refreshing the page you should now be able to select "Farmer" from the popup that appears when hovering over the profile picture in the top right corner of the view. All roles to which you have been invited for the app will appear here.

Other ways to invite yourself to an app as multiple roles include:

- Automatically inviting yourself when deploying snapshot apps from the HeliumDev client: This is achieved by adding invite user functions. Each role in the application can have an accompanying invite user function. In addition you'll need to set the roles value for the project details in the HeliumDev client to include all roles for which you need to be invited. The roles need to be specified as a comma separated list. More details on this can be found here and here.
- Inviting yourself to an app from the Helium core web app: Using this method you will be able to invite yourself to an application for any role that also has an invite user function. This is only applicable when working with non-snapshot apps. More details on this can be found here.

Append on Collections

Appending to a collection works by simply using the append function and providing the object instance to append as an argument.

```
// Link the selected crop type to the farmer
string addCropType() {
    farmer.cropTypes.append(cropType);
    return null;
}
```

Note that the following could also have used to link the farmer and crop types despite appending to an intermediate collection instead of the origin relationship collection.

```
 // Link the selected crop type to the farmer
string addCropType() {
    Stock[] cropTypes = farmer.cropTypes;
    cropTypes.append(cropType);
    return null;
}
```

Remove on Collections

Removing from an arbitrary position in a collection by selecting a specific object instance and using the remove function is slightly more complex. The reason being that remove needs the index of the item to remove from the collection while the row action binds to an object instance and does not specify a line number. We therefore loop over the collection and use the internal object ids to find the index of the object instance to remove. Note also the use of the `length` function which provides a count of the number of items in a collection. This allows a for loop to be used to iterate over the crop types for a farmer.

```
// Unlink the selected crop type from the farmer
string removeFarmerCropType() {
    Stock[] farmerCropTypes = farmer.cropTypes;
    for(int i = 0; i < farmerCropTypes.length(); i++)
        Stock currentFarmerCropType = farmerCropTypes.
        if(currentFarmerCropType._id == cropType._id)
            farmerCropTypes.remove(i);
        }
    }
    cropType = null;
    return null;
}
```

Looping to Remove Multiple Items From a Collection

When looping over a collection to remove multiple items special caution needs be exercised. With the exception of the all selector, collections are populated based on certain criteria. If any object in that collection changes in such a way that it no longer satisfies that criteria, it will immediately and automatically be removed from the collection. Thus, when making changes to objects in a collection while looping over them, it is possible that the collection is altered while looping over it. This will result in some objects being skipped while it may not have been the intention of the developer. As a workaround to this, collections can be iterated over from back to front. This will allow modifications to the objects in the collection with the guarantee that all objects will be visited. The example below shows a function that is used to remove all crop types from a farmer:

```
// Unlink all crop types from the farmer
string removeAllFarmerCropTypes() {
    Stock[] farmerCropTypes = farmer.cropTypes;
    for(int i = farmerCropTypes.length() - 1; i >= 0; i
        farmerCropTypes.remove(i);
    }
    return null;
}
```

Lesson Source Code

Lesson 8.zip

No labels