Lesson 10: Scheduled Functions, Flow Control, Basic Messaging

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

> *As a Farmer I want to set user preferences and receive stock level notification messages accordingly.*

Lesson Outcomes

By the end of this lesson you should:

- Be able to populate **`boolean`** variables from views using the checkbox input widget
- Be able to create scheduled functions that execute on the Helium backend
- Be able to use Helium built-in functions to send basic E-mail messages
- Be able to use Helium built-in function to send SMS messages

New & Modified App Files

`./model/objects/FarmerStockNotificationEmail.mez`

`./model/objects/FarmerStockNotificationSms.mez`

`./model/roles/Farmer.mez`

`./services/ScheduledMessaging.mez`

`./web-app/presenters/farmer_profile/FarmerProfile.mez`

`./web-app/presenters/farmer_profile/FarmerProfileMenu.mez`

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

`./web-app/views/farmer_profile/FarmerProfile.vxml`

`./web-app/views/farmer_profile/FarmerProfileCropTypes.vxml`

`./web-app/views/farmer_profile/FarmerProfileMenu.vxml`

`./web-app/views/farmer_profile/FarmerProfileMessaging.vxml`

Model Additions

As a first step we must once again make additions to our data model. The following changes will be added:

- Attributes of type **`boolean`** on our `Farmer` object to indicate whether the farmer has opted in for e-mail and SMS messages.

- An additional relationship on the `Farmer` object to indicate the shop that the farmer might be interested in receiving stock level notifications for.
- Attributes of type **`datetime`** on our `Farmer` object to keep track of when last the messaging preferences or crop types of the farmer were updated. The detailed usage of these attributes won't be discussed here. For details of their usage, the lesson source code can be used as reference.
- Two additional objects to keep track of e-mail and SMS stock level notification messages sent to farmers. These objects will be linked to a stock update and to the farmer recipient.

Additional farmer attributes and relationship

```
@Role("Farmer")
persistent object Farmer {
    .
    .
    bool smsMessaging;
    bool emailMessaging;
    .
    .
    datetime cropTypeProfileUpdatedOn;
    datetime messagingProfileUpdatedOn;
    .
    .
    @ManyToMany
    Shop messagingShops via notifiedFarmers;
    .
    .
}
```

E-mail stock notification object

```
persistent object FarmerStockNotificationEmail {
    datetime sentOn;

    @ManyToOne
    StockUpdate stockUpdate via stockUpdateEmails;

    @ManyToOne
    Farmer farmer via farmerStockUpdateEmails;
}
```

SMS stock notification object

```

persistent object FarmerStockNotificationSms {
    datetime sentOn;

    @ManyToOne
    StockUpdate stockUpdate via stockUpdateSmses;

    @ManyToOne
    Farmer farmer via farmerStockUpdateSmses;
}
```

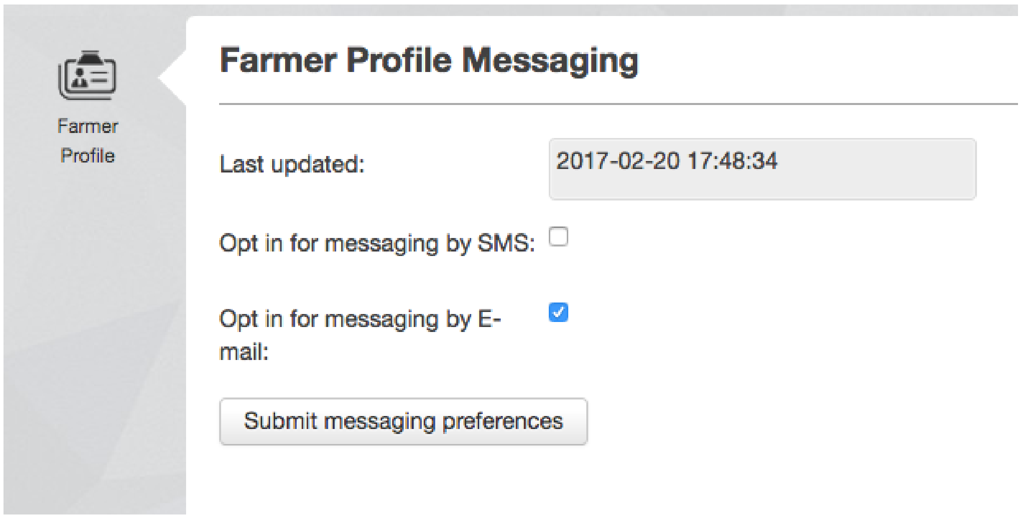
Populating Boolean Attributes From a Form

In previous lessons we have used **`boolean`** values in our model that were populated using units. In this case, we want the farmer user to specify the value for the attributes. The following code snippets were extracted from the `FarmerProfileMessaging` view:

```
<checkbox label="checkbox.opt_in_sms_messaging">
    <binding variable="farmer">
        <attribute name="smsMessaging"/>
    </binding>
</checkbox>

<checkbox label="checkbox.opt_in_email_messaging">
    <binding variable="farmer">
        <attribute name="emailMessaging"/>
    </binding>
</checkbox>
```

On the frontend check boxes appear as follows:



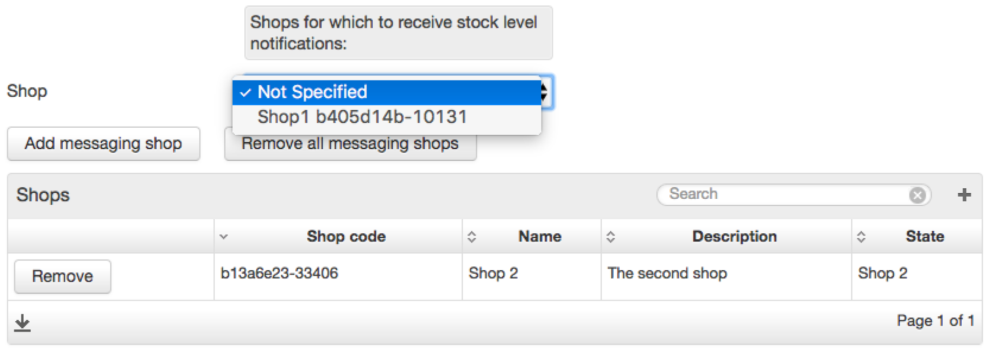
Farmer Profile Messaging

Last updated: 2017-02-20 17:48:34

Opt in for messaging by SMS: ☐

Opt in for messaging by E-mail: ✓

In addition to the above we have once again added a select box, submit button and data table in order for farmers to select shops that they are interested in. These view components are hidden and become visible as soon as one of the boolean attributes mentioned above is submitted as true:



Shops for which to receive stock level notifications:

Shop	✓ Not Specified / Shop1 b405d14b-10131			

	Shop code	Name	Description	State
Remove	b13a6e23-33406	Shop 2	The second shop	Shop 2

Scheduled Functions

Helium provides functionality for executing functions on the backend using a timer. These are called scheduled functions and they are denoted by the use of the **@Schedule** annotation. The schedule is specified using cron like syntax. For example:

```
// Scheduled function to run every day at 2:15 AM
@Scheduled("15 2 * * *")
```

The schedule is therefore represented as follows:

`@Scheduled("<minute> <hour> <day of month> <month> <day of week>"`

For our use case we would like to message farmers once a week at 08:00 AM on Monday morning. Messages will be sent only to farmers that have opted in for messaging, have selected shops to receive messages for and are linked to crop types which have stock updates at the shops that they have selected. The scheduled function was added to a new unit called `ScheduledMessaging` under the services folder and main logic is shown below:

```

@Scheduled("0 8 * * 0")
void sendFarmerStockNotificationMessages() {

    // Get farmers that have not been deleted and have
    Farmer[] farmers = Farmer:union(
        and(
            equals(deleted, false),
            equals(smsMessaging, true)
        ),
        and(
            equals(deleted, false),
            equals(emailMessaging, true)
        )
    );

    // Loop over farmers that are eligible for message
    for(int i = 0; i < farmers.length(); i++) {
        Farmer farmer = farmers.get(i);

        // Get the stock updates for shops that the fa
```

Note that Helium also provides a **foreach** loop that can be used when iterating over collections. For more information see the following:

Quick Reference

DSL References

```
22          StockUpdate[] stockUpdates = StockUpdate:relat
23
24          // For each farmer crop type filter the stock
25          for(int j = 0; j < farmer.cropTypes.length();
26
27              Stock currentCropType = farmer.cropTypes.g
28              StockUpdate[] cropSpecificUpdates = stockU
29                  relationshipIn(stock, currentCropType)
30              );
31
32              cropSpecificUpdates.sortDesc("stocktakeDat
33              messageFarmer(farmer, cropSpecificUpdates.
34          }
35      }
36  }
37
38  // Determine which messages should be sent and invoke
39  void messageFarmer(Farmer farmer, StockUpdate stockUpd
40      if(farmer.smsMessaging == true) {
41          smsFarmer(farmer, stockUpdate);
42      }
43
44      if(farmer.emailMessaging == true) {
45          emailFarmer(farmer, stockUpdate);
46      }
    }
```

Note the use of the `union` selector which is used to combine the result of two collections without duplicates. In this case the union acts as an or condition for farmers that have either opted in for SMS messaging, e-mail messaging or both. Also note the use of the select built-in function on line 27. This allows selectors to be executed against collections in memory instead of directly against the database.

Finally, take note of the fact that while a traditional for loop, with a variable to keep track of iterations, is used above, Helium also provides a **`foreach`** loop that can be used when iterating over a collection. Consider the loop above that loops over the `farmer` collection. This can be replaced by the following:

```
foreach(Farmer farmer: farmers) {
    .
    .
    .
}
```

Sending an SMS

Helium provides functionality to send SMS messages using a single built-in function. For our use case, we create and save an instance of the

Note that some configuration is required in order for apps to be enabled to send SMS

`FarmerStockNotificationSms` object for the purpose of historic record keeping. We then invoke the `Mez:sms` function. The code snippet below demonstrates this:

```
void smsFarmer(Farmer farmer, StockUpdate stockUpdate)

    // Build the message content using concat
    Shop shop = stockUpdate.shop;
    string messageContent = Strings:concat("Stock leve

    FarmerStockNotificationSms notificationSms = Farme
    notificationSms.sentOn = Mez:now();
    notificationSms.stockUpdate = stockUpdate;
    notificationSms.farmer = farmer;
    notificationSms.messageContent = messageContent;
    notificationSms.save();

    Send SMS and reference message content using a lar
    Mez:sms(farmer, "mobileNumber", "messaging.sms.mes
}
```

Note the arguments required for the `Mez:sms` function. The first is the object instance representing the recipient of the message. This can, however, be any object containing a mobile number field and does not need to be a role. The second argument represents the attribute on this object representing the mobile number of the recipient and, lastly, the third argument represents the key of a lang file entry that contains the message content. In this case the lang file entry simply references a function scoped variable.

```
messaging.sms.message_content = {messageContent}
```

messages. Although this configuration will not be needed for the tutorial app, it might be necessary for future apps that you develop. In order to request this configuration, follow the process as described here.

Be aware that Helium validates mobile numbers and will not attempt to send a message to an invalid mobile number.

Sending an E-mail

Similar to sending SMS messages, we also keep track of e-mail stock level notifications using the `FarmerStockNotificationEmail` object. To send the actual e-mail we will be using the `Mez:email` function. There are three versions of this function available. Two of these are demonstrated below while a third, that can be used to send Jasper reports as attachments is discussed in Lesson 13.

E-mails in Helium can also be used to send Jasper report attachments. This is discussed in lesson 13.

Mez:email to the Helium user/identity

```
Mez:email(farmer, "messaging.email.stock_level_descrip
```

The above function uses the e-mail address that has been captured as part of the user's Helium profile. Seeing as the e-mail address is not required for a user's profile to be created, we recommend using the function as follows:

Mez:email to e-mail address directly

```
Mez:email(farmer.emailAddress, "messaging.email.stock_
```

In this case the e-mail address that was captured in the application as part of creating the farmer is used. This is a required field in our app and we are therefore guaranteed that it will be populated. For our use case, and many others, this is a much safer option.

For the above function calls the following lang file entries were added:

```
messaging.email.stock_level_description = Stock level r
messaging.email.stock_level_subject = Stock level
messaging.email.stock_level_body = {messageContent}
```

A Note On Complex Selectors

If possible, use complex selectors instead of multiple simple selectors to populate collections.

In this lesson we briefly discussed a selector that uses a union selector, with nested and selectors and multiple equals selectors nested within the and selectors.

```
// Get farmers that have not been deleted and have sig
Farmer[] farmers = Farmer:union(
    and(
        equals(deleted, false),
        equals(smsMessaging, true)
    ),
    and(
        equals(deleted, false),
        equals(emailMessaging, true)
    )
);
```

These types of selectors are referred to as complex selectors.

It is possible, and in some cases necessary, to resort to simple selectors, loops and if statements to populate collections. For many use cases, however, it is possible to make use of complex selectors such as the one above.

Whenever possible, a single complex selector should always be preferred above multiple simple selectors. This is due to the fact that Helium will automatically optimise the backing data base queries for selectors. Using complex selectors instead of multiple simple intermediate selectors might therefore provide a significant performance advantage.

Lesson Source Code

Lesson 10.zip

No labels