Lesson 6: Relationships

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

> *As a System Admin I want to invite, view and edit Shop Owners and link them to a shop.*

- Lesson Outcomes
- New & Modified App Files
- Data Model Additions
- Multiplicity
- One to One Relationships
- One to Many Relationships
- Many to Many Relationships
- Self Referencing Relationships
- Lesson Source Code

Lesson Outcomes

By the end of this lesson you should:

- Be able to declare relationships between objects
- Understand multiplicity and support for different relationships in Helium
- Be able to set and reference relationships
- Make use of the relationshipIn selector

New & Modified App Files

`./model/Shop.mez`

`./model/ShopOwner.mez`

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

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

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

`./web-app/views/ShopDetails.vxml`

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

`./web-app/views/ShopOwnerDetails.vxml`

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

Data Model Additions

In order for us to demonstrate relationships between objects, we will add an additional user role, namely, a `"Shop Owner"`. In addition to the object and role we also add CRUD functionality for the `ShopOwner` object. Seeing as this type of functionality has already been demonstrated in the tutorial, we will not discuss it again here. The source code for this lesson can be used a reference for this.

Multiplicity

Multiplicity refers to how few and how many of one entity can be connected to another entity where the entities in this case are shop and shop owners. Helium supports the following:

- One to One
- Many to One
- One to Many
- Many to Many

One to One Relationships

For a one to one relationship each shop has only one owner and an owner only has one shop. To demonstrate this we need to declare the relationship on either the `Shop` or `ShopOwner` model object. In this example we will add the relationship to the `Shop` object.

```
persistent object Shop {

    .
    .
    .

    // Related shop owner
    @OneToOne
    ShopOwner owner via shop;
}
```

As can be seen above, the relationship is declared using the **`@OneToOne`** annotation. This is followed by the type of the related object and the names that will be used to reference the relationship. In this case `owner` will be used for reference from the `Shop` object and `shop` will be used as reference from the `ShopOwner` object. See the example functions below as a demonstration of this.

```
ShopOwner getShopOwner(Shop shop) {
    return shop.owner;
}
```

```
Shop getShop(ShopOwner shopOwner) {
    return shopOwner.shop;
}
```

In addition to the declaration of the relationship, we will also make changes to the `ShopMgmt` view and `ShopMgmt` unit to set and display the relationship. For the view we need to add a select box:

```
<select label="select.shop_owner">
    <binding variable="shop">
        <attribute name="owner"/>
    </binding>
    <collectionSource function="getShopOwners">
        <displayAttribute name="firstName"/>
        <displayAttribute name="lastName"/>
    </collectionSource>
</select>
```

Note the similarity between the select box used above and the select box covered in Lesson 5. Both binds the selected value to a variable but whereas the previous example specified a custom enum as the source for the dropdown, we now specify a collection. With this we then also need to specify the attributes for the collection objects that will be displayed. In this case we selected attributes representing the first and last name of the shop owner. Also take note that validation using validators on the model is not supported for relationships. For this reason manual validation needs to be done using an **if** statement and a popup message in the `saveShop` functions:

Manual validation using an if statement and alert is required as model validators are not supported for relationships.

```
if(shop.owner == null) {
    Mez:alertError("alert.shop_owner_required");
    return null;
}
```

We now add a column to the shop table:

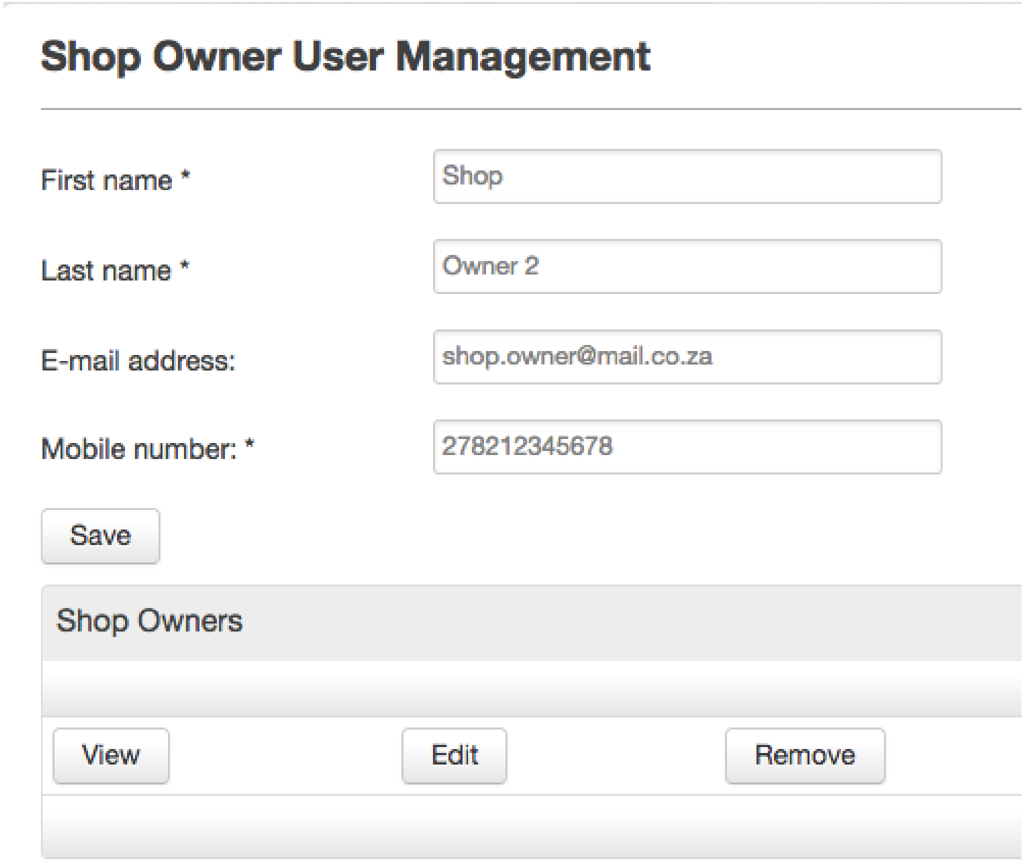
```
<column heading="column_heading.shop_owner">
    <attributeName>owner.firstName</attributeName>
    <attributeName>owner.lastName</attributeName>
</column>
```

Note that we can reference attributes of the related `ShopOwner` object by first referencing the relationship and then the attributes in the related object.

The final addition that we will make to demonstrate the one to one relationship is info widgets on the ShopOwnerDetail view:

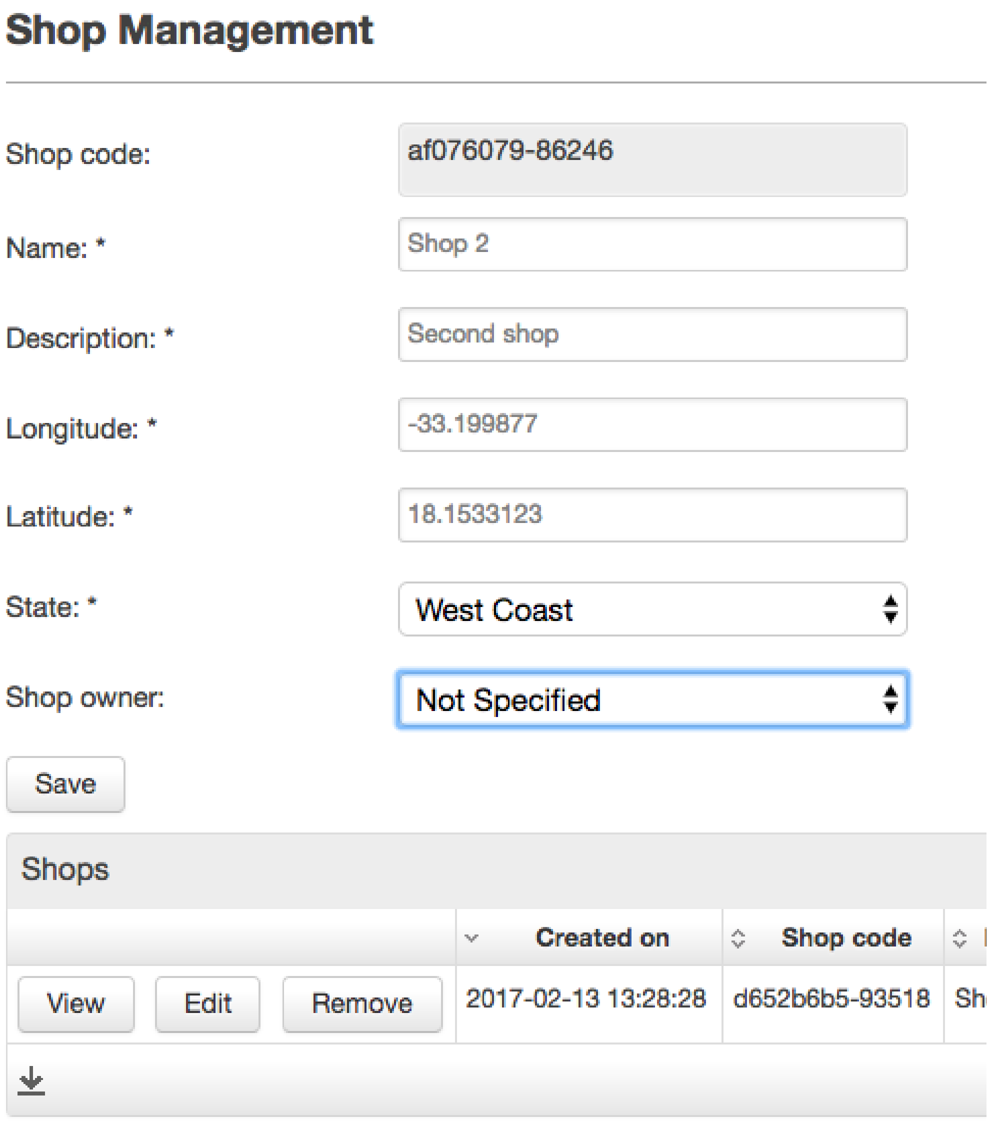
```

<info label="info.shop_owner_firstname">
    <binding variable="shop">
        <attribute name="owner.firstName"/>
    </binding>
</info>
<info label="info.shop_owner_lastname">
    <binding variable="shop">
        <attribute name="owner.lastName"/>
    </binding>
</info>

<info label="info.shop_owner_mobile_number">
    <binding variable="shop">
        <attribute name="owner.mobileNumber"/>
    </binding>
</info>

<info label="info.shop_owner_email_address">
    <binding variable="shop">
        <attribute name="owner.emailAddress"/>
    </binding>
</info>
```

The screenshots below further illustrates the additions up to this point.



Shop Owner User Management

First name *	Shop
Last name *	Owner 2
E-mail address:	shop.owner@mail.co.za
Mobile number: *	278212345678

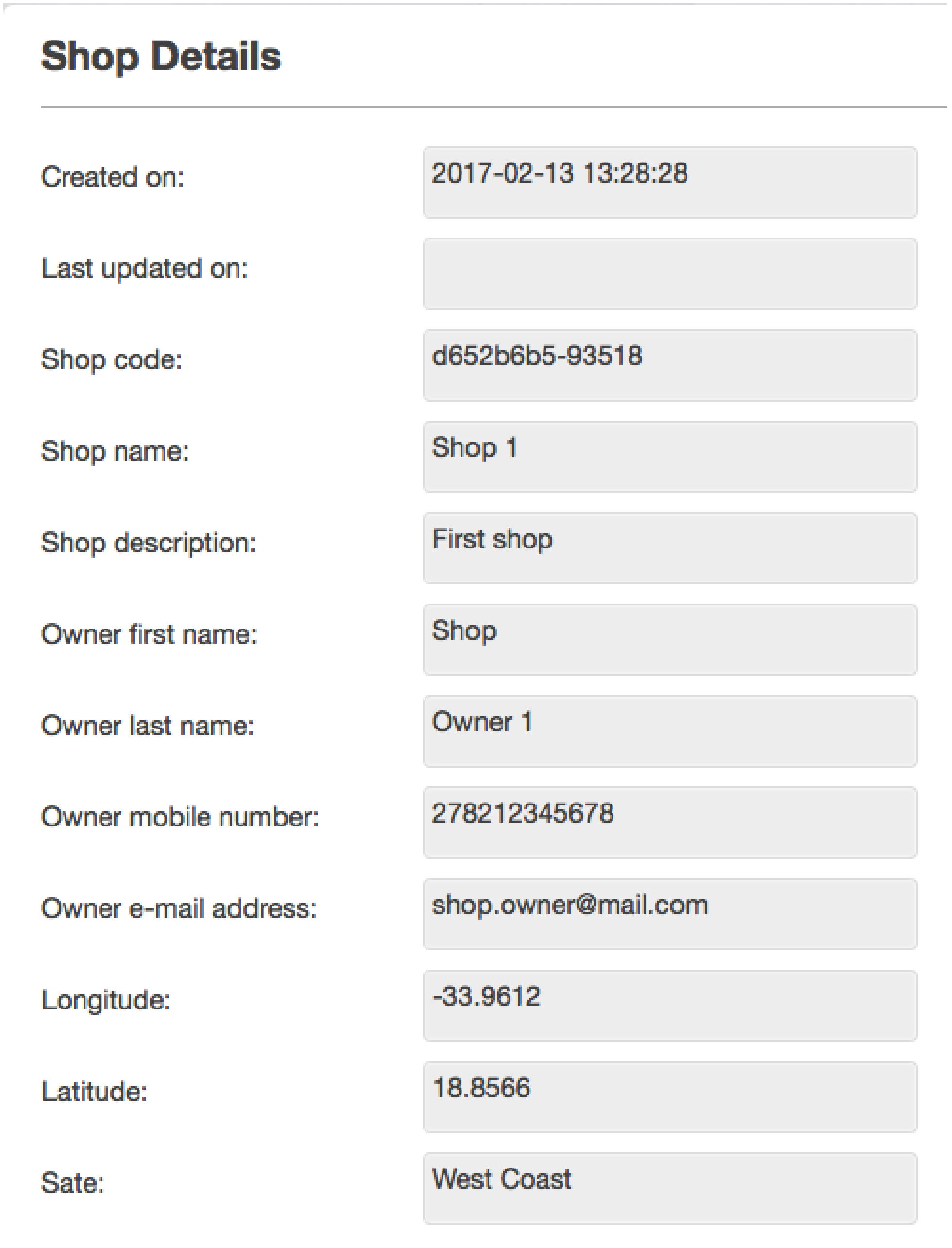


Shop Management

Shop code:	af076079-86246
Name: *	Shop 2
Description: *	Second shop
Longitude: *	-33.199877
Latitude: *	18.1533123
State: *	West Coast
Shop owner:	Not Specified

Shops

	Created on	Shop code	
View Edit Remove	2017-02-13 13:28:28	d652b6b5-93518	Sh



Shop Details

Created on:	2017-02-13 13:28:28
Last updated on:	
Shop code:	d652b6b5-93518
Shop name:	Shop 1
Shop description:	First shop
Owner first name:	Shop
Owner last name:	Owner 1
Owner mobile number:	278212345678
Owner e-mail address:	shop.owner@mail.com
Longitude:	-33.9612
Latitude:	18.8566
Sate:	West Coast

One to Many Relationships

For a many to one relationship each shop can have multiple owners. We will keep the relationship on the `Shop` object. We will, however, change the multiplicity and rename the relationship accordingly.

Seeing as each shop can now have more than one owner, a select box alone is not enough. In addition to a select box, we will also need a button to confirm the linking of a shop owner with a shop and a table to display the selected owners of a shop. We will include a similar table on the `ShopDetails` view.

First we need to make the model changes:

```

persistent object Shop {
    .
    .
    .
    @OneToMany
    ShopOwner owners via shop;
}
```

If we were to place the relationship on the `ShopOwner` object at this point, we would have used the **`@ManyToOne`** annotation instead.

We can then replace the single select box on the `ShopMgmt` view with a select box, submit button and table as follows:

```
 <select label="select.shop_owner">
    <binding variable="ownerToAdd"/>
    <collectionSource function="getAllShopOwners">
        <displayAttribute name="firstName"/>
        <displayAttribute name="lastName"/>
    </collectionSource>
</select>

<submit label="submit.add_shop_owner" action="addShopO

<table title="table_title.shop_owners">
    <collectionSource function="getCurrentShopOwners"/
    <column heading="column_heading.name">
        <attributeName>firstName</attributeName>
        <attributeName>lastName</attributeName>
    </column>
    <rowAction label="button.remove" action="removeSho
        <binding variable="ownerToRemove"/>
    </rowAction>
</table>
```

The backing functions for these view components are:

```

// Return all shop owners in the system
ShopOwner[] getAllShopOwners() {
    return ShopOwner:equals(deleted, false);
}
// Return the shop owners that are linked to the curre
ShopOwner[] getCurrentShopOwners() {
    return shop.owners;
}
string addShopOwner() {
    shop.owners.append(ownerToAdd);
    ownerToAdd = null;
    return null;
}
string removeShopOwner() {
    ShopOwner[] shopOwners = getAllShopOwners();
```

On the "One" side of a relationship, it can be set using a normal assignment operator or binding from select box widget and cleared by setting it to null. On the "Many" side of a relationship it can be treated as a collection where append can be used to set the relationship and remove to clear it.

```
    for(int i = 0; i < shopOwners.length(); i++) {
        ShopOwner currentShopOwner = shopOwners.get(i)
        if(ownerToRemove._id == currentShopOwner._id)
            shop.owners.remove(i);
        }
    }
    return null;
}
```

We can alternatively use the `relationshipIn` selector in the `getCurrentShopOwners` function to check if a relationship is set to a specific value or collection of values. In the example below the first argument is the name of the relationship and the second either an object instance or collection of object instances :

```
ShopOwner[] getCurrentShopOwners() {
    return ShopOwner:relationshipIn(shop, shop);
}
```

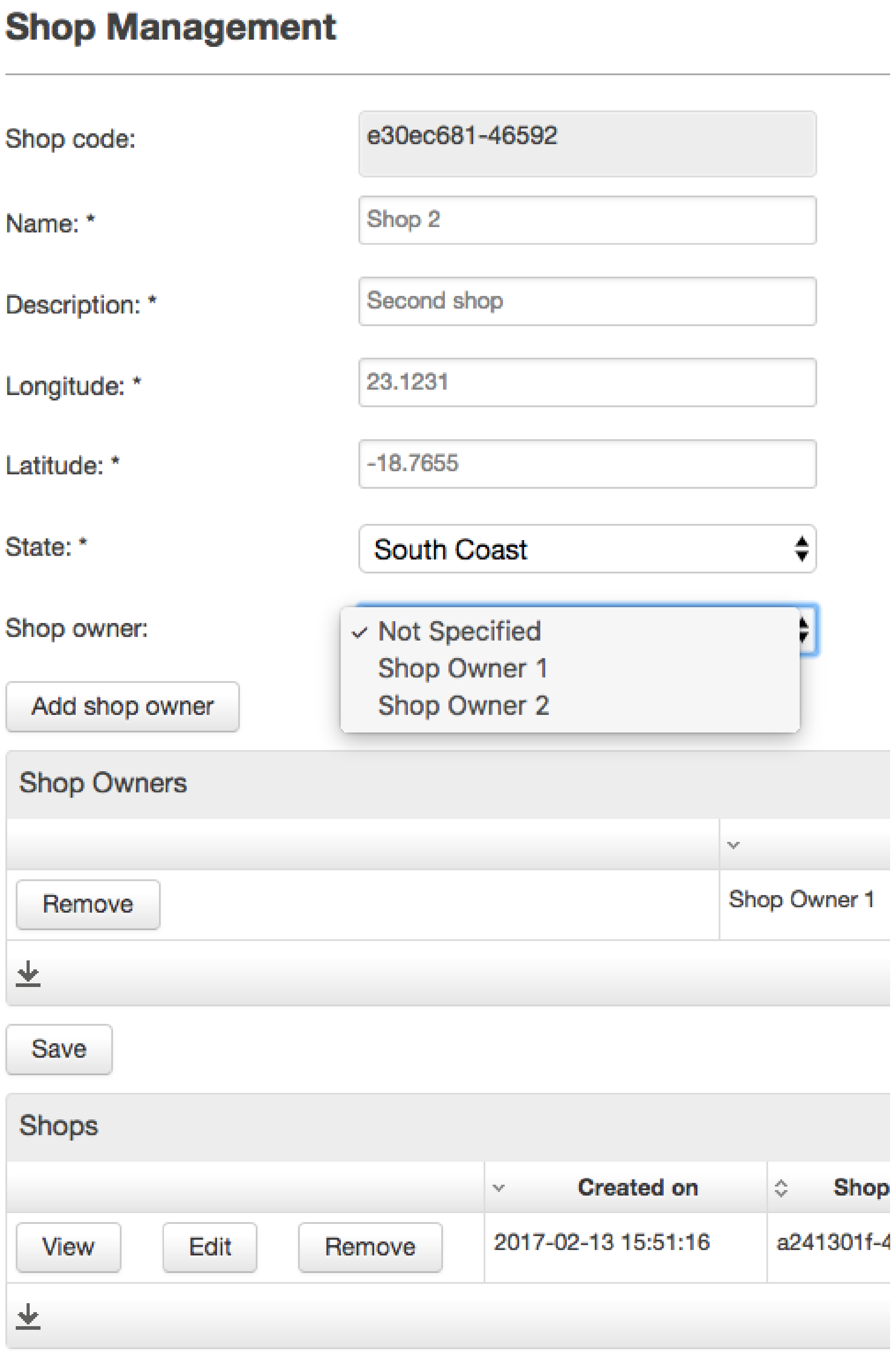
We can now also replace the info widgets on the `ShopDetails` view with a similar table to display the details of the owners of the shop:

```
 <table title="table_title.shop_owners">
    <collectionSource function="getCurrentShopOwners"/
    <column heading="column_heading.first_name">
        <attributeName>firstName</attributeName>
    </column>
    <column heading="column_heading.last_name">
        <attributeName>lastName</attributeName>
    </column>
    <column heading="column_heading.mobile_number">
        <attributeName>mobileNumber</attributeName>
    </column>
    <column heading="column_heading.email_address">
        <attributeName>emailAddress</attributeName>
    </column>
</table>
```

For completeness' sake we also add a info widget on the `ShopOwnerDetails` view to display the name of the shop of the selected owner:

```
 <info label="info.shop">
    <binding variable="shopOwner">
        <attribute name="shop.name"/>
    </binding>
</info>
```

The screenshots below further illustrates the additions up to this point.



Shop Management

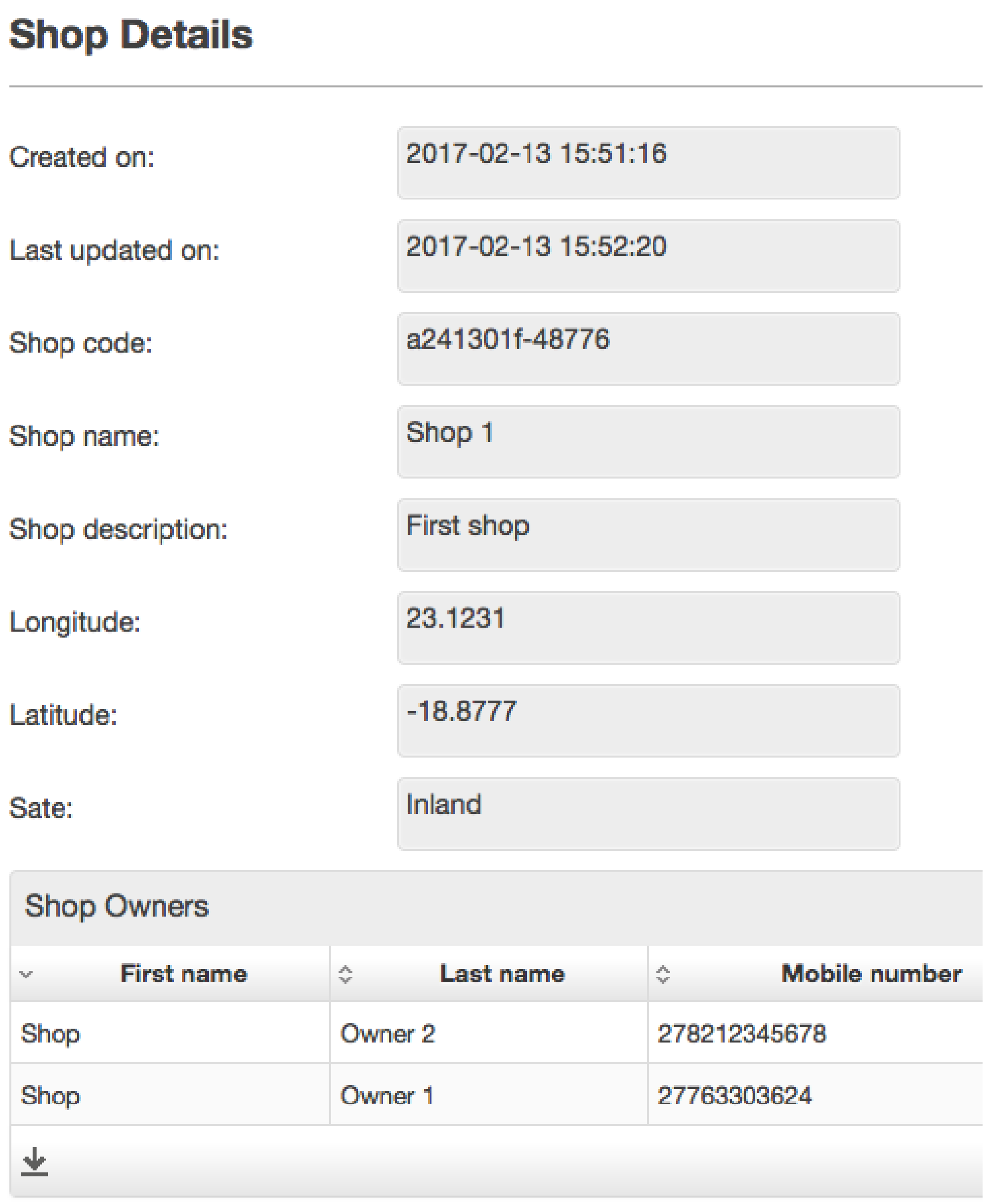
Shop code:	e30ec681-46592
Name: *	Shop 2
Description: *	Second shop
Longitude: *	23.1231
Latitude: *	-18.7655
State: *	South Coast
Shop owner:	[✓ Not Specified] Shop Owner 1 Shop Owner 2

Shop Owners

Remove	Shop Owner 1

Shops

	Created on	Shop
View Edit Remove	2017-02-13 15:51:16	a241301f-4

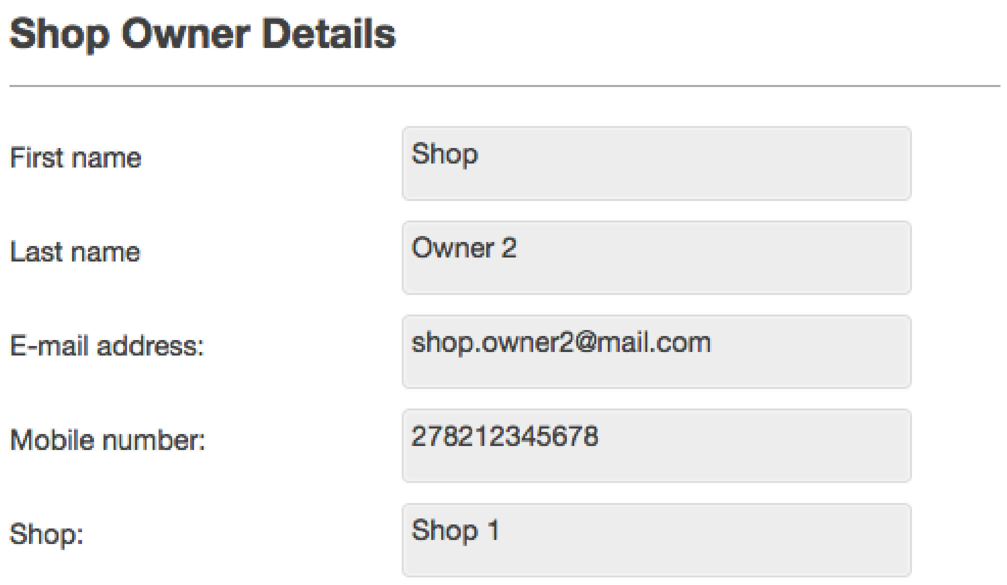


Shop Details

Created on:	2017-02-13 15:51:16
Last updated on:	2017-02-13 15:52:20
Shop code:	a241301f-48776
Shop name:	Shop 1
Shop description:	First shop
Longitude:	23.1231
Latitude:	-18.8777
Sate:	Inland

Shop Owners

First name	Last name	Mobile number
Shop	Owner 2	278212345678
Shop	Owner 1	27763303624



Shop Owner Details

First name	Shop
Last name	Owner 2
E-mail address:	shop.owner2@mail.com
Mobile number:	278212345678
Shop:	Shop 1

Many to Many Relationships

For a many to many relationship each shop can have multiple owners but a shop owner can also be a partial owner of more than one shop. We will once again change the multiplicity of the relationship in our model. We also update the via reference for our relationship from `shop` to `shops` to reflect this change:

```
persistent object Shop {
    .
    .
    .
    @ManyToMany
    ShopOwner owners via shops;
}
```

In this case, however, the only view changes we need to make are to the `ShopOwnerDetails` view where we now need to replace the info widget showing shop details to a table seeing as a shop owner can now also be an owner of multiple shops.

```
 <table title="table_title.shop_owner_shops">
    <collectionSource function="getShopOwnerShops"/>
    <column heading="column_heading.name">
        <attributeName>name</attributeName>
    </column>
    <column heading="column_heading.description">
        <attributeName>description</attributeName>
    </column>
</table>
```

```
Shop[] getShopOwnerShops() {
    return shopOwner.shops;
}
```

Self Referencing Relationships

In this lesson, we used two different objects and linked them using relationships. Helium, however, also support relationships between the same object.

Lesson Source Code

Lesson 6.zip

No labels