Lesson 27: Expanding Inbound REST API Functionality

Created by Jacques Marais, last modified on Sep 14, 2018

> *As an external client I would like to have access to an API that allows me interact more extensively with the app data.*

Lesson Outcomes

By the end of this lesson you should:

- Know how to add inbound API get, delete and put functions to a Helium DSL application
- Know which parameters are supported for inbound API get, delete and put functions
- Know which return types are supported for inbound API get, delete and put functions
- Know how query and path parameters can be used in inbound API functions
- Know how to structure the JSON returned by API functions using custom objects and the **`@ResponseInclude`** and **`@ResponseExpand`** annotations
- Know how to structure the JSON returned by API functions using the **`json`** and **`jsonarray`** types as API function return types
- Know how to invoke inbound API get, delete and put functions from outside a Helium DSL application

Note that while many of the topics mentioned above, is covered in this lesson, it also recommended the the following reference documentation be read for a more thorough coverage of the relevant topics:

- Reference documentation for json and jsonarray types
- Reference documentation for inbound API annotations

App Use Case Scenario

In Lesson 25 of this tutorial we introduced inbound API functionality to the tutorial app. In this lesson we will expand on this by covering the following use cases.

Get details for a farmer given the mobile number of the farmer

For this use case an external client wants to get details of a farmer using REST. The mobile number for the farmer should be provided as a path parameter. In addition to showing fields that describe the farmer, the response should also include the purchases

made by the farmer, the crop types that the farmer cultivates, and the details of whether his documentation has been updated and if so, when.

The implementation of this use case will showcase the **`@GET`** API annotation as well as how an existing data model can be used together with the **`@ResponseExclude`** and **`@ResponseExpand`** annotations to structure the JSON response provided by the API. It also shows how a path parameter can be used with inbound API functions.

Update an existing support ticket

In Lesson 25 we introduced API functionality for external clients to post support tickets to the app. Post, as implemented in Helium, is strictly that of object creation. For this reason put functionality has to be introduced in order to update an existing support ticket.

In this use case we will cover the **`@PUT`** api annotation.

Delete an existing support ticket

For this use case we want external clients with access to our API to be able to delete and existing support ticket given the unique id of the ticket. We will be providing an API that allows users to both archive the support ticket, by simply marking it as deleted as was previously demonstrated in this tutorial for other entities, and completely delete a support ticket from the app. For this we will be making use of a query parameter.

The implementation of this use case demonstrates query parameters and the **`@DELETE`** api annotation.

Get the oldest unresolved support tickets and a summary of support tickets in the system

For this use case we want external clients with access to our API to be able to get details of the five oldest, unresolved support tickets in our system as well as an overview of how many tickets in various states are currently in our system. For this, we will be constructing a custom JSON value using the native **`json`** and **`jsonarray`** types. These types are useful in many cases where JSON values need to be constructed or manipulated while not necessarily conforming to an existing app model.

New & Modified App Files

`./services/ApiFarmerResource.mez`

`./services/ApiSupportResource.mez`

`./model/objects/SupportTicket.mez`

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

Enabling Your App for Inbound REST API Functionality

In Lesson 25 of this tutorial we cover the topic of how to enable your app for inbound REST API functionality. Please see here for details.

Get Farmer Details Use Case

For this use case we will make use of the existing data model so no additional model objects need to be created. We start by modifying **`ApiFarmerResource.mez`** to include a

function that will be used as an API get function. We also make use of an existing utility method to find the farmer given their mobile number:

```
@GET("v1/farmer/mobileNumber/{mobileNumber}")
Farmer getFarmerWithMobileNumber(string mobileNumber) {
    return findFarmerWithNumber(mobileNumber);
}
```

Note how we declare the path parameter `mobileNumber` above. To specify it as a path parameter we add it to the API path by surrounding it with curly brackets. In addition to declaring the parameter in the path, we need to declare a matching function parameter. In this case it will be of type **`string`**.

For details on the logic being executed as part of the `findFarmerWithNumber` function as used above, please consult the source code for this lesson.

As an example of invoking our newly introduced API function we can use the **`curl`** tool as follows:

```
curl -u 'user:pass' \
-H "Content-Type: application/json" \
-X GET "https://dev.mezzanineware.com/rest/mezzanine-tut-les
```

The results from invoking our function as shown above is:


```
{
    "_id": "1d83f331-6b36-4b96-bb87-bf6eb786c90b",
    "_locale": "en_US",
    "_timeZone": "Africa/Johannesburg",
    "cropTypeProfileUpdatedOn": 1536837422295,
    "deleted": false,
    "documentationProfileUpdatedOn": 1536838001836,
    "emailAddress": "jmarais@mezzanineware.com",
    "farmAddress": "Some address",
    "farmSize": 10.0,
    "firstName": "Jacques",
    "governmentAssistanceCertificate": "YWRmYWRmYXNkZiBhZGZh
    "governmentAssistanceCertificateId": "1c83f331-6b36-4b96
    "governmentAssistanceCertificate_fname__": "ga_certifica
    "governmentAssistanceCertificate_mtype__": "text/plain",
    "governmentAssistanceCertificate_size__": 30,
    "lastName": "Marais",
    "lastShopVisit": 1536048000000,
    "latitude": -43.123123,
    "longitude": 10.123123,
    "mobileNumber": "27761231234",
    "registeredOn": 1536826600716,
    "state": "Inland"
}
```

Although we are now one step closer to providing the needed farmer details through a REST API, some details are still missing. The farmer purchases, crop types and documentation details still need to be provided. In addition our result also contains some meta data fields that API users might not be interested in, namely the `_locale` and `_timeZone` fields. These fields are included because of the fact that the `Farmer` object also represents an app role.

There are also data fields related to the `governmentAssistanceCertificate` **`blob`** field namely, `governmentAssistanceCertificate_fname__`, `governmentAssistanceCertificate_mtype__`, `governmentAssistanceCertificate_size__`, and the binary data field itself, `governmentAssistanceCertificate`.

Note also the inclusion of the `_id` field. This represents the internal identifier used to uniquely identify the database record for the farmer in Helium and might be useful for users of our API.

To structure the JSON result in such a way that it includes all the data we require but excludes the data we do not want in the response, we introduce the use of the **`@ResponseExclude`** and **`@ResponseExpand`** annotations for our API function:

```
@ResponseExclude("_locale")
@ResponseExclude("_timeZone")
@ResponseExclude("deleted")
@ResponseExclude("governmentAssistanceCertificate_fname__")
@ResponseExclude("governmentAssistanceCertificate_mtype__")
@ResponseExclude("governmentAssistanceCertificate_size__")
@ResponseExclude("governmentAssistanceCertificate")
@ResponseExpand("purchases")
@ResponseExpand("cropTypes")
@ResponseExclude("purchases._id")
@ResponseExclude("purchases.farmer")
@ResponseExpand("purchases.shop")
@GET("v1/farmer/mobileNumber/{mobileNumber}")
Farmer getFarmerWithMobileNumber(string mobileNumber) {
    return findFarmerWithNumber(mobileNumber);
}
```

Note how fields are excluded above using the attribute name. Also note how relationships are expanded. By default relationships that represents many related object instances are excluded entirely from the result and relationships that represent a single related object instance are represented by only the id of the related object instance. Expanding relationships expands the appropriate JSON array or object based on the relationship multiplicity.

Attributes and relationships on related objects can also be excluded / expanded by providing the entire path to that attribute or relationship. For example "purchases.farmer" and "purchases.shop" as used above. With our modified API function the following result is now produced:

```
{
    "_id": "1d83f331-6b36-4b96-bb87-bf6eb786c90b",
    "cropTypeProfileUpdatedOn": 1536837422295,
    "cropTypes": [
        {
            "_id": "61f57a09-f23e-4b42-ba11-fa799f1ea9e8",
            "deleted": false,
```

```
            "name": "Cowpea",
            "stockType": "crop_seed"
        },
        {
            "_id": "80dc5655-9600-440b-86c0-614ccaef11fe",
            "deleted": false,
            "name": "Corn",
            "stockType": "crop_seed"
        },
        {
            "_id": "a720ad0f-3026-47f5-8d12-fb24bcbfa93f",
            "deleted": false,
            "name": "Beans",
            "stockType": "crop_seed"
        }
    ],
    "documentationProfileUpdatedOn": 1536838001836,
    "emailAddress": "jmarais@mezzanineware.com",
    "farmAddress": "Some address",
    "farmSize": 10.0,
    "firstName": "Jacques",
    "governmentAssistanceCertificateId": "1c83f331-6b36-4b96
    "lastName": "Marais",
    "lastShopVisit": 1536048000000,
    "latitude": -43.123123,
    "longitude": 10.123123,
    "mobileNumber": "27761231234",
    "purchases": [
        {
            "discount": 0.0,
            "finalCost": 5000,
            "goodsCost": 5000.0,
            "purchasedOn": 1536796800000,
            "quantity": 100,
            "shop": {
                "_id": "1b649b79-9b49-42e5-816a-8b8387db283a
                "createdOn": 1536831092802,
                "deleted": false,
                "description": "This is a little shop of hor
                "latitude": -43.1231,
                "longitude": 23.1231,
                "mobileNumber": "27765551234",
                "name": "Little shop of horrors",
                "shopCode": "1b649b79-2696",
                "state": "Inland"
            },
            "stock": "59edd369-f663-4932-b3a1-2991bc98c5f1",
            "unitPrice": 50.0
        },
```

```
        {
            "discount": 0.0,
            "finalCost": 600,
            "goodsCost": 600.0,
            "purchasedOn": 1536796800000,
            "quantity": 10,
            "shop": {
                "_id": "1b649b79-9b49-42e5-816a-8b8387db283a
                "createdOn": 1536831092802,
                "deleted": false,
                "description": "This is a little shop of hor
                "latitude": -43.1231,
                "longitude": 23.1231,
                "mobileNumber": "27765551234",
                "name": "Little shop of horrors",
                "shopCode": "1b649b79-2696",
                "state": "Inland"
            },
            "stock": "a720ad0f-3026-47f5-8d12-fb24bcbfa93f",
            "unitPrice": 60.0
        }
    ],
    "registeredOn": 1536826600716,
    "state": "Inland"
}
```


Update an Existing Support Ticket Use Case

Once again no data model additions are needed. We simply need to create the appropriate API functions that can be used to update an existing support ticket. For completeness sake, we also provide a get API for support tickets:

```
@GET("v1/support/ticket/ticketId/{ticketId}")
SupportTicket getSupportTicket(uuid ticketId) {
    SupportTicket ticket = SupportTicket:read(ticketId);
    return ticket;
}
```

Note that if the post API was used to post a support ticket with an id that already exists, the call will fail due to a duplicate id violation. To update an existing persistent object instance we have to include a put API function:

```
@PUT("v1/support/ticket")
void updateSupportTicket(SupportTicket supportTicket) {
    supportTicket.save();
}
```

Let's consider that a support ticket with id 0e83d825-963e-4c9c-8340-75269d7c3f57 currently exists in our app. To get all the details of that support ticket we can use our newly introduced get API:

```
curl -u 'user:pass' \
-H "Content-Type: application/json" \
-X GET "https://dev.mezzanineware.com/rest/mezzanine-tut-les
```

```
 {
    "_id": "0e83d825-963e-4c9c-8340-75269d7c3f57",
    "receivedTime": 1536841629172,
    "resolved": false,
    "senderNumber": "27761231234",
    "spam": false,
    "text": "I require assistance in performing a stock upda
}
```

The response that we received from invoking the get API can be modified and used as a body for our put API. If we do not specify values for any specific fields, they will default to **`null`** and clear any values that might have already been set for those fields / attributes. For this reason it's important to include all fields in the body when updating persistent objects if the intention is not to nullify certain attributes values.

```
 curl -u 'user:pass' \
-H "Content-Type: application/json" \
-X PUT "https://dev.mezzanineware.com/rest/mezzanine-tut-les
-d '{
    "_id": "0e83d825-963e-4c9c-8340-75269d7c3f57",
    "receivedTime": 1536841629172,
    "resolved": true,
    "senderNumber": "27761231234",
    "spam": false,
    "text": "I require assistance in performing a stock upda
}'
```

Delete an Existing Support Ticket Use Case

For this use case we will need to add an attribute to the `SupportTicket` model object. This attribute will indicate whether the support ticket should be treated as archived by the application:

```
persistent object SupportTicket {
    datetime receivedTime;
    string text;
    string senderNumber;
```

```
    bool spam;
    bool resolved;
    bool deleted;
}
```

This implies some additional modification. In the `Support` unit located in **`Support.mez`**, we need to set a value of **`false`** for the newly introduced attribute. To achieve this, we modify the `receiveSms` method accordingly:

```
@ReceiveSms("Inbound Message Function")
void receiveSms(string number, string text) {
    SupportTicket ticket = SupportTicket:new();
    ticket.receivedTime = Mez:now();
    ticket.text = text;
    ticket.senderNumber = number;
    ticket.resolved = false;
    ticket.spam = false;
    ticket.deleted = false;
    ticket.save();
}
```

We also need to take the value of the `deleted` attribute into account when providing a collection source for the wall widget that displays support tickets. In the same `Support` unit, we modify the `getUnresolvedTickets` function:

```
 SupportTicket[] getUnresolvedTickets() {
    return SupportTicket:and(
        equals(spam, false),
        equals(resolved, false),
        equals(deleted, false)
    );
}
```

In addition to the above, we also need to modify our inbound API post function for support tickets in **`ApiSupportResource.mez`**:

```
@POST("v1/support/ticket")
void postSupportTicket(SupportTicket supportTicket) {
    supportTicket.receivedTime = Mez:now();
    supportTicket.spam = false;
    supportTicket.resolved = false;
    supportTicket.deleted = false;
    supportTicket.save();
}
```

Now that everything is in place we can introduce a delete API function in the `ApiSupportResource` method:

```
@DELETE("v1/support/ticket/ticketId/{ticketId}")
void deleteSupportTicket(uuid ticketId, bool purge) {
```

```
    SupportTicket ticket = SupportTicket:read(ticketId);
    if(ticket != null) {
        if(purge != true) {
            ticket.deleted = true;
        }
        else {
            SupportTicket:delete(ticket);
        }
    }
}
```

Once again we introduce `ticketId` as a path parameter. Note, however, the additional function parameter, `purge`. This will be available to us as a query parameter. Any primitive parameter, with the exception of **`blob`**, **`json`** and **`jsonarray`** types, that is not specified as a path parameter, is available to be used as a query parameter. Values for query parameters are seen as optional by Helium and as such, no validation will be done to check if a value is specified for a query parameter when the API function is invoked. API functions in the DSL should therefore make provision for **`null`** values for query parameters.

To invoke the API we simply do so as usual, while also including a query string:

```
curl -u 'user:pass' \
-H "Content-Type: application/json" \
-X DELETE "https://dev.mezzanineware.com/rest/mezzanine-tut-
```

If no query string was provided the value for `purge` will be **`null`**. If we had specified more than one query parameter in our delete API function, values for both could have been specified in the query string above by using '&' as a separator:

```
?purge=true&keepAudit=true
```

Support Ticket Overview Use Case

For this use case we introduce an API get function that gathers all the data we require for our response and then proceeds to construct a **`json`** variable using implicit casting along with the `jsonPut` built-in function.

```
@GET("v1/support/ticket/overview")
json getSupportTicketsOverview() {

    // Native SQL select query to get the oldest active supp
    SupportTicket[] tickets = sql:query("select * from suppo

    // Foreach ticket, we construct a custom json object and
    json[] ticketsSummary;
    for(int i = 0; i < tickets.length(); i++) {
        SupportTicket ticket = tickets.get(i);
        ticketsSummary.append(convertSupportTicketToJson(tic
```

```
    }

    // Convert the json[] to jsonarray so that we can nest i
    jsonarray ticketsSummaryArray = ticketsSummary;

    // Construct our final json result
    json result = "{}";
    result.jsonPut("oldestActiveTickets", ticketsSummaryArra
    result.jsonPut("activeTickets", countOpenSupportTickets(
    result.jsonPut("spamTickets", countSpamSupportTickets())
    result.jsonPut("resolvedTickets", countResolvedSupportTi
    result.jsonPut("deletedTickets", countDeletedSupportTick
    return result;
}
```

Helper functions referenced in the above code segment:

```
int countDeletedSupportTickets() {
    ...
}
int countSpamSupportTickets() {
    ...
}
int countResolvedSupportTickets() {
    ...
}
int countOpenSupportTickets() {
    ...
}
json convertSupportTicketToJson(SupportTicket ticket) {
    json result = "{}";
    result.jsonPut("dateLogged ", Strings:concat(ticket.rece
    result.jsonPut("loggedBy ", ticket.senderNumber);
    result.jsonPut("description", ticket.text);
    return result;
}
```

We can invoke the API as follows:

```
curl -u 'user:pass' \
-H "Content-Type: application/json" \
-X GET "https://dev.mezzanineware.com/rest/mezzanine-tut-les
```

The result of our API is as follows:

```
{
```

```
    "activeTickets": 5,
    "deletedTickets": 0,
    "oldestActiveTickets": [
        {
            "dateLogged ": "2018-09-14 11:35:30",
            "description": "I need assistance with a task",
            "loggedBy ": "27761231234"
        },
        {
            "dateLogged ": "2018-09-14 11:35:48",
            "description": "There is an emergency at my curr
            "loggedBy ": "27761230000"
        },
        {
            "dateLogged ": "2018-09-14 11:36:11",
            "description": "My crops are not growing. I need
            "loggedBy ": "27761230022"
        },
        {
            "dateLogged ": "2018-09-14 11:39:22",
            "description": "I need guidance on crop selectio
            "loggedBy ": "27763300909"
        },
        {
            "dateLogged ": "2018-09-14 11:41:33",
            "description": "I need to locate a shop close to
            "loggedBy ": "27763300909"
        }
    ],
    "resolvedTickets": 1,
    "spamTickets": 1
}
```


Additional Resources

For more related documentation see the following:

- Reference documentation for json and jsonarray types
- Reference documentation for inbound API annotations
- Tutorial lesson introducing inbound API post functionality

Lesson Source Code

Lesson 27.zip

No labels