Lesson 25: Supporting Inbound REST API Post Functionality

Created by Jacques Marais, last modified on Nov 23, 2018

> *As an external client I would like to have access to an API that allows me to post stock updates, farmer documentation and support tickets to the tutorial app.*

Lesson Outcomes

By the end of this lesson you should:

- Know how to add an inbound API post function to a Helium DSL application
- Know which parameters are supported for inbound API post functions
- Know which return types are supported for inbound API post functions
- Know how to invoke an inbound API post function from outside a Helium DSL application
- Understand how JSON arguments are mapped to DSL objects and object attributes

App Use Case Scenario

The Helium DSL provides a mechanism by which developers can specify that a function is to be exposed as an inbound REST API function. This allows external clients to integrate with Helium DSL apps directly. In this tutorial lesson we will specifically cover API post functionality to demonstrate how external clients can post data to Helium DSL apps. To demonstrate this, we will cover three use cases for our tutorial app. Each use case showcases a slightly different use and behaviour in order to demonstrate the scope of inbound API post functionality in the Helium DSL.

Stock Update Use Case

In this use case an external client wishes to post stock updates to our tutorial app using REST. The payload, for the stock update REST call, will consist of JSON specifying all the details for the stock update, including the shop and stock item for which the update is to be performed. A response object that specifies the timestamp when the request was received, timestamp when the request was processed, and a response message will be returned from the app back to the client. The implementation for this use case shows how relationships can be specified as part of the JSON arguments as well as how a unique identifier should be specified when posting persistent objects.

Farmer Documentation Use Case

In this use case an external client wishes to post farmer documentation using REST. This use case shows how objects with blob attributes can be posted to the API as well as how non-persistent objects can be posted without the need to specify an identifier.

Support Request Use Case

In the final use case an external client wishes to post a support ticket to the app using REST. This use case shows how an inbound API function can have a void return type.

New & Modified App Files

./model/enums/Enums.mez

./model/objects/ApiResponse.mez

./model/objects/ApiFarmerDocumentation.mez

./services/ApiFarmerResource.mez

./services/ApiStockResource.mez

./services/ApiSupportResource.mez

./web-app/lang/en.lang

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

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

Enabling Your App for Inbound REST API Functionality

In order to make use of the inbound REST API functionality in an app, certain values must be set for the app in Helium. To set these values for an existing app, use your browser to navigate to the Helium Core web app. Select the apps menu item, find your app and select "Update". You will see three relevant fields namely "Team Name", "User Friendly Description" and "User Friendly Api Name". The meaning of these fields are as follows:

Field	Description
Team Name	This is the name for the team for the app. At this stage this value is fixed as "Mezzanine". This will be changed in the future to allow third party team names to be specified.
User Friendly Description	This refers to a user friendly string that will describe your app in general and will be used to uniquely identify your app's REST API endpoint. This is the only field of the three mentioned here that can be updated.
User Friendly Api Name	This is the unique name that will identify the endpoint for your app's REST API. It is constructed by concatenating the value for the "Team Name" and "User Friendly Description" fields, changing the result to lower case and replacing all spaces with dashes. This value has to be unique between apps.

In order to make use of the REST API a value for the field "User Friendly Description" should be set in order for Helium to generate a user friendly API name. Once the appropriate values have been updated, save your app. Take note of the generated user friendly API name as this will be needed to invoke your app API.

The the examples in this tut we set the value of "User Friendly Description" to "Tut Lesson 25" which results in a user friendly api name of "mezzanine-tut-lesson-25".

These values can also be set when creating new apps on the Helium Core web app.

Stock Update Use Case

Data Model Additions for Stock Updates

For the stock update use case, we will specify a return type for our API post functions. We add the **`ApiResponse.mez`** model file and define the `ApiResponse` object as follows:

```
object ApiResponse {
    datetime requestReceived;
    datetime requestProcessed;
    string message;
    bool success;
}
```

In this case we've created the object as non-persistent. Alternatively, if we wished to keep track of API responses from our app in the database, we could have made use of a persistent object instead.

Post Functions for Stock Updates

Once we have defined the custom object that will be used as a return type, we can proceed to define our API post functions and thus the API resources that will be available to users of the API. In order to define a function as a API post function, we make use of the **`POST`** annotation. As part of the annotation the API resource is also defined. In this case we provide the `"v1/stock/stockupdate"` and `"v1/stock/stockupdates"` resources. It is common for the path of REST resources to indicate the inherent relationship between components. For example, in this case, we are performing a stock operation and specifically a stock update. This is the recommended convention to use when defining REST resources for DSL apps.

Note that for this use case we provide two post functions. One of these functions has an object as parameter and the other a collection of objects as parameter. This allows users of the API to decide which function to invoke to either post a single stock update, or multiple stock updates with a single API call.

The functions below follow a simple pattern of creating a response object, validating the input, and then, depending of whether the provided data is correct or not, it will return a response with fields populated to indicate whether it was successful or not. If the posted stock update data is correct, it will be saved to the database.

Below are the API post functions added to a new unit namely, `ApiStockResource`, located in **`ApiStockResource.mez`**:

```
@POST("v1/stock/stockupdate")
ApiResponse postStockUpdate(StockUpdate stockUpdate) {

    // initiate the response
    ApiResponse response = ApiResponse:new();
    response.requestReceived = Mez:now();

    // validate and persist
    ApiResponse updatedResponse = persistStockUpdate(stockU
    if(updatedResponse.success == false) {
        return updatedResponse;
    }

    // return the result
    updatedResponse.requestProcessed = Mez:now();
    return updatedResponse;
}

@POST("v1/stock/stockupdates")
ApiResponse postStockUpdates(StockUpdate[] stockUpdates) {
```

```
22
23      // initiate the response
24      ApiResponse response = ApiResponse:new();
25      response.requestReceived = Mez:now();
26
27      // for each stock update validate and persist
28      ApiResponse updatedResponse;
29      foreach(StockUpdate stockUpdate: stockUpdates) {
30          updatedResponse = persistStockUpdate(stockUpdate, r
31          if(updatedResponse.success == false) {
32              return updatedResponse;
33          }
34      }
35
36      // return the result
37      updatedResponse.requestProcessed = Mez:now();
38      return updatedResponse;
    }
```

For the full source code example of code excerpt above, please see the complete app source code attached to this lesson.

Invoking the API to Post Stock Updates

For a Single Stock Update

In order to invoke our API externally we will post data in JSON format corresponding to the post function's parameters. For example, to invoke the `"v1/stock/stockupdate"` resource we need to post JSON that can be converted to a `StockUpdate` object by Helium. An example payload is shown below:

Payload example

```
{
 "_id":"fb94f312-2f99-4d40-889a-414d4b09f1ac",
 "level":200,
 "price":100,
 "stocktakeDate":1522326277000,
 "shop":"4575470d-ee0a-474d-bd5f-1c370c6fc817",
 "stock":"80dc5655-9600-440b-86c0-614ccaef11fe"
}
```

Note how the field names correspond to the attributes in the `StockUpdate` object. Specific attention should be payed to the format of the data as is expected by the API.

In the example above we can see the following fields being posted:

Field Name	Description

Field Name	Description
_id	This field is an implied and required field for all persistent object parameters. It should be a string literal representing a uuid. The value of this field will be used as the unique identifier for the object in Helium. If an inbound API function uses a persistent object or a persistent object collection as parameter, and the appropriate identifiers are not specified an exception will be generated and the API will fail with the appropriate error code. In cases where the API fails due to an exception, all operations associated with the API call will be rolled back. In any case where an exception is generated, the Helium Logging Service should be used for diagnosing and debugging the issue. Note that the Helium DSL implementation of post for persistent objects is strictly that of object creation. This means that if more than one post for the same object type is made, and the same identifier value, a duplicate key exception will be generated and the API call will fail.
level	This corresponds to the attribute on `StockUpdate` with the same name. The attribute is of type **`int`** and thus an integer numeric value is expected as argument.
price	Similar as above but for the `price` attribute
stockTakeDate	This field corresponds to the attribute on `StockUpdate` with the same name. The attribute is of type **`date`**. Arguments for **`date`** and **`datetime`** attributes should be sent as a numeric value representing the number of milliseconds since the UNIX Epoch as in the example above.
shop	This field corresponds to a relationship on `StockUpdate` with the same name. The value for this field should be the unique identifier of the related object. It should therefore be a string representing a uuid. This is supported for all relationship multiplicities except many to many. In the case of many to many, relationships cannot be set using the inbound API. It might not always be practical to specify the id of the related object since external clients might not, for example, know what the id for a for shop is. This example is simply to show how relationships can be set using the API. In reality an alternative approach should probably use such as specifying the generated shop code as discussed in Lesson 5 and then using that to map the stock update to the shop.
stock	Same as above but for the relationship between the `StockUpdate` and `Stock` objects.

Now that we have created our REST resource and we have determined what the payload should be we can use the **`curl`** command from a command line terminal to invoke our API:

CURL example

```
 curl -u 'user:pwd' \
-X POST "https://dev.mezzanineware.com/rest/mezzanine-tut-lesson-1
-H "Content-Type: application/json" \
-d '{
  "_id":"fb94f312-2f99-4d40-889a-414d4b09f1ac",
  "level":200,
  "price":100,
  "stocktakeDate":1522326277000,
  "shop":"4575470d-ee0a-474d-bd5f-1c370c6fc817",
  "stock":"80dc5655-9600-440b-86c0-614ccaef11fe"
}'
```

See the table below for a description of the different sections to the call above:

```
curl
```

```
-u 'user:pwd'
```

```
-X POST
```

```
-H "Content-Type: application/json"
```

```
-d '{...}'
```

```
"https://dev.mezzanineware.com/rest/mezzanine-tut-lesson-25/v1/stock/
```

Note that although we use CURL as an example in this tutorial, the API can be invoked using any REST client.

For multiple Stock Updates

Similarly to the previous example we can also invoke the `"v1/stock/stockupdates"` resource by posting more than one stock update as shown below:

Payload example

```
 [
  {
    "_id":"b54fb253-7f61-4a5a-9ae8-fcf42c495892",
    "level":200,
    "price":100,
    "stocktakeDate":1522326277000,
    "shop":"4575470d-ee0a-474d-bd5f-1c370c6fc817",
    "stock":"80dc5655-9600-440b-86c0-614ccaef11fe"
  },
  {
    "_id":"ae9b5e85-9c7f-4062-9cbd-84060dc2267d",
    "level":500,
    "price":160,
    "stocktakeDate":1522326277000,
    "shop":"4575470d-ee0a-474d-bd5f-1c370c6fc817",
    "stock":"d5f7cb6d-69ef-4e01-953d-151d89792155"
  }
]
```

In this example we simply specify a JSON array which can then contain one or more JSON objects to represent a `StockUpdate` collection.

As before we can use CURL to invoke the API:

CURL example

```
curl -u 'user:pwd' \
-X POST "https://dev.mezzanineware.com/rest/mezzanine-tut-lesson-1
-H "Content-Type: application/json" \
-d '[
  {
    "_id":"b54fb253-7f61-4a5a-9ae8-fcf42c495892",
    "level":200,
    "price":100,
    "stocktakeDate":1522326277000,
    "shop":"4575470d-ee0a-474d-bd5f-1c370c6fc817",
    "stock":"80dc5655-9600-440b-86c0-614ccaef11fe"
  },
  {
    "_id":"ae9b5e85-9c7f-4062-9cbd-84060dc2267d",
    "level":500,
    "price":160,
    "stocktakeDate":1522326277000,
    "shop":"4575470d-ee0a-474d-bd5f-1c370c6fc817",
    "stock":"d5f7cb6d-69ef-4e01-953d-151d89792155"
  }
]'
```

For both the `"v1/stock/stockupdates"` and `"v1/stock/stockupdate"` resources as described in this section `ApiResponse` is used as the return type. This will result in JSON representing an `ApiResponse` instance as follows:

ApiResponse return example

```
{
  "success":true,
  "requestProcessed":1522914814126,
  "message":"Success",
  "requestReceived":1522914814117
}
```

If the return type of the DSL API function is a collection a JSON array will be returned.

In addition to the above any successful execution where a value is returned will also respond with the http response code **`200`**. If the function has a void return type and has executed successfully, the http response code will be **`204`** representing successful execution with no content returned.

Farmer Documentation Use Case

Data Model Additions for Farmer Documentation

For the farmer documentation use case we will only be updating a selection of attributes on an existing farmer. For cases like this, the inbound API feature allows for the posting of non-persistent objects. We therefore create a the following non-persistent object representing only the fields we wish to update. This object is located in the **`ApiFarmerDocumentation.mez`** model file.

```
1  object ApiFarmerDocumentation {
2      string farmerMobileNumber;
3      uuid governmentAssistanceCertificateId;
4      blob governmentAssistanceCertificate;
5  }
```

Post Functions for Farmer Documentation

Next we define our post functions as before. In this case we create a new presenter file for the purpose namely, **`ApiFarmerResource.mez.`** Note the naming convention for our API resources indicating the inherent relationship between a farmer, his profile and specifically the documentation for his profile. Also note the version convention as mentioned before.

```
1
2  @POST("v1/farmer/profile/documentation")
3  ApiResponse postFarmerDocumentation(ApiFarmerDocumentation
4
5      // initiate the response
6      ApiResponse response = ApiResponse:new();
7      response.requestReceived = Mez:now();
8
9      // validate the arguments
10     if(farmerDocumentation.farmerMobileNumber == null) {
11         response.success = false;
12         response.message = "Invalid arguments, please speci
           return response;
```

```
    }
    if(farmerDocumentation.governmentAssistanceCertificateI
        response.success = false;
        response.message = "Invalid arguments, please speci
        return response;
    }
    if(farmerDocumentation.governmentAssistanceCertificate
        response.success = false;
        response.message = "Invalid arguments, please speci
        return response;
    }

    // get the farmer for the specified mobile number
    Farmer farmer = findFarmerWithNumber(farmerDocumentatio
    if(farmer == null) {
        response.success = false;
        response.message = Strings:concat("A farmer with th
        return response;
    }
    farmer.governmentAssistanceCertificateId = farmerDocume
    farmer.governmentAssistanceCertificate = farmerDocument
    farmer.save();

    response.requestProcessed = Mez:now();
    response.success = true;
    response.message = "Farmer documentation successfully u
}
.
.
.
```

Invoking the API to Post Farmer Documentation

Once again we define the JSON payload:

Payload example

```
{
  "farmerMobileNumber":"27763303624",
  "governmentAssistanceCertificateId":"7c3c66dd-992d-489a-a4e0-99
  "governmentAssistanceCertificate":"WW91IGFyZSBhcHByb3ZlZCBmb3Ig
  "governmentAssistanceCertificate_fname__":"FarmerGovernmentAssi
  "governmentAssistanceCertificate_size__":43,
  "governmentAssistanceCertificate_mtype__":"text/plain"
}
```

Note that no `_id` field is specified here as we are not posting data for any persistent objects.

As part of the farmer documentation we have a **`blob`** attribute on our `ApiFarmerDocumentation` object. Data for the **`blob`** attribute, `governmentAssistanceCertificate`, should be sent as a base64 encoded string of the byte array representing the file content. Along

with this though, Helium also requires meta data fields related to the file content. These fields use the attribute name for our blob attribute appended with the relevant type description of the meta data for that field. For example:

Field Name	Description
governmentAssistanceCertificate_fname__	A string field representing the file name of our blob data
governmentAssistanceCertificate_size__	A number field representing the file size in bytes
governmentAssistanceCertificate_mtype__	A string field representing the mime type of our blob data

As before we can then post the data specified above using the **curl** command line tool:

CURL example

```
 curl -u 'user:pass' \
-X POST "https://dev.mezzanineware.com/rest/mezzanine-tut-lesson-1
-H "Content-Type: application/json" \
-d '{
  "farmerMobileNumber":"27763303624",
  "governmentAssistanceCertificateId":"7c3c66dd-992d-489a-a4e0-99
  "governmentAssistanceCertificate":"WW91IGFyZSBhcHByb3ZlZCBmb3Ig
  "governmentAssistanceCertificate_fname__":"FarmerGovernmentAssi
  "governmentAssistanceCertificate_size__":43,
  "governmentAssistanceCertificate_mtype__":"text/plain"
}'
```

Support Ticket Use Case

Posting support tickets is perhaps the simplest use case discussed in this tutorial lesson. In this case we will not need any data model additions as the return type of our post function will be **void** and the parameter will be a persistent object type.

Post Functions for Support Tickets

We define our post function and API resource in a new file, **ApiSupportResource.mez**, using the same conventions as before:

```
1  @POST("v1/support/ticket")
2  void postSupportTicket(SupportTicket supportTicket) {
3      supportTicket.receivedTime = Mez:now();
4      supportTicket.spam = false;
5      supportTicket.resolved = false;
6      supportTicket.save();
7  }
```

Invoking the API to Post Support Tickets

Seeing as we are posting data for a persistent object, a value has to be specified for the implied `_id` field:

Payload example

```
{
  "_id":"0e83d825-963e-4c9c-8340-75269d7c3f57",
  "receivedTime":1522738620123,
  "text":"I require assistance in performing a stock update",
  "senderNumber":"27763303624"
}
```

To invoke the function, we use the following curl call:

CURL example

```
curl -u 'user:pass' \
-X POST "https://dev.mezzanineware.com/rest/mezzanine-tut-lesson-
-H "Content-Type: application/json" \
-d '{
  "_id":"0e83d825-963e-4c9c-8340-75269d7c3f57",
  "receivedTime":1522738620123,
  "text":"I require assistance in performing a stock update",
  "senderNumber":"27763303624"
}'
```

Summary of Inbound API Post Functionality

Function Return Types

The following DSL return types are supported for functions annotated with **`POST`**:

- **`void`**
- A persistent or non-persistent object
- A persistent or non-persistent object collection

Function Parameters

The following parameter types are supported for inbound REST API functions in the DSL:

- No parameters
- A single parameter that is a persistent or non-persistent custom object type
- A single parameter that is a collection of a persistent or non-persistent custom object type
- A single parameter that is of type json or jsonarray

Argument Attributes and Mappings

- **`string`** attribute values are to be specified as strings, contained inside quotes
- **`int`** attribute values are to be specified as numbers without any decimal parts
- `enum` attribute values are to be specified as if they are strings, using only the enum value not including the enum type
- For persistent objects, an `_id` field has to be specified. The value for the field should be a string representing a valid UUID.
- For non-persistent objects an _id is not required but can optionally be specified.

- **`date`** and **`datetime`** attribute values are to be represented by a numeric value representing the number of milliseconds since the unix epoch.
- **`blob`** values are to be posted as base64 encoded string of file content byte arrays.
- In addition to blob content as specified above, meta data should also be included for blob attributes. The name for these fields should be the name of the blob attributes appended with a specific key representing the type of meta data. Consider a case where our blob attributes name is "data". Values for the following fields should then also be posted:
  - `data_fname__` is a sting field that represents the file name for the blob
  - `data_size__` is an integer field that represents the file size for the blob
  - `data_mtype__` is a sting field that represents the mime type for the blob
- Values for relationships can be posted using a string representation of the unique identifier of the related object and using a field name that corresponds to the relationship name on the DSL object. Posting values for many to many relationships is not supported.

Lesson Source Code

Lesson 25.zip

inbound rest api