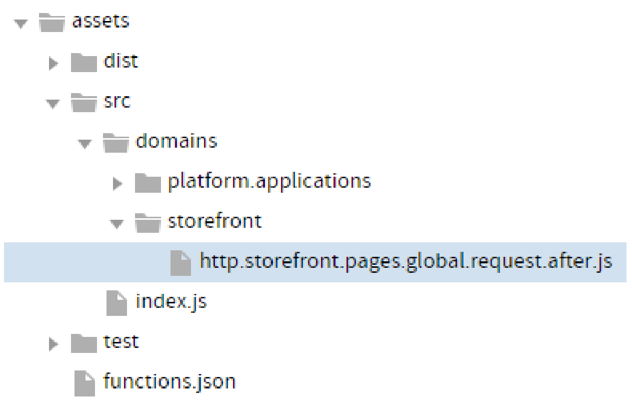
Programming Patterns

Every action has a mapping to a JavaScript file that contains a custom function. These JavaScript files are named according to the Type.Domain.Action.Occurrence syntax.

The files are grouped in folders organized by the domain the actions pertain to. To create the necessary action files and directory structure for the first time, use the recommended Yeoman Generator scaffolding tool, as described in the Quickstart. The following image shows the `http.storefront.pages.global.request.after` action file located in the `storefront` domain folder.



```
assets
  dist
  src
    domains
      platform.applications
      storefront
        http.storefront.pages.global.request.after.js
    index.js
  test
  functions.json
```

The JavaScript files share the following basic structure:

```
module.exports = function(context, callback) {
   // Your custom code here
   callback();
};
```

When you code the custom function for an action, you have access to two arguments: `context` and `callback`.

Context

The `context` argument provides the function access to relevant objects and methods that interface with Kibo. The following table summarizes a few of the `context` objects and methods you can use to enhance the functions you develop. Refer to the reference content for full details about the `context` argument available for each action you are coding a custom function for, or log the `context` object to view its data.

apiContext	Provides mostly read-only access to standard API variables, such as the tenant, site, master catalog, user claims, etc.
configuration	Provides read-only access to configuration data that you set in the Action Management JSON editor.
exec	Provides read and write access to functions that act on objects. The methods accessible by `exec` vary depending on the action you are coding the function for.

get	Provides read-only access to certain objects. The objects accessible by `get` vary depending on the action you are coding the function for.

Examples:

```
var url = context.apiContext.baseUrl;
var customData = context.configuration.customField;
context.exec.removeItem(allocation.referenceItemId);
var cartItem = context.get.cartItem();
```

In addition, when you code for HTTP actions, you have access to `context` objects and methods related to the HTTP response and request, such as the following objects:

request	Provides read access to the HTTP request made to the API endpoint the action accesses.
response	Provides read and write access to the HTTP response made to the API endpoint the action accesses.

Variable Assignment

If you assign a variable to the value of a resource, and then use an `exec` method to update the resource, the variable in your JavaScript file does not automatically update with the new value of the resource. For example, if you set a variable to the value of an order:

```
order = context.get.order();
```

...and then update the order using an `exec` method:

```
context.exec.setData("customField", "Amazing-Order");
```

...the order resource updates with the new custom field but the `order` variable still retains the original value of the order resource at the time of the GET operation. If you want the variable to update to the latest value of the order resource, you have to use the `context.order.get` method again.

callback

The `callback` argument enables non-blocking, asynchronous program flow by allowing you to propagate errors and other important information across function calls whenever needed. The `callback` argument follows the established JavaScript callback pattern: it takes an error as the first argument (or null if there is no error) and a result as the second argument (not required for API Extensions applications because you use `context` methods instead to perform state changes). To ensure that your function does not halt the runtime or break storefront functionality, arrange callbacks so that they are the last pieces of code to execute within your function.

```
// Note: This code demonstrates using callbacks within a simplified if-else function structure. The r
ecommended structure for API Extensions functions is the try catch statement, as explained in the
 next section.
module.exports = function(context, callback) {
   if (A) {
      // Do something.
      callback();
   }
   else if (B) {
      // Do something else.
      callback();
   }
   else {
      // Last option.
      callback();
   }
};
```

Try Catch Error Handling

The try catch statement is useful for handling any errors that may occur in your API Extensions functions. It tells Kibo to `try` to execute a block of code, and `catch` errors if they occur in the `try` block. Note, however, that the `catch` block does not detect errors that occur within a Promise chain. To catch these errors, use a `.catch` statement at the end of the Promise chain.

To ensure that your function does not halt the runtime or break storefront functionality, place callbacks within the try catch statement. If you are familiar with try catch statements, you may know that you can also employ a `finally` block to execute a final block of code, regardless of whether an error occurred in the `try` block. Counterintuitively, however, the `finally` block executes before Promise chains, and because you will likely employ Promises in your functions, use of the `finally` block is generally not recommended.

The following code block demonstrates a high-level example of using the try catch statement within an action file:

```
module.exports = function(context, callback) {
   try {
      // For starters, run some cool code.
      var meaningfulVariable = 42;
      coolFunction(meaningfulVariable);

      // Run this cool Promise chain.
      someFunction().then(function(name) {
         if (something) {
            return somethingCool;
         } else {
            // Do something else.
            callback();
         }
      })
      .then(function() {
         callback();
      })
      .catch(function(err) {
         // Catch errors that occur in the Promise chain...
         console.error(err);
         callback();
      });
   }
   catch (error) {
      // Catch errors that occur in the try block, outside the Promise chain...
      console.error(error);
      callback();
   }
};
```

Interested in learning more about try catch statements? The Mozilla Developer Network has agood writeup on the topic.

Call a Third-Party Package

One of the benefits of the API Extensions framework is that it allows you to leverage the thousands of NPM packages developed by the Node.js community. The following code blocks provide an example of loading the Needle HTTP client for use with API Extensions.

When integrating third-party packages, always load the smallest package possible. If the functionality you require is offered as a subset of the complete module, load the smaller module to minimize delays on your storefront.

1. Install the package using npm by opening a command prompt in your project folder and entering `npm install` *`package name`*.

   Alternatively, you can list the SDK as a dependency in your `package.json` file by adding the following code to the file, where you can specify a specific version of the package using a caret followed by the version number or match any version by using an asterisk:

   ```
   "devDependencies": {
      "needle": "*"
   }
   ```

2. Require the package or a helper file that loads the package (shown) in one of your JavaScript files, and then use the package as needed. For instance, this package may be in *assets/src/domains/someDomain/someActionFile.js*:

```
var serviceCall = require('../../util/needleRequest').makeSampleCall;
module.exports = function (context, callback) {
    var sampleData = "" + context.request.url + "";
    serviceCall(sampleData)
        .then(function (responseFromCall) {
            console.log(responseFromCall);
            callback();
        })
        .catch(function(err) {
            console.error(err);
            callback();
        });
}
```

And in *assets/src/util/needleRequest.js*:

```
var needle = require('needle');
exports.makeSampleCall = function (bodyStr) {
    var promise = new Promise(function (resolve, reject) {
        try {
            needle.post('https://someServer.com',
            {
                headers: {
                    'Content-Type': 'text/xml'
                },
                body: bodyStr
            }, function (error, response) {
                if (!error && response.statusCode == 200) {
                    console.log('Inside needle request');
                    resolve(response);
                } else {
                    reject(response.statusCode);
                }
            });
        } catch (err) {
            reject(console.error(err));
        }
    });

    return promise;
};
```

Call the Node.js SDK

While the `context` argument provides each function access to pertinent methods and data, you may at times need access to the complete set of API microservices. To access this broader array of functionality, call the Node.js SDK from within API Extensions by completing the following steps:

1. Install the SDK using npm by opening a command prompt in your project folder and entering `npm install --save mozu-node-sdk`.

Alternatively, you can list the SDK as a dependency in your `package.json` file by adding the following code to the file, where you can specify a specific version of the SDK using a caret followed by the version number or match any version by using an asterisk:

```
"devDependencies": {
   "mozu-node-sdk": "*"
}
```

2. After installation, require and use the Node.js SDK in your JavaScript files. The following code block is an example of retrieving an entity list within an action file.

```
var entityListClientFactory = require('mozu-node-sdk/clients/platform/entityList');
module.exports = function(context, callback) {
   var entityListClient = entityListClientFactory(context);
   entityListClient.context['user-claims'] = null; // Remove shopper user claims
   // Run calls...
   entityListClient.getEntities({
      entityListFullName: 'mylist@' + context.nameSpace
   }).then(function(list) {
      // Handle the response...
      callback();
   }).catch(callback); // Pass the callback to the error handler to pass errors to runtime
}
```

Notes for Using the Node.js SDK

- Require client factories using the “deep require” method shown in the preceding code block, which requires only the subset of the module that is needed. This saves you upload space compared to requiring the entire SDK using `require('mozu-node-sdk')`.
- The `context` object includes user claims for the currently active user in a store. These user claims have a “shopper” level of permissions, but the example requires claims that can make calls to the API. If you want to make calls that require more privileges than the shopper has, set the `context` user claims to null, as shown above in the preceding code block.
- The Node.js SDK convention of looking for a `mozu.config.json` file for initialization does not work with API Extensions. Instead, pass the `context` object to the client factory (or constructor) so that the SDK client initializes with the appropriate tenant, site, catalog, and authentication details.
- When handling errors, communicate a success by calling `callback()` with no arguments. Communicate an error by calling `callback()` with the error provided. Note that you can pass `callback()` directly into a `.catch()` method on the Promise returned by the SDK. If you want to handle errors differently than described, you can write your own error handler.

Add or Remove User Claims

When an API Extensions action executes a custom function, it applies the user claims that belong to the user initiating the action. In some cases, user claims have sufficient permissions to execute your needed operations, such as manipulating items in a shopper's cart. In other cases, however, you may need API-level instead of user-level claims to execute your needed operations, such as when you want to query the API regarding a resource. When you need more permissions, remove the user claims as follows:

```
var someResourceFactory = require('mozu-node-sdk/pathToResource');

module.exports = function(context, callback) {
   var someResource = someResourceFactory(context.apiContext);
   someResource.context["user-claims"] = null;
}
```

What would a real-world example look like? The following code block shows a function that adds a shopper's recent purchases as extended properties of the cart (you could then use these properties in your theme to display a list of recently purchased items to returning shoppers). Because this function is bound to the `embedded.commerce.carts.addItem.after` action, the user claims correspond to the permissions of a shopper who has just added an item to their cart. Given this information, the function requires:

- Retention of user claims on the cart resource, so that the extended properties are scoped to the shopper (in general, cart operations require user-level permissions to execute correctly).
- Removal of user claims from the order resource, so that you can query the API about order history, which is beyond the scope of the shopper's user-level permissions.

In *assets/src/domains/commerce.carts/embedded.commerce.carts.addItem.after.js*:

```
//Require the components of the Node.js SDK that you need. These "Factories" help you generate
Kibo eCommerce resources with appropriate context.
//Each of these Factories will need to receive an apiContext object to gain authorization to make A
PI requests.
var CartExtendedPropertiesResourceFactory = require('mozu-node-sdk/clients/commerce/carts/ext
endedProperty');
var OrderResourceFactory = require('mozu-node-sdk/clients/commerce/order');

module.exports = function(context, callback) {
   //Wrap functionality in a try/catch block. This prevents errors from stalling your storefront.
   try {
      //The context.apiContext contains both app-claims and user-claims.
      //The user-claims allow for making API requests scoped to the user who initiated this API Exte
nsions action (adding an item to the cart).
      //The app-claims allow for making API requests scoped beyond the permissions of a user and
are used to affect data in the workflow.
      //Since Order API calls are scoped outside of the permissions that a user is allowed, you'll rem
ove the user-claims from the Order Resource you create.
      //Note: The original context.apiContext object is not altered.
      var orderResource = OrderResourceFactory(context.apiContext);
      orderResource.context["user-claims"] = null;

      //Carts are scoped directly to the users who own them.
      //You retain the user-claims so you can write extended properties on the cart, which belongs t
o the shopper.
      var cartExtendedPropertyResource = CartExtendedPropertiesResourceFactory(context.apiCo
ntext);

      var cart = context.get.cart();
      var productCodeString = '';
      var extendedProperty;
      var hasExistingProperty = (cart.extendedProperty && cart.extendedProperty.length > 0) ? tru
```

```
        var hasExistingProperty = (cart.extendedProperty && cart.extendedProperty.length > 0) ? tru
e : false;

        orderResource.getOrders({ filter: 'userId eq ' + cart.userId })
            .then(function(orderCollection) {

            if (orderCollection.totalCount > 0) {
                //console.log(orderCollection.totalCount);
                var reduceHandler = function(previousOrderItem, currentOrderItem) {
                    productCodeString += previousOrderItem.product.productCode + "," + currentOrderI
tem.product.productCode + ',';
                };
                for (var i = 0; i < context.configuration.orderHistoryNumberOfOrders; i++) {
                    orderCollection.items[i].items.reduce(reduceHandler);
                }
                productCodeString = productCodeString.slice(0, productCodeString.lastIndexOf(','));

                if (productCodeString.length > 0) {
                    extendedProperty = [{
                        key: context.configuration.extendedPropertyKey,
                        value: productCodeString
                    }];

                    if (hasExistingProperty) {
                        cart.extendedProperties.forEach(function(property, index) {
                            if (property.key === context.configuration.extendedPropertyKey) {
                                hasExistingProperty = true;
                            }
                        });

                        if (existingProperty) {
                            return cartExtendedPropertyResource.deleteExtendedProperty({ key: context.co
nfiguration.extendedPropertyKey })
                                .then(function() {
                                    return cartExtendedPropertyResource.addExtendedProperties({}, { body: e
xtendedProperty });
                                });
                        }
                }
                return cartExtendedPropertyResource.addExtendedProperties({}, { body: extendedProp
erty });
            } else {
                callback();
            }
        } else {
            callback();
        }
    })
    .then(function() {
        callback();
    })
    //Use the .catch statement to handle errors in the Promise chain.
    //Be sure to execute a callback in case of errors to refrain from halting Mozu storefront executio
n.
    .catch(function(err) {
        console.error(err);
        callback();
    });
    //Use a catch block to handle errors that occur in the try block, outside of Promise chains.
    //Be sure to execute a callback in case of errors to refrain from halting Mozu storefront executio
```

```
n.
  } catch (error) {
     console.error(error);
     callback();
  }
};
```

Send Custom Data to Your Theme

You can set custom data within **storefront HTTP *after*** actions, and then expose that data in your theme for use on storefront pages. On the API Extensions side, you set the custom data in the `context.response.viewData` object. On the theme side, you access the data from the `viewData` variable, where it is stored as a top-level JSON object. However, you should note that ViewData is used for storefront functions that return HTML which must be processed through Kibo. This means that if your custom route manually returns some value that is only calculated in your function, it will have no effect. You must ensure that the data you are sending is compatible with the HTML designed for this process so that it can be processed and displayed on your site theme.

The `context.response.viewData` object contains an object called `model`, which represents the model for the current page. While you can access and modify the `model` object through API Extensions, you cannot add custom data that doesn't already exist in the `model` object. You can only set custom data at the `context.response.viewData` level.

When you set custom data for your theme in an API Extensions action, you must:

- Use the *after* version of a storefront HTTP action.
- Access the data within a theme template that executes the same route as the API Extensions action in which you set the data. The following table demonstrates example relationships between actions and the theme templates that can access their custom data.

Action File Where You Set Data	Theme Templates Where You Access the Data
`http.storefront.pages.cart.after.js`	`templates/pages/cart.hypr` and any included templates
`http.storefront.pages.checkout.after.js`	`templates/pages/checkout.hypr` and any included templates
`http.storefront.pages.productDetails.after.js`	`templates/pages/product.hypr` and any included templates
`http.storefront.pages.myAccount.after.js`	`templates/pages/my-account.hypr` and any included templates

Example

The following is an example of setting custom data in an action file (remember, this must be an HTTP storefront*after* action):

```
module.exports = function(context, callback) {
   context.response.viewData.yourCustomField = "yourCustomValue";
   callback();
};
```

...which you can then access in an appropriate theme template:

```
......
```

Securely Store and Access Sensitive Data

API Extensions allow you to access encrypted data on your tenant. You first post the sensitive data using the API, which encrypts the data and stores it at the tenant level as a credential store entry. Once the data is on your tenant, only an API Extensions application tied to your developer account can decrypt the entry—this is for security reasons; not even the API can decrypt the entry. The following sections provide more information about the process:

Store Sensitive Data Using the API

To securely store data on your tenant, use the StoreCredentials operation to POST your data to the `platform/extensions/credentialStore` API resource in the form of a credential store entry, which consists of the following fields:

```
{
   "fullName": "name@namespace",
   "value": [
      {
         // Custom data
      }
   ]
}
```

Field Name	Description
`fullName`	A name for your entry, in the form of `name@namespace`, where the namespace must match the namespace of your developer account. If the `name` you choose already exists within the namespace, then the POST operation overwrites the previous value.
`value`	A JSON object that represents whatever sensitive information you want to encrypt.

Access Sensitive Data

While the API receives, encrypts, and stores each credential store entry, it does not support retrieving an entry. To decrypt and access a credential store entry, use the `getCredential` method available in the `context` object of an API Extensions function. The application that executes the function must be created within your developer account—other applications will not have the required permissions to run the `getCredential` method within your namespace.

```
context.getCredential('name@namespace', function (err, result){
   // Access sensitive data within the result variable
};
```

For its first argument, the `getCredential` method takes the value of the `fullName` field of the credential store entry you want to access. For the second argument, the method takes a callback. After you decrypt data using the `getCredential` method, you can assign the result to a variable and access the sensitive data from the entry's `value` object. See the following example use case for additional details.

Example Use Case

Let's say you want to encrypt an API key for a third-party service, and then access that key in an API Extensions function. Here's how you would go about doing that:

1. In the request body, POST the following credential store entry to the `platform/extensions/credentialStore` API resource. Remember that the namespace you choose should match the developer account that will contain the application for decrypting the sensitive data.

   ```
   {
      "fullName": "myCredentials@myNamespace",
      "value": [
         {
            "accountName" : "myAccountName",
            "apiKey": "123456789"
         }
      ]
   }
   ```

2. Create an API Extensions application within the same developer account namespace where you posted the credential entry.
3. Within an API Extensions custom function, access the encrypted data:

   ```
   module.exports = function(context, callback) {
      context.getCredential('myCredentials@myNamespace', function (err, result){
         if(err){
            callback(err);
            return;
         }

         console.log("My third-party account name is " + result.accountName);
         console.log("My third-party API key is " + result.apiKey);
      };
      callback();
   };
   ```

Calculating Currency

When you are calculating currency amounts in KCCP, you need to make sure that the decimals are correctly rounded. For example, all amounts are rounded to two decimal places (or three depending on your currency).

JavaScript uses floating point values for numbers. That means `.1 + .2 = 0.30000000000000004` in some implementations including the API Extensions.

Thus, if you are setting a currency value anywhere in API Extensions, you need to apply the "Round" function and amount in a statement like this:

`Number(amount.toFixed(2))`

Which will convert an amount to `.3` if the original value was `0.30000000000000004`.