API Extension Examples

Use the examples in this topic as a reference point for developing your own custom functionality using API Extensions.

Customer Normalization

This API Extension application limits unnecessary accounts from being created for the same shopper. When a shopper checks out on your site, API Extensions check whether the shopper's email address is already associated with an existing customer account, and then handles the shopper's account based on the following scenarios:

When a shopper decides to check out as a guest:

- If their email address is not associated with an existing customer account, a new anonymous account is created.
- If their email address is associated with an existing anonymous customer account, the new order is associated with the existing anonymous account.
- If their email address is associated with an existing registered shopping account, then a new anonymous customer account is created with all the previously existing customer attributes of the existing registered account.

When a shopper decides to check out as a registered shopper:

- If their email address is not associated with an existing customer account, a new registered account is created.
- If their email address is associated with an existing anonymous customer account, the new order is associated with the existing anonymous account, but the anonymous account is converted into a registered shopper account.
- If their email address is associated with an existing registered shopping account, then the shopper is logged into the existing registered account.

Notable Files in the API Extension Assets

The assets for the customer normalization application are available on GitHub. The following table lists the notable files that make the application work. You can customize these files to suit your needs.

File Name	Description
cnAddAccountAndLoginBefore.js cnAddAccountBefore.js cnUpdateAccountBefore.js	These files execute API Extension functions before registered or anonymous customer accounts are created or updated.

File Name	Description
customerservice.js	This file contains the main functions of the application. These functions define what the application does in response to the different customer account scenarios.
embedded.platform.applications.install.js	This file provides the installation configuration for the application.
commerce.customer.manifest.js platform.applications.manifest.js	These manifest files define the relationships between API Extension actions, the functions they execute, and their configured name.

Installation

Create an application in Dev Center to house the customer normalization assets:

1. In your Dev Center Console, click **Develop** > **Applications**.
2. Click **Create Application**.
3. Specify whatever name you wish and click **Save**. You should now see your application in the Applications grid.
4. Double-click your new application to view it.
5. Note the **Application Key**. You will need this value to authenticate your customer normalization assets.

Configure and install the customer normalization assets:

1. Clone or download the `customer-normalization` repository available on GitHub. If you want to allow customers to checkout as a guest using an email address that is associated with a registered user, set the `enableAnonymousAndRegistered` variable to `true` in all files that use the variable. By default, the files that use this variable are `assets/src/domains/platform.applications/embedded.platform.applications.install.js` and `assets/src/domains/customerservice.js`.
2. In the root directory of the cloned assets, create a file called `mozu.config.json`. Specify your application configuration data within this file, as shown in the following code block, replacing the placeholder values with your application-specific values. You must remove the comments from the code below. They are provided for instructional purposes but will error your application because the JSON format does not support comments.

```
{
   "baseUrl": "https://t00000.sandbox.mozu.com", // your base url for generating auth tickets
   "developerAccountId": 1234, // unique ID of the developer account - you can find this value in the Dev Center Launchpad
   "developerAccount": {
      "emailAddress": "you@email.com" // the dev account email
   },
   "workingApplicationKey": "yourApplicationKey"
}
```

3. In the root directory, open a command prompt and run `npm install` to install project dependencies.
4. After dependencies install successfully, enter `grunt` to upload the assets to your Dev Center application.
5. Return to Dev Center to view your application.
6. To confirm that the assets uploaded successfully, navigate to the **Packages** tab and click **Assets**.
7. Install the application to the sandbox of your choice.

Add New Customers to a New Customers Segment

This function uses an HTTP action to add all new customers to a new customers segment.

To use this example:

1. In Admin, use the Customers module to create a new customer segment called "New Customers".
2. Use the Yeoman Generator to scaffold an application that includes the `http.commerce.customer.accounts.addAccountandLogin.after` action.
3. Code the `http.commerce.customer.accounts.addAccountandLogin.after.js` file as shown in the following code block:

```
var _ = require('underscore');
var CustomerSegmentFactory = require('mozu-node-sdk/clients/commerce/customer/custom
erSegment');
module.exports = function (context, callback) {
    var customerSegmentResource = CustomerSegmentFactory(context.apiContext);
    customerSegmentResource.context['user-claims'] = null;
    //console.info("Hello from Add Account and Login!");
    //console.info("Request...");
    //console.info(context.request.body);
    //console.info("Response...");
    //console.info(context.response.body);
    //console.info(context.response.body.customerAccount);

    var account = context.response.body.customerAccount;
    customerSegmentResource.getSegments({ filter: "name eq 'New Customers'" })
    .then(function (segmentCollection) {
        if (_.first(segmentCollection.items)) {
            var newCustomerSegment = _.first(segmentCollection.items);
            var accountId = [account.id];
            return customerSegmentResource.addSegmentAccounts({id: newCustomerSegment.i
d}, {body: accountId});
        }
    })
    .then(function() {
        //console.info("Successfully added " + account.firstName + " " + account.lastName + " t
o the New Customers segment.");
        callback();
    })
    .catch(function(err) {
        console.error(err);
        callback();
    });
};
```

4. Run `grunt` to upload the assets.
5. Install the application to your sandbox.

Ideas to expand on this function include applying discounts to the New Customers segment, automatically removing customers from the segment after a specified period of time, or only adding customers to the segment if they opt in to receive promotional material during the checkout process.

Limit Cart Items of the Same Product Type

This function uses an embedded action to limit how many items of the same product type can be added to the cart. The product type to be limited is read from configuration data set in the Action Management JSON Editor. Whenever an item cannot be added to the cart, the function logs an error message to the `data` array on the cart object.

To use this example:

1. Use the Yeoman Generator to scaffold an application that includes the `embedded.commerce.carts.addItem.before` action.
2. Code the `embedded.commerce.carts.addItem.before.js` file as shown in the following code block:

```
var _ = require('underscore');
module.exports = function (context, callback) {
    var cart = context.get.cart();
    var cartItem = context.get.cartItem();
    var cartProductTypes = [];
    var oneTypePerCart = context.configuration.oneTypePerCart;

    console.info(cart);
    console.info(cartItem);

    if(cart.items.length > 1) {
        for(var i = 0; i < cart.items.length; i++) {
            //console.info("Cart Item #" + (i + 1));
            //console.info(cart.items[i].product.name);
            //console.info(cart.items[i].product.productType);
            if (cartItem.id !== cart.items[i].id) {
                cartProductTypes.push(cart.items[i].product.productType);
            }
        }
    }
    var cartItemProductType = cartItem.product.productType;
    //console.info("Cart Item To Add:");
    //console.info(cartItem.product.name);
    //console.info(cartItemProductType);

    if(cartItemProductType == oneTypePerCart && _.contains(cartProductTypes, cartItemProd
uctType)) {
        //console.info("Removing cart item...");
        var itemsToRemove = [];
        _.each(cart.items, function(item) {
            if(item.product.productType == cartItemProductType) {
                itemsToRemove.push(item);
            }
        });
        try{
            if(itemsToRemove.length > 0) {
                var itemToRemove = _.first(itemsToRemove);
                var errorMessage = itemToRemove.product.name + " has been removed from your
 cart. You can have only one item of the type " + oneTypePerCart + " in your cart at a time.";
                context.exec.removeItem(itemToRemove.id);
                //console.info("Removed the following item: ");
                //console.info(itemToRemove.product.name);
                context.exec.setData("removedItemMessage", errorMessage);
                context.exec.setData("removedItemId", itemToRemove.product.productCode);
            }

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

3. In Admin, go to **System** > **Customization** > **API Extensions** to open the Action Management JSON Editor.
4. Add configuration data to specify which product type should be limited to only one item per cart. For example, the following code block designates the `"Hazardous"` product type as the type to limit.

```
{
  "actions": [
    {
      "actionId": "embedded.commerce.carts.addItem.before",
      "contexts": [
        {
          "customFunctions": [
            {
              "applicationKey": "yourApplicationKey",
              "functionId": "embedded.commerce.carts.addItem.before",
              "enabled": true
            }
          ]
        }
      ]
    },

  ],
  "configurations": [
    {
      "applicationKey": "yourApplicationKey",
      "configuration": {
        "oneTypePerCart": "Hazardous"
      }
    }
  ],
  "defaultLogLevel": "info"
}
```

The preceding example shows the `oneTypePerCart` configuration data at the application-level, which makes it accessible to every action in your application. If you want to limit the configuration data to one action (for a clearer organization or to reuse variable names with different values, for example), you can place the data within the appropriate `customFunctions` array. For more information about the JSON configuration options, refer to the Action Management JSON Editor topic.

5. Run `grunt` to upload the assets.
6. Install the application to your sandbox.

Ideas to expand on this function include limiting specific product combinations in the cart (instead of product types), allowing Admin users to specify prohibited combinations in the Action Management JSON Editor, and leveraging your theme to expose the message in the cart data during the checkout process so that shoppers know why they can't add a particular item.

Validate Purchase Orders During the Checkout Process

This function uses an embedded action to augment the purchase order feature. With this function, you can communicate with your ERP system to validate a purchase order in the following ways:

- Set the payment term based on the shopper's shipping address and PO number.
- Display an error to the shopper if they enter an invalid PO number.
- If a customer account is not in good standing, display an error to the shopper informing them that they cannot

use purchase orders as a payment option.

To use this example:

1. Enable purchase orders as a payment type on your site and then enable the ability for specific customers to use purchase orders during checkout.
2. Use the Yeoman Generator to scaffold an application that includes the `embedded.commerce.payments.action.before` action.
3. Code the `embedded.commerce.payments.action.before.js` file as shown in the following code block. The sample code provided below uses hard-coded values to validate different scenarios. When creating a production application, you would expand on this code to communicate with your ERP system in order to retrieve the correct payment term, validate if a purchase order number is valid, and determine whether a customer account is in good standing.

```
var OrderResourceFactory = require('mozu-node-sdk/clients/commerce/order');
var CustomerResourceFactory = require('mozu-node-sdk/clients/commerce/customer/accounts/customerPurchaseOrderAccount');

module.exports = function(context, callback) {
   var purchaseOrderNumber = context.get.payment().billingInfo.purchaseOrder.purchaseOrderNumber;
   var payment = context.get.payment();
   var orderId = context.get.payment().orderId;

   var orderResource = OrderResourceFactory(context.apiContext);
   var customerPurchaseOrderAccountResource = CustomerResourceFactory(context.apiContext);

   /***** Set the payment term based on the shipping address and PO# *****/
   orderResource.getOrder({ orderId: orderId})
   .then(function (order) {
      var address = order.fulfillmentInfo.fulfillmentContact.address;
      // console.info('address :', address);

      // In this example the conditions are hard-coded, but you can expand on the code to communicate with your ERP system
      if(address.postalOrZipCode == '78750' && purchaseOrderNumber == '123456')
         {
            var paymentTerm = {
               "Code": "30-days",
               "Description" : "30 Days"
            };
            context.exec.setPaymentTerm(paymentTerm);
            // console.info('payment after :', context.get.payment());
         }
   })
   .then(function() {
      callback();
   })
   .catch(function(err) {
   console.error(err);
   callback();
   });


   /***** Verify that the purchase order is valid *****/
   var expectedPurchaseOrderNumber = 123456; // In a real-life scenario, you can check the PO# against your ERP or similar system

   if(expectedPurchaseOrderNumber != purchaseOrderNumber)
      // Displays an error message to the user on the checkout page
      throw new Error('Invalid purchase order number!');


   /****** Check whether the customer account is in good or bad standing ********/
   var isCustomerFlagged = true; // In a real-life scenario, you can check the account against your ERP or similar system
   if(isCustomerFlagged)
      // Displays an error message to the user on the checkout page
      throw new Error('Your purchase order account is not in good standing. Please use another payment method.');
};
```

4. Run `grunt` to upload the assets.
5. Install the application to your sandbox.

Set a Tax Response While Short-Circuiting the API Call

This example demonstrates how you can use API Extensions to skip an API call and provide your own response information. Specifically, it does so to calculate tax using a custom function for Minnesota shoppers, while using the default call for everyone else.

To use this example:

1. Ensure you have configured tax settings for your site.
2. Use the Yeoman Generator to scaffold an application that includes the `http.commerce.catalog.storefront.tax.estimateTaxes.before.js` action.
3. Code the `http.commerce.catalog.storefront.tax.estimateTaxes.before.js` file as shown in the following code block:

```
module.exports = function(context, callback) {
   //console.info("Start: storefront.tax.estimateTaxes.before");
   var taxOrderInfo = context.request.body;
   //console.info("request: %j", context.request);
   //console.info("request.body: %j", taxOrderInfo);
   //console.info("response: %j", context.response);
   //console.info("Order #: " + taxOrderInfo.OrderNumber);

   // If the condition is met, end the call to skip prevent calling the built-in Mozu route which
would otherwise overwrite the custom response.
   if (taxOrderInfo.TaxContext.DestinationAddress.StateOrProvince === 'MN') {
      calculateMnTax(taxOrderInfo, function (responseBody){
         //console.info("%j", responseBody);
         context.response.body = responseBody;
         context.response.end();
         //console.info("Special MN Taxing for this state! Tax State: " + taxOrderInfo.TaxConte
xt.DestinationAddress.StateOrProvince);

         callback();
      });
   } else {
      // If the destination is not MN, calculate tax using the default tax engine.
      //console.info("Using default Taxing for this state! Tax State: " + taxOrderInfo.TaxContex
t.DestinationAddress.StateOrProvince);
      callback();
   }
};

function calculateMnTax(taxOrderInfo, callback) {
   var responseBody = {
      "itemTaxContexts" : [],
      "shippingTax" : 0.00,
      "handlingFeeTax" : 0.00,
      "orderTax" : 0.00
   };
```

```
    // Make sure to get current tax in order to add it to the total.
    var orderTotalTax = 0.0;
    // for each
    if (taxOrderInfo.LineItems && taxOrderInfo.LineItems.length > 0) {
        var itemTaxAmount = 0.00;
        for (var i = 0; i < taxOrderInfo.LineItems.length; i++) {
            var lineItem = taxOrderInfo.LineItems[i];
            //console.info("LineItemPrice: " + lineItem.LineItemPrice);
            // Only apply special tax to Minnesota shoppers. Skip tax-exempt
            if (!taxOrderInfo.TaxContext.TaxExemptId && lineItem.IsTaxable) {

                itemTaxAmount = lineItem.LineItemPrice * 0.10275;
                //console.info("Adding a item tax Amount of: " + itemTaxAmount);
            } else {
                //console.info("Tax exempt customer (or item).  TaxID: " + taxOrderInfo.TaxContext
.TaxExemptId);
            }
            responseBody.itemTaxContexts.push({
                "id" : lineItem.Id,
                "productCode" : lineItem.ProductCode,
                "quantity" : lineItem.Quantity,
                "tax" : itemTaxAmount.toFixed(2),
                "shippingTax" : 0.0
            });
            orderTotalTax += itemTaxAmount;
        }
        responseBody.orderTax = orderTotalTax.toFixed(2);

        //console.info("End: storefront.tax.estimateTaxes.before. Total Tax = " + orderTotalTax);
    }
    callback(responseBody);
}
```

Add Custom Tax Data to an Order

This example demonstrates how to add custom tax data (a random amount for purposes of the example) to orders and order items.

To use this example:

1. Use the Yeoman Generator to scaffold an application that includes the `http.commerce.catalog.storefront.tax.estimateTaxes.after` action.
2. Code the `http.commerce.catalog.storefront.tax.estimateTaxes.after.js` file as shown in the following code block:

```
module.exports = function(context, callback) {
   function readAndUpdateTax(target) {
      var customTax = Math.random() * 10;
      target.CustomTax = parseFloat(customTax.toFixed(2));
   }

   context.response.body.taxData = context.response.body.taxData || {};
   readAndUpdateTax(context.response.body.taxData);

   context.response.body.itemTaxContexts.forEach(function(item) {
      item.taxData = item.taxData || {};
      readAndUpdateTax(item.taxData);
   });

   callback();
};
```

Shorten the Duration of Hot Authentication

This example demonstrates how to shorten the duration of hot authentication down to 15 minutes. By default, if shoppers do not log out of the system, they remain under hot authentication for 24 hours (even if they close their browsers). After this period, shoppers enter warm authentication, which means their cart status is saved, but they must re-enter their credentials to check out or access their My Account page. If you wish for shoppers to enter warm authentication quicker than 24 hours, use API Extensions to delete the refresh tokens from their accounts.

To use this example:

1. Use the Yeoman Generator to scaffold an application that includes the `http.commerce.customer.authTickets.createUserAuthTicket.after` action.
2. Code the `http.commerce.customer.authTickets.createUserAuthTicket.after.js` file as shown in the following code block:

```
module.exports = function(context, callback) {
   var accessTokens = context.response.body;
   accessTokens.refreshToken = "";
   context.response.body = accessTokens;
   // console.log(context.response.body);

   callback();
};
```

Add/Update Custom Reasons

This API Extension application allows you to customize appeasement and cancellation reasons so that an appropriate list of business reasons will be listed.

Below are steps to create and configure the cancellation reason API Extension on any tenant.

Installation

Here are the installation steps:

1. Create an application in Dev Center.
    - In your Dev Center Console, click **Develop** > **Applications**.
    - Click **Create Application**.
    - Specify any name you wish and click **Save**. You should now see your application in the Applications grid.
    - Double-click your new application to view it.
    - Please note the Application Key. You will require this value for authenticating your customer normalization assets.
2. Create an API Extension application for the `http.commerce.orders.cancellationReasons.after action` or clone the repository available on GitHub.
3. Update the Mozu.config file with the appropriate application key, baseUrl, and developer account details. The sample Mozu.Config for an example site should look like the one below.

```
{
 "baseUrl": "https://example.com",
 "developerAccountId": 2159,
 "developerAccount": {
  "emailAddress": "admin@kibocommerce.com"
 },
 "workingApplicationKey": "dev_center_application_for_cancellation_reason_key(Ex. a0842
dd.CustomRefundReasons.1.0.0.Release)",
}
```

4. Add or Update cancellation reasons. The API cancellation is in `context.response.body.items` .
    - To append reasons, add your custom reason to `context.response.body.items` . For Example, the below code should add 'ArrivedTooLate' reason to API response.

```
var response = context.response.body;
response.items.push({
"reasonCode": "ArrivedTooLate",
"name": "Arrived too late",
"needsMoreInfo": false,
"categories": []
});
```

    - To override the API reasons and use different ones, clear the API response and assign a new list of reasons to `context.response.body.items` . For example, the code below should override the API response and provide a completely custom set of cancellation reasons.

```
var response = context.response.body;
response.items = [
{
"reasonCode": "ArrivedTooLate",
"name": "Arrived too late",
"needsMoreInfo": false,
"categories": []
},
{
"reasonCode": "CustomerChangedMind",
"name": "Customer changed mind",
"needsMoreInfo": false,
"categories": []
},
{
"reasonCode": "DamagedOrDefective",
"name": "Damaged or defective",
"needsMoreInfo": false,
"categories": []
}];
```

5. Save the API Extension application and upload it by firing the grunt command. It should upload the application to your DevCenter application.
6. Install the application to the desired tenant.
7. Login into the target tenant from step 6.
8. Navigate to the Applications page (**System** > **Customization** > **Applications**).
9. Search for the cancellation reason application and enable it.
10. Navigate to the Action Management page (**System** > **Customization** > **API Extension Application**) and check if it has an entry for your cancellation reason application, and ensure that it is enabled. The entry should look like the code snippet below. If you don't find any entry, add a script similar to the one below and save it.

```
{
  "actionId": "http.commerce.orders.cancellationReasons.after",
  "contexts": [
    {
      "customFunctions": [
        {
          "applicationKey": "dev_center_application_for_cancellation_reason_key(Ex. a084
2dd.CustomRefundReasons.1.0.0.Release)",
          "functionId": "http.commerce.orders.cancellationReasons.after",
          "enabled": true,
          "timeoutMilliseconds": 25000
        }
      ]
    }
  ]
}
```

11. When you invoke the Order, Shipment, or related APIs, it should list the custom reasons from the API Extension application.