Email Template Customization

Themes contain the design templates of your customer email templates. This guide explains how to create and apply themes in your implementation (which must be done by a developer) and how to edit an existing theme (which can be done by a non-developer, such as a marketing representative).

All out-of-the-box emails are listed in the General Settings of the Admin UI. However, some emails may not be utilized by your implementation such as if you are an Order Management-Only implementation and do not support B2B Commerce.

> This guide does not include instructions for creating a new email template. As emails are triggered by events within the Kibo Composable Commerce Platform (KCCP), code updates will need to be done by developers and Kibo Engineering to support new email types.

Create and Apply a Theme

While Kibo provides a default Core theme, many implementations may have customized themes or multiple versions of themes installed. Before you can edit a theme, it must be uploaded to the Dev Center and installed on your tenant. The steps to do so are:

1. Create a Theme
2. Prepare Your Development Environment
3. Use the Theme Generator
4. Make a Quick Change
5. Upload Your Theme Files
6. Install Your Theme in a Sandbox
7. Apply Your Theme

The sections below provide instructions for each of these actions.

Create a Theme

Themes live in the Dev Center. To upload theme files you've created or modified, you must first create a theme record in Dev Center. To create a theme record:

1. In the Dev Center Console, click **Develop** > **Themes**.
2. Click **Create Theme**.
3. In the **Create Theme** dialog box, specify the theme Name and ID as follows:
    1. **Name**: MyFirstTheme
    2. **Theme ID**: MyTheme
4. Click **Save**. You should now see your theme in the Themes grid.

5. Double-click your new theme to edit it.
6. Note the **Application Key**. You will need this value to configure the Theme Generator and Dev Center Sync tools later in this tutorial.

Prepare Your Development Environment

Themes are designed to leverage inheritance, so all of your theme development should extend the latest Core theme. We recommend using the Theme Generator to manage your theme assets. The Theme Generator:

- Creates new themes that inherit from the latest Core theme or clones an existing theme from a Git repository to a local directory. You can also use the generator to upgrade existing themes.
- Configures your local theme directory as a Git repository.
- Connects the Core theme Git repository (or any other theme Git repository) to your theme as a remote so you can merge upstream changes.

Use The Theme Generator

This tool is a Yeoman plugin that generates the scaffolding (e.g., directory structure, reference files, and build files) necessary to package a theme and upload it to Dev Center. It's designed to augment, not overwrite, existing themes. If you have a theme that extends the Core theme, you can safely run this tool in that directory without overwriting your existing files. Whenever Kibo releases a Core theme upgrade, you can use this tool to merge changes from the Core theme (or any other theme Git repository) with your theme.

For this tutorial, create a brand new theme that inherits from the latest Core theme:

1. Open a Terminal (OS X) or a Command Prompt (Windows).
2. Install the Yeoman command-line tool globally:
   `npm install -g yo`
3. Install the Grunt command-line tool globally:
   `npm install -g grunt-cli`
4. Install the Theme Generator globally:
   `npm install -g generator-mozu-theme`
5. Create a new folder for your theme on your local machine and navigate to it:
   `mkdir your_theme && cd your_theme`
6. Run the Yeoman generator inside your theme directory:
   `/your_theme/$ yo mozu-theme`
7. If you have an old version of the tool installed, you'll be prompted to update it. Press <Ctrl+C> to exit the application and enter the following command: `npm install -g generator-mozu-theme`
8. Select **Brand new theme**.
9. Enter a public name for your theme.

10. (Optional) Enter a short description of your theme.
11. Enter the initial version.
12. Select **Kibo eCommerce Core Theme** as the base theme from which your new theme will inherit.
13. Select which version of the Core theme from which your new theme will inherit.
14. Enter your theme's Dev Center Application Key.
15. Enter your Developer Account login email.
16. Enter your Developer Account password.
17. Select your developer account.

You now have a blank theme based on the latest Core theme, which you can modify, build, and upload to Dev Center. Since the Core theme Git repository is connected to this Git repository as a remote, you'll be able to merge upstream updates from the Core theme with your theme in the future.

Make a Quick Change

Now that you have your own copy of the latest Core theme, you can begin making modifications.

1. Open theme.json in your project's root directory for editing. This file contains the majority of your theme settings, structured in JSON format.
2. In the about object, change the value associated with the name property from whatever you chose as a name to Quickstart.
3. Save and close theme.json.

Your local copy of your theme now contains changes that you can upload to Dev Center.

Upload Your Theme Files

You need to build and upload your theme files to Dev Center in order to apply your theme to a site. Kibo provides build tools that take care of this process for you. Complete the following steps to take advantage of these build tools:

1. Open a command prompt in your project's root directory.
2. Run `grunt` to build your project assets and upload them to Dev Center.

What does `grunt` do and what are the common options you can use with it?

`grunt:`

- Checks your JSON and JavaScript for syntax and style errors
- Compares your theme with the remote base theme and notifies you if updates are available for merging
- Compiles your theme's JavaScript according to the `./build.js` file that you either inherit or override
- Uploads changed files to Dev Center into the theme specified by the Application Key you provided when configuring the Theme Generator tool

- If you've added new files at the root level of your theme directory, you must add each file name to the mozusync.upload.src section of Gruntfile.js to upload them using the `grunt` command.

`grunt build-production:`

- Checks your JSON and JavaScript for syntax and style errors
- Compiles your theme's JavaScript according to the ./build.js file that you either inherit or override
- Compress and minify the compiled JavaScript for production
- Creates a .zip containing your theme files suitable for sharing or manually uploading within Dev Center (The ZIP file you upload contains only the contents of the theme folder that have changed and not the theme folder itself.)

`grunt mozusync:wipe && grunt:`

- Cleans up the theme in Dev Center by deleting all files and then re-uploading your theme files

`grunt watch:`

- Listens for any changes to your theme files
- If you save a change to a theme file, `grunt` automatically builds and uploads your theme to Dev Center.

Manually Uploading Your Theme Files to Dev Center

As an alternative to the build tools, you can manually upload a built and zipped package of your theme using the following steps. However, Kibo recommends using the build tools in most cases.

1. In the Dev Center Console, click **Develop** > **Themes**.
2. In the Themes grid, double-click **MyFirstTheme**.
3. Click **More** > **Upload**.
4. Drag your zipped theme file into the **Upload files** dialog box.
5. When the upload is complete, click **Done**.
6. Click **Packages**. You should see the contents of your theme ZIP file.

Install Your Theme in a Sandbox

After your theme builds, install your theme to a sandbox so you can later apply the theme to a site. You only need to do this once. After installation, your theme remains installed on the sandbox until you remove it.

1. In the theme toolbar at the top right, click **Install**.
2. In the Select a Tenant dialog box, select the sandbox you previously created.
3. Click **OK**.

Now your theme is installed in your sandbox. The next step is to apply it to a site within your sandbox.

Apply Your Theme

The Content Editor is a module in Admin that merchants use to interact with themes. You can use this to test the functionality and appearance of your theme. To apply your theme to a site in your sandbox:

1. In the Dev Center Console, click **Sandboxes**.
2. In the Sandboxes grid, right-click the sandbox you installed your theme to and select **View**.
3. Log in to Admin.
4. Click **Main** > **Content** > **Themes**.
5. Click the dots on the `Quickstart` theme and click **Apply**.

Edit Email Templates

Once a theme has been installed, non-developers can customize the content. This section introduces the theme's file directory and how to format the email template files.

Theme Structure

Once you have been given access to the theme from your developers, you will be presented with a file directory. From the top level of the directory, there are two main areas that you will be navigating to for editing email templates.

The *labels* folder contains the language files used to manage text strings, which will be *en-US.json* (English) for the purposes of this guide. These labels will be where you edit the actual text of the email body. Other language files may or may not exist depending on which translations your implementation supports (note that *de-DE.json* may be included by default as a placeholder). The processes documented here can be followed for any language file.

The *templates* folder contains subfolders for Hypr templates of KCCP's user interfaces, site pages, and emails. The *emails* subfolder is where you will go to format the layout of emails and insert labels or API variables. It contains templates such as *order-confirmation.hypr* (Order Confirmation), *giftcard-created.hypr* (Gift Card Created), and so on.

Hypr Templates

Email templates are stored and edited as Hypr code in the *templates/emails* folder of the theme, but they can also be viewed in the Admin UI at **Main** > **Content** > **Editor**. In the Pages sidebar on the right, click the **Email Templates** folder and select a template. Here the content is displayed with the formatting that would be seen by an email recipient. You are also able to change the header settings and send test emails.

In the theme itself under *templates/emails,* Hypr templates use a labeling system to determine the text content and reference KCCP APIs for variable information such as order numbers and customer names. For instance, this is the default template for the Backorder email which you can edit for customization:

```
{% extends "email/email" %}

{% block body-content %}
   <dl class="mz-orderheader">
      <dt>{{ labels.backorder }}</dt>
      <dt>{{ labels.orderNo }} <a href="">{{ model.orderNumber }}</a></dt>
     <dt>{{ labels.externalOrderId }} {{ model.order.externalId }}</dt><dd></dd>
  </dl>
   <br>
   <p>{{ labels.orderWelcome }} {{ model.origin.firstName }} {{ model.origin.lastNameOrSurna
me }}!</p>
   <br>
   <p>{{ labels.backorderBlob|string_format(siteContext.generalSettings.websiteName, model.or
derNumber, domainName)|safe }}</p>

   <table class="mz-ordersummary">
   <thead>
      <tr>
         <th class="mz-ordersummary-header-product">{{ labels.item }}</th>
           <th class="mz-ordersummary-header-available-on">{{ labels.availableOn }}</th>
         <th class="mz-ordersummary-header-subtotal">{{ labels.subtotal }}</th>
      </tr>
   </thead>
  <tbody>
      {% for item in model.items %}
         <tr>
            <td class="mz-ordersummary-item-product">
               {{ item.name }}
               <dl>
                  <dd>{{ item.productCode }}</dd>
               </dl>
            </td>
            <td>
               {{ item.backorderReleaseDate }}
            </td>
            <td align="right"><span class="mz-item-price">{% filter currency %} {{ item.actualPr
ice|multiply(item.quantity) }} {% endfilter %}</span></td>
         </tr>
      {% endfor %}
  </tbody>
  </table>

  {{ labels.backorderNote|safe }}

  {{ labels.emailClosing|string_format(siteContext.generalSettings.websiteName)|safe }}

{% endblock body-content %}
```

Content Formatting

While the general layout of content in the*templates/emails* folder is formatted with HTML, Hypr elements will process string variables and insert data from the API. This section will explain how labels and data fields are used to build these templates.

More technical details about Hypr can be found in the Templating System documentation as needed.

API Variables

When an email is generated, it will pull variables from the API for the particular customer, order, shipment, or other element. These variables can be inserted anywhere into a *templates/email* Hypr file with the general format *{{ model.API.object.field }}*. However, the object may not be required if the particular variable is a top-level field in the API model not contained within a smaller object.

The below examples are used in the Order Confirmation template referring to the Order API:

```
{{ model.orderNumber }}
{{ model.shopperNotes.comments }}
{{ model.billingInfo.billingContact.firstName }}
```

If the API object is a list, such as a set of fulfillment locations, then 0 can be used as a placeholder where the particular entry in the list would be identified. The value will be filled in by the system when it generates the email:

```
{{ model.locations.0.address.address1 }}
```

These variables can also be used in logical decisions, such as only displaying a block of text in the Order Confirmation email if the order type is for Curbside Delivery. In this case, the logical statement and its closing tag is enclosed by {% %}:

```
{% if model.isCurbside == true %}
...
{% endif %}
```

While you can refer to the API documentation to view API models, it may be advised to check with developers for the proper way to reference certain fields that are not already provided in the template for you to copy the format of.

Labels

A label is a variable with a string value. Templates use these variables because any changes to the value are reflected across all templates where the labels is used by editing a single point of maintenance. This allows email content to remain consistent and more easily updated, as well as supports the ability to translate a template into different languages.

A few examples in the *labels/en-US.json* language file are shown below. You can edit any of these string values as well as create new labels by using the *"name":"value",* format on a new line. HTML formatting is supported such as for creating bulleted lists, underlining text, and creating links.

```
"accountMissing": "Please select an Account",
"accountName": "Account Name",
"accountNoCredits": "You have no store credits.",
"accountNoOrders": "You have not placed any orders.",
```

Once a label exists, it can be used in any file from the *templates/emails* folder. By plugging in a label name, its string value will be displayed when the email is generated. The syntax for inserting a label is *{{ labels.labelVariableName }}*.

```
{{ labels.accountName }}
```

Labels with Variables

If you want to use a placeholder value where a variable can be inserting into the label, use the *{}* notation. For example, the "milesAway" label uses a placeholder value to display the distance to a location:

```
"milesAway": "{0} miles away",
```

This label can then be inserted into an email template, using the following *string_format* syntax with the API variable identified in parentheses. You can copy this format with any label and appropriate API field.

```
{{ labels.milesAway|string_format(location.distance) }}
```

Combining Labels and API Variables

The combination of HTML formatting, labels, and API variables make up the full email template. For instance, this block within the default *templates/email/order-confirmation.hypr* template for the Order Confirmation email shows how to display the address of a store. Note the inclusion of both simple labels (this particular example does not display any placeholders) and object data.

```
<!--- Store Details --->
<div class="mz-store-details">
  <div>
    <strong>{{ labels.storeDetails }}</strong>
  </div>
  <div>
     <div> {{ labels.storeLocation }} : {{ model.locations.0.name }}</div>
     <div> {{ model.locations.0.address.address1 }} </div>
     <div> {{ model.locations.0.address.cityOrTown }},</div>
     <div> {{ model.locations.0.address.stateOrProvince }},</div>
     <div> {{ model.locations.0.address.postalOrZipCode }} </div>
     <div> {{ model.locations.0.phone }}</div>
 </div>
</div>
```

The output will look something like this:

```
Store Details
Store Location: Dallas Store
123 Example Road
Dallas
Texas
75201
972-000-0000
```

You should also now be able to look back at the earlier example of the Backorder email and see how it is building the email, as well as know which theme files to navigate to to make any edits to the labels, data, or HTML format.