Hypr Templating System

Kibo provides the Hypr templating system to facilitate the design of your web pages. Templates specify how to generate the HTML on your site—within a template, you specify the location of static and dynamic content on a page. Many web frameworks use templates to layout web pages quickly and to reuse content throughout a web site. If you are familiar with how templates work on Django, you will find it just as easy to use Hypr templates. The following code block is an example of a Hypr template:

```
{% extends "page" %}

{% block title-tag-content %}{% firstof pageContext.metaTitle model.name %}  - {% parent %}{
% endblock title-tag-content %}

{% block body-tag-classes %} mz-category {% endblock body-tag-classes %}

{% block body-content %}

{% require_script "pages/category" %}

{% include "pages/category-interior" %}

{% endblock body-content %}
```

Tags and Filters

Hypr templates use tags and filters to add dynamic functionality on top of static templates and to process and transform values.

Tags execute logic inside templates in order to render complex data or affect control flow. Some tags take arguments and some do not. The general syntax is:

`{% tag *argument* %}`

Tags that enclose content must be opened and closed:

`{% for %}`
Do something in the loop...
`{% endfor %}`

Other tags are self-contained:

`{% require_script "widgets/myAwesomeWidgetScript" %}`

Additional examples of tags include:

dropzone	Renders an area (dropzone) in Site Builder where Admin users can drag and drop widgets on a page.
for	Loops over each item in an array.

include_products	Retrieves a list of products from the catalog.

Filters perform operations on a value. Some filters take arguments, enclosed in parentheses, and some do not. The general syntax is:

`{{ value|filter(argument1, argument2) }}`

For example, you can convert a string to uppercase using a filter, or truncate words from a string:

`{{ "Hello Sweet World"|upper }}` -----> "HELLO SWEET WORLD"
`{{ "Hello Sweet World"|truncatewords(1) }}` -----> "Hello..."

Additional examples of filters include:

findwhere	Searches a list of objects to find the object that contains a matching property value.
string_format	Formats a string that contains placeholders (such as " `{0}` " and " `{1}` ") by injecting the filter arguments into the string according to the numbering of the placeholder and the order of the arguments.
urlencode	Converts special characters in a string into their URL-encoded equivalents.

To learn more about tags and filters and see them in use, refer to thereference help.

Point out "page" and scripts directory "pages/category" no extension assumed location. If in an extends template, everything has to be in a block because otherwise it's not clear where that content goes. So mixing HTML is mostly page.hypr and partials and things that you would include only. In with tags, the value will be available in any includes you put within the with as well.

Access Global Variables

You can access several global variables within Hypr templates. These variables give you access to relevant Kibo eCommerce information such as the site ID, theme settings, payment settings, etc. For example, you can access the Boolean value for whether the page is being viewed through Site Builder using the following variable:

`{{ pageContext.isEditMode }}`

In addition to these global variables, you can also access the `model` variable for each page. The model variable changes depending on which template you are editing. For example, the model for a category template is a category object, a home page template is a CMS document, a widget template is a widget object, and so on.

To learn more about global variables, refer to the reference help.

Mix in HTML Content

Within templates, you can use standard HTML markup alongside Hypr tags and filters. For example:

```
   {% if pageContext.user.isAuthenticated %}
      <img src="" model.mainImage with max=themeSettings.listProductThumbSize quality=75 a
s_parameter %}" />
   {% endif %}
```

However, if you are editing a template that extends another template (which you usually are) then all your code must be enclosed within a block tag. Any content outside of a block tag, including HTML content, will not be rendered. The next section explains this in more detail.

Templates Can Extend Other Templates

`{% extends "page" %}`

Most templates in your theme extend from `page.hypr`, which is the base template located in the `templates` directory. This means that a category template, for example, `category.hypr`, located in the `templates/pages` directory, inherits its content from `page.hypr` unless it explicitly overwrites the parent content.

Let's take a closer look at extending templates.

```
      {% block firstname-form %}
   <div class="row">
      <input type="text" name="firstname" value="{{ model.firstName }}" data-mz-value="{{ a
ddresscontext }}FirstName">
   </div>
{% endblock firstname-form %}

<div class="invisible-man">child templates can't see me!</div>

{% comment %}
The template outputs:
<div class="firstname-form">
   <div class="row">
      <input type="text" name="firstname" value="Jerms" data-mz-value="ShippingAddress.First
Name">
   </div>
</div>
{% endcomment %}
```

In this simple example, `firstname-form` renders a form for entering a first name. The second template, `checkout-firstname-form`, extends the form from `firstname-form` while also adding a Next button.

```
{% extends "modules/firstname-form" %}
{% block firstname-form %}
   {% parent %}
   <button>Next<button>
{% endblock firstname-form %}

{% comment %}
The template outputs:
<div class="firstname-form">
   <div class="row">
      <input type="text" name="firstname" value="Jerms" data-mz-value="ShippingAddress.First
Name">
   </div>
   <button>Next</button>
</div>
{% endcomment %}
```

How does this work? Here's the breakdown on extending templates:

- `firstname-form` encloses content that other templates can extend within `block` tags. If `firstname-form` contained content outside of `block` tags, other templates would not be able to extend that content.
- `checkout-firstname-form` extends `firstname-form` using the tag `{% extends "modules/firstname-form" %}`. This pulls in all the content from `firstname-form` that is enclosed by `block` tags. Note that Hypr knows to look in the `templates` directory to find the template being extended.
- To modify the div that contains the form, `checkout-firstname-form` includes the `block` that contains the div in question. In this case, it's `{% block firstname-form %}`. Including a `block` tag that exists in the parent template overwrites the parent content instead of pulling it into the child template. So if you insert `block` tags that exist in the parent template and don't include any code between the tags, then you essentially delete the parent content that is otherwise included in that block.
- However, in this case you want to add to the form from the parent template rather than simply deleting or overwriting it. To pull in the parent content from the block, `checkout-firstname-form` uses the `{% parent %}` tag. This tag copies the contents of the block it is in, `{% block firstname-form %}`, from the parent template into `checkout-firstname-form`.
- Then, to add the button to the form pulled in from the parent template, `checkout-firstname-form` adds the code `<button>Next</button>`.

You must enclose all child content within `block` tags that exist in the parent template. Otherwise, there is no way for Hypr to know where to render the child content in relation to the parent content. This means that if you wanted to add more content to `checkout-firstname-form` that was not related to `{% block firstname-form %}`, you would have to enclose that content within separate `block` tags that existed in the parent template. You could also create a new block so long as you include it within an existing parent template block, as shown in the following example, where the `help` block does not exist in the parent template but is nested within a block that does exist in the parent template.

```
{% extends "modules/firstname-form" %}
{% block firstname-form %}
   {% parent %}
   <button>Next<button>

   {% block help %}
      {% require_script "modules/help" %}
   {% endblock help %}
{% endblock firstname-form %}
```

Templates Can Include Other Templates

In addition to extending templates, you can include the contents of a template within another template. This is particularly useful for including small pieces of reusable content, such as the templates you find in the `modules` folder. Unlike when you extend templates, including templates also pulls in content that is outside of `block` tags. The following example includes rather than extends the template from the previous section, `firstname-form`:

```
<div class="firstname-form">
   {% include "modules/firstname-form" with addresscontext="ShippingAddress." model=model.fu
lfillmentInfo.fulfillmentContact %}
</div>

{% comment %}
This template outputs:
<div class="firstname-form">
   <div class="row">
      <input type="text" name="firstname" value="Jerms" data-mz-value="ShippingAddress.First
Name">
   </div>
</div>
<div class="invisible-man">child templates can't see me!</div>
{% endcomment %}
```

In the example, notice how you can use the *`with`* parameter alongside the `include` tag to define variables for use in the included template. In this case, you set the variables `addresscontext` and `model` and the `firstname-form` template uses their values to render its content.

Assign Variables to Expensive Operations

Hypr allows you to set variables to the result of expensive operations or conditional statements. This improves site performance by reducing the number of operations that need to take place. The best way to set variables to the result of an operation or conditional is to use the `with` tag, as shown in the following example. Recall that the scope of the variables is limited to the *`with`* tag.

```
{% with themeSettings.freshmen|findwhere('userId', user.userId) as freshman %}
   {% with user.isAnonymous or freshman and now|is_after("9:00 PM") as curfew %}
      {% if curfew %}
         FRESHMAN {{ freshman.name|upper }}!!! CURFEW IS IN EFFECT. BACK TO THE DORMS!!!
      {% endif %}
      <p>Welcome, freshmen, to a very important school that is not at all a half-remembered send
-up of a Nickelodeon show from 1994 or so.</p>
      {% if curfew %}
      ARE YOU STILL READING THIS? CURFEW IS IN EFFECT!!!!!
      {% endif %}
   {% endwith %}
{% endwith %}
```

Insert Content Within Items in a List

Hypr is a powerful language that provides you plenty of flexibility for rendering content. An example is inserting content between or within items in a list. The following code demonstrates how to do this with `for`, `if`, and `with` tags. Notice how the `forloop.last` property is used to avoid inserting a comma after the last color is rendered. Refer to the reference help for full details on how tags help make theme development easier.

`templates/modules/list-colors.hypr:`

```
{% with get_product_attribute_values("tenant~colors") as colors %}
   {% if colors.length > 0 %}
      <li><strong>Available Colors</strong>:
         {% for color in colors %}
            {{ color.stringValue }}{% if not forloop.last %}, {% endif %}
         {% endfor %}
      </li>
   {% endif %}
{% endwith %}

{% comment %}
This template outputs:
<li><strong>Available Colors</strong>: White, Blue, Red, Viridian</li>
{% endcomment %}
```

Substitute and Filter Text

Hypr provides several filters for processing text. For example, Hypr doesn't evaluate inside your labels file, but you can use the `string_format` filter to create labels with placeholder values that a template later fills in:

```
...
   "acceptsMarketing": "Yes, I would love to receive fantastic offers from {0} at the following emai
l address: {1}",
...
```

```
<input type="checkbox" name="acceptsMarketing" id="acceptsMarketing" value="on"><label>f
or="acceptsMarketing">labels.acceptsMarketing|string_format(siteContext.generalSettings.websit
eName, user.emailAddress)</label>
```

Another useful Hypr feature is to use the `filter` tag to apply filters to a section of content. For example, you can set a

filter to replace certain words in a dropzone where a Site Builder user might add text to a widget. Note, too, that Hypr lets you string multiple filters together.

```
{% filter replace("AC", "Awesome Company")|replace("awesome company", "Awesome Company")
|replace("Awesome Co.","Awesome Company") %}
   {% dropzone "product-description" %}
{% endfilter %}
```

Autoescape Text

For security reasons, `page.hypr` autoescapes its entire contents. This converts special characters into HTML entities to prevent cross-site scripting attacks, and it applies the autoescaping to included templates as well. Since every page template extends `page.hypr`, all your templates are autoescaped by default. If you ever want to output raw HTML, you can use the `safe` filter.

```
{% autoescape on %}
...
{# page.hypr contents #}
...
{% autoescape off %}
```

Hypr versus HyprLive

In working with themes, you may come across the terms 'Hypr' and 'HyprLive'. What's the difference? Put simply:

- **Hypr** is the server-side rendering engine. Hypr templates are denoted by the `.hypr` extension, but Hypr can also render templates with the `.hypr.live` extension.
- **HyprLive** is the client-side rendering engine. HyprLive templates are denoted by the `.hypr.live` extension.

Hypr renders all your pages on the server during initial page load. HyprLive renders page updates on the client after the initial server rendering. The rendering process is composed of the following steps:

1. Hypr renders every page on initial page load, including pages that have a `hypr.live` extension.
2. Some pages contain JavaScript, which is used to render page updates after the initial Hypr rendering. The JavaScript code specifies a Backbone view, which in turn specifies a HyprLive template for rendering the new data. Typically, pages that contain JavaScript are located in the `scripts/pages` directory and are required with a `require_scripts` tag.
3. The Backbone views take effect when the JavaScript code runs. The Backbone views also subscribe to events from Kibo eCommerce (like a model update). In either case, the views trigger HyprLive to render an updated template for the current page.

Syntax and Support for Tags and Filters

To learn about the commonalities and differences in syntax between Hypr and HyprLive, as well as the tags and filters that each rendering engine supports, refer to the reference help.

Use Labels in Email Templates

Use labels to easily customize certain user-facing values in Hypr templates. Labels see particularly widespread use in the email templates located in the `templates/email` directory. You can edit your theme's labels within the `labels` directory.

For example, you can edit the labels `inStockBlob1` and `inStockBlob2` in the `en-US` JSON file...

```
...
"inStockBlob1": "<p>Thank you for shopping at {0}!  We have good news for you-- we now have t
he following item(s) in stock:</p>",
"inStockBlob2": "<p>TO ADD THE ITEMS TO YOUR CART: <a href=\"http://{0}/p/{1}\">CLICK HER
E</a>.</p>",
...
```

...and then use the values you specified in the labels JSON file within the `product-instock.hypr` email template:

```
...
{{ labels.inStockBlob1|string_format(siteContext.generalSettings.websiteName)|safe }}

...

{{ labels.inStockBlob2|string_format(domainName, model.productCode)|safe }}

...
```

There are many other labels available for use in your templates, and you can also create your own and then access them within a Hypr template using the `labels` variable, as shown in the preceding example.

For more information about labels, refer to the Labels topic.