Flex

A brief introduction to the 'Flex' codebase.

Terminology and General Architecture:

- **C8** Avetti's Commerce-8 platform: a multi-vendor ecommerce platform.
- **cluster**:
  - a cluster denotes a logical grouping of EC2 instances (virtual servers) comprising staging (preview) and production (shop) environments. C3 denotes *cluster 3* and the staging/preview environment is ggc8admin3.avetti.ca and the production/shop environment is ggc8prod3.avetti.ca
  - Backend-driven:
    - API built on Java/Spring.
    - Velocity (Java-based templating engine) renders VTL templates into markup server-side, which is returned as an HTTP response
    - Templates and shop-specific data stored in MySQL database. We interface with templates via Avetti's built-in template editor, or through a Sublime plugin which abstracts away the underlying template data stored in MySQL.
    - NGINX is used as a reverse-proxy, allowing clients to interface with backend API, and for serving static content (images, stylesheets), and handles domain mapping.
- **shop**
  - A discrete site containing it's own set of velocity templates, configuration, and static files
  - Each shop has a corresponding Vendor ID (vid)
  - On C8, a shop's static files are located in the directory: /avetti/httpd/htdocs/content/preview/store/{vid}
  - css files are stored in ../{vid}/assets/themes/blaze_en/css"
- **Master Template Library (MTL)**
  - A shop on each cluster which acts as a central repository for velocity templates Globally scoped, templates and methods leveraged by all shops on a cluster.

Flex MTL:

About

- Coined *Flex* after the CSS layout, *Flexbox* to distinguish the fact that our sites are responsive and mobile friendly.
- As of 2020, all new sites are developed in Flex.
- Flex employs modular design patterns (components, dependencies and partials)
- Syntactically Awesome Stylesheets (SASS) is used on all Flex sites.

Architecture

The Flex MTL codebase is separated into two parts: **velocity templates**, and **skeleton library**. On the velocity template side, there are three core patterns on Flex, each of which provide a layer of abstraction over underlying static templates:

2.

Dependencies

Making use of dependency patterns allows us to break coherent functionality off into its own discrete 'contract', leading to code that is easier to manage, test, and re-use.

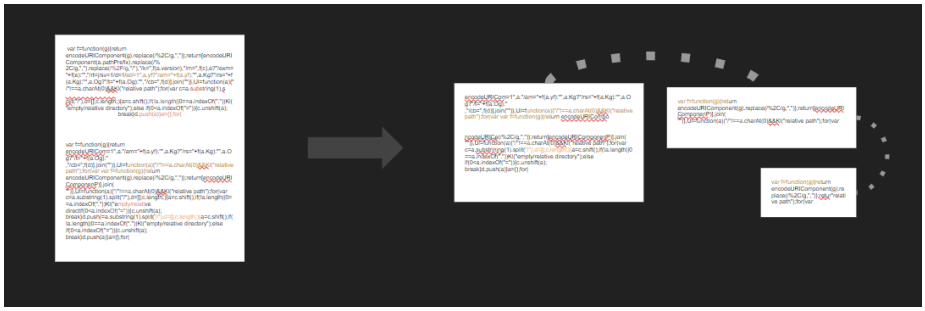

Figure 1: dependency.png

Defining, Using, and Rendering Dependencies

Define a dependency:

```
#defineDependency('my-dependency', {
    'headMarkup' : [
        "<script src='...'></script>",
"/$vendorSettingsDTO.vendorId/$vendorSettingsDTO.themeId/js_template.vm",
] ,
    'footMarkup' : [
        "Asset_1",
        "Asset_2",
             ...
        "Asset_n"
        ] ,

'requires' : ['jquery', 'dep_2', ... 'dep_n']

})
```

The `#defineDependency` macro is passed a dependency name, and a hashmap which may contain section-name/static-asset pairs along with a list of sub-dependencies required. Above, 'headMarkup' maps to a list containing a script and a local template.

Use and render:

```
#useDependency('my-dependency')
#renderDependencies('headMarkup')
```

The `useDependency` macro flags a dependency as being used. Will render out any dependency sections, if `#renderDependencies(sectionName)` is called.

Dependencies in action:

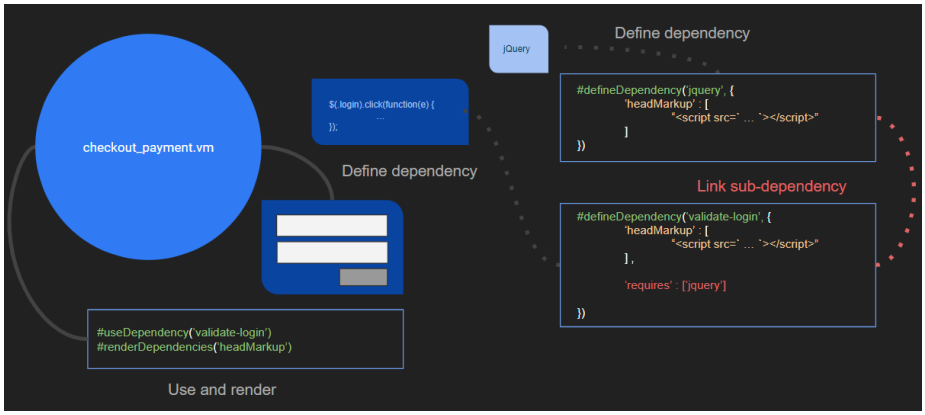

Figure 2: dependency_in_action.png

Note: when adding a new dependency to a local store, and not the MTL, include it as a new entry in `$configDependencies`, found at the bottom of template `configs.vm`

The MTL template `lib_macros_dependencies.vm` contains the code responsible for using and rendering dependencies. Open this template to explore the underlying mechanics if desired!

3.

components

Components allow you to split the UI into independent, reusable pieces and provide abstraction over a predefined set of properties (DefaultSettings) and any underlying code describing how a segment of the user interface should be rendered. The key takeaway is modularity and reusability.

Creating a component:

We create local component template `comp_componentName.vm` and call the `#registerComponent()` macro, which is defined in lib_macros_components.vm in the MTL.

Component meta-data including author, version, and HelpURL are then set:

```
#registerComponent('componentName')
#setComponentAuthor('cpweb@geiger.com')
#setComponentVersion('1.0.0')
#setComponentHelpURL('')
```

We can use the `#addComponentDefaultSetting('name', 'value')` macro to instantiate default settings. State is managed via component settings. Think of settings as private objects within the component:

```
#registerComponent('componentName')
#setComponentAuthor('cpweb@geiger.com')
#setComponentVersion('1.0.0')
#setComponentHelpURL('')
#addComponentDefaultSetting('settingName', 'value')
```

We define a render macro. componentName should match the registered component name and because macro names are globally scoped across the cluster, we are using vids to ensure names are unique. Within the render macro, we can use the `#getComponentSetting()` macro to return the value associated with a setting name.

```
#registerComponent('componentName')
#setComponentAuthor('cpweb@geiger.com')
#setComponentVersion('1.0.0')
#setComponentHelpURL('')
#addComponentDefaultSetting('settingName', 'value')

#macro(render_componentName_20181005604)
    #getComponentSetting('settingName')
#end
```

Testing macro is set:

```
#registerComponent('componentName')
#setComponentAuthor('cpweb@geiger.com')
#setComponentVersion('1.0.0')
#setComponentHelpURL('')
#addComponentDefaultSetting('settingName', 'value')

#macro(render_componentName_20181005604)

#end
```

```
#macro(test_componentName_20181005604)

#end
```

Using a component:

We can now use and render our component in another template on the site:

```
#useComponent('componentName')
```

Calling the #setComponentSetting macro allows us to override a componentDefaultSetting:

```
#useComponent('componentName')
#setComponentSetting('componentName', 'settingName', ‘value’)
```

`#renderedComponent` is called, which renders the component:

```
#useComponent('componentName')
#setComponentSetting('componentName', 'settingName', ‘value’)
#renderComponent('componentName')
```

The MTL template `lib_macros_components.vm` contains the code responsible for using and rendering dependencies. Open this template to explore the underlying mechanics if desired!

skeleton library:

There are 23 javascript files which comprise the Flex Skeleton Library:

- ss.breakpoint-imaging.js
- ss.cache.js
- ss.categories.js
- ss.checkout-address.js
- ss.checkout-basket.js
- ss.checkout-payment.js
- ss.checkout-review.js
- ss.custom-modals.js
- ss.customers.js
- ss.custprops-admin.js
- ss.date.js
- ss.global.js
- ss.invoicing-admin.js
- ss.js
- ss.message-box.js
- ss.minicart.js
- ss.navBuilder.js
- ss.points-admin.js

- ss.price.js
- ss.product-detail.js
- ss.product-tabs.js
- ss.shoppergroup-admin.js
- ss.wishlist.js

Each of the files above are loaded through a separate dependency, defined in `lib_configs.vm` in the Flex MTL. On the preview environment, these assets are located in the `/avetti/httpd/htdocs/content/preview/store/20170604234/assets/js` directory.

Each .js file is of the following form:

```
ss.dependency_name = (function (){

    var dependency_name = {}

    dependency_name.defaultConfigs = {
        'key_1' : 'value_1',
        ...,
        'key_n : 'value_n'
    }

    dependency_name.globalFunction = function() {

    }

    var localFunction = function() {

    }

    return dependency_name
})()
```

Along with the .js file on disk, the skeleton store dependencies defined in `lib_configs.vm` may also leverage a separate velocity template used to initialize values in the .js file.

The `ss.product-detail` dependency is an example of this:

```
#defineDependency('ss.product-detail', {
    'requires': ['ss', 'ss.cache', 'ss.price', 'ss.minicart', 'dialog-polyfill', 'jquery', '
    'footMarkup': [
        "<script src='store/$configMTLVID/assets/js/ss.product-detail.js'></script>",
        "lib_js_product_detail.vm"
    ]
})
```

On a shop's `item.vm` template, `#useDependency('ss.product-detail')` is called, the 9 other dependencies required by `ss.product-detail` are loaded, and when `#renderDependencies('footMarkup')` is rendered at bottom of the item.vm template, `store/$configMTLVID/assets/js/ss.product-detail.js` is rendered, alongside the content of MTL template `lib_js_product_detail.vm`.

Taking a look at the contents of lib_js_product_detail.vm reveals, amongst other things, the initialization of a number of default settings for product_detail:

```
ss.product_detail.defaultConfigs.excludeProductImage = 'noimage.jpg';
ss.product_detail.itemCode = "$esc.javascript($item.itemCode)";
ss.product_detail.itemID = "$esc.javascript($item.itemid)";
...
```

Your Flex Store:

I've setup a test site that you can modify and learn on:

When logging in via https://ggc8admin3.avetti.ca/preview/login.admin

You'll find the website named `CJORDAN SANDBOX` with store id 20210422954

Clicking the store id will allow you to access settings for this Shop. From there, feel free to poke around. 'templates -> manage templates' will allow you to view all templates for this shop. Template `home.vm` is rendered when hitting your site link, on preview:

https://ggc8admin3.avetti.ca/preview/store.html?vid=20210422954

Things to cover...

1. General store configuration
2. Defining new local dependencies
3. Changing default settings for components
4. Overriding an MTL dependency
5. Stylesheets: Skinning process, skinning tool
6. Transfering files via sFTP
7. Interfacing with velocity templates using sublime