Marketplace Documentation

Release 0.1

Marketplace Developers

September 27, 2017

Contents

In-depth developer documentation for Firefox Marketplace frontend projects.

CHAPTER 1

Related Documentation

- Firefox Marketplace high-level documentation
- Firefox Marketplace API documentation
- Old Commonplace Wiki

CHAPTER 2

Frontend Projects and Components

Marketplace frontend projects all share a common framework. Current projects include:

- Marketplace Frontend
- Marketplace Template
- Marketplace Curation Tools
- Marketplace Communication Dashboard
- Marketplace Operator Dashboard
- Marketplace Statistics
- Marketplace Style Guide

Pieces that make up the framework include:

- Marketplace Core Modules
- Marketplace Gulp
- Marketplace Node Modules
- Marketplace API Mock
- Marketplace Elements
- Marketplace jQuery

CHAPTER 3

Contents

Usage

This section is about installation, command-line interface, and configuration of Marketplace frontend projects.

Installation

You will need Node and NPM installed.

Setting up an instance of the Marketplace frontend for development is very simple. For this example, we'll check out Fireplace:

```
git clone git@github.com:mozilla/fireplace
npm install
make install
make serve
```

This will:

- Clone the project
- Install Node and Bower dependencies
- Copy assets from *bower_components* into the project source directory
- Generate a local settings file at *src/media/js/settings_local.js*
- Generate a require.js with an injected paths and shim configuration
- Generate an index.html file with an injected LiveReload script
- Start a local webserver with a filesystem watcher

Pulling in Updates

When fetching updates (e.g., after *git pull*), due to often-updating Bower and Node dependencies, you will often have to run:

```
make install
```

This will also do every part of the installation step above (except generate a local file since that has already been done).

About the Webserver

The webserver launched by *make serve* will watch the filesystem for changes and recompile anything if necessary. Here is what the webserver watches for:

- When a Stylus file is modified, then only that specific Stylus file will be recompiled
- When a Stylus library file is modified (*css/lib* or *css/lib.styl*), every Stylus files will be recompiled
- When a HTML/Nunjucks template is modified, *src/templates.js* will be recompiled via Nunjucks
- When a root HTML file is modified (*src/*.html*), *index.html* will be re-generated from the active template

To run the webserver on a different port:

```
MKT_PORT=8000 make serve
```

To serve with compressed assets (bundled CSS/JS/templates with no RequireJS), pass in the *MKT_COMPILED* flag. This is useful for testing in a more production-like environment:

```
MKT_COMPILED=1 make serve
```

The webserver will also launch a LiveReload server for CSS modifications. When CSS is recompiled, the browser sessions will automatically refresh their CSS stylesheets live without a page refresh. To disable LiveReload:

```
NO_LIVERELOAD=1 make serve
```

The webserver will rewrite `src/media/js/lib/marketplace-core-modules` to the `bower_components` directory such that the modules don't need to be copied into the project. Our build tools at `marketplace-gulp` will be able to find the JS modules in the `bower_components` directory. We plan on doing this with our other JS dependencies.

Generated index.html

Note that *src/index.html* is a generated file. This allows us to:

- Easily serve different templates when we want by copying other templates to *src/index.html* (such as when specifying *TEMPLATE* like above)
- Inject a LiveReload script (with the correct ports) into the body
- Have a single dev.html template that can switch back and forth between serving development assets and compiled assets

Building the Project for Production

Outside of the dev environment, we bundle all of our CSS, JS, and templates. We use Gulp to build our projects; our Gulp code lives in Marketplace Gulp. To run the build system:

```
make build
```

About the CSS builds:

- CSS listed in *src/dev.html* will be minified and concatenated into *src/media/css/include.css* in the order that they are listed in *src/dev.html*
- Extra CSS bundles can be configured in *config.js*, such as we do for *src/media/css/splash.css*. The files used to create the bundle and the bundle itself will not be minified into *src/media/css/include.css*

About the JS builds:

- The JS will be bundled into *src/media/js/include.js*
- JS is run through a RequireJS optimizer
- The RequireJS optimizer will build a dependency tree by parsing our defines and requires to only include modules that are used. Although, it is configurable to include and exclude what we want.
- The RequireJS configuration used for local development is also passed in to the our RequireJS optimizer
- almond, a lightweight AMD loader, will be prepended onto the bundle
- A sourcemap will be generated at *src/media/js/include.js.map*

About template builds:

- We use Nunjucks to compile all our templates into *src/templates.js*
- We have Node modules that monkeypatch the official Nunjucks compiler to perform various (non-upstream compatible) optimizations to reduce the size of our templates bundle

Other things that will be generated:

- *src/media/build_id.txt* contains the timestamp during that build, which is used to cachebust our assets in production
- *src/media/imgurls.txt* contains image URLs found in our CSS stylesheets, which is used to generate an appcache manifest on the server

If you want to test the project with the built bundles above, serve with a template that uses the bundle, such as *src/app.html*. Read about the webserver above for more details.

If you want to disable uglification and minification of JS and CSS:

```
MKT_NO_MINIFY=1 make build
```

Additional Command-Line Interface

Most of our commands are brought to you by our build system and task runner, Gulp. And most of the useful ones have been aliased with Makefile directives such that we don't have to expect developers to have Gulp installed globally. For commands that do not have Makefile aliases, if you don't have Gulp installed globally, you can run Gulp through *node_modules/.bin/gulp*.

You won't often need these, but here is a list of commands not mentioned above:

- *make clean* - deletes generated and temporary files
- *make lint* - lints the project's JS with JSHint
- *gulp bower_copy* - performs the Bower copying step described in *Installation*
- *gulp require_config* - performs the require.js generation described in *Installation*
- *gulp css_compile* - compiles Stylus files
- *gulp templates_build* - compiles Nunjucks templates
- *node_modules/.bin/commonplace langpacks* - extracts locales into JS modules

Changing API Settings

To conveniently change the API settings (i.e., which server the project points to) in `settings_local.js`, set the environment variable `API` to one of the server names below:

```
prod, dev, stage, altdev, paymentsalt, localhost, mpdev, mock, mocklocal
```

For example:

```
API=prod make serve
API=mock make test
```

This will simply overwrite the `api_url` and `media_url` in your current `settings_local.js`. To view the mappings, check out the Marketplace Node modules config.

Bower and RequireJS Configuration

Above we mentioned the installation and update steps will:

- Copy assets from *bower_components* into the project
- Generate a require.js with an injected paths and shim configuration

These two things, setting up Bower and RequireJS, do not happen magically. They are both specifically configured (though with reusable code and handy loops).

The base configuration lives in Commonplace, our Node modules, in *lib/config.js*. This configuration ships and is required with every frontend project. It sets up Bower copying paths, and RequireJS paths and shims for modules that we know ships with every frontend project (e.g., marketplace-core-modules).

There are two exported configuration objects, one for Bower and one for RequireJS.

Bower Configuration

We use Gulp to copy files from *bower_components* into our project source. This is standard. Bower recommends not serving up the *bower_components* directory statically for security reasons, and using a build tool such as Gulp or Bower to process components.

The Bower configuration, *require('commonplace').bowerConfig*, for example may look like:

```
{
    'jquery/jquery.js': 'src/media/js/lib/',
    'marketplace-frontend/src/templates/feed.html': 'src/templates'
}
```

RequireJS Configuration

The keys of the object specifies the source path of the file within *bower_components*. The values of the object specify the destination path. The RequireJS configuration, *require('commonplace').requireConfig*, for example may look like:

```
{
    paths: {
        'jquery': 'lib/jquery'
    },
    shim: {
        'underscore': {
            'exports': '_'
        }
    }
}
```

This will be used to generate a *require.js* file that contains an injected *require.config*. It is also used during our RequireJS optimization build step. Our project runs on AMD so understanding RequireJS configuration is very helpful.

Note that anything you wish to shim must be specified with a valid *exports*. If your module doesn't export/expose anything, just set it to *window* or something.

Extending the Base Configuration

The base Commonplace configuration can be extended within frontend projects in *config.js*. It will become straightforward once you check out the file. We extend the base configuration usually if we want to add a module or component that only matters one of the several Marketplace frontend projects.

Packaged App

Read about Packaged Apps in the top-level documentation.

Marketplace Framework

This section describes on a high-level the framework that we use when developing our frontend projects. Let's call it the Marketplace Framework. The Marketplace Framework is an in-house MVC framework comprised of AMD modules and Nunjucks templates that allows us build to performant single-page apps. We'll go over in-depth what the Marketplace Framework looks like.

If you are curious why we have our own framework, and not use something like Backbone or Angular? Simply put for Backbone, it had a lot stuff we didn't need like syncing and data manipulation as the main Marketplace is largely read-only, and it didn't suit our needs. Angular was tempting, but we wanted the flexibility to render templates server-side, and the l10n support wasn't quite there.

If you want a quick run-down, you can watch a 20-minute video of the "masta" of the framework, Matt Basta, talking about the framework and the decisions behind it. We highly recommend watching it.

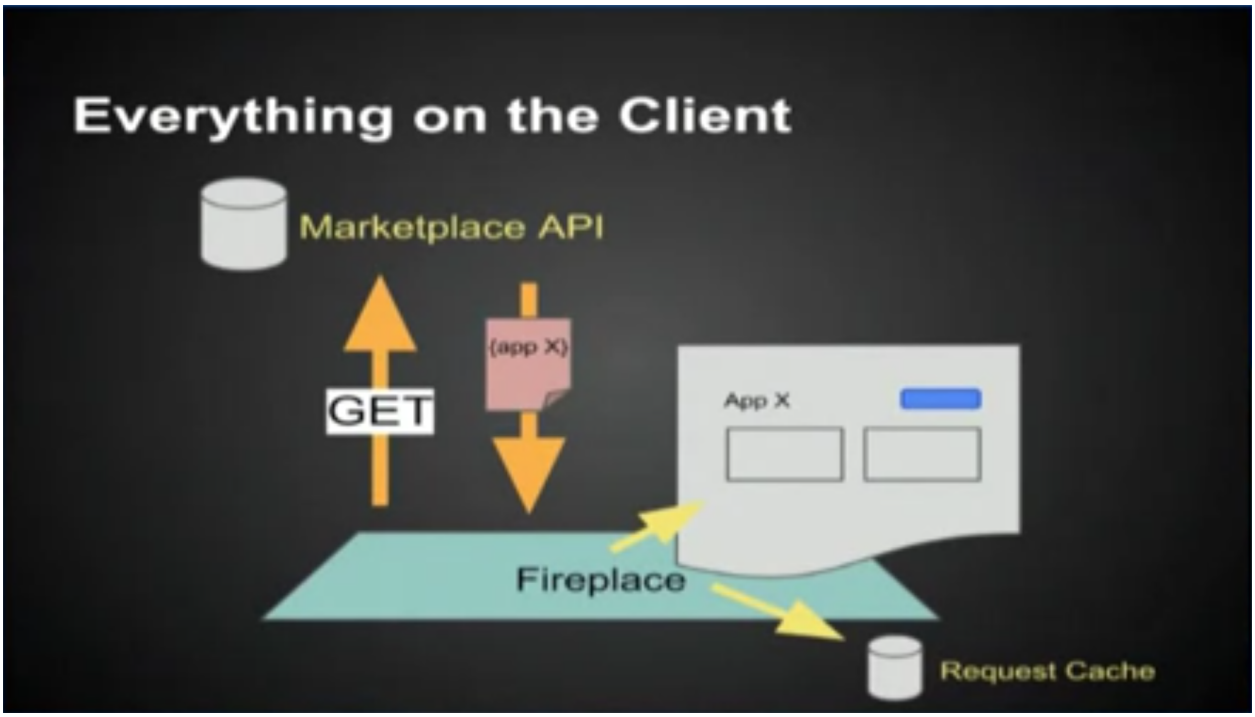
Goals of the Marketplace Framework

Performance (and perceived performance) on low-end devices is the headlining goal:

- Have everything client-side (even the templates) for less requests
- Use RESTful APIs for everything for asynchronous and deferred loading
- Always show something (throbbers) to the user while data is being fetched
- Cache as much as possible
- Modular components and reusable code
- Pave the way for the Marketplace to become a packaged app.

Everything Client-Side

Here is a simple diagram of the client-side Marketplace (codename Fireplace). It's pretty straightforward description of a single-page app:



- Data is requested from the API
- The client receives the data
- The client renders the page with the data
- The client stores the data in its request cache (key-value store keyed by URL)

When we navigate to another page, we simply have to fetch data from another endpoint since the page rendering happens client-side. When we navigate to an already-visited page, the request goes to the cache. Less requests, better performance.

Fetching and Rendering Data from RESTful APIs

While an asynchronous request to an API endpoint is being made, we generally want to show something to the user in the process, like a spinner. And when the data comes in, we want to replace that *placeholder* with rendered markup. To do so, the Marketplace Frontend implements in the page rendering engine something called a *defer block*, which is used from the templates. Here is a visual representation using for example an Marketplace app detail page:

From the templates, it might look like:

```
{% defer (url='https://marketplace.firefox.com/api/v2/app/twitter') %}
  <h1 class="app-name">{{ this.name }}</h1>
{% placeholder %}
  <p>Loading app data...</p>
  <div class="spinner"></div>
{% end %}
```

Each defer block is a request to an API endpoint. In the defer block *signature*, you pass in a URL. In the defer block

body, you add the templating markup you want displayed once the data comes in. In the *placeholder block*, you add templating markup you want displayed while the data is loading. The defer block is one of the magical facilities that the Marketplace Framework provides.

Caching

Caching makes the Marketplace fast. When data is cached, the client doesn't have to wait for API responses and can render data almost immediately. On top of caching data in memory, we also persistently cache the data in LocalStorage so it stays in effect on subsequent visits. We have two forms of caching:

- request caching
- model caching

Request Caching

Remember the defer blocks above? Under the hood, it caches data from its API endpoint. It does this by invoking our requests module, *request-caches* all GET requests by default into memory, which will later be pesistently cached in LocalStorage. An entry in the request cache is keyed by the API endpoint URL which points to the value of the response. For example:

```
{
    'https://marketplace.firefox.com/api/v2/app/twitter': {
        'author': 'Twitter',
        'name': 'Twitter',
        ...
    }
}
```

Model Caching

Model caching is a more granular form of caching. It is useful when we request a list of data, and each entry in that list represents an object. For instance, an app listing endpoint returns a list of apps. Model caching allows us to extract from the list and store each object in the cache individually such that we can access that specific object individually.

The defer block handles this as well, though we need to pass in some parameters to its signature. For instance:

```
{% defer (url='https://marketplace.firefox.com/api/v2/feed/collection/list-of-apps',
          pluck='apps', as='app', key='slug')
  ...
{% end %}
```

This requests an endpoint that returns a list of apps. We tell the defer block to *model-cache* these *as* an *app* and *key* them in the cache by their slug field. Whereas request caching invokes our requests module, model caching invokes our models module (although the models module will call our requests module if the model cache hasn't been primed). The model cache might look like:

```
{
    'apps': {
        'facebook': {
            'author': 'Facebook',
            'name': 'Facebook',
            'slug': 'facebook',
        },
        'twitter': {
            'author': 'Twitter',
```

```
            'name': 'Twitter',
            'slug': 'twitter',
            ...
        }
    }
}
```

Modular Components and Reusable Code

The Marketplace Framework is split into many different repositories, all hosted on an appropriate package manager (NPM or Bower). These components are reusable, as in any projects can consume them. And they are modular, such that they can be updated one piece at a time, and projects and pull in those updates.

With stuff separately hosted in NPM and Bower, when a component is updated and a frontend project wishes to pull in the updates, all that needs to be done is to bump the version number in the *package.json* or *bower.json*.

For example, we have a component, Marketplace Core Modules. Whenever a project wishes to update these modules, they push an update to the repository and tag a version. Then other projects are able to enjoy the benefits of the updated modules by bumping their `bower.json` to the updated tag.

If you want a complete listing of all the modules and components, head to the index page of this documentation.

Page Building

This section details the entire process of page building, from routing to views to the builder object to the defer block. It is MVC-esque and is the meat of the Marketplace Framework.

Routing

All pages should have a route that which is used to resolve the appropriate view. To make a page accessible, you must create a route in the project's `src/media/js/routes.js`. A route is represented by an object in the list of routes. The `routes.js` file may look like:

```
function() {
    // The list of routes.
    var routes = window.routes = [
        {'pattern': '^$', 'view_name': 'homepage'},
        {'pattern': '^/app/([^/<>"\']+)/?$', 'view_name': 'app'},
    ];

    define('routes', [
        'views/app',
        'views/homepage'
    ], function() {
        for (var i = 0; i < routes.length; i++) {
            var route = routes[i];
            var view = require('views/' + route.view_name);
            route.view = view;
        }
        return routes;
    });
}();
```

Each route object should match the following prototype:

```
{
    'pattern': '<regexp>',  // Regular expression to match current path.
    'view_name': '<view name>'  // Name of view module w/o 'views/' prefix.
}
```

In the define call, the list of routes is looped over and the respective view modules are attached to the route objects. Note that the define call **must** list every view's module name as a dependency as seen above.

When navigating or loading a page, `marketplace-core-modules/views.js` will loop over all the routes in sequence and try to match it with the current path. If no match is found, it will just redirect the browser to the navigating URL. When a pattern is matched, the view (which returns a builder object as we will see later) will be invoked to build the page.

Regular Expressions

There are some quirks with regular expressions in routes:

- **non-matching groups may not be used** - cannot use a group simply to tell the parser that something is optional. Groups are only for parameters during matching the current path and reverse URL resolving.
- **? may not be used to perform greedy matching** - the optional operator can only be performed on static, non-repeating components.
- **multiple optional groups are not allowed** - optional groups are not evaluated greedily. Use multiple routes instead.

Building Views

Each view is represented by its own module in the `src/media/js/views` directory. Here's an example view:

```
define('views/myview', ['l10n', 'z'], function(l10n, z) {
    // -- MODULE LEVEL --.
    var gettext = l10n.gettext;

    z.page.on('click', 'button', function() {
        console.log('Put your delegated event handlers on this level.');
    });

    return function(builder) {
        // -- BUILDER LEVEL --.
        builder.z('title', gettext('My View Page Title'));
        builder.z('type', 'leaf');

        builder.start('some_template.html', {
            someContextVar: true
        });
    };
});
```

At the the module level, the view usually consists of mostly delegated event handlers. At the *Builder level*, inside the function returned by the view, consists of code that handles the actual page rendering using the injected *Builder object*.

Guidelines

Here are some important guidelines around building views:

- You should **not** perform **any** delegated event bindings on elements that are not a part of the page (e.g., `z.page.on('click', '.element-from-another-page'...)`).
- And all delegated event bindings on non-page elements should be bound at the module level. This prevents *listener leaks*.
- Expensive lookups, queries, and data structures should be done **outside** the Builder level. This ensures that any value is computed only once.
- Delegated events should be used whenever possible, and state should be preserved on affected elements (e.g., via data attributes) rather than within variables.
- State variables should never exist at the module level unless it represents a persistent state.

The Builder Object

A Builder object is injected into the function returned by the view when. `marketplace-core-modules/views.js` matches the respective route and invokes the view. The Builder object is defined in `marketplace-core-modules/builder.js`. It is the pure meat behind page rendering, handling Nunjucks template rendering, defer block injections, pagination, requests, callbacks, and more.

And to note **the Builder object itself is a promise object** (jQuery Deferred-style). It has accepts `.done()` and `.fail()` handlers. These **represent the completion of the page as a whole** (including asynchronous API requests via defer blocks). However, the promise methods should not be used to set event handlers or modify page content. The `builder.onload` callback should be used instead which happens when each defer block returns, which makes the view more reactive and non-blocking. This will be described more below.

In the barebones example above, we set some context variables with `z`. Then we calls `builder.start` to start the build process, passing in the template to render and a context to render it with. After some magic, the page will render.

For more details and functionality, below describes the Builder object's API:

`builder.`**`start`**(*template*[, *context*])

Starts the build process by rendering a base template to the page.

Parameters

- **template** – path to the template
- **context** – object which provides extra context variables to the template

Return type the Builder object

`builder.`**`z`**(*key*, *value*)

Sets app context values. Any values are accepted and are stored in require('z').context. However, certain keys have special behaviors. 'title' sets the window's page title. 'type' sets the body's data-page-type data attribute.

Parameters

- **key** – name of the context value (e.g., 'title', 'type')
- **value** – the context value

Return type the Builder object

`builder.`**`onload`**(*defer_block_id*, *handler_function*)

Registers a callback for a defer block matching the defer_block_id. Depending on whether the defer block's URL or data was request-cached or model-cached, it may fire synchronously.

Given an example defer block `{% defer (url=api('foo'), id='some-id') %}`, we can register a handler for when data is finished fetching from the 'foo' endpoint like `builder.onload('some-id', function(endpoint_data) {...`

Parameters

- **defer_block_id** – ID of the defer block to register the handler to
- **handler_function** – the handler, which is given the resulting data from the defer block's API endpoint

Return type the Builder object

`builder.`**`terminate`**`()`
While the Builder object is often not accessible, if a reference to it is available and the current viewport is changing pages, calling builder.abort will terminate all outstanding HTTP requests and begin cleanup. This should never be called from a builder.onload handler.

Return type the Builder object

`builder.`**`results`**
An object containing results from API responses triggered in defer blocks. The results will be keyed by the ID of the defer blocks.

Defer Blocks

In the *Marketplace Framework* section about *Fetching and Rendering Data from RESTful APIs*, we introduced **defer blocks** as a page rendering component, invoked from the templates, that asynchronously fetches data from APIs and renders a template fragment once finished. We *heavily* recommend reading that section if you have not already. As we described the Builder as the meat of the framework, defer blocks are the magic.

`{% defer (api_url[, id[, pluck[, as[, key[, paginate[, nocache]]]]]]) %}`
Fetches data from api_url, renders the template in the body, and injects it into the page.

Parameters

- **api_url** – the API endpoint URL. Use the `api` helper and an API route name to reverse-resolve a URL. The response will be made available as variables `this` and `response`
- **id** – ID of the defer block, used to store API results and fire callbacks on the Builder object (Builder.onload)
- **pluck** – extracts a value from the returned response and reassigns the value to `this`. Often used to facilitate model caching. The original value of `this` is preserved in the variable `response`.
- **as** – determines which model to cache `this` into
- **key** – determines which field of `this` to use as key to store the object in the model. Used in conjunction with `as`
- **paginate** – selector for the wrapper of paginatable content to enable continuous pagination
- **nocache** – will not request-cache the response if enabled

A basic defer block looks like:

```
{% defer (url=api('foo')) %}
  <h1 class="app-name">{{ this.name }}</h1>
{% placeholder %}
  <div class="spinner"></div>
{% end %}
```

In this example, the defer block asynchronously loads the content at api('foo'). While waiting for the response, we can show something in the meantime with the `placeholder`. Once it has loaded, the content within the defer block

is injected into the page's template by replacing the placeholder. `this` will then contain the API response returned form the server.

The entire response, unless otherwise specified with `nocache`, will automatically be stored in the request cache. The request cache is an object with API endpoints as the key and the responses as the value. Note that if two defer blocks use the same API endpoint, the request will only be made once in the background.

{% placeholder %}

Defer blocks are designed with the idea that the base template is rendered immediately and content asynchronously loads in over time. This means that some placeholder content or loading text is often necessary. The placeholder extension to the defer block facilitates this:

```
{% defer (url=api('foo')) %}
  I just loaded {{ this.name }} from the `foo` endpoint.
{% placeholder %}
  This is what you see while the API is working away.
{% end %}
```

Note that placeholder blocks have access to the full context of the page.

{% except %}

If the request to the server leads to a non-200 response after redirects, an `{% except %}` extension block, if specified, will be rendered instead:

```
{% defer (url=api('foo')) %}
  I just loaded {{ this.name }}.
{% except %}
  {% if error == 404 %}
    Not found
  {% elif error == 403 %}
    Not allowed
  {% else %}
    Error
  {% endif %}
{% end %}
```

The variable `error` is defined in the context of the except block. `error` contains the erroring numeric HTTP response code, or a falsey value if there was a non-HTTP error. A default except template can be rendered if specified, which can be set in `settings.fragment_error_template`.

{% empty %}

If the value of `this` (i.e., the plucked value) is an empty list, an alternative `{% empty %}` extension is rendered. This is useful for lists where the amount of content is unknown in advance:

```
{% defer (url=api('foo')) %}
  <ul>
    {% for result in this %}
      <li>I just loaded {{ result.slug }}.</li>
    {% endfor %}
  </ul>
{% empty %}
  <div class="empty">
    Nothing was loaded.
```

```
  </div>
{% end %}
```

If `{% empty %}` is not defined, the defer block will run with the empty list with the current `this` variable.

Model Caching

In *Model Caching*, we briefly introduced model caching through defer blocks. Again, it allows us to cache API responses at an object level (e.g., an app from a list of apps) such that they can be retrieved individually (e.g., looked up at the app detail page). Here is an example:

```
{% defer (url='https://marketplace.firefox.com/api/v2/feed/collection/list-of-apps',
          pluck='apps', as='app', key='slug')
  ...
{% end %}
```

The defer block will make a request the endpoint. Consider the API response returns:

```
{
    'name': 'List of Apps',
    'slug': 'list-of-apps',
    'apps': {
        'facebook': {
            'author': 'Facebook',
            'name': 'Facebook',
            'slug': 'facebook',
        },
        'twitter': {
            'author': 'Twitter',
            'name': 'Twitter',
            'slug': 'twitter',
        },
    }
}
```

Let's go over it attribute by attribute:

- **`pluck='apps'`** tells the defer block to `pluck` the key `apps` will reassign the value of `this` in the defer block, which would normally contain the whole response, to the value of `apps` in the response. This will help isolate what we wish to model cache
- **`as='app'`** store it in the `app` model cache. Setting up a model cache will be described in the Caching section
- **`key='slug'`** use the app's `slug` field as a key to store in the model cache. Default keys can be configured in the settings which will be described in the Caching section

Once the response comes in, the apps will then automatically be model-cached. The `apps` model cache may then look like:

```
{
    'facebook': {
        'author': 'Facebook',
        'name': 'Facebook',
    },
    'twitter': {
        'author': 'Twitter',
        'name': 'Twitter',
    }
}
```

Pagination

Pagination is done by passing in a selector to the defer block's `paginate` and having an element `.loadmore button` inside the defer block. When the button is clicked, the defer block will re-run using the button's `data-url` as the API URL. The template is re-rendered with the new (next page) content while keeping what was previously within the pagination container. Thus, it essentially seem as if the new content was just appended to the pagination container. For example:

```
{% defer (url=api('foo'), paginate='ul.list') %}
  <ul class="list">
    {% for result in this %}
      <li>{{ this.bar }}</li>
    {% endfor %}
  </ul>
  <div class="load_more">
    <button data-url="{{ api('this.meta.next_page') }}">Load More</button>
  </div>
{% end %}
```

If pagination fails, a fallback template will be loaded instead as specified by `settings.pagination_error_template`. Note that this is separate behavior from the `{% except %}` block which renders when an API request errors. The error template's context is passed the variable `more_url`, the URL originally requested and failed. This is useful for creating a retry button.

Developing Components

This section is about developing, testing, and updating Marketplace Framework components such as:

- Marketplace Core Modules
- Marketplace Gulp.
- Commonplace.

We have two types of components: Bower components and Node modules.

Bower Components

Bower is the package manager we use for frontend assets. The assets are downloaded to `bower_components` and most are currently copied into the project with `make install`.

Development and Testing Workflow

To develop changes for `marketplace-core-module` first checkout a local copy using `git clone`. Once you have the code you can `cd` into the folder and run `bower link` (see the bower docs for more information). `bower` will now know that this is your local copy of `marketplace-core-modules`. To use your local copy in a frontend project like `fireplace` you can run `bower link marketplace-core-modules` in the frontend project's directory to tell `bower` to use your local copy. If you are not making local changes to `marketplace-core-modules` then you should not use a local copy to ensure that you are up to date. To stop using your local copy simply uninstall it with `bower uninstall marketplace-core-modules` and then run `make install` to get the latest version.

For most other Bower components, they are copied into JS and CSS `lib` directories into the source tree where the project can locate them and then gitignored. You can employ some strategies for developing these such as setting up symlinks. We are sorry for this.

Updating a Component

When pushing an update for a component:

- Bump the version in `bower.json` (e.g., *1.5.2* to *1.6.0*), following semver
- Commit the patch and push (e.g., `git commit -m v1.6.0 important bug fix`)
- Git tag the version (e.g., `git tag v1.6.0`)
- Push to Github (i.e., `git push origin v1.6.0 && git push origin master`)

To **consume** the updated component from a frontend project:

- Bump the component's version in the frontend project's `bower.json`
- Run `make install` to get these modules copied into your project and into your RequireJS development configuration
- Commit and push the `bower.json` to deploy the updated component for the project

Node Modules

We have node modules such as Commonplace and Marketplace Gulp.

Development and Testing Workflow

When developing a node module, it is annoying to have to continually rebuild it or copy files back and forth between a version-tracked directory. `npm link` eases this:

- Go to the project directory of the node module you are developing (e.g., marketplace-gulp)
- Run `npm link` to create a global link
- Go to the project directory of the frontend project you want test it with (e.g., fireplace)
- Run `npm link <PACKAGE_NAME>` to link-install the package

Then changes are made in the node module's project directory will be reflected within the frontend project it is being tested with.

When doing `npm link` for marketplace-gulp, make sure your project's gulpfile looks similar to this reference gulpfile. We require tasks exposed by marketplace-gulp, and attach it to the local Gulp object. This is a workaround for issues with `npm link` and Gulp.

Updating a Module

When pushing an update for a module:

- Bump the version in `package.json` (e.g., *1.5.2* to *1.6.0*)
- Commit and push to Github (i.e., `git push origin master`)
- Run `npm publish` to publish it to npm

To **consume** the updated component from a frontend project:

- Bump the module's version in the frontend project's `package.json`
- Run `make install` to update the project's node modules
- Commit and push the `package.json` to deploy the updated module for the project

Guidelines

When updating a module:

- Try to bump the `bower.json` or `package.json` in the same commit
- Prepend the commit message with the version (e.g., *v1.5.3 updated logs*)
- Use semver's Semantic Versioning

Testing

Marketplace frontend projects have both unit tests and end-to-end tests. These tests help catch regressions, validate user flow, and add confidence the codebase. When adding a feature, try to look to cover it with tests.

Our tests expect that you use the Marketplace API Mock. You can do so by setting in your local settings:

```
api_url: 'https://flue.paas.allizom.org',
media_url: 'https://flue.paas.allizom.org'
```

Expect to have to modify the Marketplace API Mock when writing tests for new features. When doing so, try to keep the responses predictable rather than random as to not introduce flakiness into our tests.

When modifying a module that has defined input and output, write a unit test. When modifying the UI or something that affects multiple pages and URLs, then write an end-to-end test.

Unit Tests

Unit tests live in:

- `<frontend-project>/tests/unit` for frontend projects
- `marketplace-core-modules/tests` for Marketplace Core Modules

To run the unit tests once:

```
make unittest
```

To continuosly run unit tests when files change:

```
make unittest-watch
```

Troubleshooting

If you encounter an error where the `karma` command cannot be found try running `rm -rf node_modules && npm install` to get a fresh copy of the node dependencies.

How They Work

The unit tests are powered by RequireJS, in terms of being able to "import" modules and unit test their interfaces. The tests use the following libraries:

- Mocha for defining tests.
- Chai for assertions.
- Sinon for mocks, stubs and spies.

- Squire for managing RequireJS.
- Karma to run the tests in a real browser.

Writing a Unit Test

A basic unit test may look like:

```
define('tests/unit/some-module', ['tests/unit/helpers'], function(h) {
    describe('someModule.someFunction', function() {
        it('gives the expected value',
            h.injector()
            .mock('someModuleToMock', {mockKey: mockValue})
            .run(['someModule'], function(someModule) {
                assert.equal(someModule.someFunction(),
                             'My Expected Value');
            }));
    });
});
```

Important things to note:

- `describe` and `it` come from Mocha. When the tests run the strings passed to `describe` and `it` will be combined to describe the test in the output. This test will be called "someModule.someFunction gives the expected result".
- `h.injector()` is a shorthand for `new Squire()`. It also takes a variable number of functions as arguments to intialize the mocking. These functions should accept the injector and call its `mock` method. There are some helpers included in `tests/unit/helpers.js`, such as `mockSettings` which can be used as: `h.injector(h.mockSettings({mySetting: 'foo'}))`.
- Calling `run` on the injector will automatically end the test for synchronous code. If you have asynchronous code you will need to use `require` instead and call Mocha's `done()` function when complete.
- See the Squire page for documentation on how to use Squire.

End-to-End Tests

We use CasperJS (*v1.1.0-beta3*) to write our end-to-end, or integration, tests. These tests live in the `tests` directory. This directory comprises of:

- `captures` contains screenshots taken whenever a test fails.
- `lib/constants.js` holds reusable constants.
- `lib/helpers.js` helps power our tests on top of CasperJS. Contains various assertion facilities, utilities, and sets up necessary callbacks.
- `ui` holds the actual test suites.

If you wish to run end-to-end tests with just one of the browser targets, you can run `make uitest-phantom` or `make uitest-slimer`. To run end-to-tests targetting both PhantomJS and SlimerJS, run:

```
make uitest
```

You can run a single test file or folder by setting `UITEST_FILE` environment variable:

```
UITEST_FILE=tests/ui/search.js make uitest-slimer
```

SlimerJS runs against an external Firefox binary. By default the tests will try to use `/Applications/Firefox.app/Contents/MacOS/firefox`. This only works on Mac and uses the installed version of Firefox. We recommended downloading and running tests with Firefox 30. You can then set the version of Firefox to use with `SLIMERJSLAUNCHER`. For convenience, include `export SLIMERJSLAUNCHER=/path/to/firefox` in your shell's setup script.

```
SLIMERJSLAUNCHER=/Applications/Firefox-30.app/Contenst/MacOS/firefox make uitest-slimer
```

Writing an End-to-End Test

The tests *usually* consist of telling CasperJS what to click and then asserting that a selector is visible. An example test:

```
casper.test.begin('Test Some Selector', {
    test: function(test) {
        helpers.startCasper({path: '/some/path'});

        helpers.waitForPageLoaded(function() {
            // Run an assertion.
            test.assertVisible('.some-selector',
                               'Check that Some Selector is visible');
            casper.click('.go-to-some-page');
        });

        casper.waitForSelector('.some-page', function() {
            test.assertVisible('.some-page',
                               'Check navigated to Some Page');
        });

        helpers.done(test);  // Required for test to run!
    },
});
```

`helpers` is always available and contains useful boilerplate such as initializing CasperJS. We pass a path to `startCasper` which CasperJS will tell the browser to initially load. Try to use `startCasper` within the `test function` to keep the Casper environment isolated.

We begin a test, named *Test Some Selector*, which takes an object. The `test` function is injected with the CasperJS test module which contains assertion facilities and callbacks. Then we run the test, but make sure that the `helpers.done(test)` callback is invoked at the end.

Check out the CasperJS docs and our existing Fireplace tests for clues on how to write end-to-end tests for our frontend projects.

Mocking Login

To mock login, run `helpers.fake_login()`. This will, within the PhantomJS browser context, set a fake shared-secret token, set user's apps and settings, add a login state on the body, and then asynchronously reload the page.

Usually, you will run `fake_login()` and then immediately use a `helpers.waitForPageLoaded()` to wait for the `fake_login()` to reload the page.

Executing Code Within the Browser Environment

The code within the tests themselves executes in Node runtime, not PhantomJS browser runtime. CasperJS handles the communication to the PhantomJS browser. If you wish to run something within browser environment, you can use `casper.evaluate`:

```
var returnValue = casper.evaluate(function() {
    window.querySelector('.some-selector').setAttribute('data-value', value);
    return window.querySelector('.some-selector').getAttribute(value);
});
```

`casper.evaluate` runs synchronously and is allowed to return primitive values up to the Node runtime.

Using waitFor's

waitFor methods are useful for making CasperJS wait until a condition is met before running assertions. Generally, timeouts should be avoided with *casper.wait*.

For example, on many tests, we tell CasperJS to `waitForSelector` on `body.loaded` which is how we know the page is done rendering. We can also do this when we click around with `casper.click`, and tell CasperJS to wait until a selector we expect to be visible is loaded.

Here is a list of commonly used *waitFor* methods:

- **waitForSelector** - wait for selector to exist in the DOM
- **waitWhileVisible** - wait for selector to disappear
- **waitUntilVisible** - wait for selector to appear
- **waitForUrl** - wait until casper has moved to the desired or matching url
- helpers.waitForPageLoaded - a custom waitFor helper we wrote that waits for page to load (`body.loaded`)

You can make custom waitFor by defining a function that returns true when a custom condition is met.

Debugging Tests

Some useful tips when debugging a failing test:

- Set the system environment variable, `SHOW_TEST_CONSOLE`, to see every

`console.log` that is sent to the client-side console. This is useful for debugging tests. - Set the environment varible, `FILTER_TEST_CONSOLE`, along with `SHOW_TEST_CONSOLE` to see only logs that start with whatever is passed to `FILTER_TEST_CONSOLE`. Often, you can console.log `[debug]` in a casper.evaluate context and filter on that to poke around. - Whenever a test fails, CasperJS will automatically take a screenshot using PhantomJS. The screenshot is stored in the `tests/captures` directory. Check it out to see what the page looked like when an assertion fails.

Tips and Guidelines

- Keep tests organized. Ideally, each test file tests a page or component, and each test (`casper.begin('Test...')`) tests a specific part of that page or component.
- If testing a page, place the test file in a location that would match the route of the page.
- If you write something reusable, consider adding it to `helpers.js`
- If you use a constant, consider adding it to `constants.js`
- Keep selectors short and specific. We don't want tests to break as UI changes are made. One-class-name selectors are preferred over element selectors.
- Avoid specific string checking as the test may break if strings are updated.
- If `setUp` is firing too early, then try running the code within `casper.once('page.initialized', function() {...)`.

Continuous Integration (Travis)

On every commit (on projects that have a `.travis.yml`), a Travis build is triggered that runs the project's test suite (both unit and end-to-end tests). `.travis.yml` sets up the continuous integration testing process.

For the Marketplace frontend, tests are run using the Marketplace Mock API. A specific settings file for is used for Travis, found in `src/media/js/settings_local_test.js`.

Results of each build are posted to the IRC channel, `irc.mozilla.org#amo-bots`.

Coding Guidelines

This section is about coding guidelines, or coding style guide. Marketplace developers, like many other Mozilla developers, can be strict on following the style guide. We can be nitty and tell you to put periods at the end of all your comments, add linebreaks to logically separate your code, or alphabetize your CSS rules. This is to make our code consistent with itself and improve readability and ease decisions how to style code.

When in doubt, just follow what the code looks like.

Mozilla Webdev Style Guide

As a foundation, we follow Mozilla's web developer style guide. But we will have additional guidelines on top of that.

- JS Style Guide
- HTML Style Guide
- CSS Style Guide

Indenting

We use 2-space indents for HTML, 4-space indents everywhere else.

For a line of code that spans multiple lines, use a visual-based indent or hanging-ident, preference for visual-based indents.

Visual Indent

```
// On the wrapped line, line the parameter to the start of the previous line's
// function call.
createShip(captain='Mal', address='Serenity',
           class='Firefly')
```

Hanging Indent

```
// The next line starts 4 spaces after the previous line.
createShip(
    captain='Mal', address='Serenity', class='Firefly')
```

AMD Definitions

We make heavy use of AMD modules in the frontend.

- Docstring at the top of all modules that describes the module.
- The *define* line, the dependency list, and the function signature all begin

on separate lines. - Generally, alphabetize dependencies in module definitions based on the name of the dependency module. - Keep with 80-chars. When a dependency list grows too long, line-break it and use a visual-based indent. - If there is a line break in the dependency list, match the line break in the function signature.

For example:

```
/*
    Here is what this module is about.
*/
define('myModule',
    ['foo', 'bar', ..., 'baz',
     'qux'],
    function(foo, bar, ..., baz,
             qux) {
});
```

Comments

Keep comments short and concise. Make them span as few lines as possible. This can be done by getting rid of excess words like definite articles and pronouns.

```
/* This comment is too long. */
// This function is the centralized handler for all apps in a list.

/* This comment is concise. */
// Centralized handler for apps in list.
```

All comments end with a period.

Additional Guidelines

Some additional guidelines:

- When importing Nunjucks macros from a file, specify each macro rather than importing the whole macro file using the `from "x.html" import x` syntax.
- When creating an event to trigger, prefix the event name by the module which is triggering it followed by two hyphens, and then the event name. For example `builder--post-render`.
- Media queries should be defined at the end of CSS files in order of viewport size.
- Stylus variables should be defined all at the top of the file, not in the middle.
- Prefix Stylus classes by page or component (e.g., `.big-dropdown-label` rather than just `.label`).
- Try to contain as much view logic in JS as you can rather than Nunjucks templates. Nunjucks templates usually generate more code than JS.

B

R