Community Plugin Development Documentation

central 1

Introduction

A plugin consists of two required files:

1. plugin manifest
2. plugin template

Prerequisites

- Familiarity with Git
- Familiarity with Handlebars templating language
- Familiarity with using the terminal command line interface
- Node 14+

Community CLI

The Community CLI provides useful commands that will help you with creating plugins, validating the plugin, and perform a variety of tasks to help with plugin development.

Install

The community-cli is not currently available from public NPM repositories. Please reach out to your Community contact to get a copy of the community-cli installation file. This file will be named something like:

```
c1-community-cli-X.X.X-external.tgz
```

Option 1 - Global installation (recommended)

To install the Community CLI globally on your system, open a terminal window in the directory you saved the TGZ file and run the following command:

```
npm install -g c1-community-cli-X.X.X-external.tgz
```

Verify that the CLI was installed successfully by running:

```
community-cli --version
```

To execute community-cli commands from a global system installation, you need to call the community-cli executable directly. Eg:

```
community-cli --help
community-cli serve
community-cli validate
community-cli generate
```

Option 2 - Package level installation

You may install the community-cli locally within an existing NPM package or project.
Save a copy of the installation TGZ file into the root directory of your local NPM package.
From the root directory that contains your Forge Community plugin code, execute the following command within your terminal:

```
npm install c1-community-cli-X.X.X-external.tgz
```

When you install the community-cli tool in this manor you will be required to add scripts to your package.json file in order to execute the community-cli. Eg:

```
{
  "name": "...",
  "scripts": {
    "validate": "community-cli validate -m ./src/manifest.json",
    "serve": "community-cli serve -m ./src/manifest.json"
  },
  ...
}
```

To execute any of these commands you will need to use npm run <COMMAND> . Eg:

```
npm run validate
npm run serve
```

Create a Plugin

We recommended that you use the Community CLI to generate a plugin. It is possible to manually create the files needed for a plugin, but highly discouraged.

Create a Git repository

First, create a new Git repository for your new plugin using git init:

```
git init my-new-plugin
```

Generate the plugin using the CLI

From the newly created directory, in a terminal window and run the CLI command generate:

```
cd my-new-plugin;

community-cli generate
```

Follow the CLI's prompts to create your plugin.
You will need a valid **Vendor ID** in order to submit the plugin, this will be provided to you by Central 1.

Defining Your Manifest

The plugin manifest is a JSON file that provides Central 1 with some details about who developed the application, as well as the various configuration points required by the application (eg. a client identifier).
See API docs for the complete plugin manifest spec.

Defining configuration points for your plugin

If your plugin needs to behave differently based on specific configurations, define that list of configurations in the configurable_preferences property in the manifest.
For example, we can define a configuration point named my_url as a *String* with a default value:

```
{
   "configurable_preferences": [
      {
```

```
        "key": "my_url",
        "label": "My URL",
        "description": "URL to embed",
        "data_type": "string",
        "default_value": "https://www.youtube.com/embed/y8Kyi0WNg40",
        "editor_role": "content_editor"
    }
  ]
}
```

How your plugin will receive configuration values at runtime

Any configurable preferences you define will be made available to the plugin at runtime via Handlebars expressions with the same name as the key as the configurable preference.
For example, we can reference the configuration preference my-url as follows:

```
<p>The configured link is: <a href="{{my_url}}">{{my_url}}</a></p>
```

Read about the Plugin Template for more details.

String preferences

You can configure your plugin to accept string values as a configurable preference.

```
{
  "configurable_preferences": [
    {
      "key": "client_name",
      "label": "Client Name",
      "description": "The name of your client",
      "data_type": "string",
      "default_value": "John",
      "editor_role": "content_editor"
    }
  ]
}
```

Number preferences

Available in community-cli version 1.9.x or higher
You can configure your plugin to accept number values as a configurable preference.

```
{
  "configurable_preferences": [
    {
      "key": "client_age",
      "label": "Client Age",
      "description": "The age of your client",
      "data_type": "number",
      "default_value": 29,
      "editor_role": "content_editor"
    }
  ]
}
```

Boolean preferences

Available in community-cli version 1.9.x or higher
You can configure your plugin to accept Boolean values as a configurable preference.

```
{
   "configurable_preferences": [
      {
         "key": "thing_enabled",
         "label": "Is feature enabled",
         "description": "Whether to enable a feature",
         "data_type": "boolean",
         "default_value": true,
         "editor_role": "content_editor"
      }
   ]
}
```

Using the Boolean value in your template

You can use the Boolean value in your template to enable or disable sections of your code by using an if handlebars helper:

```
{{#if thing_enabled}}
<p>You're setting is enabled</p>
{{else}}
<p>You're setting is not enabled</p>
{{/if}}
```

Including the Boolean in your template as a JavaScript variable

You can introduce the Boolean preference value into a JavaScript application by putting the following in your handlebars template:

```
<script>
   let isThingEnabled = {{thing_enabled}};
</script>
```

Date preferences

Available in community-cli version 1.9.x or higher
You can configure your plugin to accept Date values as a configurable preference.

```
{
   "configurable_preferences": [
      {
         "key": "appointment_date",
         "label": "Enter a date",
         "description": "The date of your appointment",
         "data_type": "date",
         "default_value": "2021-05-30",
         "editor_role": "content_editor"
      }
```

```
  ]
}
```

Date default value

When you configure a date preference with a default value, it is *required* that the date be in the ISO date format YYYY-MM-DD. Eg: “2021-01-30”

Including the date in your template as a string

When you include your date preference value in your template, it will automatically be formatted to look like January 30, 2021
Example:

```
<p>Your appointment is booked for: {{appointment_date}}</p>
```

Will render to:

```
<p>Your appointment is booked for: January 30, 2021</p>
```

Including the date in your template as a JavaScript object

You can introduce the date preference value into a JavaScript application by putting the following in your handlebars template:

```
<script>
  let myAppointmentDate = new Date('{{appointment_date}}');
</script>
```

Formatting your dates

You can format the date preference by using our included dateFormat handlebars helper. The handlebars helper has the following signature:

```
{{dateFormat date ["format"] [format="format"]}}
```

The helper comes with several predefined date formats:

Format Name	Used Format	Example Output	Usage
short	M/dd/YY	6/30/09	{{dateFormat datePreferenceKey "short"}}{{dateFormat datePreferenceKey format="short}}
medium	MMM d, yyyy	Jan 1, 2009	{{dateFormat datePreferenceKey "medium"}}{{dateFormat datePreferenceKey format="medium}}
long	MMMM d, yyyy	June 30, 2009	{{dateFormat datePreferenceKey "long"}}{{dateFormat datePreferenceKey format="long}}
full	EEEE, MMMM d, yyyy	Tuesday, June 30, 2009	{{dateFormat datePreferenceKey "full"}}{{dateFormat datePreferenceKey format="full}}

Or you can define your own format. The handlebars helper accepts any java SimpleDateFormat string.

```
<p>Your appointment is booked for: {{dateHelper appointmentDate "yyyy-M-dd"}}</p>
<!-- Or -->
<p>Your appointment is booked for: {{dateHelper appointmentDate format="yyyy-M-dd"}}</p>
```

Would output:

Your appointment is booked for: 2021-01-30

Rich Text preferences

Available in community-cli version 1.12.x or higher
You can configure your plugin to accept Rich Text values as a configurable preference. The format will be in HTML, be sure to only use valid HTML.


```
{
   "configurable_preferences": [
      {
         "key": "legal_agreement",
         "label": "Legal Agreement text",
         "description": "The legal agreement",
         "data_type": "richtext",
         "default_value": "<p><strong>AGREEMENT</strong></p><p>By accessing and using Electronic Services, you ac
knowledge and accept our Online Banking Privacy Policy and other privacy policies, as amended from time to time. We st
rongly encourage you to review our <a href=\"http://some.url\">Online Banking Privacy Policy</a>.</p>",
         "editor_role": "content_editor"
      }
   ]
}
```


Localization

Available in community-cli version 1.10.x or higher.
You can make a configurable preference support localized values by setting:

"localizable": true

For example:


```
{
   "configurable_preferences": [
      {
         "key": "greeting",
         "label": "Greeting",
         "description": "A greeting",
         "data_type": "string",
         "default_value": "Hello",
         "localizable": true,
         "editor_role": "content_editor"
      }
   ]
}
```


Providing default values for different locales

You can specify default values for each locale by providing a localized_default_value object.
The keys for this object can be:

- language code (i.e. fr)
- language code with locale (i.e. fr_CA).

```
{
   "configurable_preferences": [
      {
         "key": "greeting",
         "label": "Greeting",
         "description": "A greeting",
         "data_type": "string",
         "default_value": "Hello",
         "localizable": true,
         "localized_default_value": {
            "fr": "Bonjour",       // all French locales
            "fr-CA": "Allo"        // French-Canadian locale
         },
         "editor_role": "content_editor"
      }
   ]
}
```

You can set default values for any number of languages and locales.
Canada has two official languages, since English is the predominant language, it is assumed as the default language. It would be prudent to provide a localized default value for French.

Plugin Template

The plugin template is the HTML required to be put onto a page in order to bootstrap a web application. This can be standard HTML that include a Script tag that loads JavaScript from an external system, or CSS styles required for the application or a combination of these items. This plugin template is written in the Handlebars templating language and will be provided the information collected by Central 1.

Handlebars template

The file defined in the plugin manifest start_template will be a starting point for your plugin and is typically as a Handlebars template. The start_template should load the required assets and define the necessary HTML elements needed to make the plugin work.

Hello World example

This example displays a greeting based on the configurable preference greeting:

```
<div class="c1-hello-world">
   <h1 class="display-1">{{greeting}}, world!</h1>
</div>
```

Including external assets

To load external assets for your plugin, you can define them like you normally would on an HTML page.
For example:

```
<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.0.1/dist/css/bootstrap.min.css"
   rel="stylesheet">

<script src="https://yourdomain.com/your-asset.js"></script>

<div class="my-plugin">
```

```
    <my-app></my-app>
</div>
```

** IMPORTANT: All external JavaScript assets must be bundled correctly so that they do not introduce code into the *global* browser scope. See the section on including third party libraries. **

Reserved Variables

We provide some special reserved variables in the Handlebars rendering context that may be useful for plugins.
The following variables are available to the plugin template:

Variable Name	What it is	Example Usage
auth_code	When the user is authenticated, the auth_code is an authentication code that will can be used to make authenticated API calls.	?auth={{auth_code}}
instance_id	A temporary GUID assigned to an instance of a plugin when the page is rendered.	
plugin_root	An absolute URL to where the plugin and its assets are hosted. This should be used to load any assets that are packaged with the plugin.	<script src="{{plugin_root}}/my-app.js"></script>

Including third party libraries

Background

The forge community plugin platform provides you the flexibility to use whatever frameworks and libraries you wish. However, depending on your integration method, extra precaution needs to be taken when introducing JavaScript frameworks and libraries such that they do not interfere with the operation of *other* plugins when operating on the same page.
This means that no plugin is allowed to introduce JavaScript frameworks or libraries that would be operate in the *global* (window) browser scope.
Use one of the following options to ensure compliance with these guidelines.

Option #1 - Iframe Integration

This is the easiest option for integrating your plugin using 3rd party libraries. Since your plugin is operating in a *scoped* iframe window you are free to use whatever frameworks and libraries you would like within your iframe.
If your plugin requires JavaScript code outside of your scoped iframe window, you will need to follow Option 2 for static code bundling.

Option #2 - Static code bundling

Use a static module bundler (like Webpack or RollUp) to bundle and scope all of your dependencies into a single JavaScript asset.
These bundling tools will ensure that any frameworks that are required by your plugin will be available without interfering with the browsers global scope or any other plugins on the same page.
Please reference our webpack bundling example to see how you can integrate with various JavaScript frameworks.

Bundling Your Plugin With Webpack

Install Webpack

```
npm install --save-dev webpack
```

Configure webpack

In your root plugin directory, create a webpack.config.js file

```
const path = require('path');

/**
 * Webpack config
 * @see https://webpack.js.org/configuration/
 */
module.exports = {

    // define the starting point of the application and Webpack starts bundling from
    entry: './src/app/index.js',    // our app's starting point is the index.js file

    // define how Webpack will output the bundled files
    output: {

        // target directory for all output files (absolute path)
        path: path.resolve(__dirname, 'dist'),  // we want to output the files into ./dist

        // the file name of the output file
        filename: 'app.bundle.js'   // we are naming it app.bundle.js for clarity
    }
}
```

Bundled plugin - sample code

The following code snippets are an example of a simple plugin using bundled third party libraries (in this case jQuery and chart.js)
If you would like to run the following code examples, you will need to install some NPM dependencies in your project first:

```
npm install --save chart.js jquery
```

- /src/manifest.json

```
{
    "start_template": "index.hbs",
    "name": "Hello World - Bundled example",
    "description": "Hello World plugin demonstrating bundling",
    "vendor_id": "c1",
    "id": "c1/community-hello-bundled",
    "type": "micro_frontend",
    "version": "1.0.0"
}
```

- /src/index.hbs

```
<!-- /src/index.hbs -->

<div class="plugin_c1_community_hello_bundled">
    <p>Here's a chart:</p>
    <ul class="list-group"></ul>
    <div class="chart">
        <canvas></canvas>
```

```
    </div>
</div>
<script type="text/javascript" src="{{plugin_root}}/app.bundle.js"></script>
<script type="text/javascript">
    // Quick test to make sure Jquery isn't in the global (Window) scope
    console.log(window.jQuery, 'should be undefined');
    console.log(window.jquery, 'should be undefined');
</script>
```

- /src/app/index.js

```
/**********************/
/* /src/app/index.js */
/**********************/

import jquery from 'jquery';    // import the Jquery library
import {getData} from './data'; // application specific module
import {drawChart} from './utils';  // application specific module

/**
 * Application entry point.
 * The code is wrapped inside an immediately invoked function. (see https://developer.mozilla.org/en-US/docs/Glossary/IIFE)
 */
(function Main() {
    const $container = jquery('.plugin_c1_community_hello_bundled');    // use the Jquery library
    const data = getData();

    drawChart(data, $container.find('.chart canvas')[0]);

}())
```

- /src/app/data.js

```
/**********************/
/* /src/app/data.js */
/**********************/

/**
 * Sample data for the app to use
 */
export function getData() {
    return {
        xs: 1,
        sm: 3,
        md: 2,
        lg: 1
    };
}
```

- /src/app/utils.js

```
/**********************/
/* /src/app/utils.js */
```

```
/********************/

/**
 * Example how to use the Chart.js library (https://www.chartjs.org/) in our project
 *
 * Following their integration instructions on https://www.chartjs.org/docs/latest/getting-started/integration.html
 * to import Chart.js library using a module loader.
 */
import Chart from 'chart.js/auto';  // import the Chart library

export function drawChart(data, element) {
    const labels = Object.keys(data);
    const dataValues = labels.map(function(key) {
        return data[key]
    });
    const chartConfig = {
        type: 'pie',
        data: {
            labels: labels,
            datasets: [{
                label: 'One',
                backgroundColor: [
                    'rgb(255, 99, 132)',
                    'rgb(54, 162, 235)',
                    'rgb(255, 205, 86)',
                    'rgb(90, 85, 205)',
                ],
                hoverOffset: 4,
                data: dataValues
            }]
        },
        options: {}
    };
    const chart = new Chart(element, chartConfig);  // using the Chart library
}
```

Testing

The CLI provides important commands that will help you verify that your plugin has been built correctly. This will help ensure a smooth ingestion process when submitting the plugin to Central 1.

Validating the plugin

You can easily validate the plugin to check if it adheres to the required specifications for a plugin using the CLI.
From the same directory as your plugin, run the CLI command validate:

```
community-cli validate
```

Resolving validation errors

If there are any errors, they will be presented to you as a list with a reason as to why it failed validation and how to resolve the issue.
For example, if the plugin manifest is invalid:

```
***** Your manifest file contains the following errors *****
[
  {
    instancePath: '/configurable_preferences/0/editor_role',
    schemaPath: '#/properties/configurable_preferences/items/properties/editor_role/enum',
    keyword: 'enum',
    params: { allowedValues: [ 'admin', 'content_editor' ] },
    message: 'must be equal to one of the allowed values'
  },
```

Refer to the instancePath to locate the area in the plugin manifest that is causing the error and resolve the issue.

Running the plugin locally

You can test the plugin in your browser to verify that the template is correctly implemented.
From the same directory as your plugin, run the CLI command serve:

```
community-cli serve
```

A local web server will be started at http://localhost:8000/ and you will see your plugin rendered on that page.

Plugin Submission

The plugin can be submitted to Central 1 using Git and triggered using Git tags. If there are any errors in the plugin submission, the ingestion process will fail. Be sure to run community-cli validate to make sure your plugin is valid.

Submission Steps

1. Provide Central 1 with read access to your plugin's Git repository.
2. Put the required plugin files and place it in the **/dist** directory.
3. Create a **Git tag** with the version number at the specific commit containing the plugin files.
4. Push the Git tag to the Git repository.
5. Central 1 will detect the new Git tag in the repository and ingests the plugin. If the ingestion process is successfully, your plugin will eventually be made available on the pertinent environments.
