+++ title = "Legacy code style guide" aliases = ["/docs/grafana/latest/plugins/developing/code-styleguide/"] +++

Legacy code style guide

This guide has two parts. The first part describes the metadata and the second part is a styleguide for HTML/CSS and JavaScript in Grafana plugins and applies if you are using ES6 in your plugin. If using TypeScript then the Angular TypeScript styleguide is recommended.

Metadata

The plugin metadata consists of a plugin.json file and the README.md file. These two files are used by Grafana and Grafana.com.

Plugin.json (mandatory)

The plugin.json file is the same concept as the package.json file for an npm package. When Grafana starts it will scan the plugin folders and mount every folder that contains a plugin.json file unless the folder contains a subfolder named `dist`. In that case grafana will mount the `dist` folder instead.

The most important fields are the first three, especially the id. The convention for the plugin id is **[github username/org]-[plugin name]-[datasource|app|panel]** and it has to be unique.

Examples:

```
raintank-worldping-app
grafana-simple-json-datasource
grafana-piechart-panel
mtanda-histogram-panel
```

For more information about the file format for `plugin.json` file, refer to [metadata]({{< relref "../metadata.md" >}}).

Minimal plugin.json:

```
{
  "type": "panel",
  "name": "Clock",
  "id": "yourorg-clock-panel",

  "info": {
    "description": "Clock panel for grafana",
    "author": {
      "name": "Grafana Labs",
      "url": "https://grafana.com"
    },
    "keywords": ["clock", "panel"],
    "version": "1.0.0",
    "updated": "2015-03-24"
  },

  "dependencies": {
```

```
    "grafanaVersion": "3.x.x",
    "plugins": [ ]
  }
}
```

README.md

The README.md file is rendered both in the grafana.com plugins page, and within the Grafana application. The only difference from how GitHub renders markdown is that html is not allowed.

File and Directory Structure Conventions

Here is a typical directory structure for a plugin.

```
johnnyb-awesome-datasource
|-- dist
|-- spec
|   |-- datasource_spec.js
|   |-- query_ctrl_spec.js
|   |-- test-main.js
|-- src
|   |-- img
|   |   |-- logo.svg
|   |-- partials
|   |   |-- annotations.editor.html
|   |   |-- config.html
|   |   |-- query.editor.html
|   |-- datasource.js
|   |-- module.js
|   |-- plugin.json
|   |-- query_ctrl.js
|-- Gruntfile.js
|-- LICENSE
|-- package.json
|-- README.md
```

Most JavaScript projects have a build step and most Grafana plugins are built using Babel and ES6. The generated JavaScript should be placed in the `dist` directory and the source code in the `src` directory. We recommend that the plugin.json file be placed in the src directory and then copied over to the dist directory when building. The `README.md` can be placed in the root or in the dist directory.

Directories:

- `src/` contains plugin source files.
- `src/partials` contains html templates.
- `src/img` contains plugin logos and other images.
- `spec/` contains tests (optional).
- `dist/` contains built content.

HTML and CSS

For the HTML on editor tabs, we recommend using the inbuilt Grafana styles rather than defining your own. This makes plugins feel like a more natural part of Grafana. If done correctly, the html will also be responsive and adapt to smaller screens. The `gf-form` css classes should be used for labels and inputs.

Below is a minimal example of an editor row with one form group and two fields, a dropdown and a text input:

```
<div class="editor-row">
  <div class="section gf-form-group">
    <h5 class="section-heading">My Plugin Options</h5>
    <div class="gf-form">
      <label class="gf-form-label width-10">Label1</label>
      <div class="gf-form-select-wrapper max-width-10">
        <select
          class="input-small gf-form-input"
          ng-model="ctrl.panel.mySelectProperty"
          ng-options="t for t in ['option1', 'option2', 'option3']"
          ng-change="ctrl.onSelectChange()"
        ></select>
      </div>
      <div class="gf-form">
        <label class="gf-form-label width-10">Label2</label>
        <input
          type="text"
          class="input-small gf-form-input width-10"
          ng-model="ctrl.panel.myProperty"
          ng-change="ctrl.onFieldChange()"
          placeholder="suggestion for user"
          ng-model-onblur
        />
      </div>
    </div>
  </div>
</div>
```

Use the `width-x` and `max-width-x` classes to control the width of your labels and input fields. Try to get labels and input fields to line up neatly by having the same width for all the labels in a group and the same width for all inputs in a group if possible.

Build Scripts

Our recommendation is to use whatever you usually use - Grunt, Gulp or npm scripts. Most plugins seems to use Grunt so that is probably the easiest to get started with if you do not have a preferred build system. The only requirement is that it supports systemjs which is required by Grafana to load plugins.

Linting

We recommend that you use a linter for your JavaScript. For ES6, the standard linter is eslint. Rules for linting are described in an .eslintrc that is placed in the root directory. For an example of linting rules in a plugin, refer to .eslintrc.

ES6 features

1. Use `const` if a variable is not going to be reassigned.

2. Prefer to use `let` instead `var` (Exploring ES6)

3. Use arrow functions, which don't shadow `this` (Exploring ES6):

```
testDatasource() {
  return this.getServerStatus()
  .then(status => {
    return this.doSomething(status);
  })
}
```

   better than

```
testDatasource() {
  var self = this;
  return this.getServerStatus()
  .then(function(status) {
    return self.doSomething(status);
  })
}
```

4. Use native *Promise* object:

```
metricFindQuery(query) {
  if (!query) {
    return Promise.resolve([]);
  }
}
```

   better than

```
metricFindQuery(query) {
  if (!query) {
    return this.$q.when([]);
  }
}
```

5. If using Lodash, then be consistent and prefer that to the native ES6 array functions.