+++ title = "Legacy review guidelines" aliases = ["/docs/grafana/latest/plugins/developing/plugin-review-guidelines/"] +++

Legacy review guidelines

The Grafana team reviews all plugins that are published on Grafana.com. There are two areas we review, the metadata for the plugin and the plugin functionality.

Metadata

The plugin metadata consists of a `plugin.json` file and the README.md file. The `plugin.json` file is used by Grafana to load the plugin, and the README.md file is shown in the plugins section of Grafana and the plugins section of https://grafana.com.

README.md

The README.md file is shown on the plugins page in Grafana and the plugin page on Grafana.com. There are some differences between the GitHub markdown and the markdown allowed in Grafana/Grafana.com:

- Cannot contain inline HTML.
- Any image links should be absolute links. For example: https://raw.githubusercontent.com/grafana/azure-monitor-datasource/master/dist/img/grafana_cloud_install.png

The README should:

- describe the purpose of the plugin.
- contain steps on how to get started.

Plugin.json

The `plugin.json` file is the same concept as the `package.json` file for an npm package. When the Grafana server starts it will scan the plugin folders (all folders in the data/plugins subfolder) and load every folder that contains a `plugin.json` file unless the folder contains a subfolder named `dist`. In that case, the Grafana server will load the `dist` folder instead.

A minimal `plugin.json` file:

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

  "info": {
    "description": "Clock panel for grafana",
    "author": {
      "name": "Author Name",
      "url": "http://yourwebsite.com"
    },
    "keywords": ["clock", "panel"],
    "version": "1.0.0",
    "updated": "2018-03-24"
  },
```

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

- The convention for the plugin id is **[grafana.com username/org]-[plugin name]-[datasource|app|panel]** and it has to be unique. The org **cannot** be `grafana` unless it is a plugin created by the Grafana core team.

  Examples:

  - raintank-worldping-app
  - ryantxu-ajax-panel
  - alexanderzobnin-zabbix-app
  - hawkular-datasource
- The `type` field should be either `datasource` `app` or `panel`.
- The `version` field should be in the form: x.x.x e.g. `1.0.0` or `0.4.1`.

The full file format for `plugin.json` file is in plugin.json.

Plugin Language

JavaScript, TypeScript, ES6 (or any other language) are all fine as long as the contents of the `dist` subdirectory are transpiled to JavaScript (ES5).

File and Directory Structure Conventions

Here is a typical directory structure for a plugin.

```
johnnyb-awesome-datasource
|-- dist
|-- 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. 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.
- `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.

Data Sources

For more information about data sources, refer to the basic guide for data sources.

Configuration Page Guidelines

- It should be as easy as possible for a user to configure a URL. If the data source is using the `datasource-http-settings` component, it should use the `suggest-url` attribute to suggest the default URL or a URL that is similar to what it should be (especially important if the URL refers to a REST endpoint that is not common knowledge for most users e.g. `https://yourserver:4000/api/custom-endpoint`).

```
<datasource-http-settings current="ctrl.current" suggest-
url="http://localhost:8080"> </datasource-http-settings>
```

- The `testDatasource` function should make a query to the data source that will also test that the authentication details are correct. This is so the data source is correctly configured when the user tries to write a query in a new dashboard.

Password Security

If possible, any passwords or secrets should be saved in the `secureJsonData` blob. To encrypt sensitive data, the Grafana server's proxy feature must be used. The Grafana server has support for token authentication (OAuth) and HTTP Header authentication. If the calls have to be sent directly from the browser to a third-party API, this will not be possible and sensitive data will not be encrypted.

Read more here about how [authentication for data sources]({{< relref "../add-authentication-for-data-source-plugins.md" >}}) works.

If using the proxy feature, the Configuration page should use the `secureJsonData` blob like this:

- good: `<input type="password" class="gf-form-input" ng-model='ctrl.current.secureJsonData.password' placeholder="password"></input>`
- bad: `<input type="password" class="gf-form-input" ng-model='ctrl.current.password' placeholder="password"></input>`

Query Editor

Each query editor is unique and can have a unique style. It should be adapted to what the users of the data source are used to.

- Should use the Grafana CSS `gf-form` classes.
- Should be neat and tidy. Labels and fields in columns should be aligned and should be the same width if possible.
- The data source should be able to handle when a user toggles a query (by clicking on the eye icon) and not execute the query. This is done by checking the `hide` property - an example.
- Should not execute queries if fields in the Query Editor are empty and the query will throw an exception (defensive programming).
- Should handle errors. There are two main ways to do this:
  - use the notification system in Grafana to show a toaster pop-up with the error message. For an example of a pop-up with the error message, refer to code in triggers_panel_ctrl.
  - provide an error notification in the query editor like the MySQL/Postgres data sources do. For an example of error notification in the query editor, refer to code in query_ctrl and in the html.