+++ title = “Plugin migration guide” +++

Plugin migration guide

Introduction

This guide helps you identify the steps you need to take based on the Grafana version your plugin supports and explains how to migrate the plugin to the 8.2.x or a later version.

> **Note:** If you’ve successfully migrated your plugin using this guide, then share your experiences with us! If you find missing information, then we encourage you to submit an issue on GitHub so that we can improve this guide!

Table of contents

    - Field options
    - Backend plugins
  * Migrate a plugin from Angular to React
    - Migrate a panel plugin
    - Migrate a data source plugin
    - Migrate to data frames
  * Troubleshoot plugin migration

From version 8.3.x to 8.4.x

This section explains how to migrate Grafana v8.3.x plugins to the updated plugin system available in Grafana v8.4.x. Depending on your plugin, you need to perform one or more of the following steps.

Value Mapping Editor has been removed from @grafana-ui library

Removed due to being an internal component.

Thresholds Editor has been removed from @grafana-ui library

Removed due to being an internal component.

8.4 deprecations

LocationService replaces getLocationSrv In a previous release, we migrated to use a new routing system and introduced a new service for managing locations, navigation, and related information. In this release, we are making that new service the primary service.

Example: Import the service.

```
// before
import { getLocationSrv } from '@grafana/runtime';

// after
import { locationService } from '@grafana/runtime';
```

Example: Navigate to a path and add a new record in the navigation history so that you can navigate back to the previous one.

```
// before
getLocationSrv.update({
  path: '/route-to-navigate-to',
  replace: false,
});

// after
locationService.push('/route-to-navigate-to');
```

Example: Navigate to a path and replace the current record in the navigation history.

```
// before
getLocationSrv.update({
  path: '/route-to-navigate-to',
  replace: true,
});

// after
locationService.replace('/route-to-navigate-to');
```

Example: Update the search or query parameter for the current route and add a new record in the navigation history so that you can navigate back to the previous one.

```
// How to navigate to a new path
// before
getLocationSrv.update({
  query: {
    value: 1,
  },
  partial: true,
  replace: false,
});

// after
locationService.partial({ value: 1 });
```

Example: Update the search or query parameter for the current route and add replacing it in the navigation history.

```
// before
getLocationSrv.update({
  query: {
    'var-variable': 1,
  },
  partial: true,
  replace: true,
});

// after
locationService.partial({ 'var-variable': 1 }, true);
```

From version 7.x.x to 8.x.x

This section explains how to migrate Grafana v7.x.x plugins to the updated plugin system available in Grafana v8.x.x. Depending on your plugin, you need

to perform one or more of the following steps. We have documented the breaking changes in Grafana v8.x.x and the steps you need to take to upgrade your plugin.

Backend plugin v1 support has been dropped

Use the new plugin sdk to run your backend plugin running in Grafana 8.

1. Add dependency on grafana-plugin-sdk-go Add a dependency on the `https://github.com/grafana/grafana-plugin-sdk-go`. We recommend using go modules to manage your dependencies.

2. Update the way you bootstrap your plugin Update your `main` package to bootstrap via the new plugin sdk.

```
// before
package main

import (
  "github.com/grafana/grafana_plugin_model/go/datasource"
  hclog "github.com/hashicorp/go-hclog"
  plugin "github.com/hashicorp/go-plugin"

  "github.com/myorgid/datasource/pkg/plugin"
)

func main() {
  pluginLogger.Debug("Running GRPC server")

  ds, err := NewSampleDatasource(pluginLogger);
  if err != nil {
    pluginLogger.Error("Unable to create plugin");
  }

  plugin.Serve(&plugin.ServeConfig{
    HandshakeConfig: plugin.HandshakeConfig{
            ProtocolVersion:  1,
            MagicCookieKey:   "grafana_plugin_type",
            MagicCookieValue: "datasource",
        },
    Plugins: map[string]plugin.Plugin{
            "myorgid-datasource": &datasource.DatasourcePluginImpl{Plugin: ds},
    },
    GRPCServer: plugin.DefaultGRPCServer,
  })
}
```

```go
// after
package main

import (
  "os"

  "github.com/grafana/grafana-plugin-sdk-go/backend/log"
  "github.com/grafana/grafana-plugin-sdk-go/backend/datasource"

  "github.com/myorgid/datasource/pkg/plugin"
)

func main() {
  log.DefaultLogger.Debug("Running GRPC server")

  if err := datasource.Manage("myorgid-datasource", NewSampleDatasource, datasource.ManageOp
                log.DefaultLogger.Error(err.Error())
                os.Exit(1)
        }
}
```

3. Update the plugin package Update your `plugin` package to use the new plugin sdk.

```go
// before
package plugin

import (
  "context"

  "github.com/grafana/grafana_plugin_model/go/datasource"
  "github.com/hashicorp/go-hclog"
)

func NewSampleDatasource(pluginLogger hclog.Logger) (*SampleDatasource, error) {
    return &SampleDatasource{
        logger: pluginLogger,
    }, nil
}

type SampleDatasource struct{
  logger hclog.Logger
}

func (d *SampleDatasource) Query(ctx context.Context, tsdbReq *datasource.DatasourceRequest)
  d.logger.Info("QueryData called", "request", req)
```

```
  // logic for querying your datasource.
}

// after
package plugin

import (
  "context"

  "github.com/grafana/grafana-plugin-sdk-go/backend"
  "github.com/grafana/grafana-plugin-sdk-go/backend/instancemgmt"
  "github.com/grafana/grafana-plugin-sdk-go/backend/log"
  "github.com/grafana/grafana-plugin-sdk-go/data"
)

func NewSampleDatasource(_ backend.DataSourceInstanceSettings) (instancemgmt.Instance, erro
    return &SampleDatasource{}, nil
}

type SampleDatasource struct{}


func (d *SampleDatasource) Dispose() {
    // Clean up datasource instance resources.
}

func (d *SampleDatasource) QueryData(ctx context.Context, req *backend.QueryDataRequest) (*b
  log.DefaultLogger.Info("QueryData called", "request", req)
  // logic for querying your datasource.
}

func (d *SampleDatasource) CheckHealth(_ context.Context, req *backend.CheckHealthRequest) (
  log.DefaultLogger.Info("CheckHealth called", "request", req)
  // The main use case for these health checks is the test button on the
  // datasource configuration page which allows users to verify that
  // a datasource is working as expected.
}
```

Sign and load backend plugins

We strongly recommend that you not allow unsigned plugins in your Grafana installation. By allowing unsigned plugins, we cannot guarantee the authenticity of the plugin, which could compromise the security of your Grafana installation.

To sign your plugin, see Sign a plugin.

You can still run and develop an unsigned plugin by running your

Grafana instance in development mode. Alternatively, you can use the allow_loading_unsigned_plugins configuration setting.

Update react-hook-form from v6 to v7

We have upgraded react-hook-form from version 6 to version 7. To make your forms compatible with version 7, refer to the react-hook-form-migration-guide.

Update the plugin.json

The property that defines which Grafana version your plugin supports has been renamed and now it is a range instead of a specific version.

```
// before
{
"dependencies": {
    "grafanaVersion": "7.5.x",
    "plugins": []
  }
}

// after
{
  "dependencies": {
    "grafanaDependency": ">=8.0.0",
    "plugins": []
  }
}
```

Update imports to match emotion 11

Grafana uses Emotion library to manage frontend styling. We have updated the Emotion package and this can affect your frontend plugin if you have custom styling. You only need to update the import statements to get it working in Grafana 8.

```
// before
import { cx, css } from 'emotion';

// after
import { cx, css } from '@emotion/css';
```

Update needed for app plugins using dashboards

To make side navigation work properly - app plugins targeting Grafana `8.+` and integrating into the side menu via [addToNav]({{< relref “metadata.md#properties-4” >}}) property need to adjust their `plugin.json` and all dashboard json files to have a matching `uid`.

plugin.json

```
json "linenos=inline,hl_lines=7,linenostart=1" {   "id": "plugin-id",
// ...   "includes": [     {       "type": "dashboard",       "name":
"(Team) Situation Overview",       "path": "dashboards/example-dashboard.json",
"addToNav": true,       "defaultNav": false,       "uid": "l3KqBxCMz"
}   ]   // ... }
```

dashboards/example-dashboard.json

```
{
  // ...
  "title": "Example Dashboard",
  "uid": "l3KqBxCMz",
  "version": 1
  // ...
}
```

8.0 deprecations

Grafana theme v1 In Grafana 8 we have introduced a new improved version of our theming system. The previous version of the theming system is still available but is deprecated and will be removed in the next major version of Grafana.

You can find more detailed information on how to apply the v2 theme here.

How to style a functional component

The `useStyles` hook is the preferred way to access the theme when styling. It provides basic memoization and access to the theme object.

```
// before
import React, { ReactElement } from 'react';
import css from 'emotion';
import { GrafanaTheme } from '@grafana/data';
import { useStyles } from '@grafana/ui';

function Component(): ReactElement | null {
  const styles = useStyles(getStyles);
}

const getStyles = (theme: GrafanaTheme) => ({
  myStyle: css`
    background: ${theme.colors.bodyBg};
    display: flex;
  `,
});
```

```
// after
import React, { ReactElement } from 'react';
import { css } from '@emotion/css';
import { GrafanaTheme2 } from '@grafana/data';
import { useStyles2 } from '@grafana/ui';

function Component(): ReactElement | null {
  const theme = useStyles2(getStyles);
}

const getStyles = (theme: GrafanaTheme2) => ({
  myStyle: css`
    background: ${theme.colors.background.canvas};
    display: flex;
  `,
});
```

How to use the theme in a functional component

```
// before
import React, { ReactElement } from 'react';
import { useTheme } from '@grafana/ui';

function Component(): ReactElement | null {
  const theme = useTheme();
}

// after
import React, { ReactElement } from 'react';
import { useTheme2 } from '@grafana/ui';

function Component(): ReactElement | null {
  const theme = useTheme2();
  // Your component has access to the theme variables now
}
```

How to use the theme in a class component

```
// before
import React from 'react';
import { Themeable, withTheme } from '@grafana/ui';

type Props = {} & Themeable;

class Component extends React.Component<Props> {
  render() {
    const { theme } = this.props;
    // Your component has access to the theme variables now
```

```
  }
}

export default withTheme(Component);

// after
import React from 'react';
import { Themeable2, withTheme2 } from '@grafana/ui';

type Props = {} & Themeable2;

class Component extends React.Component<Props> {
  render() {
    const { theme } = this.props;
    // Your component has access to the theme variables now
  }
}

export default withTheme2(Component);
```

Gradually migrating components

If you need to use both the v1 and v2 themes due to using migrated and non-migrated components in the same context, use the `v1` property on the `v2` theme as described in the following example.

```
function Component(): ReactElement | null {
  const theme = useTheme2();
  return (
    <NonMigrated theme={theme.v1}>
      <Migrated theme={theme] />
    </NonMigrate>
  );
};
```

From version 6.2.x to 7.4.0

Legend components

The Legend components have been refactored and introduced the following changes within the `@grafana/ui` package.

```
// before
import { LegendItem, LegendOptions, GraphLegend } from '@grafana/ui';

// after
import { VizLegendItem, VizLegendOptions, VizLegend } from '@grafana/ui';
```

- `LegendPlacement` has been updated from `'under' | 'right' | 'over'` to `'bottom' | 'right'` so you can not place the legend above the visualization anymore.
- The `isVisible` in the `LegendItem` has been renamed to `disabled` in `VizLegendItem`.

From version 6.5.x to 7.3.0

getColorForTheme changes

The `getColorForTheme` function arguments have changed from `(color: ColorDefinition, theme?: GrafanaThemeType)` to `(color: string, theme: GrafanaTheme)`.

```
// before
const color: ColorDefinition = {
  hue: 'green';
  name: 'dark-green';
  variants: {
    light: '#19730E'
    dark: '#37872D'
  };
}
const themeType: GrafanaThemeType = 'dark';
const themeColor = getColorForTheme(color, themeType);

// after
const color = 'green';
const theme: GrafanaTheme = useTheme();
const themeColor = getColorForTheme(color, theme);
```

From version 6.x.x to 7.0.0

What's new in Grafana 7.0?

Grafana 7.0 introduced a whole new plugin platform based on React. The new platform supersedes the previous Angular-based plugin platform.

Plugins built using Angular still work for the foreseeable future, but we encourage new plugin authors to develop with the new platform.

New data format Along with the move to React, the new plugin platform introduced a new internal data format called data frames.

Previously, data source plugins could send data either as time series or tables. With data frames, data sources can send any data in a table-like structure. This gives you more flexibility to visualize your data in Grafana.

Improved TypeScript support While the previous Angular-based plugin SDK did support TypeScript, for the React platform, we've greatly improved the support. All our APIs are now TypeScript, which might require existing code to update to the new stricter type definitions. Grafana 7.0 also introduced several new APIs for plugin developers that take advantage of many of the new features in Grafana 7.0.

Grafana Toolkit With Grafana 7.0, we released a new tool for making it easier to develop plugins. Before, you'd use Gulp, Grunt, or similar tools to generate the minified assets. Grafana Toolkit takes care of building and testing your plugin without complicated configuration files.

For more information, refer to @grafana/toolkit.

Field options Grafana 7.0 introduced the concept of *field options*, a new way of configuring your data before it gets visualized. Since this was not available in previous versions, any plugin that enables field-based configuration will not work in previous versions of Grafana.

For plugins prior to Grafana 7.0, all options are considered *Display options*. The tab for field configuration isn't available.

Backend plugins While backend plugins were available as an experimental feature in previous versions of Grafana, the support has been greatly improved for Grafana 7. Backend plugins for Grafana 7.0 are backwards-compatible and will continue to work. However, the old backend plugin system has been deprecated, and we recommend that you use the new SDK for backend plugins.

Since Grafana 7.0 introduced signing of backend plugins, community plugins won't load by default if they're unsigned.

To learn more, refer to Backend plugins.

Migrate a plugin from Angular to React

If you're looking to migrate a plugin to the new plugin platform, then we recommend that you release it under a new major version. Consider keeping a release branch for the previous version to be able to roll out patch releases for versions prior to Grafana 7.

While there's no 1-to-1 migration path from an Angular plugin to the new React platform, from early adopters, we've learned that one of the easiest ways to migrate is to:

1. Create a new branch called `migrate-to-react`.
2. Start from scratch with one of the templates provided by Grafana Toolkit.
3. Move the existing code into the new plugin incrementally, one component at a time.

Migrate a panel plugin Prior to Grafana 7.0, you would export a MetricsPanelCtrl from module.ts.

src/module.ts

```
import { MetricsPanelCtrl } from 'grafana/app/plugins/sdk';

class MyPanelCtrl extends MetricsPanelCtrl {
  // ...
}

export { MyPanelCtrl as PanelCtrl };
```

Starting with 7.0, plugins now export a PanelPlugin from module.ts where MyPanel is a React component containing the props from PanelProps.

src/module.ts

```
import { PanelPlugin } from '@grafana/data';

export const plugin = new PanelPlugin<MyOptions>(MyPanel);
```

src/MyPanel.tsx

```
import { PanelProps } from '@grafana/data';

interface Props extends PanelProps<SimpleOptions> {}

export const MyPanel: React.FC<Props> = ({ options, data, width, height }) => {
  // ...
};
```

Migrate a data source plugin While all plugins are different, we'd like to share a migration process that has worked for some of our users.

1. Define your configuration model and `ConfigEditor`. For many plugins, the configuration editor is the simplest component so it's a good candidate to start with.
2. Implement the `testDatasource()` method on the class that extends `DataSourceApi` using the settings in your configuration model to make sure you can successfully configure and access the external API.
3. Implement the `query()` method. At this point, you can hard-code your query, because we haven't yet implemented the query editor. The `query()` method supports both the new data frame response and the old TimeSeries response, so don't worry about converting to the new format just yet.
4. Implement the `QueryEditor`. How much work this requires depends on how complex your query model is.

By now, you should be able to release your new version.

To fully migrate to the new plugin platform, convert the time series response to a data frame response.

Migrate to data frames Before 7.0, data source and panel plugins exchanged data using either time series or tables. Starting with 7.0, plugins use the new data frame format to pass data from data sources to panels.

Grafana 7.0 is backward compatible with the old data format used in previous versions. Panels and data sources using the old format will still work with plugins using the new data frame format.

The `DataQueryResponse` returned by the `query` method can be either a LegacyResponseData or a DataFrame.

The toDataFrame() function converts a legacy response, such as `TimeSeries` or `Table`, to a `DataFrame`. Use it to gradually move your code to the new format.

```
import { toDataFrame } from '@grafana/data';

async query(options: DataQueryRequest<MyQuery>): Promise<DataQueryResponse> {
  return {
    data: options.targets.map(query => {
      const timeSeries: TimeSeries = await doLegacyRequest(query);
      return toDataFrame(timeSeries);
    }
  };
}
```

For more information, refer to Data frames.

Troubleshoot plugin migration

As of Grafana 7.0, backend plugins can now be cryptographically signed to verify their origin. By default, Grafana ignores unsigned plugins. For more information, refer to Allow unsigned plugins.