Red Hat Developer Hub 1.8

Installing and viewing plugins in Red Hat Developer Hub

Installing plugins in Red Hat Developer Hub

Last Updated: 2026-01-22

Red Hat Developer Hub 1.8 Installing and viewing plugins in Red Hat Developer Hub

Installing plugins in Red Hat Developer Hub

Legal Notice

Abstract

Administrative users can install and configure plugins to enable other users to use plugins to extend Red Hat Developer Hub (RHDH) functionality.

Table of Contents

CHAPTER 1. INSTALLING DYNAMIC PLUGINS IN RED HAT DEVELOPER HUB

The dynamic plugin support is based on the backend plugin manager package, which is a service that scans a configured root directory (**dynamicPlugins.rootDirectory** in the **app-config.yaml** file) for dynamic plugin packages and loads them dynamically.

You can use the dynamic plugins that come preinstalled with Red Hat Developer Hub or install external dynamic plugins from a public NPM registry.

1.1. INSTALLING DYNAMIC PLUGINS WITH THE RED HAT DEVELOPER HUB OPERATOR

You can store the configuration for dynamic plugins in a **ConfigMap** object that your **Backstage** custom resource (CR) can reference.

Dynamic plugins might require certain Kubernetes resources to be configured. These resources are referred to as **plugin dependencies**. For more information, see Dynamic plugins dependency management.

In Red Hat Developer Hub (RHDH), you can automatically create these resources when the Backstage CR is applied to the cluster.

NOTE

If the **pluginConfig** field references environment variables, you must define the variables in your ***<my_product_secrets>*** secret.

Procedure

1. From the OpenShift Container Platform web console, select the **ConfigMaps** tab.
2. Click **Create ConfigMap**.
3. From the **Create ConfigMap** page, select the **YAML view** option in **Configure via** and edit the file, if needed.

Example ConfigMap object using the GitHub dynamic plugin

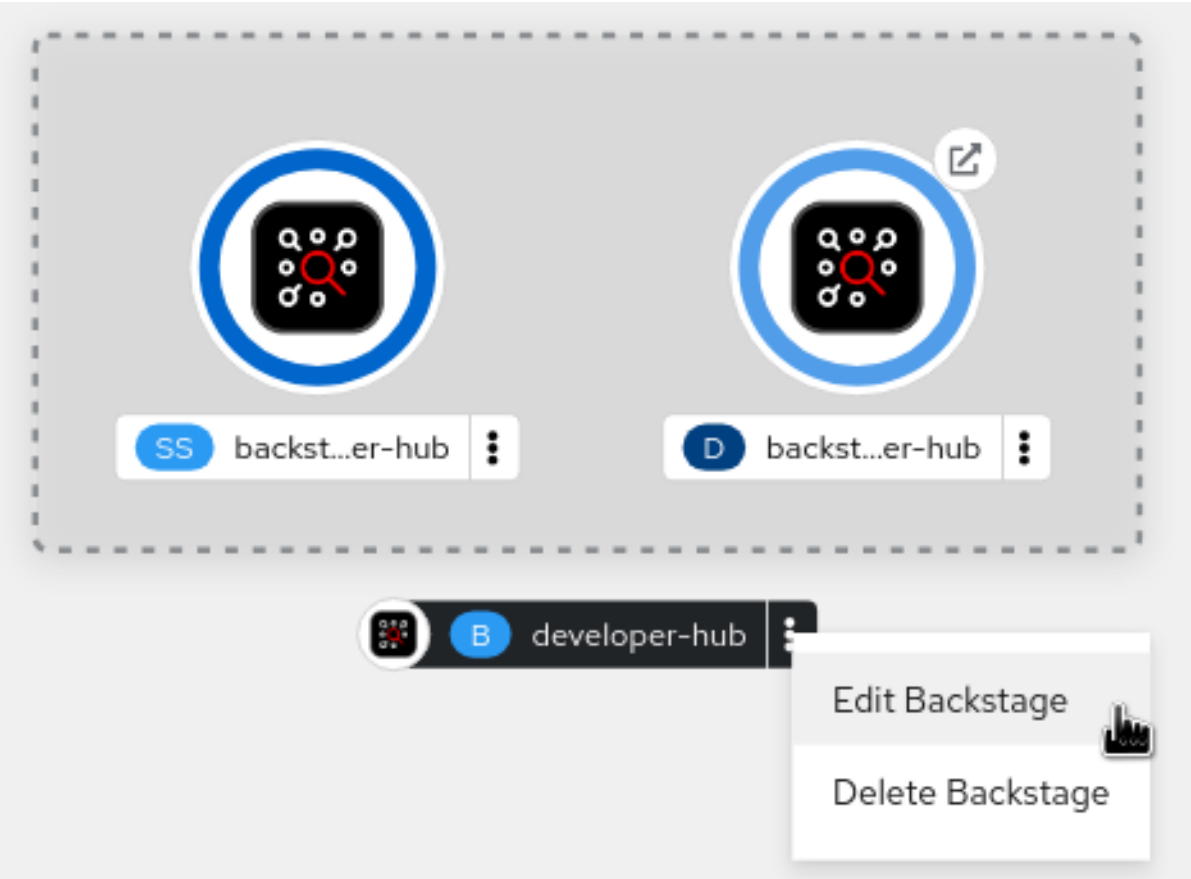
```
kind: ConfigMap
apiVersion: v1
metadata:
  name: dynamic-plugins-rhdh
data:
  dynamic-plugins.yaml: |
    includes:
      - dynamic-plugins.default.yaml
    plugins:
      - package: './dynamic-plugins/dist/backstage-plugin-catalog-backend-module-github-dynamic'
        disabled: false
        pluginConfig:
          catalog:
```

```
        providers:
          github:
            organization: "${GITHUB_ORG}"
            schedule:
              frequency: { minutes: 1 }
              timeout: { minutes: 1 }
              initialDelay: { seconds: 100 }
```

4. Click **Create**.

5. Go to the **Topology** view.

6. Click on the overflow menu for the Red Hat Developer Hub instance that you want to use and select **Edit Backstage** to load the YAML view of the Red Hat Developer Hub instance.



7. Add the **dynamicPluginsConfigMapName** field to your **Backstage** CR. For example:

```
apiVersion: rhdh.redhat.com/v1alpha3
kind: Backstage
metadata:
  name: my-rhdh
spec:
  application:
# ...
    dynamicPluginsConfigMapName: dynamic-plugins-rhdh
# ...
```

8. Click **Save**.

9. Navigate back to the **Topology** view and wait for the Red Hat Developer Hub pod to start.

10. Click the **Open URL** icon to start using the Red Hat Developer Hub platform with the new configuration changes.

Verification

- Ensure that the dynamic plugins configuration has been loaded, by appending **/api/dynamic-plugins-info/loaded-plugins** to your Red Hat Developer Hub root URL and checking the list of plugins:

  Example list of plugins

```
[
  {
    "name": "backstage-plugin-catalog-backend-module-github-dynamic",
    "version": "0.5.2",
    "platform": "node",
    "role": "backend-plugin-module"
  },
  {
    "name": "backstage-plugin-techdocs",
    "version": "1.10.0",
    "role": "frontend-plugin",
    "platform": "web"
  },
  {
    "name": "backstage-plugin-techdocs-backend-dynamic",
    "version": "1.9.5",
    "platform": "node",
    "role": "backend-plugin"
  },
]
```

1.2. DYNAMIC PLUGINS DEPENDENCY MANAGEMENT

Dynamic plugins configured for the Backstage custom resource (CR) may require certain Kubernetes resources to be configured to make the plugin work. These resources are referred to as **plugin dependencies**. In Red Hat Developer Hub (RHDH), you can automatically create these resources when the Backstage CR is applied to the cluster.

1.2.1. Cluster level plugin dependencies configuration

You can configure plugin dependencies by including the required Kubernetes resources in the **/config/profile/{PROFILE}/plugin-deps** directory. You must add the required resources as Kubernetes manifests in YAML format in the **plugin-deps** directory.

Example showing how to add **example-dep1.yaml** and **example-dep2.yaml** as plugin dependencies:

```
config/
  profile/
    rhdh/
     kustomization.yaml
     plugin-deps/
        example-dep1.yaml
        example-dep2.yaml
```

-

NOTE

- If a resource manifest does not specify a namespace, it will be created in the namespace of the Backstage CR.
- Resources may contain **{{backstage-name}}** and **{{backstage-ns}}** placeholders, which will be replaced with the name and namespace of the Backstage CR, respectively.

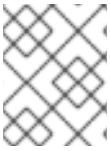
The **kustomization.yaml** file must contain the following lines:

```
configMapGenerator:
  - files:
      - plugin-deps/example-dep1.yaml
      - plugin-deps/example-dep2.yaml
    name: plugin-deps
```

1.2.2. Plugin dependencies infrastructure

To install infrastructural resources that are required by plugin dependencies, for example, other operators or custom resources (CR), you can include these in the **/config/profile/{PROFILE}/plugin-infra** directory.

To create these infrastructural resources (along with the operator deployment), use the **make plugin-infra** command.



NOTE

On a production cluster, use this command with caution as it might reconfigure cluster-scoped resources.

1.2.3. Plugin configuration

You must reference the plugin dependenciesin the **dependencies** field of the plugin configuration when the Backstage CR is applied.

The Operator creates the resources described in the files located in the **plugin-deps** directory.

You can reference plugin dependencies in the **dynamic-plugins** ConfigMap which can either be part of the default profile configuration for all Backstage custom resources or part of the ConfigMap referenced in the Backstage CR. In Red Hat Developer Hub, you can include plugin dependencies in the dynamic plugin configuration.

Each **dependencies.ref** value can either match the full file name or serve as a prefix for the file name. The operator will create the resources described in the files contained in the **plugin-deps** that start with the specified **ref** value or exactly match it

Example showing how to add **example-dep** plugin dependency:

```
apiVersion: v1
kind: ConfigMap
metadata:
```

```
  name: default-dynamic-plugins
data:
  dynamic-plugins.yaml: |
    includes:
      - dynamic-plugins.default.yaml
    plugins:
      - disabled: false
        package: "path-or-url-to-example-plugin"
        dependencies:
          - ref: example-dep
```

1.3. INSTALLING DYNAMIC PLUGINS USING THE HELM CHART

You can deploy a Developer Hub instance by using a Helm chart, which is a flexible installation method. With the Helm chart, you can sideload dynamic plugins into your Developer Hub instance without having to recompile your code or rebuild the container.

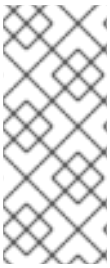
To install dynamic plugins in Developer Hub using Helm, add the following **global.dynamic** parameters in your Helm chart:

- **plugins**: the dynamic plugins list intended for installation. By default, the list is empty. You can populate the plugins list with the following fields:
  - **package**: a package specification for the dynamic plugin package that you want to install. You can use a package for either a local or an external dynamic plugin installation. For a local installation, use a path to the local folder containing the dynamic plugin. For an external installation, use a package specification from a public NPM repository.
  - **integrity** (required for external packages): an integrity checksum in the form of **<alg>-<digest>** specific to the package. Supported algorithms include **sha256**, **sha384** and **sha512**.
  - **pluginConfig**: an optional plugin-specific **app-config.yaml** YAML fragment. See plugin configuration for more information.
  - **disabled**: disables the dynamic plugin if set to **true**. Default: **false**.
  - **forceDownload**: Set the value to **true** to force a reinstall of the plugin, bypassing the cache. The default value is **false**.
  - **pullPolicy**: Similar to the **forceDownload** parameter and is consistent with other image container platforms. You can use one of the following values for this key:
    - **Always**: This value compares the image digest in the remote registry and downloads the artifact if it has changed, even if the plugin was previously downloaded.
    - **IfNotPresent**: This value downloads the artifact if it is not already present in the dynamic-plugins-root folder, without checking image digests.

NOTE

The **pullPolicy** setting is also applied to the NPM downloading method, although **Always** will download the remote artifact without a digest check. The existing **forceDownload** option remains functional, however, the **pullPolicy** option takes precedence. The **forceDownload** option may be deprecated in a future Developer Hub release.

- **includes**: a list of YAML files utilizing the same syntax.



NOTE

The **plugins** list in the **includes** file is merged with the **plugins** list in the main Helm values. If a plugin package is mentioned in both **plugins** lists, the **plugins** fields in the main Helm values override the **plugins** fields in the **includes** file. The default configuration includes the **dynamic-plugins.default.yaml** file, which contains all of the dynamic plugins preinstalled in Developer Hub, whether enabled or disabled by default.

1.3.1. Example Helm chart configurations for dynamic plugin installations

The following examples demonstrate how to configure the Helm chart for specific types of dynamic plugin installations.

Configuring a local plugin and an external plugin when the external plugin requires a specific configuration

```
global:
  dynamic:
    plugins:
      - package: <alocal package-spec used by npm pack>
      - package: <external package-spec used by npm pack>
        integrity: sha512-<some hash>
        pluginConfig: ...
```

Disabling a plugin from an included file

```
global:
  dynamic:
    includes:
      - dynamic-plugins.default.yaml
    plugins:
      - package: <some imported plugins listed in dynamic-plugins.default.yaml>
        disabled: true
```

Enabling a plugin from an included file

```
global:
  dynamic:
    includes:
      - dynamic-plugins.default.yaml
    plugins:
      - package: <some imported plugins listed in dynamic-plugins.custom.yaml>
        disabled: false
```

Enabling a plugin that is disabled in an included file

```
global:
  dynamic:
    includes:
      - dynamic-plugins.default.yaml
    plugins:
      - package: <some imported plugins listed in dynamic-plugins.custom.yaml>
        disabled: false
```

1.4. INSTALLING DYNAMIC PLUGINS IN AN AIR-GAPPED ENVIRONMENT

You can install external plugins in an air-gapped environment by setting up a custom NPM registry.

You can configure the NPM registry URL and authentication information for dynamic plugin packages using a Helm chart. For dynamic plugin packages obtained through **npm pack**, you can use a **.npmrc** file.

Using the Helm chart, add the **.npmrc** file to the NPM registry by creating a secret. For example:

```
apiVersion: v1
kind: Secret
metadata:
  name: <release_name>-dynamic-plugins-npmrc (1)
type: Opaque
stringData:
  .npmrc: |
    registry=<registry-link>
    //<registry-link>:_authToken=<auth-token>
          ...
```

1. Replace ***<release_name>*** with your Helm release name. This name is a unique identifier for each chart installation in the Kubernetes cluster.

CHAPTER 2. CUSTOM PLUGINS IN RED HAT DEVELOPER HUB

You can integrate custom dynamic plugins into Red Hat Developer Hub to enhance its functionality without modifying its source code or rebuilding it. To add these plugins, export them as derived packages.

While exporting the plugin package, you must ensure that dependencies are correctly bundled or marked as shared, depending on their relationship to the Developer Hub environment.

To integrate a custom plugin into Developer Hub:

1. First, obtain the plugin's source code.
2. Export the plugin as a dynamic plugin package. See Section 2.1, "Exporting custom plugins in Red Hat Developer Hub".
3. Package and publish the dynamic plugin. See Section 2.2, "Packaging and publishing custom plugins as dynamic plugins".
4. Install the plugin in the Developer Hub environment. See Section 2.3, "Installing custom plugins in Red Hat Developer Hub".

2.1. EXPORTING CUSTOM PLUGINS IN RED HAT DEVELOPER HUB

To use plugins in Red Hat Developer Hub, you can export plugins as derived dynamic plugin packages. These packages contain the plugin code and dependencies, ready for dynamic plugin integration into Developer Hub.

Prerequisites

- The **@red-hat-developer-hub/cli** package is installed. Use the latest version (**@latest** tag) for compatibility with the most recent features and fixes.

NOTE

Use the **npx @red-hat-developer-hub/cli@latest plugin export** command to export an existing custom plugin as a derived dynamic plugin package.

You must use this command when you have the source code for a custom plugin and want to integrate it into RHDH as a dynamic plugin.

The command processes the plugin's source code and dependencies and generates the necessary output for dynamic loading by RHDH.

For an example of using this command, see Example of installing a custom plugin in Red Hat Developer Hub.

- Node.js and NPM is installed and configured.
- The custom plugin is compatible with your Red Hat Developer Hub version. For more information, see Version compatibility matrix.
- The custom plugin must have a valid **package.json** file in its root directory, containing all required metadata and dependencies.

Backend plugins

To ensure compatibility with the dynamic plugin support and enable their use as dynamic plugins, existing backend plugins must be compatible with the new Backstage backend system. Additionally, these plugins must be rebuilt using a dedicated CLI command.
The new Backstage backend system entry point (created using **createBackendPlugin()** or **createBackendModule()**) must be exported as the default export from either the main package or an **alpha** package (if the plugin instance support is still provided using **alpha** APIs). This doesn't add any additional requirement on top of the standard plugin development guidelines of the plugin instance.

The dynamic export mechanism identifies private dependencies and sets the **bundleDependencies** field in the **package.json** file. This export mechanism ensures that the dynamic plugin package is published as a self-contained package, with its private dependencies bundled in a private **node_modules** folder.

Certain plugin dependencies require specific handling in the derived packages, such as:

- **Shared dependencies** are provided by the RHDH application and listed as **peerDependencies** in **package.json** file, not bundled in the dynamic plugin package. For example, by default, all **@backstage** scoped packages are shared.
  You can use the **--shared-package** flag to specify shared dependencies, that are expected to be provided by Red Hat Developer Hub application and not bundled in the dynamic plugin package.

  To treat a **@backstage** package as private, use the negation prefix (**!**). For example, when a plugin depends on the package in **@backstage** that is not provided by the Red Hat Developer Hub application.

- **Embedded dependencies** are bundled into the dynamic plugin package with their dependencies hoisted to the top level. By default, packages with **-node** or **-common** suffixes are embedded.
  You can use the **--embed-package** flag to specify additional embedded packages. For example, packages from the same workspace that do not follow the default naming convention.

  The following is an example of exporting a dynamic plugin with shared and embedded packages:

  Example dynamic plugin export with shared and embedded packages

  ```
  $ npx @red-hat-developer-hub/cli@latest plugin export --shared-package
  '!/@backstage/plugin-notifications/' --embed-package @backstage/plugin-
  notifications-backend
  ```

  In the previous example:

- **@backstage/plugin-notifications** package is treated as a private dependency and is bundled in the dynamic plugin package, despite being in the **@backstage** scope.

- **@backstage/plugin-notifications-backend** package is marked as an embedded dependency and is bundled in the dynamic plugin package.

Front-end plugins

Front-end plugins can use **scalprum** for configuration, which the CLI can generate automatically during the export process. The generated default configuration is logged when running the following command:

Example command to log the default configuration

```
$ npx @red-hat-developer-hub/cli@latest plugin export
```

The following is an example of default **scalprum** configuration:

Default scalprum configuration

```
"scalprum": {
  "name": "<package_name>",  // The Webpack container name matches the NPM
package name, with "@" replaced by "." and "/" removed.
  "exposedModules": {
    "PluginRoot": "./src/index.ts"  // The default module name is "PluginRoot" and doesn't
need explicit specification in the app-config.yaml file.
  }
}
```

You can add a **scalprum** section to the **package.json** file. For example:

Example scalprum customization

```
"scalprum": {
  "name": "custom-package-name",
  "exposedModules": {
    "FooModuleName": "./src/foo.ts",
    "BarModuleName": "./src/bar.ts"
    // Define multiple modules here, with each exposed as a separate entry point in the
Webpack container.
  }
}
```

Dynamic plugins might need adjustments for Developer Hub needs, such as static JSX for mountpoints or dynamic routes. These changes are optional but might be incompatible with static plugins.

To include static JSX, define an additional export and use it as the dynamic plugin's **importName**. For example:

Example static and dynamic plugin export

```
// For a static plugin
$ export const EntityTechdocsContent = () => {...}

// For a dynamic plugin
$ export const DynamicEntityTechdocsContent = {
  element: EntityTechdocsContent,
  staticJSXContent: (
    <TechDocsAddons>
      <ReportIssue />
```

```
    </TechDocsAddons>
  ),
};
```

Procedure

- Use the **plugin export** command from the **@red-hat-developer-hub/cli** package to export the plugin:

  Example command to export a custom plugin

  ```
  $ npx @red-hat-developer-hub/cli@latest plugin export
  ```

  Ensure that you execute the previous command in the root directory of the plugin's JavaScript package (containing **package.json** file).

  The resulting derived package will be located in the **dist-dynamic** subfolder. The exported package name consists of the original plugin name with **-dynamic** appended.

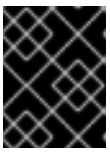
WARNING

The derived dynamic plugin JavaScript packages must not be published to the public NPM registry. For more appropriate packaging options, see Section 2.2, "Packaging and publishing custom plugins as dynamic plugins" . If you must publish to the NPM registry, use a private registry.

2.2. PACKAGING AND PUBLISHING CUSTOM PLUGINS AS DYNAMIC PLUGINS

After exporting a custom plugin, you can package the derived package into one of the following supported formats:

- Open Container Initiative (OCI) image (recommended)
- TGZ file
- JavaScript package



IMPORTANT

Exported dynamic plugin packages must only be published to private NPM registries.

2.2.1. Creating an OCI image with dynamic packages

Prerequisites

- You have installed **podman** or **docker**.
- You have exported a custom dynamic plugin package. For more information, see Section 2.1, "Exporting custom plugins in Red Hat Developer Hub".

Procedure

1. Navigate to the plugin's root directory (not the **dist-dynamic** directory).
2. Run the following command to package the plugin into an OCI image:

   Example command to package an exported custom plugin

   ```
   $ npx @red-hat-developer-hub/cli@latest plugin package --tag quay.io/example/image:v0.0.1
   ```

   In the previous command, the **--tag** argument specifies the image name and tag.
3. Run one of the following commands to push the image to a registry:

   Example command to push an image to a registry using podman

   ```
   $ podman push quay.io/example/image:v0.0.1
   ```

   Example command to push an image to a registry using docker

   ```
   $ docker push quay.io/example/image:v0.0.1
   ```

   The output of the **package-dynamic-plugins** command provides the plugin's path for use in the **dynamic-plugin-config.yaml** file.

2.2.2. Creating a TGZ file with dynamic packages

Prerequisites

- You have exported a custom dynamic plugin package. For more information, see Section 2.1, "Exporting custom plugins in Red Hat Developer Hub".

Procedure

1. Navigate to the **dist-dynamic** directory.
2. Run the following command to create a **tgz** archive:

   Example command to create a tgz archive

   ```
   $ npm pack
   ```

   You can obtain the integrity hash from the output of the **npm pack** command by using the **--json** flag as follows:

   Example command to obtain the integrity hash of a tgz archive

```
$ npm pack --json | head -n 10
```

3. Host the archive on a web server accessible to your RHDH instance, and reference its URL in the **dynamic-plugin-config.yaml** file as follows:

   Example dynamic-plugin-config.yaml file

   ```
   plugins:
     - package: https://example.com/backstage-plugin-myplugin-1.0.0.tgz
       integrity: sha512-<hash>
   ```

4. Run the following command to package the plugins:

   Example command to package a dynamic plugin

   ```
   $ npm pack --pack-destination ~/test/dynamic-plugins-root/
   ```

   TIP

   To create a plugin registry using HTTP server on OpenShift Container Platform, run the following commands:

   Example commands to build and deploy an HTTP server in OpenShift Container Platform

   ```
   $ oc project my-rhdh-project
   $ oc new-build httpd --name=plugin-registry --binary
   $ oc start-build plugin-registry --from-dir=dynamic-plugins-root --wait
   $ oc new-app --image-stream=plugin-registry
   ```

5. Configure your RHDH to use plugins from the HTTP server by editing the **dynamic-plugin-config.yaml** file:

   Example configuration to use packaged plugins in RHDH

   ```
   plugins:
     - package: http://plugin-registry:8080/backstage-plugin-myplugin-1.9.6.tgz
   ```

2.2.3. Creating a JavaScript package with dynamic packages

WARNING

The derived dynamic plugin JavaScript packages must not be published to the public NPM registry. If you must publish to the NPM registry, use a private registry.

Prerequisites

- You have exported a custom dynamic plugin package. For more information, see Section 2.1, "Exporting custom plugins in Red Hat Developer Hub".

Procedure

1. Navigate to the **dist-dynamic** directory.
2. Run the following command to publish the package to your private NPM registry:

   Example command to publish a plugin package to an NPM registry

   ```
   $ npm publish --registry <npm_registry_url>
   ```

   TIP

   You can add the following to your **package.json** file before running the **export** command:

   Example package.json file

   ```
   {
     "publishConfig": {
       "registry": "<npm_registry_url>"
     }
   }
   ```

   If you modify **publishConfig** after exporting the dynamic plugin, re-run the **plugin export** command to ensure the correct configuration is included.

2.3. INSTALLING CUSTOM PLUGINS IN RED HAT DEVELOPER HUB

You can install a custom plugins in Red Hat Developer Hub without rebuilding the RHDH application.

The location of the **dynamic-plugin-config.yaml** file depends on the deployment method. For more details, refer to Installing dynamic plugins with the Red Hat Developer Hub Operator and Installing dynamic plugins using the Helm chart.

Plugins are defined in the **plugins** array within the **dynamic-plugin-config.yaml** file. Each plugin is represented as an object with the following properties:

- **package**: The plugin's package definition, which can be an OCI image, a TGZ file, a JavaScript package, or a directory path.
- **disabled**: A boolean value indicating whether the plugin is enabled or disabled.
- **integrity**: The integrity hash of the package, required for TGZ file and JavaScript packages.
- **pluginConfig**: The plugin's configuration. For backend plugins, this is optional; for frontend plugins, it is required. The **pluginConfig** is a fragment of the **app-config.yaml** file, and any added properties are merged with the RHDH **app-config.yaml** file.

NOTE

You can also load dynamic plugins from another directory, though this is intended for development or testing purposes and is not recommended for production, except for plugins included in the RHDH container image. For more information, see Chapter 4, *Enabling plugins added in the RHDH container image* .

2.3.1. Loading a plugin packaged as an OCI image

Prerequisites

- The custom plugin is packaged as a dynamic plugin in an OCI image.
  For more information about packaging a custom plugin, see Section 2.2, "Packaging and publishing custom plugins as dynamic plugins".

Procedure

1. To retrieve plugins from an authenticated registry, complete the following steps:
   a. Log in to the container image registry.

   ```
   podman login <registry>
   ```

   b. Verify the content of the **auth.json** file created after the login.

   ```
   cat ${XDG_RUNTIME_DIR:-~/.config}/containers/auth.json
   ```

   c. Create a secret file using the following example:

   ```
   oc create secret generic _<secret_name>_ --from-
   file=auth.json=${XDG_RUNTIME_DIR:-~/.config}/containers/auth.json 1
   ```

   - For an Operator-based deployment, replace *<secret_name>* with **dynamic-plugins-registry-auth**.
   - For a Helm-based deployment, replace *<secret_name>* with **<Helm_release_name>_-dynamic-plugins-registry-auth**.

2. Define the plugin with the **oci://** prefix in the following format in **dynamic-plugins.yaml** file: **oci://*<image_name>*:<tag>!*<plugin_name>***

   Example configuration in dynamic-plugins.yaml file

   ```
   plugins:
     - disabled: false
       package: oci://quay.io/example/image:v0.0.1!backstage-plugin-myplugin
   ```

3. To perform an integrity check, use the image digest in place of the tag in the **dynamic-plugins.yaml** file as shown in the following example:

   Example configuration in dynamic-plugins.yaml file

   ```
   plugins:
   ```

```
  - disabled: false
    package:
oci://quay.io/example/image@sha256:28036abec4dffc714394e4ee433f16a59493db8017795049c831be41c02eb5dc!backstage-plugin-myplugin
```

4. To apply the changes, restart the RHDH application.

2.3.2. Loading a plugin packaged as a TGZ file

Prerequisites

- The custom plugin is packaged as a dynamic plugin in a TGZ file.
  For more information about packaging a custom plugin, see Section 2.2, "Packaging and publishing custom plugins as dynamic plugins".

Procedure

1. Specify the archive URL and its integrity hash in the **dynamic-plugins.yaml** file using the following example:

   Example configuration in dynamic-plugins.yaml file

```
plugins:
  - disabled: false
    package: https://example.com/backstage-plugin-myplugin-1.0.0.tgz
    integrity: sha512-9WlbgEdadJNeQxdn1973r5E4kNFvnT9GjLD627GWgrhCaxjCmxqdNW08cj+Bf47mwAtZMt1Ttyo+ZhDRDj9PoA==
```

2. To apply the changes, restart the RHDH application.

2.3.3. Loading a plugin packaged as a JavaScript package

Prerequisites

- The custom plugin is packaged as a dynamic plugin in a JavaScript package.
  For more information about packaging a custom plugin, see Section 2.2, "Packaging and publishing custom plugins as dynamic plugins".

Procedure

1. Run the following command to obtain the integrity hash from the NPM registry:

```
$ npm view --registry <registry-link> <npm package>@<version> dist.integrity
```

2. Specify the package name, version, and its integrity hash in the **dynamic-plugins.yaml** file as follows:

   Example configuration in dynamic-plugins.yaml file

```
plugins:
  - disabled: false
```

```
    package: @example/backstage-plugin-myplugin@1.0.0
    integrity: sha512-
9WlbgEdadJNeQxdn1973r5E4kNFvnT9GjLD627GWgrhCaxjCmxqdNW08cj+Bf47mwAtZMt1Tt
yo+ZhDRDj9PoA==
```

3. If you are using a custom NPM registry, create a **.npmrc** file with the registry URL and authentication details:

   Example code for .npmrc file

   ```
   registry=<registry-link>
   //<registry-link>:_authToken=<auth-token>
   ```

4. When using OpenShift Container Platform or Kubernetes:

   - Use the Helm chart to add the **.npmrc** file by creating a secret. For example:

     Example secret configuration

     ```
     apiVersion: v1
     kind: Secret
     metadata:
       name: <release_name>-dynamic-plugins-npmrc (1)
     type: Opaque
     stringData:
       .npmrc: |
         registry=<registry-link>
         //<registry-link>:_authToken=<auth-token>
     ```

     (1) Replace ***<release_name>*** with your Helm release name. This name is a unique identifier for each chart installation in the Kubernetes cluster.

   - For RHDH Helm chart, name the secret using the following format for automatic mounting: ***<release_name>*-dynamic-plugins-npmrc**

5. To apply the changes, restart the RHDH application.

2.4. EXAMPLE OF INSTALLING A CUSTOM PLUGIN IN RED HAT DEVELOPER HUB

This example demonstrates how to package and install dynamic plugins using the Backstage **Entity Feedback** community plugin that is not included in Red Hat Developer Hub pre-installed dynamic plugins.

Limitations

- You need to ensure that your custom plugin is built with a compatible version of Backstage. In Developer Hub, click **Settings**. Your custom plugin must be compatible with the **Backstage Version** (or the closest previous version) that is displayed in the **Metadata** section of Red Hat Developer Hub.
  For example, if you view the history of the **backstage.json** file for the **Entity Feedback** plugin, the **1fc87de** commit is closest previous version to Backstage version of 1.39.1.

Figure 2.1. **backstage.json** file history in Github

Prerequisites

- Your local environment meets the following requirements:
- **Node.js:** Version 22.x
- **Yarn:** Version 4.x
- **git CLI**
- **jq CLI:** Command-line JSON processor
- **OpenShift CLI (oc):** The client for interacting with your OpenShift cluster.
- **Container runtime:** Either podman or docker is required for packaging the plugin into an OCI image and logging into registries.
- **Container registry access:** Access to an OCI-compliant container registry (such as the internal OpenShift registry or a public registry like Quay.io).

Procedure

1. Clone the source code for the **Entity Feedback** plugin, as follows:

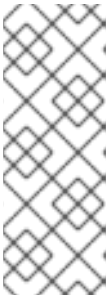
```
$ git clone https://github.com/backstage/community-plugins.git
$ cd community-plugins
```

2. Prepare your environment to build the plugin by enabling Yarn for your Node.js installation, as follows:

```
$ corepack enable yarn
```

3. Install the dependencies, compile the code, and build the plugins, as follows:

```
$ cd workspaces/entity-feedback
$ yarn install
$ yarn tsc
$ yarn build:all
```



NOTE

After this step, with upstream Backstage, you publish the built plugins to a NPM or NPM-compatible registry. In this example, as you are building this plugin to support it being loaded dynamically by Red Hat Developer Hub, you can skip the **npm publish** step that publishes the plugin to a NPM registry. Instead, you can package the plugin for dynamic loading and publish it as a container image on **Quay.io** or your preferred container registry.

4. Prepare the **Entity Feedback** frontend plugin by using the Red Hat Developer Hub CLI. The following command uses the plugin files in the **dist** folder that was generated by the **yarn build:all** command, and creates a new **dist-scalprum** folder that contains the necessary configuration and source files to enable dynamic loading:

```
$ cd plugins/entity-feedback
$ npx @red-hat-developer-hub/cli@latest plugin export
```

When this command packages a frontend plugin, it uses a default Scalprum configuration if one is not found. The Scalprum configuration is used to specify the plugin entry point and exports, and then to build a **dist-scalprum** folder that contains the dynamic plugin. The default Scalprum configuration is shown below, however a **scalprum** key can be added to the **package.json** file used by your plugin to set custom values, if necessary:

```
{
  "name": "backstage-community.plugin-entity-feedback",
  "exposedModules": {
    "PluginRoot": "./src/index.ts"
  }
}
```

The following **plugin-manifest.json** file, which Red Hat Developer Hub uses to load the plugin, is located in the **dist-dynamic/dist-scalprum** folder:

```
{
  "name": "backstage-community.plugin-entity-feedback",
  "version": "0.6.0",
  "extensions": [],
  "registrationMethod": "callback",
  "baseURL": "auto",
  "loadScripts": [
    "backstage-community.plugin-entity-feedback.fd691533c03cb52c30ac.js"
  ],
  "buildHash": "fd691533c03cb52c30acbb5a80197c9d"
}
```

5. Package the plugin into a container image and publish it to Quay.io or your preferred container registry:

```
$ export QUAY_USER=replace-with-your-username
$ export PLUGIN_NAME=entity-feedback-plugin
$ export VERSION=$(cat package.json | jq .version -r)

$ npx @red-hat-developer-hub/cli@latest plugin package \
  --tag quay.io/$QUAY_USER/$PLUGIN_NAME:$VERSION
```

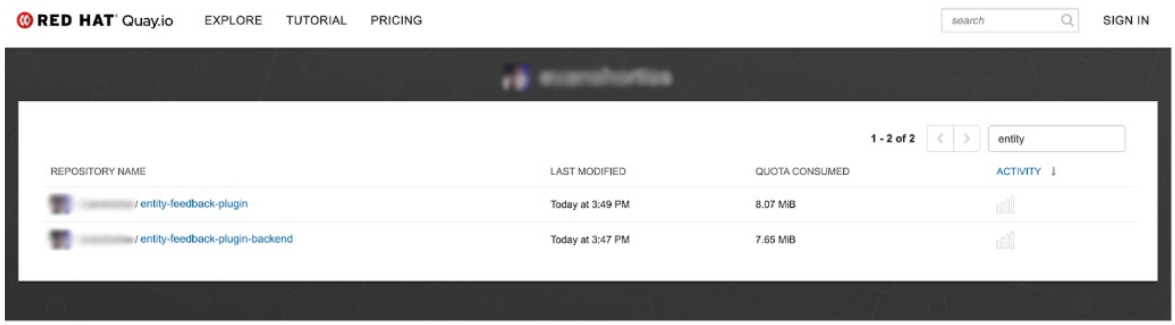
```
$ podman login quay.io
$ podman push quay.io/$QUAY_USER/$PLUGIN_NAME:$VERSION
```

6. Repeat the same steps for the backend plugin. Scalprum is not required for backend plugins, and a **dist-dynamic** folder is generated instead of a **dist-scalprum** folder:

```
$ cd ../entity-feedback-backend/
$ npx @red-hat-developer-hub/cli@latest plugin export

$ export QUAY_USER=replace-with-your-username
$ export PLUGIN_NAME=entity-feedback-plugin-backend
$ export VERSION=$(cat package.json | jq .version -r)

$ npx @red-hat-developer-hub/cli@latest plugin package \
  --tag quay.io/$QUAY_USER/$PLUGIN_NAME:$VERSION

$ podman push quay.io/$QUAY_USER/$PLUGIN_NAME:$VERSION
```

Those commands result in two container images being published to your container registry.

Figure 2.2. Container images published to Quay.io

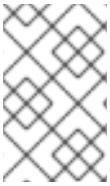


2.4.1. Adding a custom dynamic plugin to Red Hat Developer Hub

Procedure

1. To add your custom dynamic plugins to Red Hat Developer Hub. you must update the **dynamic-plugins.yaml** file by using the following configuration that is generated from the **npx @red-hat-developer-hub/cli@latest plugin package** command:

```
plugins:
  - package: oci://quay.io/_<user_name>_/entity-feedback-plugin:0.5.0!backstage-community-plugin-entity-feedback
    disabled: false

  - package: oci://quay.io/_<user_name>_/entity-feedback-plugin-backend:0.6.0!backstage-community-plugin-entity-feedback-backend
    disabled: false
```



NOTE

Ensure that your container images are publicly accessible, or that you have configured a pull secret in your environment. A pull secret provides Red Hat Developer Hub with credentials to authenticate pulling your plugin container images from a container registry.

2.4.2. Displaying the frontend plugin

Procedure

1. You need to update the **pluginConfig** section of your **dynamic-plugins.yaml** file to specify how the **Entity Feedback** should be added to the Red Hat Developer Hub UI.

dynamic-plugins.yaml file fragment

```
- package: oci://quay.io/_<user_name>_/entity-feedback-plugin:0.5.0!backstage-community-plugin-entity-feedback
  disabled: false
  pluginConfig:
    dynamicPlugins:
      frontend:
        backstage-community.plugin-entity-feedback:
          entityTabs:
            - mountPoint: entity.page.feedback
              path: /feedback
              title: Feedback
          mountPoints:
            - config:
                layout:
                  gridColumn: 1 / -1
              importName: StarredRatingButtons
              mountPoint: entity.page.feedback/cards
            - config:
                layout:
                  gridColumn: 1 / -1
              importName: EntityFeedbackResponseContent
              mountPoint: entity.page.feedback/cards
            - config:
                layout:
                  gridColumnEnd:
                    lg: span 6
                    md: span 6
                    xs: span 6
              importName: StarredRatingButtons
              mountPoint: entity.page.overview/cards
```

where:

backstage-community.plugin-entity-feedback:entityTabs

Enter the **entityTabs** array to define a new tab, named “Feedback” on the **Entity Overview** screen in Red Hat Developer Hub.

frontend:mountPoints

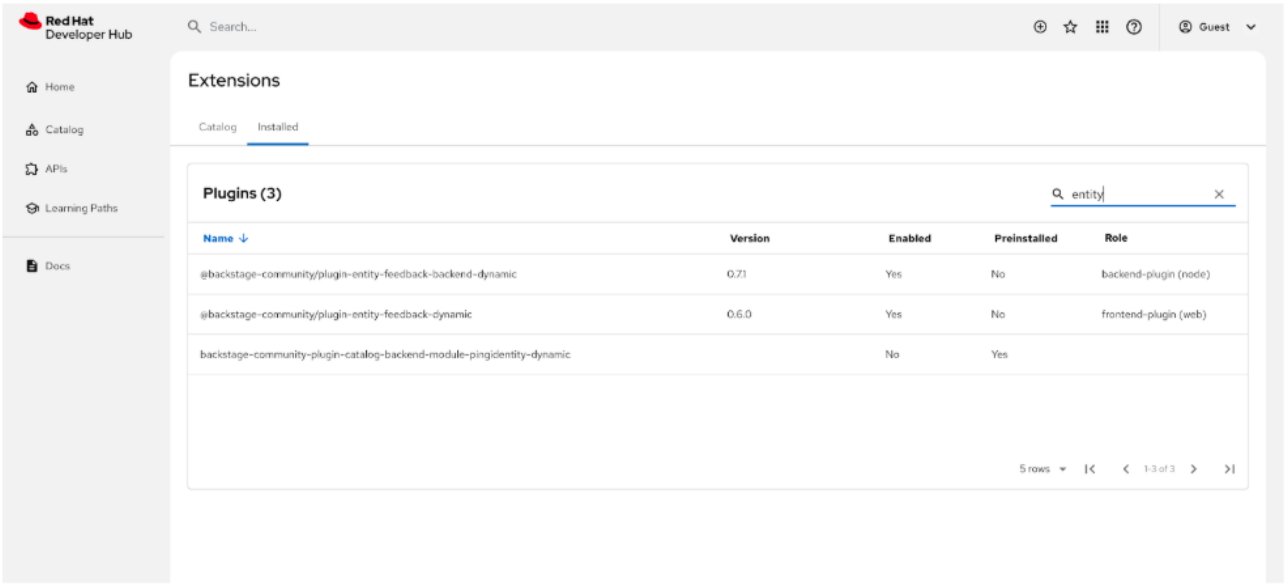
This array defines the following configurations to mount React components exposed by the plugin:

- The **StarredRatingButtons** component is added to the new **Feedback** tab defined in entityTabs.
- Similar to the **StarredRatingButtons**, the **EntityFeedbackResponseContent** is mounted on the **Feedback** tab.
- The **StarredRatingButtons** is added to the default **Overview** tab for each entity.

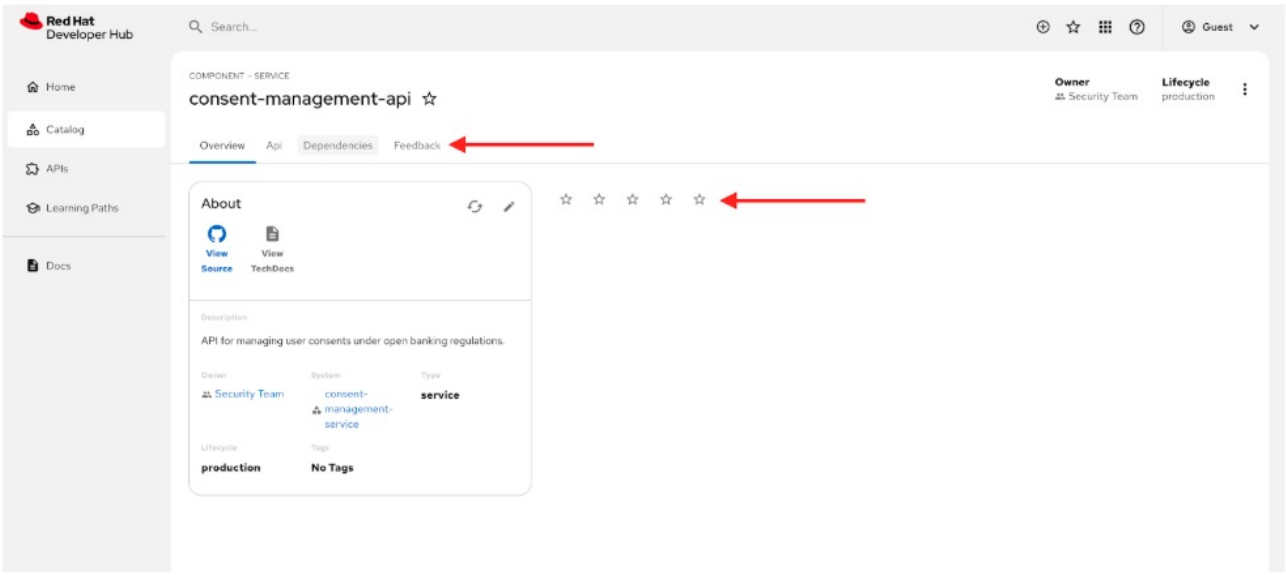
2. To complete installing the **Entity Feedback** plugins, you must redeploy your Red Hat Developer Hub instance.

Verification

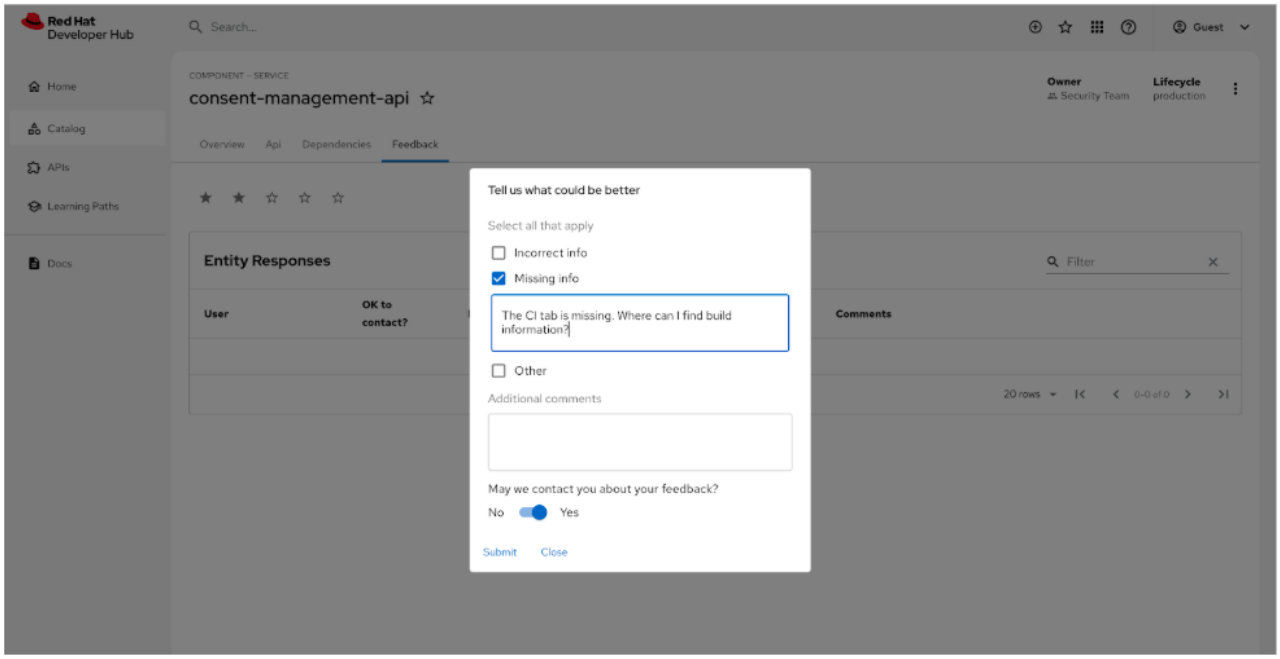
When your new instance of Red Hat Developer Hub has started, you can check that your plugins are installed and enabled by visiting the **Administration > Extensions**screen and searching for “entity” on the **Installed** tab.



When you click **Catalog**, you should see the new **Feedback** tab, and the **StarredRatingButtons** displayed, as follows:



Selecting a low star rating prompts the user to provide feedback, as follows:



NOTE

The user provided feedback is not saved if you are logged in as the Guest user.

CHAPTER 3. USING THE DYNAMIC PLUGIN FACTORY TO CONVERT PLUGINS INTO DYNAMIC PLUGINS

You can automate the conversion and packaging of standard Backstage plugins into RHDH dynamic plugins by using the RHDH Dynamic Plugin Factory tool.

IMPORTANT

The Dynamic Plugin Factory is maintained as an open-source project by Red Hat, but is not supported or subject to any service level agreement (SLA).

The core function of the Dynamic Plugin Factory tool is to streamline the dynamic plugin build process, offering the following capabilities:

Source Code Handling

Manages cloning, checking out, and applying custom patches to the plugin source.

Dependency Management

Handles yarn installation and TypeScript compilation.

Packaging

Uses the RHDH CLI to build, export, and package the final dynamic plugin.

Deployment

Offers an option to push the resulting container image to registries like Quay or OpenShift.

The Dynamic Plugin Factory tool provides a simplified, reproducible method for developers and platform engineers to create and test dynamic plugins using a pre-configured *dynamic plugin factory* container and documentation, significantly easing migration and testing.

For more information, see RHDH Dynamic Plugin Factory .

CHAPTER 4. ENABLING PLUGINS ADDED IN THE RHDH CONTAINER IMAGE

In the RHDH container image, a set of dynamic plugins is preloaded to enhance functionality. However, due to mandatory configuration requirements, most of the plugins are disabled.

You can enable and configure the plugins in the RHDH container image, including how to manage the default configuration, set necessary environment variables, and ensure the proper functionality of the plugins within your application.

Prerequisites

- You have access to the **dynamic-plugins.default.yaml** file, which lists all preloaded plugins and their default configuration.
- You have deployed the RHDH application, and have access to the logs of the **install-dynamic-plugins** init container.
- You have the necessary permissions to modify plugin configurations and access the application environment.
- You have identified and set the required environment variables referenced by the plugin's default configuration. These environment variables must be defined in the Helm Chart or Operator configuration.

Procedure

1. Start your RHDH application and access the logs of the **install-dynamic-plugins** init container within the RHDH pod.
2. Identify the Red Hat supported plugins that are disabled by default.
3. Copy the package configuration from the **dynamic-plugins.default.yaml** file.
4. Open the plugin configuration file and locate the plugin entry you want to enable.
   The location of the plugin configuration file varies based on the deployment method. For more details, see Installing and viewing plugins in Red Hat Developer Hub .
5. Modify the **disabled** field to **false** and add the package name as follows:

   #### Example plugin configuration

   ```
   plugins:
     - disabled: false
       package: ./dynamic-plugins/dist/backstage-plugin-catalog-backend-module-github-dynamic
   ```

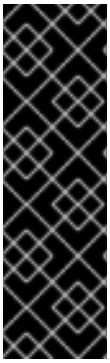
   For more information about how to configure dynamic plugins in Developer Hub, see Configuring dynamic plugins.

Verification

1. Restart the RHDH application and verify that the plugin is successfully activated and configured.

2. Verify the application logs for confirmation and ensure the plugin is functioning as expected.

CHAPTER 5. EXTENSIONS IN RED HAT DEVELOPER HUB



IMPORTANT

These features are for Technology Preview only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs), might not be functionally complete, and Red Hat does not recommend using them for production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information on Red Hat Technology Preview features, see Technology Preview Features Scope.

Red Hat Developer Hub (RHDH) includes the Extensions feature which is preinstalled and enabled by default. Extensions provides users with a centralized interface to browse and manage available plugins

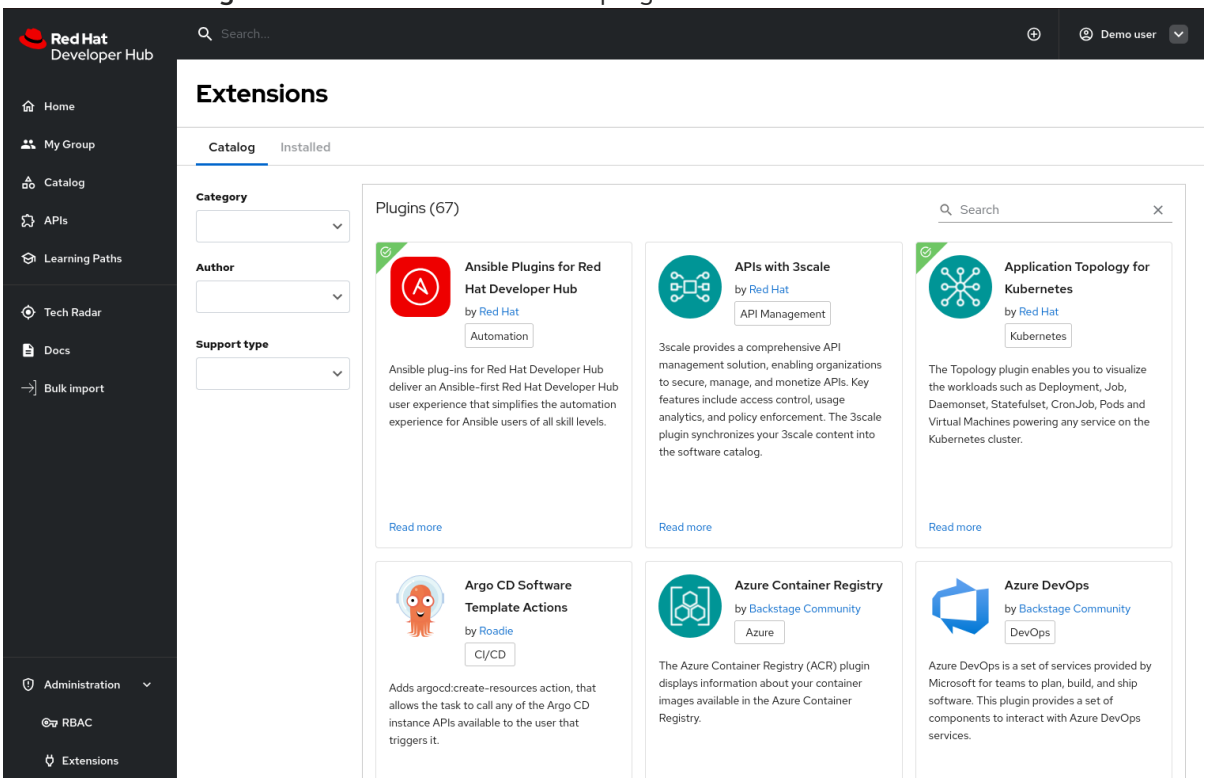
You can use Extensions to discover plugins that extend RHDH functionality, streamline development workflows, and improve the developer experience.

5.1. VIEWING AVAILABLE PLUGINS

You can view plugins available for your Red Hat Developer Hub application on the **Extensions** page.

Procedure

1. Open your Developer Hub application and click **Administration** > **Extensions**.
2. Go to the **Catalog** tab to view a list of available plugins and related information.



5.2. VIEWING INSTALLED PLUGINS

Using the Dynamic Plugins Info front-end plugin, you can view plugins that are currently installed in your Red Hat Developer Hub application. This plugin is enabled by default.

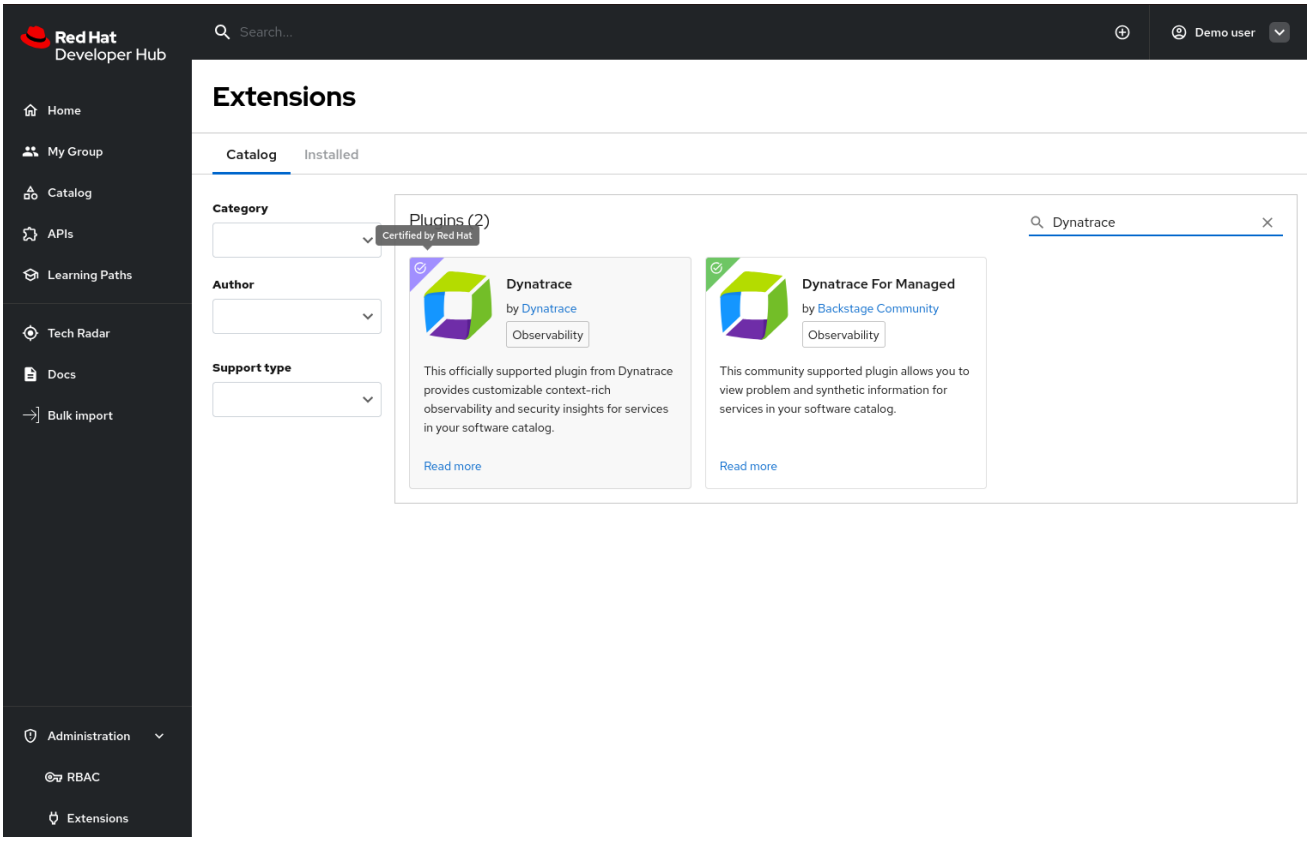
Procedure

1. Open your Developer Hub application and click **Administration** > **Extensions**.
2. Go to the **Installed** tab to view a list of installed plugins and related information.

5.3. SEARCH AND FILTER THE PLUGINS

5.3.1. Search by plugin name

You can use the search bar in the header to filter the Extensions plugin cards by name. For example, if you type “A” into the search bar, Extensions shows only the plugins that contain the letter “A” in the Name field.



Optionally, you can use the search bar in conjunction with a filter to filter only plugins of the selected filter by name. For example, you can apply the **Category** filter and then type a character into the search bar to view only Openshift plugins that contain the typed character in the name.

The following filters are available:

- Category
- Author
- Support type

5.4. REMOVING EXTENSIONS

The Extensions feature plugins are preinstalled in Red Hat Developer Hub (RHDH) and enabled by default. If you want to remove Extensions from your RHDH instance, you can disable the relevant plugins.

Procedure

1. To disable the the Extensions feature plugins, edit your **dynamic-plugins.yaml** with the following content.

   dynamic-plugins.yaml fragment

```
plugins:
  - package: ./dynamic-plugins/dist/red-hat-developer-hub-backstage-plugin-marketplace
    disabled: true
  - package: ./dynamic-plugins/dist/red-hat-developer-hub-backstage-plugin-catalog-backend-module-marketplace-dynamic
    disabled: true
  - package: ./dynamic-plugins/dist/red-hat-developer-hub-backstage-plugin-marketplace-backend-dynamic
    disabled: true
```

NOTE

If you disable the Extensions feature plugins, the **Catalog** and **Installed** tabs will also be removed. You can still view installed plugins by clicking on **Administration** > **Extensions**.

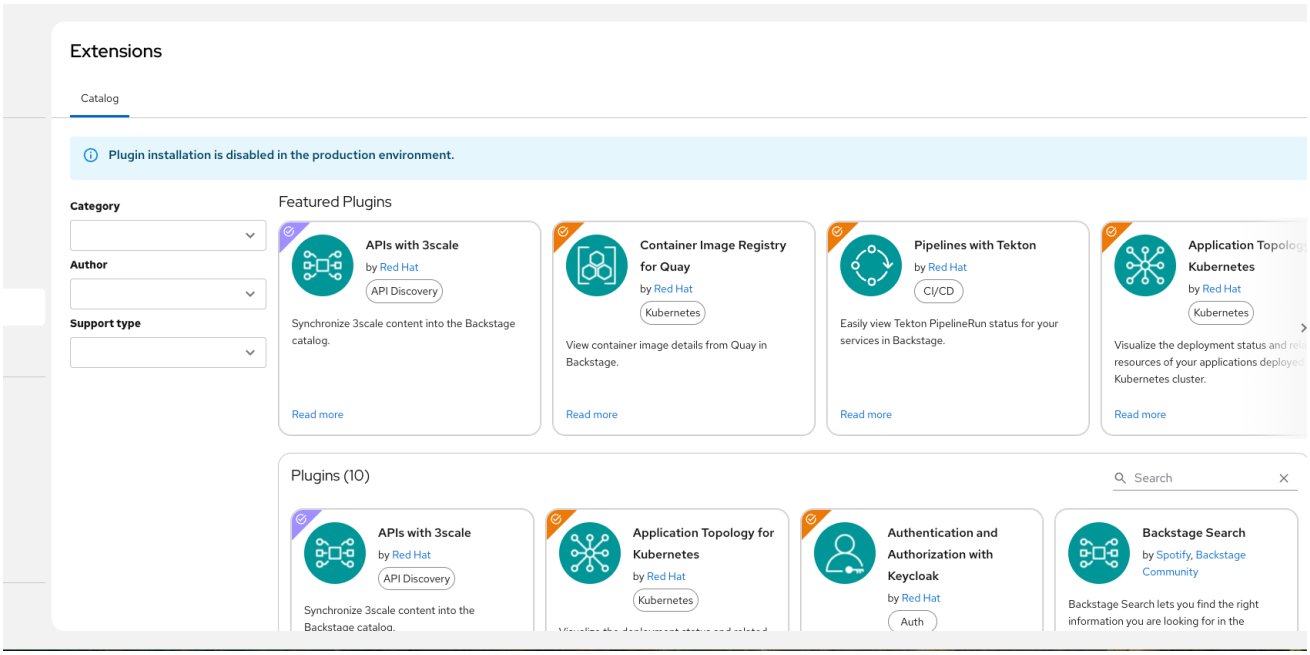
5.5. MANAGING PLUGINS BY USING EXTENSIONS

You can install and configure plugins by using **Extensions**.

WARNING

Installation and configuration of plugins by using Extensions will only work in development environments. This feature is not supported in production environments.

In a production environment, users will be notified that plugin installation is not permitted.



In a development environment:

- Administrators can install a plugin using the default configuration preloaded in the editor, or modify the configuration before installing. Upon successful installation, users will be notified that a backend restart is required to complete the installation process.
- When a plugin is installed, administrators can access the **Actions** drop-down in the side panel of the plugin. Available actions include:
  - Editing the configuration used during installation
  - Disabling or enabling the plugin
  - After performing any of these actions, users will be notified that a backend restart is required for the changes to take effect.

5.5.1. Configuring RBAC to manage Extensions

You can add Extensions permissions by creating or updating and existing RBAC role. For more information about using RBAC to manage role-based controls, see Managing role-based access controls (RBAC) using the Red Hat Developer Hub Web UI.

5.5.1.1. Creating a role in the Developer Hub UI to manage Extensions

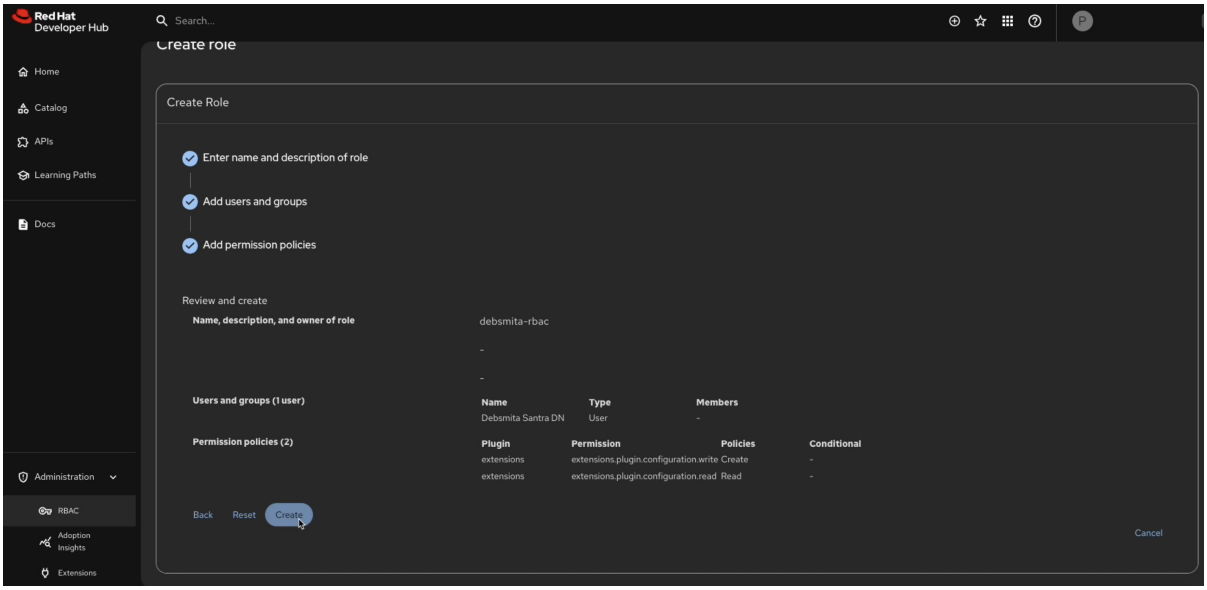
Prerequisites

- You have enabled RBAC, have a policy administrator role in Developer Hub, and have added plugins with permission.

Procedure

1. Go to **Administration** at the bottom of the sidebar in the Developer Hub.
   The **RBAC** tab appears, displaying all the created roles in the Developer Hub.
2. Click **Create** to create a role.

3. Enter the user name and description (optional) of role in the given fields and click **Next**.
4. In **Add users and groups**, select the user name, and click **Next**.
5. In **Add permission policies**, select **Extensions** from the plugins dropdown.
6. Expand **Extensions**, select both the **Create** and **Read** permissions for the Extensions plugin and click **Next**.
7. Click **Create** to create the role.



Verification

After you refresh the RHDH application, when you select a plugin, the **Actions** drop-down is active. When you click the **Actions** drop-down, you can edit the the plugin configuration, and enable or disable the plugin.

5.5.2. Configuring RHDH to install plugins by using Extensions

When you install a plugin using Extensions UI, the configuration that you use is saved to a **dynamic-plugins.extensions.yaml** file within the **dynamic-plugins-root** persistent volume. This ensures the configuration is available when you restart the application, allowing you to edit or re-enable the plugin.

You must create a persistent volume claim (PVC) to ensure that the cache persists when you restart the RHDH application. For more information about using the dynamic plugins cache, see Using the dynamic plugins cache.

Prerequisites

- You have created a persistent volume claim (PVC) for the dynamic plugins cache with the name **dynamic-plugins-root**.
- You have installed Red Hat Developer Hub using the Helm chart or the Operator.
- You have installed the OpenShift CLI (**oc**).

Procedure

1. Create the extensions configuration file and save it as **dynamic-plugins.extensions.yaml**. For example:

```
includes:
  - dynamic-plugins.default.yaml

plugins:
  - package: ./dynamic-plugins/dist/red-hat-developer-hub-backstage-plugin-marketplace
    disabled: false
    pluginConfig:
      dynamicPlugins:
        frontend:
          red-hat-developer-hub.backstage-plugin-marketplace:
            translationResources:
              - importName: marketplaceTranslations
                ref: marketplaceTranslationRef
                module: Alpha
            appIcons:
              - name: pluginsIcon
                importName: PluginsIcon
            dynamicRoutes:
              - path: /extensions
                importName: DynamicMarketplacePluginRouter
                menuItem:
                  icon: pluginsIcon
                  text: Extensions
                  textKey: menuItem.extensions
            menuItems:
              extensions:
                parent: default.admin
  - package: ./dynamic-plugins/dist/red-hat-developer-hub-backstage-plugin-marketplace-
backend-dynamic
    disabled: false
    pluginConfig:
      extensions:
        installation:
          enabled: true
          saveToSingleFile:
            file: /opt/app-root/src/dynamic-plugins-root/dynamic-plugins.extensions.yaml
```

where:

translationResources

Sets the extension point for localization.

2. Copy the file to your cluster by running the following commands:

```
oc get pods -n <your-namespace>

oc cp ./dynamic-plugins.extensions.yaml <your-namespace>/<pod-name>:/opt/app-
root/src/dynamic-plugins-root/dynamic-plugins.extensions.yaml
```

3. Update your RHDH application to use this file:

   a. For operator-based installations:

i. Update your Backstage CR to update the **NODE_ENV** environment variable to **development**, as follows:

```
apiVersion: rhdh.redhat.com/v1alpha3
kind: Backstage
metadata:
  name: developer-hub
  namespace: rhdh
spec:
  application:
    dynamicPluginsConfigMapName: dynamic-plugins-rhdh
    extraEnvs:
      envs:
        - name: NODE_ENV
          value: "development"
      secrets:
        - name: secrets-rhdh
    extraFiles:
      mountPath: /opt/app-root/src
    route:
      enabled: true
  database:
    enableLocalDb: true
```

ii. Update your **dynamic-plugins-rhdh** config map to include your extensions configuration file, as follows:

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: dynamic-plugins-rhdh
  namespace: rhdh
data:
 dynamic-plugins.yaml: |
   includes:
     - dynamic-plugins.default.yaml
     - /dynamic-plugins-root/dynamic-plugins.extensions.yaml
   plugins: []
```

b. For Helm chart installations:

i. Upgrade the Helm release to include your extensions configuration file and update the **NODE_ENV** environment variable to **development**:

```
global:
  auth:
    backend:
      enabled: true
  clusterRouterBase: apps.<clusterName>.com
  dynamic:
    includes:
      - dynamic-plugins.default.yaml
      - /dynamic-plugins-root/dynamic-plugins.extensions.yaml
upstream:
  backstage:
```

```
    extraEnvVars:
      - name: NODE_ENV
        value: development
```

ii. Click **Upgrade**

Verification

Enable a plugin by using the Extensions UI, restart your RHDH application and refresh the UI to confirm that the plugin is enabled.

5.5.3. Configuring RHDH Local to install plugins by using Extensions

You can use RHDH Local to test installing plugins by using Extensions.

Prerequisites

- You have installed RHDH Local. For more information about setting up RHDH Local, see Test locally with Red Hat Developer Hub.

Procedure

1. Update your **dynamic-plugins.override.yaml** file:

```
includes:
  - dynamic-plugins.default.yaml

plugins:
  - package: ./dynamic-plugins/dist/red-hat-developer-hub-backstage-plugin-marketplace
    disabled: false
    pluginConfig:
      dynamicPlugins:
        frontend:
          red-hat-developer-hub.backstage-plugin-marketplace:
            translationResources:
              - importName: marketplaceTranslations
                ref: marketplaceTranslationRef
                module: Alpha
            appIcons:
              - name: pluginsIcon
                importName: PluginsIcon
            dynamicRoutes:
              - path: /extensions
                importName: DynamicMarketplacePluginRouter
                menuItem:
                  icon: pluginsIcon
                  text: Extensions
                  textKey: menuItem.extensions
            menuItems:
              extensions:
                parent: default.admin
  - package: ./dynamic-plugins/dist/red-hat-developer-hub-backstage-plugin-marketplace-
backend-dynamic
    disabled: false
    pluginConfig:
      extensions:
```

```
      installation:
        enabled: true
        saveToSingleFile:
          file: /opt/app-root/src/configs/dynamic-plugins/dynamic-plugins.override.yaml
```

where:

translationResources

Sets the extension point for localization.

2. Update your **compose.yaml** file:

```
rhdh:
  container_name: rhdh
  environment:
    NODE_OPTIONS: "--inspect=0.0.0.0:9229"
    NODE_ENV: "development"
```

Verification

Enable a plugin by using the Extensions UI, restart your RHDH application and refresh the UI to confirm that the plugin is enabled.

5.5.4. Installing plugins by using Extensions

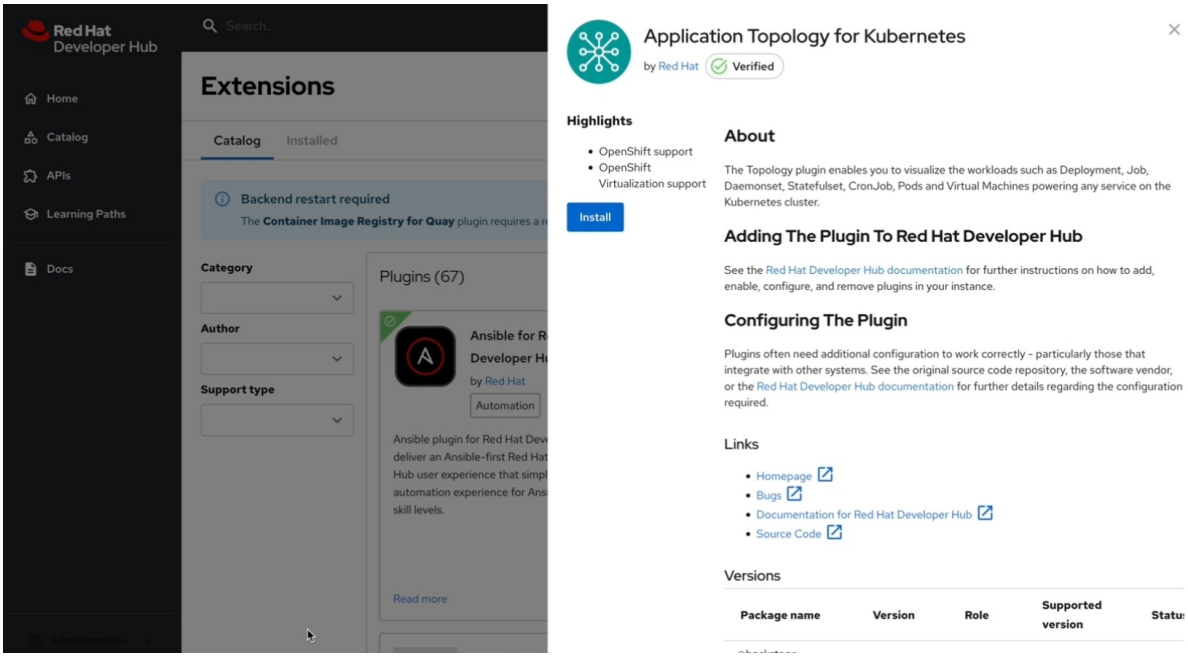
You can install and configure plugins by using Extensions.

Prerequisites

- You have configured RHDH to allow plugins installation from **Extensions**.
- You have configured RBAC to allow the current user to manage plugin configuration.

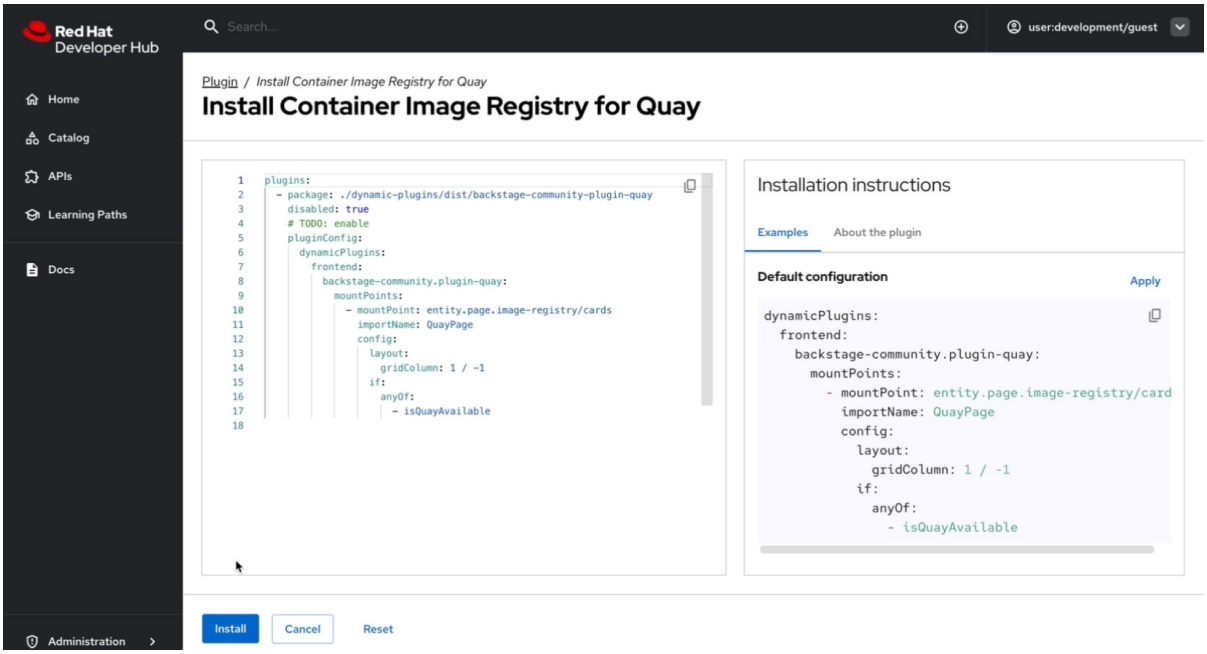
Procedure

1. Navigate to **Extensions**.
2. Select a plugin to install.
3. Click the **Install** button.



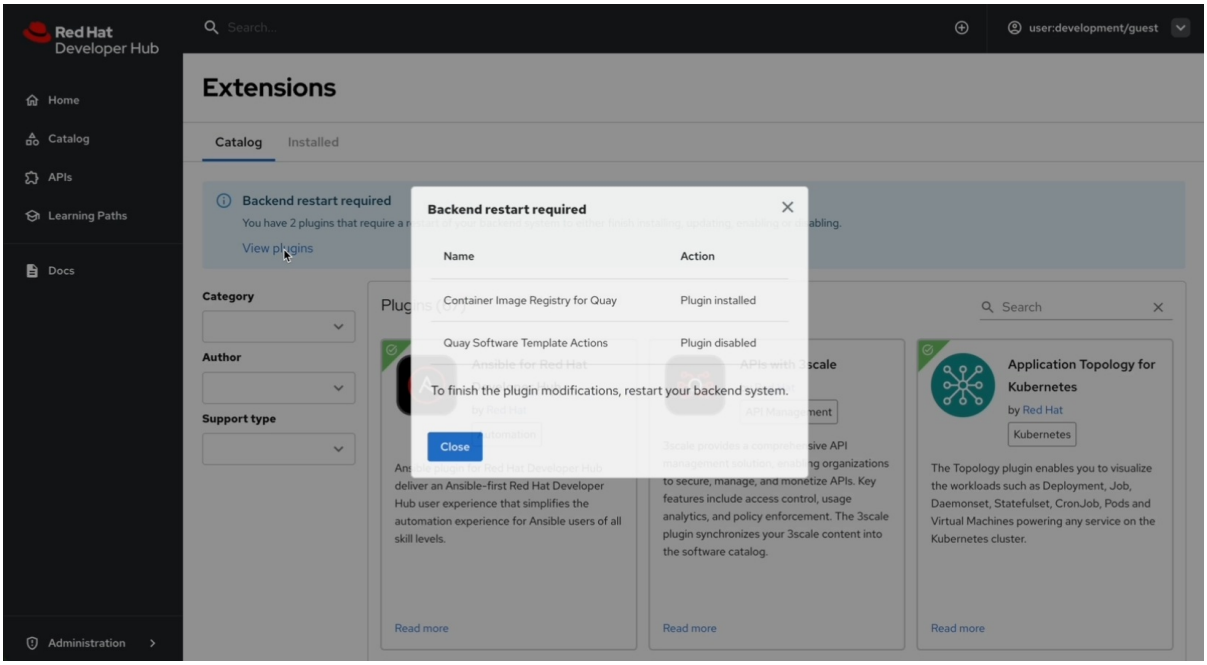
The code editor is displayed which displays the plugin default configuration.

4. Update the plugin configuration, if necessary.



5. Click **Install**

6. To view the plugins that require a restart, click **View plugins** in the alert message.



7. Restart your RHDH application.

Verification

1. After you restart your RHDH application, navigate to **Extensions**.
2. Select the plugin that you have installed.
3. The **Actions** button is displayed.

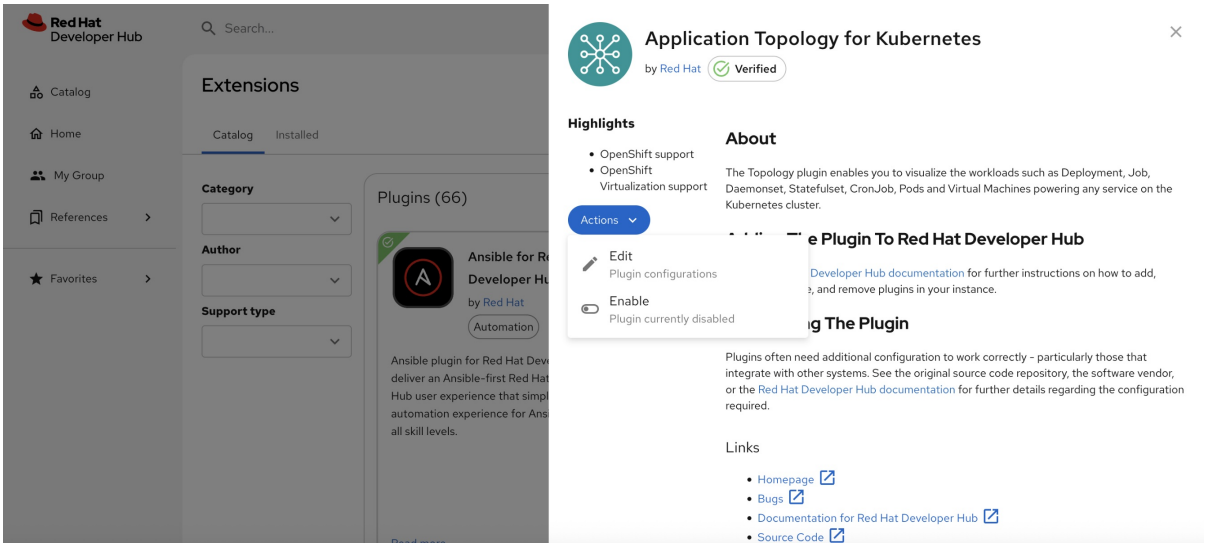
5.5.5. Enabling and disabling plugins by using Extensions

Prerequisites

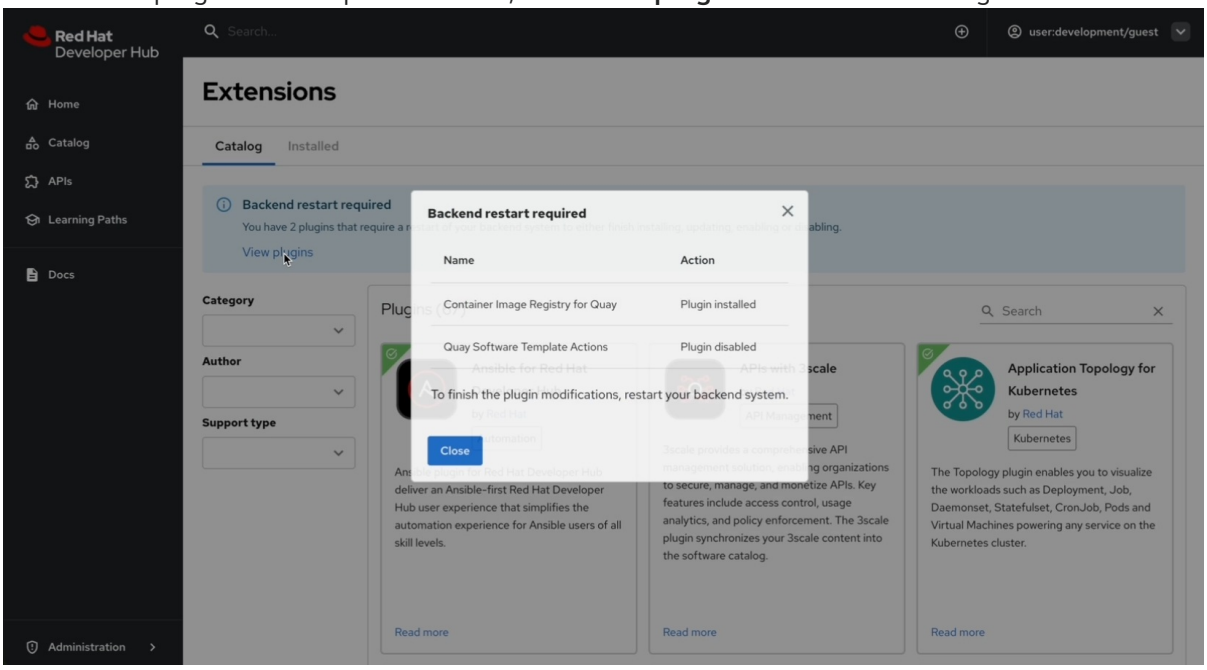
- You have configured RHDH to allow plugins installation from **Extensions**.
- You have configured RBAC to allow the current user to access to manage plugin configuration.

Procedure

1. Navigate to **Extensions**.
2. Select a plugin to enable or disable.
3. Click on the Enable/Disable slider.



4. To view the plugins that require a restart, click **View plugins** in the alert message.



5. Restart your RHDH application.

Verification

1. After you restart your RHDH application, navigate to **Extensions**.
2. Select the plugin that you have installed.
3. The **Enable/Disable** slider is updated.

CHAPTER 6. TROUBLESHOOTING PLUGINS

6.1. RHDH POD FAILS TO START AFTER ENABLING A PLUGIN

Procedure

1. Inspect your RHDH pod logs to identify if the plugin requires specific environment variables or additional configuration, for example:

```
Plugin '<PLUGIN_NAME>' threw an error during startup, waiting for X other plugins to finish before shutting down the process. Plugin '<PLUGIN_NAME>' startup failed; caused by Error: Missing required config value at '<concretePluginRequiredVariable.name>' in 'app-config.local.yaml' type="initialization"
```

2. Verify the required configuration by inspecting **dynamic-plugins.default.yaml** file that lists the required environment variables for each plugin. The variables for each plugin are in the format of **${PLUGIN_VARIABLE_NAME}**

3. If any required environment variables are missing, set the environment variables by using a **secret**. For example:

```
kind: Secret
apiVersion: v1
metadata:
  name: rhdh-secrets
  labels:
    backstage.io/kubernetes-id: developer-hub
data:
  PLUGIN_VARIABLE_NAME: 'dummy-value'
type: Opaque
```

4. Mount the secret:

   a. If RHDH is deployed by using the Operator, update your Backstage CR, as follows:

```
spec:
  application:
    extraEnvs:
      secrets:
        - name: rhdh-secrets
```

   b. If RHDH is deployed by using the Helm chart, in the **upstream.backstage** key in your Helm chart values, enter the name of the Developer Hub **rhdh-secrets** secret as the value for the **extraEnvVarsSecrets** field. For example:

```
upstream:
  backstage:
    extraEnvVarsSecrets:
      - rhdh-secrets
```

CHAPTER 7. FRONT-END PLUGIN WIRING

You can configure front-end plugins to customize icons, integrate components at mount points, and provide or replace utility APIs.

7.1. FRONT-END PLUGIN WIRING

Front-end plugin wiring integrates dynamic front-end plugin components, such as new pages, UI extensions, icons, and APIs, into Red Hat Developer Hub.

Because the dynamic plugins load at runtime, the core application must discover and connect the exported assets of the plugin to the appropriate user interface systems and locations.

7.1.1. Understand why front-end wiring is required

Because dynamic front-end plugins load their code at runtime, the Developer Hub application requires explicit instructions to integrate the plugin components in the user interface (UI).

Front-end wiring provides the metadata and instructions necessary to bridge this gap, informing the applications on how to:

- Add new pages and routes to the main application . (using **dynamicRoutes**)
- Inject custom components into existing UI pages . (using **mountPoints**)
- Configure application-wide customizations, such as new icons or themes . (using **appIcons**)
- Add new menu items . (using **menuItems**)
- Bind to existing plugins . (using **routeBindings**)

The wiring configuration, typically located in **app-config.yaml** or **dynamic-plugins-config.yaml**, gives the application the necessary metadata (including the component names, paths, and integration points) to render and use the plugin features.

7.1.2. Consequences of skipping front-end wiring

If you skip front-end wiring, the plugin is discovered but not loaded into the application front-end. As a result, the plugin features do not appear or function. This behavior is expected because while back-end plugins are discovered and loaded automatically, the core application loads front-end plugins only based on the list defined in the **dynamicPlugins.frontend** configuration.

You can expect the following behavior when you skip front-end wiring:

Disabled functionality

The Backstage application cannot integrate or use the plugin exports.

Invisible components

New pages, sidebar links, or custom cards do not render in the application UI.

Unregistered APIs

Custom utility APIs or API overrides provided by the plugin are not registered in the application API system, which can cause plugins or components to fail.

Unused assets

Icons, translations, and themes are not registered or available for use.

TIP

If a plugin is not visible even with front-end wiring, the plugin is likely misconfigured. Troubleshoot the issue by checking the **Console** tab in the *Developer Tools* of your browser for specific error messages or warnings.

7.1.3. Dynamic front-end plugins for application use

A dynamic front-end plugin requires front-end wiring when it exports a feature for integration into the main Backstage application UI. Wiring is required for the following scenarios:

Scenario	Wiring configuration	Description
Extending entity tabs	**entityTabs**	Add or customize a tab on the **Catalog entity** view.
Binding routes	**routeBindings**	Link a route in one plugin to an external route defined by another plugin.
Integrating custom APIs	**apiFactories**	Provide a custom utility API implementation or override an existing one.
Enabling new pages/Routes	**dynamicRoutes**	Add a new page and route to the application (for example, /**my-plugin**).
Extending existing pages/UI	**mountPoints**	Inject custom widgets, cards, listeners, or providers into existing pages (for example, the **Catalog entity** page).
Customizing sidebar navigation	**dynamicRoutes.menuItem**, **menuItems**	Add a new entry to the main sidebar or customize its order and nesting.
Adding icons/Theming	**appIcons**, **themes**	Add custom icons to the application catalog or define a new Backstage theme.
Scaffolder/TechDocs extensions	**scaffolderFieldExtensions**, **techdocsAddons**	Expose custom field extensions for the Scaffolder or new Addons for TechDocs.
Translation resources	**translationResources**	Provide new translation files or override default plugin translations.

Example of Front-end wiring workflow

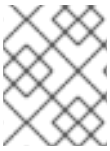
Front-end wiring is configured in the **app-config.yaml** or a dedicated **dynamic-plugins-config.yaml** file. The dynamic plugin exposes components, routes, or APIs. For example, a plugin component (**FooPluginPage**) is exported from a module (for example, **PluginRoot**).

The application administrator defines the wiring in the configuration file, using the plugin package name to register the exports, such as adding a new page with a sidebar link.

```
# dynamic-plugins-config.yaml
plugins:
  - plugin: <plugin_path_or_url>
    disabled: false
    pluginConfig:
      dynamicPlugins:
        frontend:
          my-plugin-package-name: # The plugin's unique package name
            dynamicRoutes: # Wiring for a new page/route
              - path: /my-new-page # The desired URL path
                importName: FooPluginPage # The exported component name
                menuItem: # Wiring for the sidebar entry
                  icon: favorite # A registered icon name
                  text: My Custom Page
```

When the application loads, it performs the following steps:

1. It parses the **dynamic-plugins-config.yaml**.
2. It uses the **<plugin_path_or_url>** to download the plugin bundle using the dynamic loading mechanism.
3. If the package exports the plugin object, the application adds it to the list provided to the Backstage **createApp** API, registering the plugin properly with the front-end application.
4. It uses the configuration block (**dynamicRoutes**, **menuItem**) to:
   - Add an entry to the internal router mapping /**my-new-page** to the **FooPluginPage** component.
   - Construct and render a new sidebar item labeled *My Custom Page*, pointing to the /**my-new-page** route.



NOTE

If the configuration is missing, steps 1 and 2 might still occur, but the final registration in step 3 and the wiring/rendering in step 4 are skipped, and no UI changes occur.

7.2. EXTENDING INTERNAL ICON CATALOG

You can use the internal catalog to fetch icons for configured routes with sidebar navigation menu entry.

Procedure

- Add a custom icon to the internal icon catalog for use in the menu item of a plugin by using the **appIcons** configuration as shown in the following example:

```
# dynamic-plugins-config.yaml
plugins:
  - plugin: <plugin_path_or_url>
    disabled: false
```

```
  pluginConfig:
    dynamicPlugins:
      frontend:
        my-plugin:  # The plugin package name
          appIcons:
            - name: fooIcon # The icon catalog name
              # (Optional): The set of assets to access within the plugin. If not specified, the system uses the `PluginRoot` module.
              module: CustomModule
              # (Optional): The actual component name to be rendered as a standalone page. If not specified, the system uses the `default` export.
              importName: FooIcon
```

NOTE

The **package_name** key under **dynamicPlugins.frontend must match** the **scalprum.name** value in your plugin's **package.json**. This ensures your dynamic plugin loads correctly during runtime.

7.3. DEFINING DYNAMIC ROUTES FOR NEW PLUGIN PAGES

Procedure

1. Define each route by specifying a unique **path** and, if needed, an **importName** if it is different from the **default** export.

2. Expose additional routes in a dynamic plugin by configuring **dynamicRoutes** as shown in the following example:

```
# dynamic-plugins-config.yaml
plugins:
  - plugin: <plugin_path_or_url>
    disabled: false
    pluginConfig:
      dynamicPlugins:
        frontend:
          my-plugin: # The plugin package name
            dynamicRoutes:
              # The unique path in the application. The path cannot override existing routes except the `/` home route.
              - path: /my-plugin
                # (Optional): The set of assets to access within the plugin. If not specified, the system uses the `PluginRoot` module.
                module: CustomModule
                # (Optional): The component name as a standalone page. If not specified, the system uses the `default` export.
                importName: FooPluginPage
                # Allows you to extend the main sidebar navigation and point to a new route.
                menuItem:
                  icon: fooIcon
                  text: Foo Plugin Page
                  enabled: false
                config: # (Optional): Passes `props` to a custom sidebar item
                  props: ...
```

The **menuItem** accepts the following properties:

- **text**: The label shown to the user.
- **icon**: The Backstage system icon name.
- **enabled**: Optional: Allows the user to remove a menuItem from the sidebar when it is set to false.
- **importName**: Specifies the optional name of an exported **SidebarItem** component. To configure a custom **SidebarItem** to enhance experiences such as notification badges, ensure the component accepts the following properties:

```
export type MySidebarItemProps = {
  to: string; // supplied by the sidebar during rendering, this will be the path configured for the
dynamicRoute
};
```

Example specifying a custom SidebarItem component:

```
# dynamic-plugins-config.yaml
plugins:
  - plugin: <plugin_path_or_url>
    disabled: false
    pluginConfig:
      dynamicPlugins:
        frontend:
          my-dynamic-plugin-package-name:
            dynamicRoutes:
              - importName: CustomPage
                menuItem:
                  config:
                    props:
                      text: Click Me!
                  importName: SimpleSidebarItem
                path: /custom_page
```

7.4. CUSTOMIZING MENU ITEMS IN THE SIDEBAR NAVIGATION

You can customize the order and parent-child relationships of plugin menu items in the main sidebar navigation using the menu items configuration as shown in the following example:

```
# dynamic-plugins-config.yaml
plugins:
  - plugin: <plugin_path_or_url>
    disabled: false
    pluginConfig:
      dynamicPlugins:
        frontend:
          my-plugin: # The plugin package name
            menuItems:
              # The unique name in the main sidebar navigation (for example, either a standalone menu
item or a parent menu item)
              <menu_item_name>:
```

```
        # (Optional): The icon for the menu item, which refers to a Backstage system icon
        icon: fooIcon
        # (Optional): The display title of the menu item
        title: Foo Plugin Page
        # (Optional): The order in which menu items appear. The default priority is `0`.
        priority: 10
        # (Optional): Defines the parent menu item to nest the current item under
        parent: favorites
        # (Optional): Allows you to remove a `menuItem` from the sidebar when it is set to `false`
        enabled: false
```

Handling Complex Paths:

- For simple paths like **path: /my-plugin**, the **menu_item_name** should be **my-plugin**.
- For complex paths like **/metrics/users/info**, the **menu_item_name** should represent the full path in dot notation (for example **metrics.users.info**).
- Ignore trailing and leading slashes in paths as follows:
  - For **path: /docs**, the **menu_item_name** is **docs**.
  - For **path: /metrics/users**, the **menu_item_name** is **metrics.users**.

NOTE

Red Hat Developer Hub supports up to 3 levels of nested menu items.

7.5. BINDING TO EXISTING PLUGINS

You can bind to existing plugins and their routes, and declare new targets sourced from dynamic plugins as shown in the following **routeBindings** configuration:

```
# dynamic-plugins-config.yaml
plugins:
  - plugin: <plugin_path_or_url>
    disabled: false
    pluginConfig:
      dynamicPlugins:
        frontend:
          my-plugin: # The plugin package name
            routeBindings:
              targets: # A new bind target
                # (Optional): Defaults to importName. Explicit name of the plugin that exposes the bind
target.
                - name: barPlugin
                  # (Required): Explicit import name that reference a BackstagePlugin<{}> implementation.
                  importName: barPlugin
                  # # (Optional): Same as key in `scalprum.exposedModules` key in the `package.json` file
of the plugin.
                  module: CustomModule
              bindings:
                - bindTarget: "barPlugin.externalRoutes" # (Required): One of the supported or imported
```

```
bind targets
        bindMap: #  A required map of route bindings similar to `bind` function options
          headerLink: "fooPlugin.routes.root"
```

To configure **routeBindings**, complete the following steps:

1. Define new targets using **routeBindings.targets**. Set the required **importName** to a **BackstagePlugin<{}>** implementation.
2. Declare route bindings using the **routeBindings.bindings** field by setting **bindTarget** to the name of the target to bind to. This is a dynamic or static target, such as:
   - **catalogPlugin.externalRoutes**
   - **catalogImportPlugin.externalRoutes**
   - **techdocsPlugin.externalRoutes**
   - **scaffolderPlugin.externalRoutes**
     You can extend existing pages with additional content using mount points, which are predefined identifiers available throughout the application.

7.6. USING MOUNT POINTS

Mount points are defined identifiers available across Red Hat Developer Hub. You can use these points to extend existing pages with additional content.

7.6.1. Customizing entity page

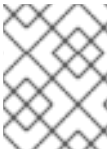
You can extend catalog components and additional views.

The available mount points include the following:

Table 7.1. Input parameters

Mount point	Description	Visible even when no plugins are enabled
admin.page.plugins	Administration plugins page	NO
admin.page.rbac	Administration RBAC page	NO
entity.context.menu	Catalog entity context menu	YES for all entities
entity.page.overview	Catalog entity overview page	YES for all entities
entity.page.topology	Catalog entity **Topology** tab	NO

Mount point	Description	Visible even when no plugins are enabled
entity.page.issues	Catalog entity **Issues** tab	NO
entity.page.pull-requests	Catalog entity **Pull Requests** tab	NO
entity.page.ci	Catalog entity **CI** tab	NO
entity.page.cd	Catalog entity **CD** tab	NO
entity.page.kubernetes	Catalog entity **Kubernetes** tab	NO
entity.page.image-registry	Catalog entity **Image Registry** tab	NO
entity.page.monitoring	Catalog entity **Monitoring** tab	NO
entity.page.lighthouse	Catalog entity **Lighthouse** tab	NO
entity.page.api	Catalog entity **API** tab	YES for entity of **kind: Component** and **spec.type: 'service'**
entity.page.dependencies	Catalog entity **Dependencies** tab	YES for entity of **kind: Component**
entity.page.docs	Catalog entity **Documentation** tab	YES for entity that satisfies **isTechDocsAvailable**
entity.page.definition	Catalog entity **Definitions** tab	YES for entity of **kind: Api**
entity.page.diagram	Catalog entity **Diagram** tab	YES for entity of **kind: System**
search.page.types	Search result type	YES, default catalog search type is available
search.page.filters	Search filters	YES, default catalog kind and lifecycle filters are visible
search.page.results	Search results content	YES, default catalog search is present



NOTE

Mount points within catalog such as **entity.page.** are rendered as tabs and become visible only if at least one plugin contributes to them, or if they can render static content.

Each **entity.page.** mount point contains the following variations:

- /**context** type that serves to create React contexts
- /**cards** type for regular React components

The following is an example of the overall configuration structure of a mount point:

```
# dynamic-plugins-config.yaml
plugins:
  - plugin: <plugin_path_or_url>
    disabled: false
    pluginConfig:
      dynamicPlugins:
        frontend:
          my-plugin: # The plugin package name
            mountPoints: # (Optional): Uses existing mount points
              - mountPoint: <mountPointName>/[cards|context]
                module: CustomModule
                importName: FooPluginPage
                config: # (Optional): Allows you to pass additional configuration to the component
                  layout: {} # Used only in `/cards` type which renders visible content
                  # Use only in `/cards` type which renders visible content. `if` is passed to
`<EntitySwitch.Case if={<here>}`.
                  if:
                    allOf|anyOf|oneOf:
                      - isMyPluginAvailable
                      - isKind: component
                      - isType: service
                      - hasAnnotation: annotationKey
                  props: {} # Useful when you are passing additional data to the component
```

The available conditions include:

- **allOf**: All conditions must be met
- **anyOf**: At least one condition must be met
- **oneOf**: Only one condition must be met

Conditions are:

- **isKind**: Accepts a string or a list of string with entity kinds. For example **isKind: component** renders the component only for entity of **kind: Component**.
- **isType**: Accepts a string or a list of string with entity types. For example **isType: service** renders the component only for entities of **spec.type: 'service'**.

- **hasAnnotation**: Accepts a string or a list of string with annotation keys. For example **hasAnnotation: my-annotation** renders the component only for entities that have defined **metadata.annotations['my-annotation']**.

- Condition imported from the **module** of the plugin: Must be function name exported from the same **module** within the plugin. For example **isMyPluginAvailable** renders the component only if **isMyPluginAvailable** function returns **true**. The function must have the following signature: **(e: Entity)** ⇒ **boolean**.

The entity page supports adding more items to the context menu at the top right of the page. The exported component is a form of dialog wrapper component that accepts an **open** boolean property and an **onClose** event handler property as shown in the following example:

```
export type SimpleDialogProps = {
  open: boolean;
  onClose: () => void;
};
```

You can configure the context menu entry using the props configuration entry for the mount point. The **title** and **icon** properties sets the text and icon of the menu item. You can use any system icon or icon added through a dynamic plugin. The following is an example configuration:

```
# dynamic-plugins-config.yaml
plugins:
  - plugin: <plugin_path_or_url>
    disabled: false
    pluginConfig:
      dynamicPlugins:
        frontend:
          my-dynamic-plugin-package:
            appIcons:
              - name: dialogIcon
                importName: DialogIcon
            mountPoints:
              - mountPoint: entity.context.menu
                importName: SimpleDialog
                config:
                  props:
                    title: Open Simple Dialog
                    icon: dialogIcon
```

7.6.2. Adding application header

You can customize global headers by specifying configurations in the **app-config.yaml** file as shown in the following example:

```
# app-config.yaml
dynamicPlugins:
  frontend:
    my-plugin:  # The plugin package name
      mountPoints:
        - mountPoint: application/header # Adds the header as a global header
          importName: <header_component> # Specifies the component exported by the global header
```

```
plugin
        config:
          position: above-main-content # Supported values: (`above-main-content`| above-sidebar`)
```

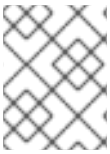
NOTE

To configure multiple global headers at different positions, add entries to the **mountPoints** field.

7.6.3. Adding application listeners

You can add application listeners using the **application/listener** mount point as shown in the following example:

```
# app-config.yaml
dynamicPlugins:
  frontend:
    my-plugin: # The plugin package name
      mountPoints:
        - mountPoint: application/listener
          importName: <exported listener component>
```



NOTE

You can configure multiple application listeners by adding entries to the **mountPoints** field.

7.6.4. Adding application providers

You can add application providers using the **application/provider** mount point. You can use a mount point to configure a context provider as shown in the following example:

```
# app-config.yaml
dynamicPlugins:
  frontend:
    my-plugin: # The plugin package name
      dynamicRoutes:
        - path: /<route>
          importName: Component # The component to load on the route
      mountPoints:
        - mountPoint: application/provider
          importName: <exported provider component>
```

NOTE

1. You can configure multiple application providers by adding entries to the **mountPoints** field.

2. The **package_name** key under **dynamicPlugins.frontend must match** the **scalprum.name** value in the **package.json** file of your plugin. This ensures your dynamic plugin loads correctly at runtime.

7.7. CUSTOMIZING AND EXTENDING ENTITY TABS

You can customize and extend the set of tabs using the **entityTabs** configuration as follows:

```
# dynamic-plugins-config.yaml
plugins:
  - plugin: <plugin_path_or_url>
    disabled: false
    pluginConfig:
      dynamicPlugins:
        frontend:
          my-plugin: # The plugin package name
            entityTabs:
              # Specify a new tab
              - path: /new-path
                title: My New Tab
                mountPoint: entity.page.my-new-tab
              # Change an existing tab's title or mount point
              - path: /
                title: General
                mountPoint: entity.page.overview
                # Specify the sub-path route in the catalog where this tab is available
              - path: "/pr"
                title: "Changed Pull/Merge Requests" # Specify the title you want to display
                priority: 1
                # The base mount point name available on the tab. This name expands to create two
mount points per tab, with` /context` and with `/cards`
                mountPoint: "entity.page.pull-requests"
              - path: "/"
                title: "Changed Overview"
                mountPoint: "entity.page.overview"
                # Specify the order of tabs. The tabs with higher priority values appear first
                priority: -6
```

You can configure dynamic front-end plugins to target the mount points exposed by the entityTabs configuration. The following are the default catalog entity routes in the default order:

Table 7.2. Input parameters

Route	Title	Mount Point	Entity Kind
/	Overview	**entity.page.overview**	Any
/topology	Topology	**entity.page.topology**	Any
/issues	Issues	**entity.page.issues**	Any
/pr	CPull/Merge Requests	**entity.page.pull-requests**	Any
/ci	CI	**entity.page.ci`**	VAny

Route	Title	Mount Point	Entity Kind
/cd	CD	**entity.page.cd**	Any
/kubernetes	Kubernetes	**entity.page.kubernetes**	Any
/image-registry	Image Registry	**entity.page.image-registry**	Any
/monitoring	Monitoring	**entity.page.monitoring**	Any
/lighthouse	Lighthouse	**entity.page.lighthouse**	Any
/api	Api	**entity.page.api**	kind: Service or kind: Component
/dependencies	Dependencies	**entity.page.dependencies**	kind: Component
/docs	Docs	**entity.page.docs**	Any
/definition	Definition	**entity.page.definition**	kind: API
/system	Diagram	**entity.page.diagram**	kind: System

NOTE

Mount points within Catalog such as `*entity.page.**` are rendered as tabs and become visible only if at least one plugin contributes to them, or if they can render static content.

7.8. USING A CUSTOM SIGNINPAGE COMPONENT

In Red Hat Developer Hub (RHDH), the **SignInPage** component manages the authentication flow of the application. This component connects one or more authentication providers to the sign-in process. By default, Developer Hub uses a static **SignInPage** that supports all built-in authentication providers.

When you configure a custom **SignInPage**:

- The system loads the specified **importName** component from your dynamic plugin.
- The component returns a configured **SignInPage** that connects the desired authentication provider factories.
- Only one **signInPage** is specified for the application at a time.

```
dynamicPlugins:
  frontend:
    my-plugin: # The plugin package name
      signInPage:
        importName: CustomSignInPage
```

NOTE

The **package_name** specified under **dynamicPlugins.frontend** must match the **scalprum.name** value in the **package.json** file of your plugin to ensure the dynamic plugin loads correctly at runtime.

The **module** field is optional and allows specifying which set of assets must be accessed within the dynamic plugin. By default, the system uses the **PluginRoot** module.

7.9. PROVIDING CUSTOM SCAFFOLDER FIELD EXTENSIONS

The Scaffolder component in Red Hat Developer Hub (RHDH) enables users to create software components using templates through a guided wizard. You can extend the functionality of the Scaffolder by providing custom form fields as dynamic plugins using the **scaffolderFieldExtensions** configuration.

Custom field extensions allow you to add specialized form fields that capture domain-specific data during the scaffolding process, such as environment selectors, input validations, or repository checks.

When you configure custom scaffolder field extensions:

- The dynamic plugin exposes the field extension component using **createScaffolderFieldExtension**.
- Each field extension requires a unique **importName** for registration.
- You register multiple field extensions by listing each in the configuration.

```
dynamicPlugins:
  frontend:
    my-plugin: # The plugin package name
      scaffolderFieldExtensions:
        - importName: MyNewFieldExtension # References the exported scaffolder field extension component from your plugin
```

NOTE

The **module** field is optional and specifies which set of assets to access within the plugin. By default, the system uses the **PluginRoot** module, consistent with the **scalprum.exposedModules** key in the **package.json** file of your package.

7.10. PROVIDING ADDITIONAL UTILITY APIS

If a dynamic plugin exports the plugin object returned by **createPlugin**, it is supplied to the **createApp** API. All API factories exported by the plugin are automatically registered and available in the front-end application.

You can add an entry to the **dynamicPlugins.frontend** configuration when a dynamic plugin contains only API factories as shown in the following example:

```
# app-config.yaml
dynamicPlugins:
  frontend:
    my-dynamic-plugin-package-with-api-factories: {}
```

However, when the dynamic plugin is not exporting the plugin object, explicitly configure each API factory that must be registered with the **createApp** API using the **apiFactories** configuration as shown in the following example:

```
# app-config.yaml
dynamicPlugins:
  frontend:
    my-plugin: # The plugin package name
      apiFactories:
        # (Optional): Specify the import name that references a `AnyApiFactory<{}>` implementation.
(Defaults to `default` export)
        - importName: BarApi
          # (Optional): An argument which specifies the assets you want to access within the plugin. If not
provided, the default module named `PluginRoot` is used
          module: CustomModule
```

The API factories initialized by the Developer Hub application shell are overridden by an API factory provided by a dynamic plugin by specifying the same API ref ID. A dynamic plugin can export **AnyApiFactory<{}>** to cater for some specific use case as shown in the following example:

```
export const customScmAuthApiFactory = createApiFactory({
  api: scmAuthApiRef,
  deps: { githubAuthApi: githubAuthApiRef },
  factory: ({ githubAuthApi }) =>
    ScmAuth.merge(
      ScmAuth.forGithub(githubAuthApi, { host: "github.someinstance.com" }),
      ScmAuth.forGithub(githubAuthApi, {
        host: "github.someotherinstance.com",
      }),
    ),
});
```

The corresponding configuration which overrides the default **ScmAuth** API factory that Developer Hub defaults to is as shown in the following example:

```
dynamicPlugins:
  frontend:
    my-plugin:  # The plugin package name
      apiFactories:
        - importName: customScmAuthApiFactory
```

7.11. ADDING CUSTOM AUTHENTICATION PROVIDER SETTINGS

You can install new authentication providers from a dynamic plugin that either adds additional configuration support for an existing provider or adds a new authentication provider. These providers are listed in the user settings section under the **Authentication Providers** tab.

You can use the **providerSettings** configuration to add entries for an authentication provider from a dynamic plugin, as shown in the following example:

```
dynamicPlugins:
  frontend:
    my-plugin: # The plugin package name
```

```
    providerSettings:
      # The title for the authentication provider shown above the user's profile image if available
      - title: My Custom Auth Provider
        # The description of the authentication provider
        description: Sign in using My Custom Auth Provider
        # The ID of the authentication provider as provided to the `createApiRef` API call.
        provider: core.auth.my-custom-auth-provider
```

NOTE

provider looks up the corresponding API factory for the authentication provider to connect the provider's Sign In/Sign Out button.

7.12. PROVIDING CUSTOM TECHDOCS ADDONS

If a plugin provides multiple addons, each **techdocsAddon** entry specifies a unique **importName** corresponding to the addon. Front-end plugins exposes the TechDocs addon component using the **techdocsAddons** configuration as shown in the following example:

```
dynamicPlugins:
  frontend:
    my-plugin: # The plugin package name
      techdocsAddons:
        - importName: ExampleAddon # The exported Addon component
          config:
            props: ... # (Optional): React props to pass to the addon
```

7.13. CUSTOMIZING RED HAT DEVELOPER HUB THEME

You can customize Developer Hub themes from a dynamic plugin with various configurations as shown in the following example:

```
import { lightTheme } from './lightTheme';
import { darkTheme } from './darkTheme';
import { UnifiedThemeProvider } from '@backstage/theme';
export const lightThemeProvider = ({ children }: { children: ReactNode }) => (
  <UnifiedThemeProvider theme={lightTheme} children={children} />
);
export const darkThemeProvider = ({ children }: { children: ReactNode }) => (
  <UnifiedThemeProvider theme={darkTheme} children={children} />
);
```

For more information about creating a custom theme, see creating a custom theme.

You can declare the theme using the **themes** configuration as shown in the following example:

```
dynamicPlugins:
  frontend:
    my-plugin: # The plugin package name
      themes:
        #  are `light` or `dark`. Using 'light' overrides the app-provided light theme
        - id: light
          title: Light
```

```
        variant: light
        icon: someIconReference
        importName: lightThemeProvider
        # The theme name displayed to the user on the *Settings* page. Using 'dark' overrides the app-provided dark theme
      - id: dark
        title: Dark
        variant: dark
        icon: someIconReference # A string reference to a system or app icon
        # The name of the exported theme provider function, the function signature should match `({ children }: { children: ReactNode }): React.JSX.Element`
        importName: darkThemeProvider
```

CHAPTER 8. UTILIZING PLUGIN INDICATORS AND SUPPORT TYPES IN THE RED HAT DEVELOPER HUB

Understanding plugin indicators and support types is crucial for effectively utilizing the Red Hat Developer Hub.

8.1. NAVIGATING THE PLUGIN MARKETPLACE AND FILTERING PLUGINS USING BADGES

The plugin marketplace includes new indicators that enhance transparency. These changes provide clearer information about the status of each plugin, support level, and other relevant details, allowing you to make informed decisions when you select plugins. By understanding these visual cues and descriptive texts, you can quickly identify the maturity, support, and origin of each plugin.

The marketplace employs the following badges and tags to provide quick visual cues:

- **Certified**
- **Generally Available** (**GA**)
- **Community Plugin**
- **Tech Preview**
- **Dev Preview**
- **Custom Plugin**

NOTE

Hovering your mouse cursor over any badge or tag (such as **GA** or **Certified**) on either the plugin list or details page displays a helper text explanation clarifying its meaning and implications. Hovering over tags such as **tech preview**, or **dev preview** will display a helper text explanation clarifying its meaning and indicating if it is in a technical preview phase or a development preview phase.

The plugin details pages include the following fields for additional context: **author**, **tags**, **category**,**publisher**, and **support provider**. The update also changes the **version** text to **Backstage compatibility version** to offer more precise compatibility information.

The following filter examples are available:

- To view only plugins that are **Generally Available** (**GA**), apply the **GA** filter.
- To find **Certified** plugins, apply the **Certified** in **Support type** filter.
- To explore plugins currently in **Tech Preview**, **Dev Preview**, or **Community Plugin** status, apply the corresponding filter.

Procedure

1. Navigate to the main plugin list page within the Red Hat Developer Hub.

2. On the plugin list page, use the **Support type** filter to refine your plugin listings based on the plugin support status. You can use this filter to quickly narrow down options to plugins that meet your specific support requirements.

3. Click the entry of a plugin list page to view its details, which provide an in-depth view of its characteristics.