Pratexo Feature DevTools

Table of Contents

Introduction

The Pratexo Feature DevTools is a collection of artifacts aimed at helping custom feature developers initialize their feature project and synchronizing their features with the Pratexo Design Studio (PDS). This document will guide custom feature developers through the necessary steps to setup their local environments.

Obtaining DevTools

The DevTools are available from the following public GitHub repository:

https://github.com/pratexo/feature-devtools

Using your favorite Git client, clone this repository to your local system and then checkout the latest branch.

Note that for the remainder of this document, the location on your local system where you cloned DevTools will be referred to as **<DEV_TOOLS_INST_DIR>**

Obtaining GCP Data

The PDS is a hosted SaaS in Google Cloud Platform. In order to properly setup your local system for feature development and synchronization with PDS, a few data items will be required. These include:

- A GCP project id
- A GCP service account application_default_credentials.json file
- The name of the GCP bucket that services as the sync inbox for custom features

All of this information can be obtained from your Pratexo support personal.

Feature Checklist

Before development of a custom feature begins, it is helpful to walk through a feature checklist to ensure that the necessary prerequisites are in place.

Pratexo Design Studio features fall into 2 broad camps: Kubernetes-based and Native. The following sections cover the prerequisites for each.

Kubernetes-based Feature

When developing a Kubernetes-based feature, the following prerequisites must be met:

- a containerized version of the application
- a corresponding Helm Chart

The container for the application is required to be pushed to the following Google Registry

gcr.io/pratexo-public-5b5a/applications

The Helm Chart for the application is required to be pushed to the following Google Registry

oci://europe-north1-docker.pkg.dev/pratexo-public-5b5a/helm-charts

Failure to do this will result in the inability to deploy an architecture from the PDS.

In order to push the container and helm chart, authentication to the registries is required. Authentication is covered in a later section.

Native Feature

To be documented in a later version

Feature Connectivity

Features typically need to support connections to/from other features and at times, external connectivity. The following table details the various types of connections:

Connectivity	Description
External Connectivity	Supports connectivity to the feature from outside the deployed architecture
Internal Inbound-Connectivity	Supports connectivity to the feature from inside the deployed architecture
Internal Outbound-Connectivity	Supports connectivity to another feature

For example, if an application supports a management UI and requires a connection to MongoDB, this would be examples of External Connectivity (the management UI) and Internal Outbound-Connectivity (the MongoDB connection).

Another example might be an application that supports receiving messages from other features into a queue and publishing aggregated messages to a another feature. This illustrates examples of Internal Inbound-Connectivity (the queue) and Internal Outbound-Connectivity (the aggregated messages).

When developing custom features, it is important to understand what the connectivity requirements are. This information will drive how the feature is initialized.

Checkpoint A

Before moving onto the next section, it is important to ensure that all necessary prerequisites have been met. The following table will aid in this determination

Prerequisite	Completed
Have you obtained a GCP project id	
Have you obtained the service account JSON cred file	
Have you obtained the GCP bucket name for feature sync	
Have you pushed your application container to gcr.io/pratexo-public-5b5a/applications	

Prerequisite	Completed
Have you pushed your Helm Chart to oci://europe-north1-docker.pkg.dev/pratexo-public-5b5a/helm-charts	
Have you determined your feature connectivity needs	

If you have answered **yes** to each of these prerequisites, you can continue onto the next section.

Setting up Local Environment

In order to develop custom PDS Features, your local development environment will require a few tools. The next section covers these requirements.

Install Docker

In order to synchronize your custom developed feature with the PDS, a running Pratexo Feature Adapter is required. This process will be executed as a Docker container, using Docker Compose to start the container. As such, your local system will require valid installations of Docker and Docker Compose. As there are several online installation guides available, it is recommended that, based on your local OS, you search for the appropriate installation guide and follow those steps.

TODO: do we need to be concerned about specific versions? For example, should we specify a minimum version level.

Install GCloud SDK

The PDS is hosted as a SaaS in Google's Cloud Platform (GCP). The PDS makes use of several GCP features natively. In order to properly synchronize a custom developed feature with PDS, the local feature development environment will require the GCloud SDK.

As with the previous section on Docker installation, there are many online resources that detail the steps necessary to install GCloud SDK on specific operating systems. Find a suitable guide for your OS and follow those steps.

Once you have installed GCloud SDK, you will need to enable interaction between GCloud SDK and Docker. In order to facilitate this, execute the following commands on your system:

```
gcloud auth login ①
gcloud auth configure-docker
```

① This command will initiate an OAuth flow where you will be required to grant access to GCP resources

Setting up GCloud SDK will create a ~/.config/gcloud directory on your system. At this point, copy the application_default_credentials.json to ~/.config/gcloud.

Create a Features Project Directory

You should create a directory that will hold all custom developed features. This step is particularly important, as it will be required info to properly start the Pratexo Feature Adapter, which is covered in the next step.

While the specific directory is not necessarily important, it is recommended that you avoid spaces in the directory name (usually only a concern on Windows systems). Examples of a suitable project directory include:

/home/svai/pratexocustomfeatures
/projects/dev/pratexo/custom-features

Take note of the directory name, as it will be required in the next two sections.

Checkpoint B

As with the previous checkpoint, the following table will help determine if the next set of prerequisites have been met.

Prerequisite	**Completed**
Have you installed Docker and verified installation	
Have you install GCloud SDK and verified installation	
Have you copied application_default_credentials.json to ~/.config/gcloud	
Have you created a directory for custom features	

If you have answered **yes** to each of these prerequisites, you can continue onto the next section.

Start Pratexo Feature Adapter

The docker-compose-fa.yml file that's included in the DevTools can be used to startup the Pratexo Feature Adapter (PFS) on your local environment. The PFA is a liaison process that serves as a bridge between your local feature files and the PDS. You will interact with the PFA via web calls (curl, wget, Postman, etc) to synchronize your local feature project with the PDS. The PFA will validate your custom feature and ensure that it can be properly integrated into the PDS.

In order to property start the PFA, you must create an environment file that contains settings that are specific to your local system and your GCP project. Using your favorite editor, create a file with the following information:

```
LOCAL_FEATURES_DIRECTORY=your_local_features_directory ①
ORG_NAME=your_organization_name
INBOX_BUCKET_NAME=your_inbox_bucket_name ②
```

① this should be the directory you created in the previous section

② should be provided by your Pratexo Support Team

After you have saved the env file, perform the following commands on your local system to start the PFA

```
cd <DEV_TOOLS_INST_DIR>
gcloud auth print-access-token | docker login -u oauth2accesstoken --password-stdin
https://gcr.io
gcloud auth configure-docker
docker-compose -f docker-compose-fa.yml --env-file <your-env-file> up
```

This should output similar to the following:

```
[+] Running 1/1
 ⠿ Container pratexo_adapter  Recreated
0.5s
Attaching to pratexo_adapter
pratexo_adapter  | 20230127-02:44:27 - INFO - Api starting...
pratexo_adapter  | 20230127-02:44:27 - DEBUG - Loaded config 'GCS_HYBRID' from ENV as:
[True]
pratexo_adapter  | 20230127-02:44:27 - INFO - Staring in hybrid mode
pratexo_adapter  | 20230127-02:44:27 - DEBUG - Loaded config
'FEATURE_ADAPTER_INBOX_BUCKET' from ENV as: [gman-fa-experiments]
pratexo_adapter  | 20230127-02:44:27 - DEBUG - Checking
/tmp/.config/gcloud/application_default_credentials.json for explicit credentials as
part of auth process...
pratexo_adapter  | 20230127-02:44:27 - DEBUG - Loaded config
'CORE_FEATURE_LIBRARY_PATH' from ENV as: [/features]
pratexo_adapter  | 20230127-02:44:27 - WARNING - Using GCS Mock
pratexo_adapter  | 20230127-02:44:27 - DEBUG - Mock root path is /features
pratexo_adapter  | 20230127-02:44:27 - DEBUG - Loaded config 'HOSTNAME' from ENV as:
[6aa1df493370]
pratexo_adapter  | 20230127-02:44:27 - DEBUG - Loaded config 'PORT' from ENV as:
[8080]
pratexo_adapter  | 20230127-02:44:27 - DEBUG - Loading config
pratexo_adapter  | 20230127-02:44:27 - DEBUG - Loaded config 'ORGANIZATION' from ENV
as: [com.biz]
pratexo_adapter  | 20230127-02:44:27 - DEBUG - inbox bucket is custom-feature-bucket
pratexo_adapter  | 20230127-02:44:27 - DEBUG - organization is com.biz
pratexo_adapter  | 20230127-02:44:27 - DEBUG - library location is /features
pratexo_adapter  | 20230127-02:44:27 - DEBUG - hostname is 6aa1df493370
pratexo_adapter  | 20230127-02:44:27 - DEBUG - port is 8080
pratexo_adapter  | 20230127-02:44:27 - INFO - Server started http://6aa1df493370:8080
```

If you desire to stop the PFA, issue the following command:

```
docker-compose -f docker-compose-fa.yml --env-file=<your-env-file> stop
```

Initialize a Feature Project

The Feature DevTools includes a script that can be used to initialize a feature project. The approach involves creating a features.properties file, where key information about your custom feature is specified. The initialization script will use this .properties file to generate the required directory structure and initialized feature files.

The Feature DevTools includes a template features properties file that you can use to define your feature properties. The following steps detail initializing your custom feature project

- Change directory to <DEV_TOOLS_INST_DIR>
- Copy <**DEV_TOOLS_INST_DIR**>/**feature.properties.template** file to **feature.properties**

At this point the **feature.properties** file contains this

```
feature_id: your_feature_id
feature_name: your_feature_name
feature_description: your_feature_description
feature_version: your_feature_version
feature_org: your_feature_organization
feature_kind: your_feature_kind
feature_install_type: your_feature_install_type
feature_helm_chart_url: your_feature_helm_chart_url
feature_category: your_feature_category
feature_internal_ingress: your_feature_has_internal_ingress
feature_external_ingress: your_feature_has_external_ingress
feature_connections: your_feature_has_connections
```

While most of these are self-explanatory, the following describes each of the properties and offers guidance as to proper values

Property	**Description**
feature_id	A short name for the feature. Should be lower case with no spaces.
feature_name	Descriptive name for the feature
feature_description	A description of the feature
feature_version	The version of the feature. Should follow major.minor.dot format (e.g. 1.0.1).
feature_org	The organization writing the feature. Should follow Internet domain format (e.g. com.pratexo)
feature_kind	The find of feature. Valid options are: **device_protocol** **ansible_script** **core_entity** **docker_compose** **helm_chart**
feature_install_type	How the feature will be installed. Valid options are: **container** **daemonSet** **native** **replicaSet**
feature_helm_chart_url	URL to the helm chart.
feature_category	Name that categorizes the feature. See Appendix A for list of valid categories.

Property	Description
feature_internal_ingress	Indication if feature supports internal connections. Valid options are: **yes** **no**.
feature_external_ingress	Indication if feature supports external connections. Valid options are: **yes** **no**.
feature_connections	Indication if feature requires connection to another feature. Valid options are: **yes** **no**.

Using your favorite editor, replace the placeholder values with the values appropriate to your feature.

You should specify **helm_chart** as the feature_kind and **replicaSet** as the feature_install_type. Other values are currently not supported.

Later version of the DevTools will include information on the other kinds and install types for features.

Once you have completed your changes, save the feature.properties file and execute the following command to initialize the feature project

```
./initialize-feature.sh -f feature.properties -d <features project directory>
```

Change directory to your feature directory
You should see the following directories and files

```
<feature_id>
    |
    | -- <feature_version>
          |
          | -- manifest.yaml
          |
          | -- replicaSet
                |
                | -- helm
                |     |
                |     | -- values.yaml
                |
                | -- routing
                |     |
                |     | -- cluster-<feature_id>-ingress.json①
                |     | -- external-<feature_id>-ingress.json②
                |     | -- egress-<feature_id>.json③
                |
                | -- schemas
                      |
                      | -- schemaParserRules.json
                      | -- ui.schema.json
```

① only present if your_feature_has_internal_ingress was set to 'yes'

② only present if your_feature_has_external_ingress was set to 'yes'

③ only present if your_feature_has_connections was set to 'yes'

Note that the following files: manifest.yaml and ui.schema.json files have been initialized with values from your feature.properties file

Complete build out of your feature

Now that your feature project is initialize, all that is left is to complete the build-out which essentially involves updating certain files in the project that the initialization script wasn't able to construct.

Build out values.yaml

Copy the text from your helm chart values.yaml file and add to the values.yaml file of your feature project.

Expose Feature Properties

While developing a custom feature, a feature developer may decide to provide the ability for an end user that is using the PDS to be able to override certain properties of the custom feature. If a given feature should not expose properties for override, this section can be skipped.

Illustrating how to expose feature properties is best shown via a specific example. This section will

assume that a custom feature with id **simpledb** has been initialized. The **simpledb** feature has the following properties:

- **adminuser** (string value)
- **adminpassword** (string value)
- **port** (numeric value)
- **autocompression** (boolean value)
- **tlssupport** (boolean value)

Exposing feature properties requires the feature developer to know the *path* to the properties in the values.yaml file. Using this knowledge, the feature developer can then modify the **ui.schema.json** file properly to expose the properties to an end user.

The subsequent sections cover these steps in detail.

Determining the property path

Properties like the ones in the previous section can be typically found in an env section of the values.yaml file, such as the following

```
....

env:
    adminuser: "root"
    adminpassword: ""
    port: 2112
    autocompression: True
    tlssupport: True

....
```

In this case, the *path* of the variables are as follows:

/env/adminuser
/env/adminpassword
/env/port
/env/autocompression
/env/tlssupport

However, another example might be the following:

```
....

env:
    vars:
        adminuser: "root"
        adminpassword: ""
        port: 2112
        autocompression: True
        tlssupport: True

....
```

In this case, the *path* of the variables are as follows:

/env/vars/adminuser
/env/vars/adminpassword
/env/vars/port
/env/vars/autocompression
/env/vars/tlssupport

For each property that a feature developer intends to expose, the *path* must be known.

Update ui.schema.json file

Once the property paths are known, the feature developer can then update the **ui.schema.json** file.

After feature project initialization, the starting contents of the generated **ui.schema.json** file is as follows:

```
{
    "$schema": "http://json-schema.org/schema#",
    "type": "object",
    "title": "simpledb",
    "properties": {
      "featureName": {
        "type": "string",
        "title": "Feature name",
        "enum": [
          "simpledb"
        ],
        "default": "simpledb"
      }
    }
}
```

For the purposes of this section, let's decide that only **adminuser**, **adminpassword**, **port** and **autocompression** should be able to be overridden by the end user. Further, let's assume that the properties are found under an **env** section of the values.yaml file.

In order to support exposing the desired properties, the **ui.schema.json** file should be modified to the following

```
{
    "$schema": "http://json-schema.org/schema#",
    "type": "object",
    "title": "simpledb",
    "properties": {
      "featureName": {
        "type": "string",
        "title": "Feature name",
        "enum": [
          "simpledb"
        ],
        "default": "simpledb"
      },
      "adminuser": {
        "type": "string",
        "title": "Admin User",
        "default": "root",
        "path": "/env/adminuser"
      },
      "adminpassword": {
        "type": "string",
        "title": "Admin Password",
        "default": "",
        "path": "/env/adminpassword"
      },
      "port": {
        "type": "number",
        "title": "Port Number",
        "default": 2112,
        "path": "/env/port"
      },
      "autocompression": {
        "type": "boolean",
        "title": "Enable Autocompression",
        "default": True,
        "path": "/env/autocompression"
      }
    }
}
```

Updating the schemaParserRules.json File

Most custom features do not require changes to this file.

Complete Output Connections Configuration

If you specified **feature_connections: yes** in your feature.properties file, then you will need to update a few of the generated files. Note that the specific configuration that is added depends largely on what existing feature your custom feature is connecting. This section will detail the steps necessary to support connections to the mongodb feature.

Update manifest.yaml File

If the features.properties file specified **yes** for the feature_connections property, the generated manifest.yaml file will contain the following:

```
    connections:
      - id: egress-<feature_id>
        description: TODO - Add connection description here
        featureReference: TODO - Add feature reference here
```

As per the **TODOs**, this file should be updated to indicate which feature is being connected. Since this section is using MongoDB as the feature, use your favorite editor to make the following changes:

```
    connections:
      - id: egress-<feature_id>
        description: mongoDB egress
        featureReference: mongodb
```

When complete, save your changes.

Future versions of DevTools will document how to connect to features other than mongodb.

Update routing/egress-<feature_id>.json File

In addition to updating the manifest.yaml file, you will be required to update the egress-<feature_id>.json file as well. As with the previous section, we will use mongo as the feature that is being connected to.

The egress-<feature_id>.json file looks like this when the feature project is initialized

```
{
  "externalVariables": [],
  "files": {},
  "returns": {}
}
```

How this file is updated depends on how the values.yaml file specifies the settings for the output connection.

For the purposes of this example, we'll assume that the values.yaml file is specifying connection information as such:

```
...
env:
  MONGODB_URL: "mongodb://localhost"
  MONGODB_PORT: 27017
 ...
```

The goal is to edit the egress-<feature_id>.json file to replace the above values with values from the mongodb feature.

Using your favorite editor, open the egress-<feature_id>.json file and replace the content with the following:

```
{
  "externalVariables": [
      "ingressConfig"
  ],
  "files": {
      "values.yaml": [
          {
              "title": "Update route to MongoDB",
              "comparison": [
                  {
                      "type": "always"
                  }
              ],
              "operations": [

                  {
                      "op": "copy",
                      "path": "/self/env/MONGODB_URL",
                      "from": "/externalVariables/ingressConfig/clusterServiceName"
                  },
                  {
                      "op": "copy",
                      "path": "/self/env/MONGODB_PORT",
                      "from": "/externalVariables/ingressConfig/clusterServicePort"
                  }

              ]
          }
      ]
  }
}
```

After making the changes, save the file.

Sync Local Feature with Pratexo Design Studio

Once you have completed building out your feature, you will be required to synchronize the feature with the PDS. The sync function is initiated via POST to the PFA that is running on your local system. The specific URI that you will POST to is as follows:

http://localhost:9080/features/<**your_feature_id**>/sync

Note that no special HTTP headers are required to initiate the POST.

How you specifically POST to your locally running PFA is a matter of choice. Using your favorite ReST client, such as curl, wget, Postman, HTTPie, etc.

For illustrative purposes, the commands using curl and wget are shown, since it is reasonable to assume that most system will have one or the other already available (and if not, they are easy to install).

Using curl to synchronize a custom feature

```
curl -X POST http://localhost:9080/features/<your_feature_id>/sync
```

Using wget to synchronize a custom feature

```
wget --post-data '' http://localhost:9080/features/<your_feature_id>/sync
```

In both examples, be sure to replace <**your_feature_id**> with id you populated in the features.properties file.

Verify Feature Has Been Synchronized

If not errors results from the HTTP command, you can now verify that your custom feature has been successfully imported into the PDS. Follow these steps

- login to the PDS
- Navigate to a folder and create a new architecture
- Expand the Features tab in the left hand navigation pane
- Enter your feature name in the search dialog
  - your feature should appear
- Clear search dialog and scroll down to the category you associated your feature
- Expand the category
  - your feature should be listed under the category

Troubleshooting

TBA

Appendix A - Available Category Names

Specifying a feature category help the PDS to organize features in a manner which makes it easier for the end user to select features when building an architecture.

The following are the supported categories. Pick the category that best describes your feature

- AI/ML
- Alerting
- Analytics
- Analyzer
- CA NoSQL Database
- Communications
- Compute Cluster
- Data Visualization/Dashboard
- Device Connectivity
- Document Database
- Event Forwarder
- Event store with dashboard
- Forward Chaining Rules System
- Generic Local Processing
- Global Event Store
- Graph Databases
- Infrastructure
- Logging
- Machine Learning
- Messaging
- ModBus
- Monitoring
- NoSQL Database
- Relational Database
- WebApi
- key-value store
- time-series Database