PRODUCT DOCUMENTATION

Pivotal Function Service®

Version 0.1.0

User's Guide

Table of Contents

Pivotal Function Service (PFS)

Page last updated:

To request early access to this PFS alpha release, please signup here.

This release of Pivotal Function Service is not intended for use in a production environment. Features are subject to change without notice in future releases.

Introduction

Pivotal Function Service (PFS) is a Function as a Service (FaaS) platform built on Kubernetes. PFS is based on the riff open source project.

PFS includes a distribution of Knative for installation on-prem or in the cloud. It also includes the pfs command line interface for managing function builds, Knative services, pub/sub channels, and subscriptions.

Developers can use PFS to build and run functions.

This release comes with buildpacks for functions using the following invokers:

- Java
- JavaScript
- Command

The release was tested with the following environments:

- Pivotal Container Service (PKS) running on Google Cloud Platform (GCP)
- Google Kubernetes Engine (GKE)
- Minikube

Key topics in the documentation include:

- PFS Concepts
- Installing PFS
- Using PFS
- Release Notes

PFS Release Notes

Page last updated:

v0.1.0

Release Date: December 7, 2018

This preview release of Pivotal Function Service is not intended for use in a production environment. Features are subject to change without notice in future releases.

Features

- Supports PKS on GCP, GKE, and Minikube. For more information, see Installing PFS.
- Supports private registries. For more information, see Registries and Relocate Container Images.
- Supports Java, JavaScript, and Command functions in one or more namespaces. For more information, see Functions and Invokers and Using PFS.
- Supports cloud-native buildpacks. For more information, see PFS Buildpacks.
- Supports in-cluster builds and for Minikube, local builds. For more information, see Using PFS.
- Supports eventing. For more information, see Eventing and Using Eventing Channels.
  - Currently only supports the in-memory `stub` bus.
- Includes the `pfs` command line interface. For more information, see The pfs Command Line Interface.
- Supports Knative. This includes the following:
  - Knative/build for building containers from source.
  - Knative/serving for deploying, managing, and autoscaling Knative container workloads.
  - Knative/eventing for connecting Knative container workloads over publish-subscribe channels.
- Supports Istio. For more information, see PFS Component Layering.

Limitations

- PKS is supported only on GCP and only with GCR as a registry.
- Installation from images relocated to a private registry is supported only from a cluster on GCP with the GCR registry.
- The `pfs` CLI runs only on macOS and Linux. Windows is not supported.
- `pfs function create` with `--local-path` is supported only when the builder and run images can be pulled from an unauthenticated registry (Minikube).
- JavaScript streaming functions are experimental.
- Custom languages, buildpacks, and invokers are not supported.
- The eventing features and event sources in Knative are under development.
- Knative monitoring features are not included in this release of PFS.
- Exporting metrics from StatsD to Prometheus is not supported. The deployment `istio-statsd-prom-bridge`, a deprecated Istio service, is omitted from PFS.
- Collection of logs under `/var/log` is not supported. The image `k8s.gcr.io/fluentd-elasticsearch` is omitted from PFS.

Known Issues

The `pfs function create` command may fail with the following error:

```
Error: failed to : find metadata: unexpected end of JSON input
```

This error occurs when a previous image has incompatible metadata. You can work around this issue by specifying a different image name in the `--image` flag or by deleting the remote image.

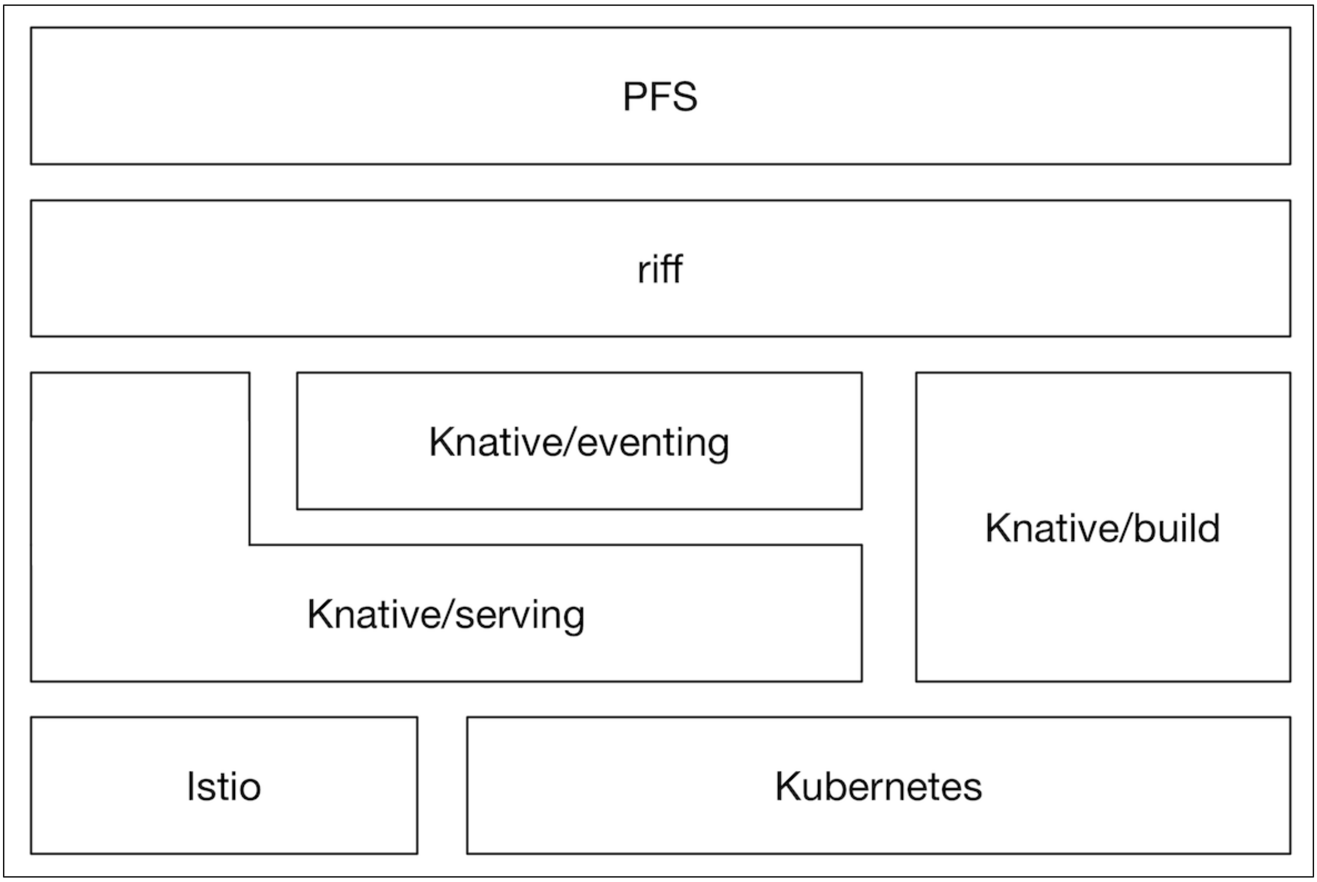
For more information, see issue 21 in the buildpack lifecycle component on GitHub.

Knative

Page last updated:

PFS is built on top of the riff open source project which is itself built on top of the Knative open source project. Knative extends Kubernetes with components which help deliver serverless capabilities for developers and operators.

The following diagram shows the layering of PFS components:



PFS uses the following Knative components:

- Build: builds containers from source code in a Kubernetes cluster.
- Serving: deploys, manages traffic, and autoscales Knative container workloads.
- Eventing: connects Knative workloads over pub/sub channels.

Knative uses Istio to manage network routing inside the cluster, and ingress into the cluster.

For more information on Knative, see the Knative documentation repository.

Functions and Invokers

This topic explains the relationship between functions and the infrastructure code that calls them.

Developers Write Functions

When developers use Pivotal Function Service (PFS), they write functions.

A function is a unit of code that performs a specific task. Functions are isolated from the infrastructure code that calls them.

Even though functions are isolated from the infrastructure code, they can rely on external dependencies such as Spring. Functions are typically built from source and can leverage dependency management solutions for their language of choice, for example, Maven or NPM. For more information, see PFS Buildpacks.

JavaScript function example:

```
module.exports = (x) => x**2;
```

Java function example:

```
package com.acme;

import java.util.function.Function;

public class Upper implements Function<String, String> {

  public String apply(String in) {
    return in.toUpperCase();
  }
}
```

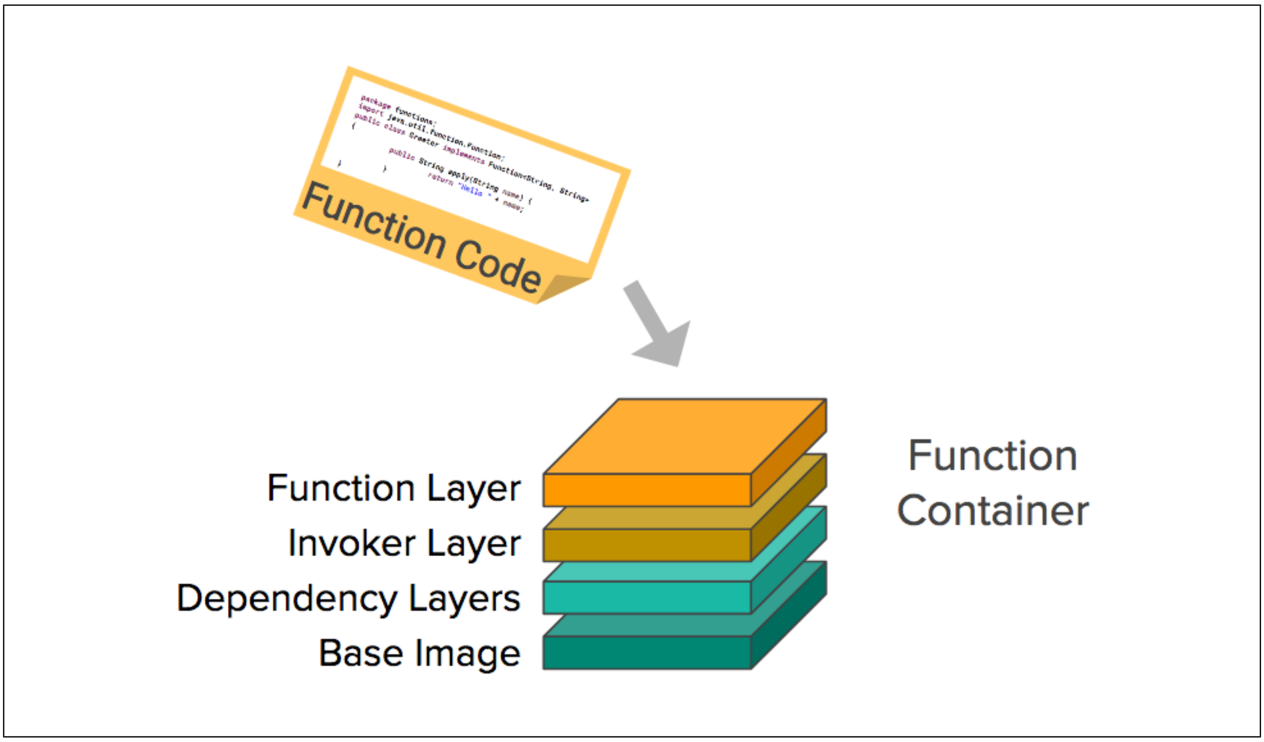
Invokers Call Functions

An invoker works as an interface between the contract of a Knative service and your function. Invokers are written idiomatically for each supported language.

Invokers are responsible for the following:

- Loading and calling functions
- Transforming input and output types

When PFS creates a function container, PFS buildpacks combine functions with invokers. For more information, see PFS Buildpacks.


Function Code
Function Layer
Invoker Layer
Dependency Layers
Base Image
Function
Container

PFS Buildpacks

Page last updated:

This topic describes buildpacks and how they are used in PFS to build container images from function source code.

Introduction

When building a container image for a given function, whether it happens locally or on cluster, PFS uses Cloud Native Buildpacks.

Buildpacks are pluggable, modular tools that translate source code into OCI images. They offer a programmatic alternative to solutions such as Dockerfiles for building images, enabling composition of fine grained, dedicated, pieces of logic (wisely named “buildpacks”) into a bigger “builder”.

When dealing with PFS functions, the user doesn’t have to worry about buildpacks and builders. However, this section explains the behavior and benefits of such an approach.

Composition

The *builder* that PFS uses is composed of several buildpacks, many of which are unaware of the particulars of functions. For example, it embeds a NPM buildpack, capable of building *any* NPM application (provided that it is well formed). Similarly, support for compiling JVM based functions is provided by a Maven and Gradle capable buildpack.

The riff buildpack is responsible for contributing the appropriate function invoker to the image.

Detection and Building

Buildpacks rely on a two-phase contract for building images:

1. The “detect” phase first determines which buildpacks will participate in the build.
2. During the “build” phase, each participating buildpack can act on the results of previous buildpacks, and contribute state to the next.

This mechanism is exploited for PFS functions as follows:

- The presence of a `pom.xml` or `build.gradle` file will trigger compilation and building of an image for running a Java function.
- A `package.json` file or an `--artifact` flag pointing to a `.js` file will build the image for running a JavaScript function.
- An `--artifact` flag pointing to a file with execute permissions will generate an image for running a Command function.

Intelligent Caching

In addition to smart detection logic, buildpacks provide some extra benefits such as the ability to craft image layers in a way that allow intelligent caching and re-use.

When re-building a function, if some aspects of the source have not changed, then buildpacks can decide to re-use layers, thus shortening the build lifecyle.

As an example of such a mechanism, if the `package-lock.json` file capturing the exact versions of NPM dependencies of a javascript function hasn’t changed between builds, then the invocation of NPM is skipped altogether.

Day 2 Operations

Another benefit of buildpacks and their layered approach is the ability to patch images by replacing layers remotely, without impacting the behavior of the function.

This is especially important when security patches need to be applied to OS-level layers of hundreds of images at once.

More information can be found at the buildpacks.io site.

Eventing

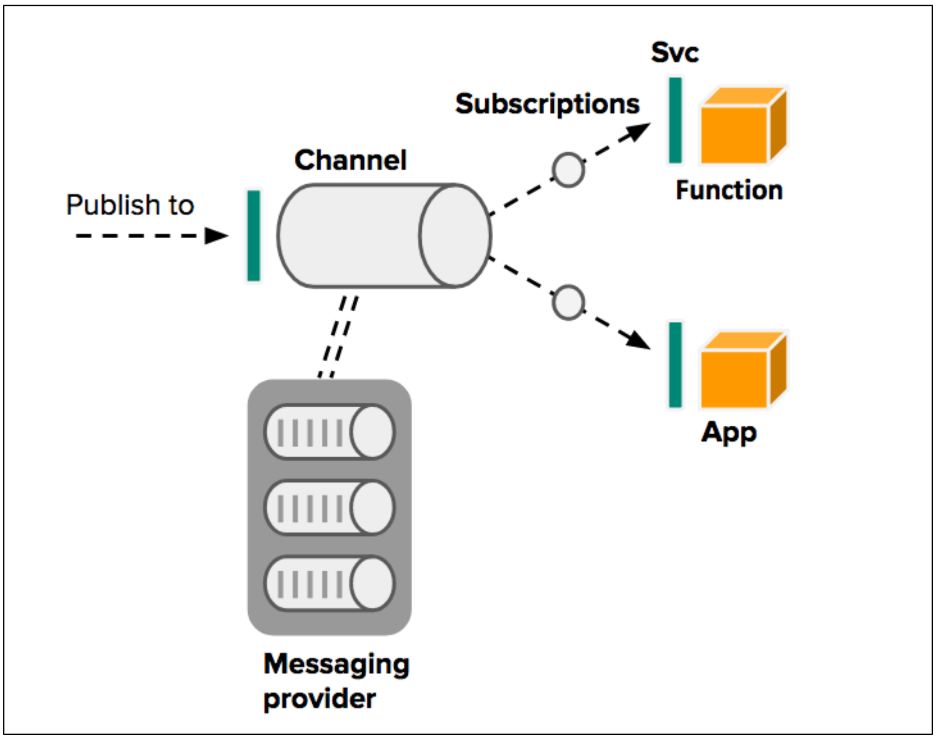
Page last updated:

PFS includes Knative/eventing components for connecting workloads over messaging.

The design of Knative eventing is still evolving and likely to change in future releases.

- Events can be delivered to Channels backed by pub/sub queues on a messaging provider.
- Subscriptions result in the delivery of events to Knative workloads.

This enables message consumers and producers to operate without prior knowledge of each other.



Registries

Page last updated:

PFS includes container images which must be put in a registry before they can be used to create containers.

The PFS distribution consists of:

- A release manifest (lists of kubernetes configuration files)
- Kubernetes configuration files
- An image manifest (map of image names, in the configuration files, to image ids)
- Images (in binary form; some, but not all, of the images in the image manifest).

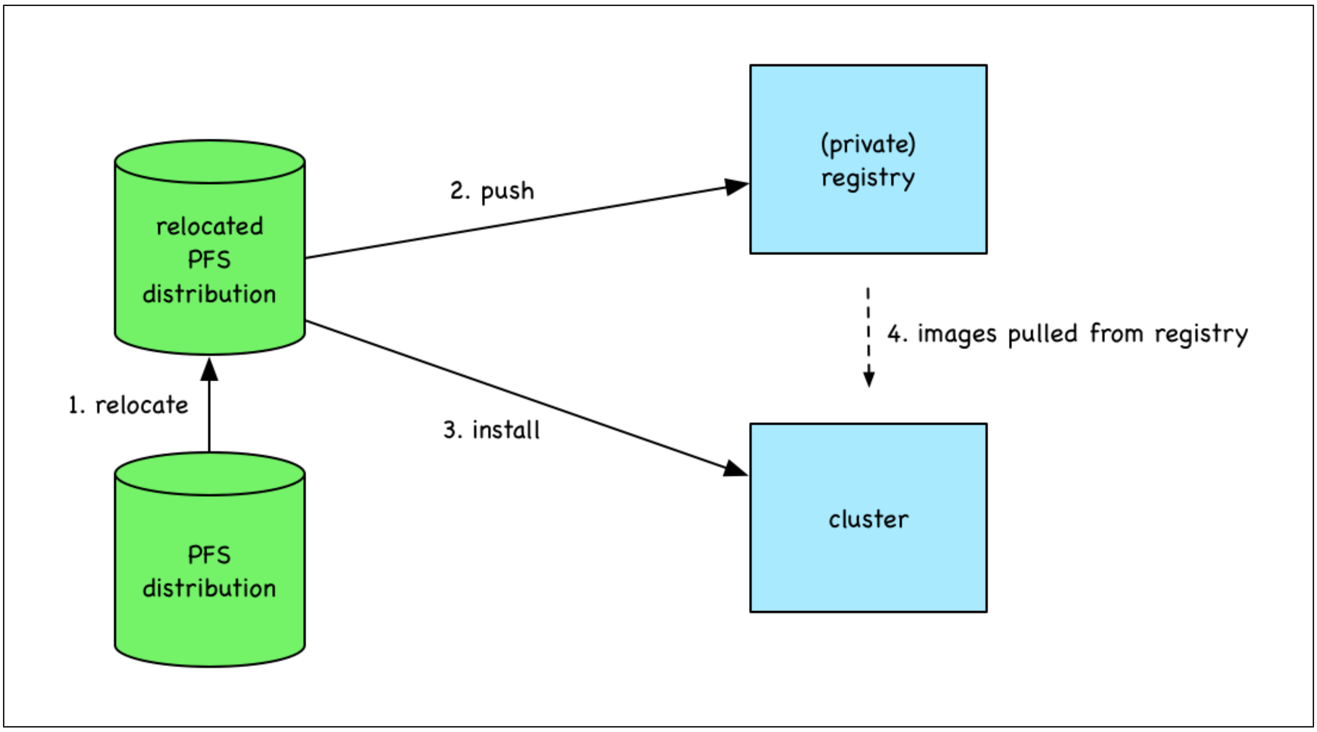
The kubernetes configuration files and the image manifest refer to images hosted in public repositories. These images are *not* supported since Pivotal does not control their content; only the images supplied in the PFS distribution are supported.

So, as part of installing PFS, you need to store the supplied images in a suitable registry and *relocate* the images (described below) in the PFS distribution to point at that registry.

Using your own registry has other advantages:

- You can control when those images are updated.
- The registry can be hosted on a private network for security or other reasons.

The following diagram shows the sequence of events.



Relocating Images

An image name consists of a domain name (with optional port) and a *path*. The image name may also contain a tag and/or a digest. The domain name determines the network location of a registry. The path consists of one or more components separated by forward slashes. The first component is, by convention, a user name providing access control to the image.

Let's look at some examples:

- The image name `docker.io/istio/proxyv2` refers to an image with user name `istio` residing in the docker hub registry at `docker.io`.
- The image name `projectriff/builder` is short-hand for `docker.io/projectriff/builder` which refers to an image with user name `projectriff` also residing at `docker.io`.

- The image name `gcr.io/cf-elafros-dog/knative-releases/github.com/knative/serving/cmd/autoscaler` refers to an image with user name `cf-elafros-dog` residing at `gcr.io`.

When an image is relocated to a registry, the domain name is set to that of the registry and the path modified so that it:

- Starts with the user name that will own the image
- Includes the original user name for readability
- Is “flattened” to accommodate registries which do not support hierarchical paths with more than two components
- Ends with a hash of the image name (to avoid collisions).

For instance, when relocated to a registry at `example.com` with user name `user`, the above image names become:

- `example.com/user/istio-proxyv2-f93a2cacc6cafa0474a2d6990a4dd1a0`
- `example.com/user/projectriff-builder-a4a25a99d48adad8310050be267a10ce`
- `example.com/user/cf-elafros-dog-knative-releases-github.com-knative-serving-cmd-autoscaler-c74d62dc488234d6d1aaa38808898140`

You can find out how to relocate images in Relocate Container Images.

Download PFS from Pivotal Network

To install PFS, The pfs CLI and distribution first need to be downloaded and unpacked. These instructions assume macOS or Linux.

1. Download 2 files from the PFS page on Pivotal Network.
   - Choose either `PFS CLI for macOS...` or `PFS CLI for Linux...`
   - and also `PFS distribution...` a large archive containing installation files and images.
2. Change to the download directory:

   This is commonly named *Downloads* under your home directory, if your system uses a different location then substitute with that.

   ```
   cd ~/Downloads
   ```

3. Unpack both of the downloaded files using a terminal as follows:

   ```
   tar xzvf pfs-cli-...tgz
   mkdir pfs-download
   tar xzv -C pfs-download -f pfs-distro-...tgz
   ```

4. Move the `pfs` CLI to a directory on your path.

   ```
   sudo mv pfs /usr/local/bin/
   ```

5. Confirm that the `pfs` CLI is working E.g.

   ```
   pfs version
   ```

   ```
   Version
     pfs cli: 0.1.0 (e5de84d12d10a060aeb595310decbe7409467c99)
   ```

The expanded PFS distribution directory `pfs-download` should look something like this:

```
pfs-download/
├── image-manifest.yaml
├── images
│   ├── sha256:0c7b69593d12904a898d8ff93fc7326a4d491fb74c7d34e3ec0a5558db3d2365
│   ├── sha256:11c68b5385fb99772ce8b39cee6839a08c5bf4552abfbee4e5e3ca227a8d3e54
│   ├── sha256:262a6e446f6ce7afa5d54d763ce3ed34410c0ba7120d2679f3e5f4d952bb946d
│   ├── sha256:31a4547a8f07e8e9bde55f268a67a198a9cfba4534b6a5eeb06754122acd588f
│   ├── sha256:3ab82c17c25822b034b125ff7b58a34184c7567ea5744efb7bbdc6d9bc34c685
│   ├── sha256:3be7ec27d893a76ccb7c63fbd52babbaba5afc9a1eb787d5779ba2bfcf6d8582
│   ├── sha256:4cd353237d9788467af6d28d3a1f31ec0cdf058d61d05bd0e3e7151b1bb302bb
│   ├── sha256:50d4ec2a16fd249a40eb5fb28d54a04553bb026f87afe220442a9563972449dc
│   ├── sha256:55c07d77a0b039b1af9acebd6d159b4f94528fad4d8695cd5f488b7365db0bef
│   ├── sha256:6e2d3acd9b1442bcdd46a8a2c28c0acd034cdaae33a99bce9829a64b6f7e847e
│   ├── sha256:6ea88f82046a1de4018a6282b0fbaca0e3d2b6c3979464c5c0f87c0ce5d6b872
│   ├── sha256:77e6870301bb24aef719bf86485158cf79788309385ee598f698a2dc8ead039f
│   ├── sha256:78a363e4b45be3741bd2c2548801d623d91b207bade213c0aca3d5bcea82ab5d
│   ├── sha256:a0e97babcda043614f6b2b7eed4dd4c171d186b8500e670941dfed203e81a6ad
│   ├── sha256:a23501d42e579d3d60e23dbbd9ec74d5fb743c22e18a5b3adf5fc267a3c40806
│   ├── sha256:b34e3ef99bf613851cace672643d6dab31df3570033f31e39375a62beffbf420
│   ├── sha256:b862513e5e886b1c2e7c2beaa2e783227e5e67f626b0f7df3d2fe47baa45ce01
│   ├── sha256:b8cfc0e19a91047c9dad45e43858ee16304ac4be98ff0ab187f8ecff4817a206
│   ├── sha256:ca4050c9fed3a2ddcaef32140686613c4110ed728f53262d0a23a7e17da73111
│   ├── sha256:d559bdcd7a8872a662cf3195c8792ca0df50922d81722b4207b14e080e420f0b
│   ├── sha256:f0db13ef4a3b70570fe853ae2048deb557f6943d7cbbe3abe06a75bd0eaedd99
│   ├── sha256:f5631a82ba31a87f0c4e082e3f542927238da8e2ddc522959d029a8b03cd6235
│   ├── sha256:fadd9ed5d5070763aa2994cd2f31b6eb56dacfa00f49ea58f82718dda9ad5ba3
│   └── sha256:fc8c2aa60865888ecc475d9bc7d4399b60da8adec74c330acb20f134c5932617
├── istio-riff-knative-serving-v0-2-2-e87abddc666ed6ee0533a6a80226d11a.yaml
├── manifest.yaml
├── release-3abd9a7d3e747faf7c1fb7ebc82748a7.yaml
├── release-6919db2dad3443aea5fdfbd57c4bc613.yaml
├── release-clusterbus-stub-545d115713c5a18364cb7b73ed35ed14.yaml
├── riff-cnb-buildtemplate-0.1.0-66c3cc7a6054c438d8505b30cf4772a9.yaml
└── serving-2939e59a4aeabd8cf043f393915218bd.yaml
```

Next: Relocate Container Images

Relocate Container Images

Page last updated:

This topic describes how to relocate the container images supplied in the PFS distribution.

Requirements

- PFS container images have been downloaded.
- The `pfs` CLI has been downloaded and installed.
- `docker` is installed and a docker daemon is running on your development machine.
- A container registry like GCR is available for hosting images. (For Minikube installs, please first follow steps 1-7 of Installing PFS on Minikube.)
- The docker CLI can push images to the registry. (See Google Cloud to configure a docker credential helper to push images to GCR.)

Step 1: Create the Relocated Distribution

The relocation process will create a new distribution of PFS in an output directory which you can then use for future installations of PFS. All the Kubernetes configuration files in the relocated directory will point to images in your own private registry, rather than public open source repos.

When you use `pfs image relocate`, you will need to specify the following:

- An output directory
- The manifest and image-manifest from your downloaded PFS distribution
- Your registry's domain name and user/repo name

For example, to use the Google Container Registry, the registry domain name is `gcr.io` and the registry user id is the same as your GCP Project ID.

```
export REGISTRY=gcr.io
export REGISTRY_USER=$(gcloud config get-value core/project)
```

For the registry in Minikube, use registry domain name `registry.kube-system.svc.cluster.local` and a dummy username.

```
export REGISTRY=registry.kube-system.svc.cluster.local
export REGISTRY_USER=testuser
```

Change to the download directory if that is not the current directory:

> This is commonly named *Downloads* under your home directory, if your system uses a different location then substitute with that.

```
cd ~/Downloads
```

If the directory containing the downloaded distribution files is `pfs-download`, the command below will produce a new output distribution in a subdirectory called `pfs-relocated`.

```
echo "relocating images to $REGISTRY/$REGISTRY_USER"

pfs image relocate \
  --output pfs-relocated \
  --manifest pfs-download/manifest.yaml \
  --images pfs-download/image-manifest.yaml \
  --registry $REGISTRY \
  --registry-user $REGISTRY_USER
```

Step 2. Push Images to the Registry

NOTE: If you are planning to run on Minikube, first install Minikube with the registry add-on.

`pfs image push` uses the image-manifest in the relocated distribution.

```
pfs image push --images pfs-relocated/image-manifest.yaml
```

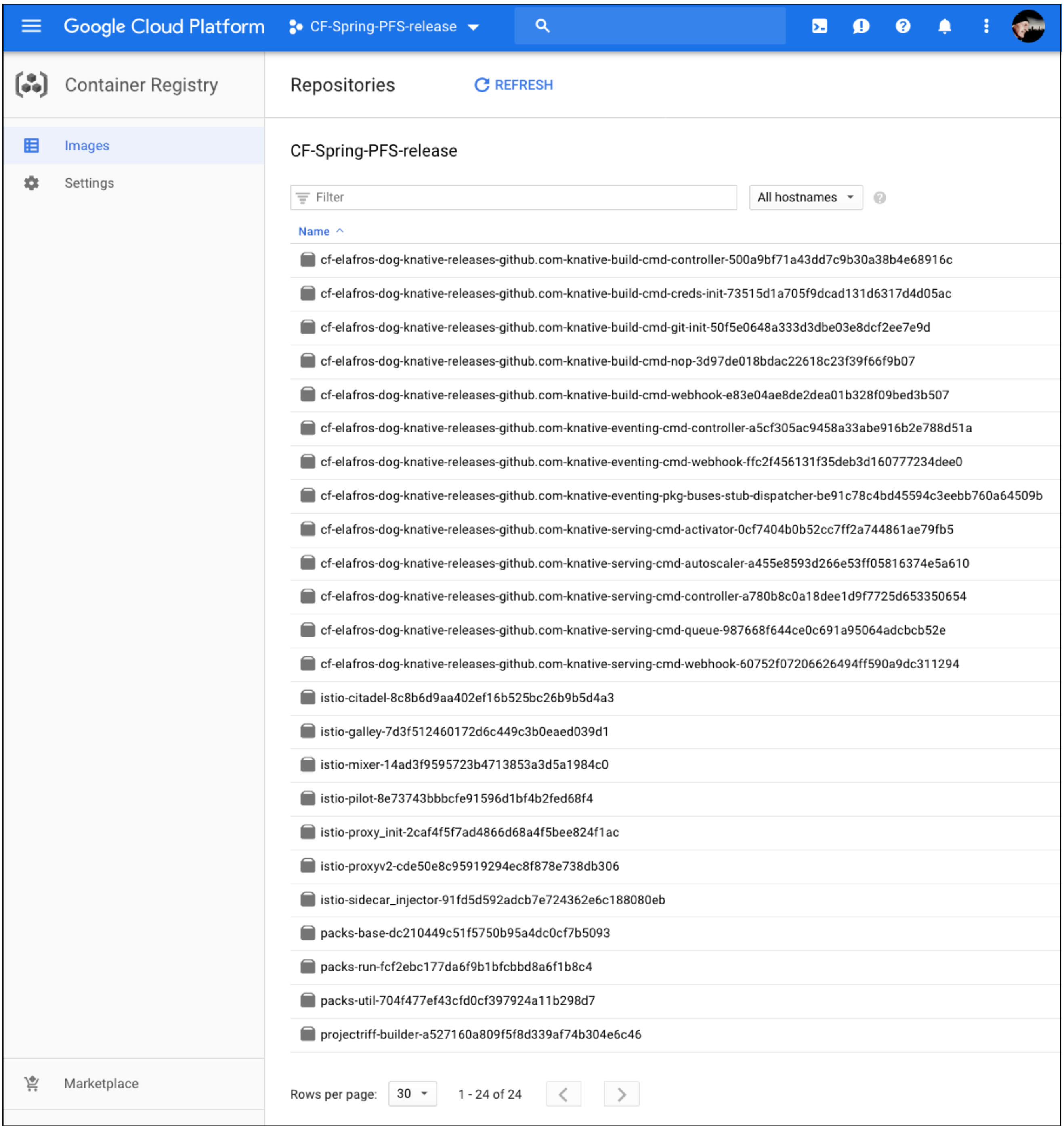
For each image this does:

- `docker load` to load the image from a file in the distribution
- `docker tag` to give the image its relocated name
- `docker push` to push the image to your registry

This process may take a while to complete depending on the available network bandwidth .

You may notice a `Skipping IMAGE-NAME` message during the loading stage. This is normal because certain non-essential images have been omitted from this PFS distribution for expediency reasons.

A sample listing of relocated images on GCR is shown below:



Installing PFS on PKS

Page last updated:

This topic describes how to install Pivotal Function Service (PFS) on Pivotal Container Service (PKS) deployed to GCP.

Requirements

- The `pks` CLI has been installed.
- The `kubectl` CLI has been installed at version 1.10 or later.
- The Google Cloud SDK which provides the `gcloud` CLI has been installed.
- The `pfs` CLI has been downloaded and installed.
- PFS container images have been downloaded and successfully relocated.
- The relocated manifest and relocated image-manifest are available.

It is assumed that when image relocation was carried out the registry host and registry user arguments targeted a GCR registry in the current GCP project.

Installation Steps

1. Verify that the Google Cloud APIs, and Container Registry API are enabled in the current GCP project.

   ```
   gcloud services list
   ```

   ```
   NAME                              TITLE
   cloudapis.googleapis.com          Google Cloud APIs
   containerregistry.googleapis.com  Container Registry API
   . . .
   ```

   If necessary enable these services using the `gcloud services enable` command.

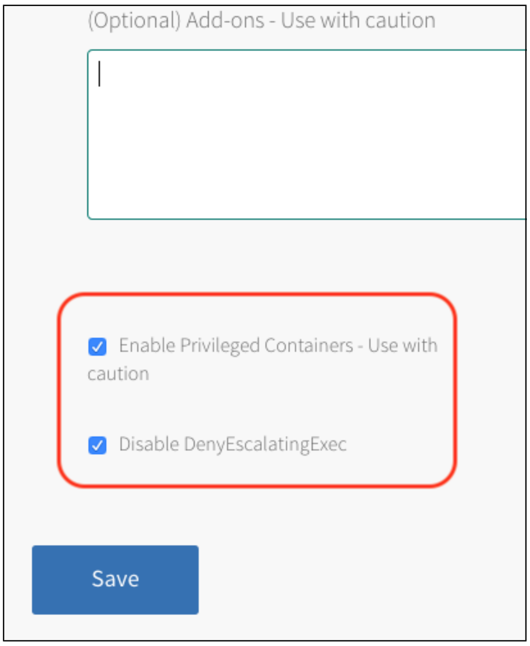
   ```
   gcloud services enable \
     cloudapis.googleapis.com \
     containerregistry.googleapis.com
   ```

2. If you haven't already done so, push the relocated images to the GCR registry in the current GCP project using the `pfs` CLI as shown below where the `-i` flag is the path to the previously relocated image manifest file.

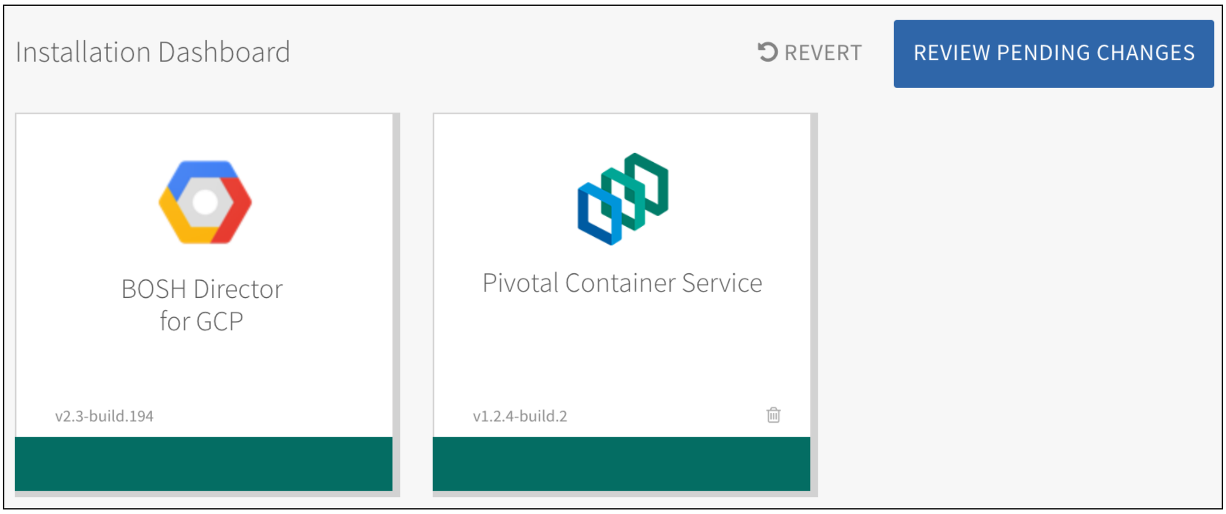
   ```
   pfs image push -i pfs-relocated/image-manifest.yaml
   ```

3. Select a PKS plan and enable privileged containers and escalated privileges.
   This step is necessary in order for Istio sidecar injection to function correctly.

   - Using PCF Ops Manager, click on the `Pivotal Container Service` tile, and select one of the plans from the menu on the left. It is recommended to choose a plan intended for large workloads.
     e.g.
     - a master node VM with 2 CPUs and 8GB of memory
     - four worker nodes each with 2 CPUs and 8GB of memory
   - Scroll to the bottom of the plan page and enable both the **Enable Privileged Containers** and **Disable DenyEscalatingExec** checkboxes as shown below.



- Save your changes and navigate back to the installation dashboard. Click **Review Pending Changes** and then apply them.



4. Log into the PKS environment using your usual credentials. To log in targeting the PKS API server `pks-api.example.com` as user `admin` with password `adminpassword` the following command would be run.

```
pks login -a pks-api.example.com -u admin -p adminpassword
```

5. Create a new PKS cluster being careful to specify the PKS plan which you configured in step 3 above.

   - To create a new cluster called `mycluster` using the `large` plan, and an external hostname of `myhostname.example.com` run:

   ```
   pks create-cluster mycluster \
       --external-hostname myhostname.example.com \
       --plan large
   ```

   Track the progress of the create using the `pks cluster` command. For example, to check on the status of a cluster named `mycluster` run `pks cluster mycluster`

   It can take up to 30 minutes for cluster creation to complete.

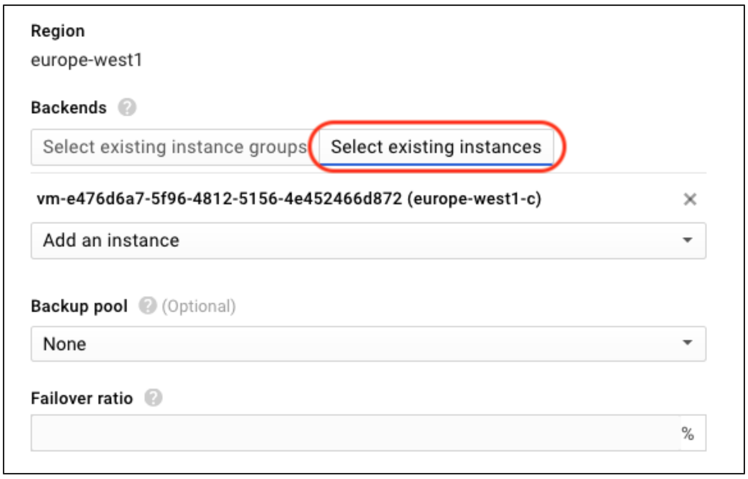
6. Configure a GCP load balancer for the created cluster. This step and the following step are required to make the new PKS on GCP cluster available on the network. Consult your PKS environment administrator if you need assistance in carrying them out.

   - Run the `pks cluster` command with your cluster name and note the value of `Kubernetes Master IP(s)`. In the example below the master node IP address is `10.0.11.10`

```
pks cluster mycluster
```

```
Name:                   mycluster
Plan Name:              large
UUID:                   65bc21d8-819f-483a-a08b-7c55b500b0a2
Last Action:            CREATE
Last Action State:      succeeded
Last Action Description:  Instance provisioning completed
Kubernetes Master Host:   myhostname.example.com
Kubernetes Master Port:   8443
Worker Nodes:           4
Kubernetes Master IP(s):  10.0.11.10
Network Profile Name:
```

- In your Google Cloud Platform Console navigate to **Compute Engine > VM instances** and locate the master VM by filtering on `Internal IP` with the value of the master node IP address. Note the name of this master node VM which will be of the form `vm-<guid>` and also its zone.
- Navigate to **Network services > Load balancing** and click **Create Load Balancer**.
- In the **TCP Load Balancing** pane, click **Start configuration**.
- Accept all of the defaults on the next page and click **Continue**.
- In the resulting page give the load balancer a name
- Click the **Backend configuration** section and set the region value to be consistent with the zone of the cluster master VM (e.g. if the master VM is in zone `europe-west1-c` then set the region to be `europe-west-1`).
- Click the **Select existing instances** tab and then select your cluster's master node VM in the dropdown.



- Click **Frontend configuration** and give it the same name as the load balancer.
- In the **IP** dropdown select the option to **Create IP address**.
- In the resulting **reserve a new static IP address** box give the same name as the frontend load balancer and then click **Reserve**.
- Enter the value `8443` in the **Port** field.
- Click **Done**.
- Review the new load balancer settings and then click **Create**.

It will take several seconds before the new load balancer becomes available.

7. Configure the DNS record for the created cluster.

- In the Google Cloud Platform Console page for the cluster load balancer (**Network services > Load balancing**) note the IP address of the **Frontend** of the load balancer.
- Navigate to **Network services > Cloud DNS** and click the appropriate DNS zone for your PKS cluster (its DNS name will be the domain of the PKS environment Ops Manager).
- Click **Add record set**.
- In the next page add a *prefix* to the DNS name so that the complete name matches the `--external-hostname` value used when the cluster was created.
- In the **IPv4 Address** field enter the IP address (only the IP address, do not include the port number) of the **Frontend** of the cluster's load balancer.
- Click **Create**.

> It can take several minutes for the DNS record information to propagate around the network. During this time "unable to connect" or "no such host" errors may occur when attempting to use kubectl with the cluster.

8. Use the `pks` CLI to retrieve credentials and change your kubectl context to your PKS cluster.
    - To change context to a PKS cluster named `mycluster` run the following:

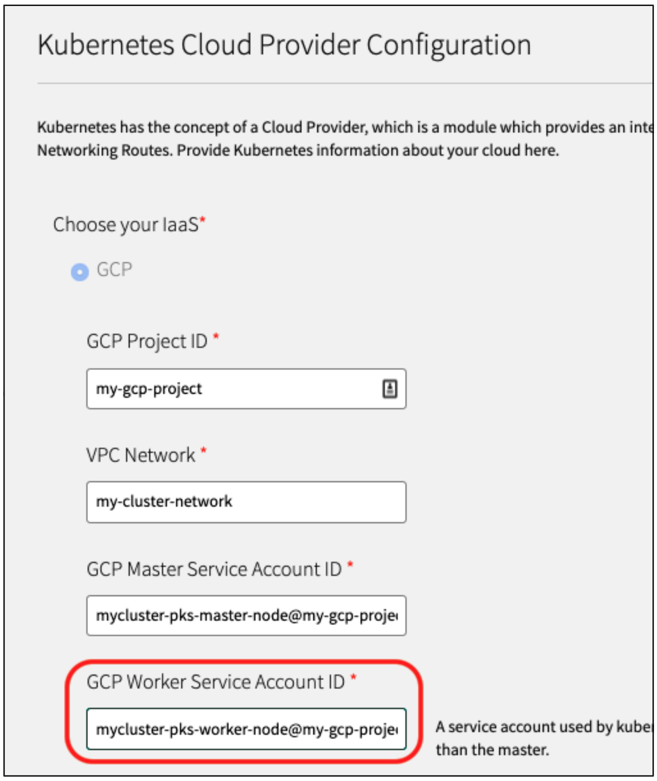
      ```
      pks get-credentials mycluster
      ```

    - Verify that the current context is as expected using `kubectl`:

      ```
      kubectl config current-context
      ```

      ```
      mycluster
      ```

9. In order to authorise your PKS cluster to pull images from your local GCP project's GCR registry it is necessary to update the IAM permissions of the service account used by the worker nodes.
    - To determine this service account go to the PCF Ops Manager set up when your PKS environment was created, click the **Pivotal Container Service** tile, and then navigate to the **Kubernetes Cloud Provider** configuration page. The **GCP Worker Service Account ID** field will contain the ID of the worker nodes' service account.



    - Export the name of the serviceaccount in an environment variable E.g.

      ```
      export WORKER_SERVICE_ACCOUNT=mycluster-pks-worker-node@my-gcp-project.iam.gserviceaccount.com
      ```

    - Use the `gsutil iam ch` command to add the objectViewer role, permitting the service account to read from the GCS storage bucket backing your project's GCR registry.

      ```
      gsutil iam ch \
        serviceAccount:$WORKER_SERVICE_ACCOUNT:objectViewer \
        gs://artifacts.$GCP_PROJECT.appspot.com
      ```

10. Install PFS using the `pfs` CLI as shown below where the `-m` flag is the path to the previously relocated manifest file.

    ```
    pfs system install -m pfs-relocated/manifest.yaml
    ```

    After the command completes pods should be successfully running in the `istio-system`, `knative-build`, `knative-eventing`, `knative-serving`, and `kube-system` namespaces similar to the output from `kubectl get pods` shown below.

    ```
    kubectl get pods --all-namespaces
    ```

```
NAMESPACE       NAME                                    READY   STATUS    RESTARTS   AGE
istio-system    istio-citadel-6dddb76bff-fhhd9          1/1     Running   0          1m
istio-system    istio-egressgateway-6ffcb89f5-s5twg     1/1     Running   0          1m
istio-system    istio-galley-76db9d5457-6whqt           1/1     Running   0          1m
istio-system    istio-ingressgateway-b44b59658-ndpkz    1/1     Running   0          1m
istio-system    istio-pilot-78668c67bd-swmj9            2/2     Running   0          1m
istio-system    istio-policy-78fd576646-l8rpd           2/2     Running   0          1m
istio-system    istio-sidecar-injector-646b6f4d4b-wt9g2 1/1     Running   0          1m
istio-system    istio-telemetry-56fbcd7b6f-s6bpg        2/2     Running   0          1m
istio-system    knative-ingressgateway-6d8d7b56f6-mg9qt 1/1     Running   0          36s
knative-build   build-controller-bcfdddf77-nb5f6        1/1     Running   0          37s
knative-build   build-webhook-7dccdd6ff8-qnb4p          1/1     Running   0          37s
knative-eventing  eventing-controller-5bb797f774-nhhj4  1/1     Running   0          31s
knative-eventing  stub-clusterbus-dispatcher-5658748c67-lhddz  2/2  Running   0      15s
knative-eventing  webhook-685f94854c-l9z7x              1/1     Running   0          31s
knative-serving   activator-779cf7cd8-2m984             2/2     Running   0          34s
knative-serving   activator-779cf7cd8-nj76w             2/2     Running   0          34s
knative-serving   activator-779cf7cd8-r6sgm             2/2     Running   0          34s
knative-serving   autoscaler-85984647cf-9d5wl           2/2     Running   0          34s
knative-serving   controller-6ff4f74bd9-cc7hz           1/1     Running   0          33s
knative-serving   webhook-5d898886bf-xlps4              1/1     Running   0          33s
kube-system     heapster-6d5f964dbd-fgrlk               1/1     Running   0          1h
kube-system     kube-dns-6b697fcdbd-76p4d               3/3     Running   0          1h
kube-system     kubernetes-dashboard-785584f46b-bd68t   1/1     Running   0          1h
kube-system     metrics-server-5f68584c5b-n9wh4         1/1     Running   0          1h
kube-system     monitoring-influxdb-54759946d4-wv29g    1/1     Running   0          1h
kube-system     telemetry-agent-68c6647967-886bd        1/1     Running   0          1h
pks-system      fluent-bit-dmrcv                        1/1     Running   0          1h
pks-system      fluent-bit-fcvhk                        1/1     Running   0          1h
pks-system      fluent-bit-lbwns                        1/1     Running   0          1h
pks-system      fluent-bit-s8j7r                        1/1     Running   0          1h
pks-system      sink-controller-7c85744bd6-4lbwq        1/1     Running   0          1h
```

PFS is now installed. Next you need to prepare one or more namespaces for your functions.

Prepare a Namespace

You need to initialize PFS resources in each Kubernetes namespace that your function pods will run in.

1. Get the name of your current GCP project:

```
export GCP_PROJECT=$(gcloud config get-value core/project)
```

2. Create a service account with the required permission to *push* function images to GCR. To create a GCP service account named `push-image` run:

```
gcloud iam service-accounts create push-image
```

To grant the `push-image` account the `storage.admin` role run the following command:

```
gcloud projects add-iam-policy-binding $GCP_PROJECT \
  --member serviceAccount:push-image@$GCP_PROJECT.iam.gserviceaccount.com \
  --role roles/storage.admin
```

3. Create a private authentication key for the push service account and store it in a local file. To create a new key for the `push-image` service account and have it stored in a file called `gcr-storage-admin.json` run the following:

```
gcloud iam service-accounts keys create \
  --iam-account "push-image@$GCP_PROJECT.iam.gserviceaccount.com" \
  gcr-storage-admin.json
```

4. Use the `pfs` CLI to initialize PFS resources in a Kubernetes namespace. Pass the path to the previously relocated manifest file using the `-m` flag and the path to the previously created private authentication key file using the `--gcr` flag. The following command initializes the `default` namespace:

```
pfs namespace init default -m pfs-relocated/manifest.yaml \
  --gcr gcr-storage-admin.json
```

You can now create your first function.

Installing PFS on GKE

Page last updated:

This topic describes how to install Pivotal Function Service (PFS) on Google Kubernetes Engine (GKE).

Requirements

- The `kubectl` CLI has been installed at version 1.10 or later.
- The Google Cloud SDK which provides the `gcloud` CLI has been installed.
- The `pfs` CLI has been downloaded and installed.
- PFS container images have been downloaded and successfully relocated.
- The relocated manifest and relocated image-manifest are available.

It is assumed that when image relocation was carried out the registry host and registry user arguments targeted a GCR registry in the current GCP project.

Installation Steps

1. Verify that the Google Cloud APIs, Kubernetes Engine API, and Container Registry API are enabled in the current GCP project.

   ```
   gcloud services list
   ```

   ```
   NAME                               TITLE
   cloudapis.googleapis.com           Google Cloud APIs
   container.googleapis.com           Kubernetes Engine API
   containerregistry.googleapis.com   Container Registry API
   ...
   ```

   If necessary enable these services using the `gcloud services enable` command.

   ```
   gcloud services enable \
     cloudapis.googleapis.com \
     container.googleapis.com \
     containerregistry.googleapis.com
   ```

2. Create a new GKE cluster if a suitable one does not already exist. In order to meet Knative guidelines, we recommend running Kubernetes version 1.10 or higher, on nodes with 4 vCPUs and 15GB of memory.
   The following `gcloud` CLI command will create a cluster named `my-gke-cluster` using the most recent version of Kubernetes available in GKE, and three `n1-standard-4` nodes.

   ```
   gcloud container clusters create my-gke-cluster \
     --cluster-version=latest \
     --machine-type=n1-standard-4 \
     --enable-autoscaling --min-nodes=1 --max-nodes=3 \
     --enable-autorepair \
     --scopes=service-control,service-management,compute-rw,storage-ro,cloud-platform,logging-write,monitoring-write,pubsub,datastore \
     --num-nodes=3
   ```

3. If you haven't already done so, push the relocated images to the GCR registry in the current GCP project using the `pfs` CLI as shown below where the `-i` flag is the path to the previously relocated image manifest file.

   ```
   pfs image push -i pfs-relocated/image-manifest.yaml
   ```

4. Use `kubectl` to verify that the target GKE cluster is referenced by the current context.

   ```
   kubectl config current-context
   ```

   The clusters create command should have added a context for the new cluster.

If necessary you can create a new context for the cluster by retreiving the credentials using a `gcloud` command. To create a new `kubectl` context for a cluster named `my-gke-cluster` in the project and region/zone currently configured for `gcloud` run:

```
gcloud container clusters get-credentials my-gke-cluster
```

5. Grant yourself cluster-admin permissions.

```
kubectl create clusterrolebinding cluster-admin-binding \
  --clusterrole=cluster-admin \
  --user=$(gcloud config get-value core/account)
```

6. Install PFS using the `pfs` CLI as shown below where the `-m` flag is the path to the previously relocated manifest file.

```
pfs system install -m pfs-relocated/manifest.yaml
```

After the command completes pods should be successfully running in the `istio-system`, `knative-build`, `knative-eventing`, `knative-serving`, and `kube-system` namespaces similar to the output from `kubectl get pods` shown below.

```
kubectl get pods --all-namespaces
```

```
NAMESPACE     NAME                                   READY   STATUS   RESTARTS   AGE
istio-system   istio-citadel-79964f8bd9-dqpbq          1/1    Running   0     2m
istio-system   istio-egressgateway-57988996cc-lt9n6     1/1    Running   0     2m
istio-system   istio-galley-556fbd8687-k7flt           1/1    Running   0     2m
istio-system   istio-ingressgateway-69fd7b888-hsqsr     1/1    Running   0     2m
istio-system   istio-pilot-5764c7f956-zqtpm            2/2    Running   0     2m
istio-system   istio-policy-57f6799f46-89nqx           2/2    Running   0     2m
istio-system   istio-sidecar-injector-7548c8cd9b-rs9kc  1/1    Running   0     2m
istio-system   istio-telemetry-db55c4c7c-jbtww         2/2    Running   0     2m
istio-system   knative-ingressgateway-545c57fd8b-c2kcn  1/1    Running   0     35s
knative-build   build-controller-655bdd95f4-zzzsp       1/1    Running   0     35s
knative-build   build-webhook-775c8f5996-lmxk9          1/1    Running   0     35s
knative-eventing   eventing-controller-ddb4d966-dcdgq    1/1    Running   0     32s
knative-eventing   stub-clusterbus-dispatcher-8679cd49bd-mbrzd   2/2    Running   0     25s
knative-eventing   webhook-6c4c867647-dtpkb             1/1    Running   0     32s
knative-serving   activator-5c4d788788-hr49t            2/2    Running   0     34s
knative-serving   activator-5c4d788788-nbtkj            2/2    Running   0     34s
knative-serving   activator-5c4d788788-shfsh            2/2    Running   0     34s
knative-serving   autoscaler-5459944c66-r6tlt           2/2    Running   0     34s
knative-serving   controller-7f7f76cb4b-k7ffn           1/1    Running   0     33s
knative-serving   webhook-59457b697c-s6599              1/1    Running   0     33s
kube-system   event-exporter-v0.2.1-7978ddf677-jqwx6     2/2    Running   0     4m
kube-system   fluentd-gcp-scaler-5d85d4b48b-47dfn        1/1    Running   0     4m
kube-system   fluentd-gcp-v3.1.0-fjqsb                   2/2    Running   0     4m
kube-system   fluentd-gcp-v3.1.0-l82bx                   2/2    Running   0     4m
kube-system   fluentd-gcp-v3.1.0-nq8tn                   2/2    Running   0     4m
kube-system   heapster-v1.6.0-beta.1-7d8cbf84b6-tg2pg    3/3    Running   0     3m
kube-system   kube-dns-548976df6c-k76tk                  4/4    Running   0     4m
kube-system   kube-dns-548976df6c-lzpjp                  4/4    Running   0     4m
kube-system   kube-dns-autoscaler-67c97c87fb-sqtns       1/1    Running   0     4m
kube-system   kube-proxy-gke-riff-cluster-2-default-pool-64c434d3-4zqc   1/1    Running   0     4m
kube-system   kube-proxy-gke-riff-cluster-2-default-pool-64c434d3-pccc   1/1    Running   0     4m
kube-system   kube-proxy-gke-riff-cluster-2-default-pool-64c434d3-x970   1/1    Running   0     4m
kube-system   l7-default-backend-5bc54cfb57-5k6fj        1/1    Running   0     4m
kube-system   metrics-server-v0.2.1-fd596d746-vgbck      2/2    Running   0     4m
```

PFS is now installed. Next you need to prepare one or more namespaces for your functions.

Prepare a Namespace

You need to initialize PFS resources in each Kubernetes namespace that your function pods will run in.

1. Get the name of your current GCP project:

```
export GCP_PROJECT=$(gcloud config get-value core/project)
```

2. Create a service account with the required permission to *push* function images to GCR. To create a GCP service account named `push-image` run:

```
gcloud iam service-accounts create push-image
```

To grant the `push-image` account the `storage.admin` role run the following command:

```
gcloud projects add-iam-policy-binding $GCP_PROJECT \
  --member serviceAccount:push-image@$GCP_PROJECT.iam.gserviceaccount.com \
  --role roles/storage.admin
```

3. Create a private authentication key for the push service account and store it in a local file. To create a new key for the `push-image` service account and have it stored in a file called `gcr-storage-admin.json` run the following:

```
gcloud iam service-accounts keys create \
  --iam-account "push-image@$GCP_PROJECT.iam.gserviceaccount.com" \
  gcr-storage-admin.json
```

4. Use the `pfs` CLI to initialize PFS resources in a Kubernetes namespace. Pass the path to the previously relocated manifest file using the `-m` flag and the path to the previously created private authentication key file using the `--gcr` flag. The following command initializes the `default` namespace:

```
pfs namespace init default -m pfs-relocated/manifest.yaml \
  --gcr gcr-storage-admin.json
```

You can now create your first function.

Installing PFS on Minikube

Page last updated:

This topic describes how to install Pivotal Function Service (PFS) on your development machine using `minikube`.

Requirements

These requirements assume a macOS or Linux environment.

- `minikube` has been installed at version v0.30.0 or later.
- A Minikube VM driver for your OS has been installed:
  - For macOS use the Hyperkit driver.
  - For Linux use the KVM2 driver.
- The `kubectl` CLI has been installed at version 1.10 or later.
- `docker` is installed and a docker daemon is running on your development machine.
- The `pfs` CLI has been downloaded and installed.
- PFS container images have been downloaded and successfully relocated.
- The relocated manifest and relocated image-manifest are available.

Running Minikube with a Registry

Minikube is a Kubernetes environment which runs in a single virtual machine on your host. Since PFS installations require a container registry, these instructions include enabling the minikube registry add-on.

To maintain consistency of image names across environments, the registry will be named `registry.kube-system.svc.cluster.local` and it will be exposed on port 80, both inside minikube as well as from the host.

> Using the minikube registry addon ties the registry's lifecycle to that of minikube. You will lose contents of the registry when you restart minikube. Also, due to limited resources, installing PFS and creating functions can be significantly slower.

Installation Steps

1. Start the minikube cluster

```
minikube start --memory=8192 --cpus=4 \
  --kubernetes-version=v1.12.3 \
  --bootstrapper=kubeadm \
  --vm-driver=hyperkit \
  --extra-config=apiserver.enable-admission-plugins="LimitRanger,NamespaceExists,NamespaceLifecycle,ResourceQuota,ServiceAccount,DefaultStorageClass,MutatingAdmissionWebhook"
```

   Note: for Linux use `kvm2` instead of `hyperkit` for the `--vm-driver`

2. Use `kubectl` to verify that the context is set to minikube.

```
kubectl config current-context
```

```
minikube
```

   If necessary set the current context.

```
kubectl config use-context minikube
```

3. In a separate terminal window, watch the pods in the cluster.

```
watch -n 1 kubectl get pod --all-namespaces
```

```
NAMESPACE    NAME                                   READY    STATUS    RESTARTS   AGE
kube-system   coredns-576cbf47c7-xqnkd              1/1      Running   0          1m
kube-system   etcd-minikube                         1/1      Running   0          1m
kube-system   kube-addon-manager-minikube           1/1      Running   0          45s
kube-system   kube-apiserver-minikube               1/1      Running   0          46s
kube-system   kube-controller-manager-minikube      1/1      Running   0          1m
kube-system   kube-proxy-djb7j                      1/1      Running   0          1m
kube-system   kube-scheduler-minikube               1/1      Running   0          1m
kube-system   kubernetes-dashboard-5bb6f7c8c6-t6vkt  1/1     Running   0          1m
kube-system   storage-provisioner                   1/1      Running   0          1m
```

4. Enable minikube registry addon to host your PFS relocated images and build images. A `registry` pod should appear in the `kube-system` namespace after a delay .

```
minikube addons enable registry
```

5. Add an entry for the registry to your dev machine `/etc/hosts` and validate the new file contents.

```
echo "127.0.0.1 registry.kube-system.svc.cluster.local" | sudo tee -a /etc/hosts
cat /etc/hosts
```

6. In a separate shell run port-forward so that the registry is reachable from your dev machine

```
sudo kubectl port-forward --namespace kube-system service/registry 80
```

7. Add an entry to /etc/hosts inside the minikube VM. First get the cluster IP of the registry service, then append an entry for that IP to /etc/hosts on the minikube vm.

```
export REG_IP=$(kubectl get svc --namespace kube-system \
  -l "kubernetes.io/minikube-addons=registry" \
  -o jsonpath="{.items[0].spec.clusterIP}")

echo REG_IP = $REG_IP
```

```
minikube ssh \
  "echo \"$REG_IP      registry.kube-system.svc.cluster.local\" \
  | sudo tee -a  /etc/hosts"
```

8. Follow the instructions to Push relocated images to the registry in Minikube, making sure that the images were relocated to `registry.kube-system.svc.cluster.local` .

   You can review the contents of the registry using:

```
curl -X GET registry.kube-system.svc.cluster.local/v2/_catalog
```

9. Install PFS using the `pfs` CLI as shown below where the `-m` flag is the path to the previously relocated `manifest.yaml` file. The `--node-port` option is required for access to Kubernetes services via NodePort rather than LoadBalancer.

```
pfs system install -m pfs-relocated/manifest.yaml --node-port
```

   After the install completes, you should see pods similar to the output below.

```
NAMESPACE        NAME                                  READY   STATUS   RESTARTS   AGE
istio-system     istio-citadel-679d658cff-ndr2q              1/1     Running   0          18m
istio-system     istio-egressgateway-7f5996f689-jf7wm        1/1     Running   0          18m
istio-system     istio-galley-7d699c7667-f6l5s               1/1     Running   0          18m
istio-system     istio-ingressgateway-dc44b97bd-c68pp        1/1     Running   0          18m
istio-system     istio-pilot-57b76f975d-q7bnd                2/2     Running   0          18m
istio-system     istio-policy-7c69978c7b-rqbgw               2/2     Running   0          18m
istio-system     istio-sidecar-injector-df9f4df88-mx8n8      1/1     Running   0          18m
istio-system     istio-telemetry-765688ffc9-6b7vd            2/2     Running   0          18m
istio-system     knative-ingressgateway-68449b7bc4-2n2zz     1/1     Running   0          15m
knative-build    build-controller-f46b45b8f-zkms2            1/1     Running   0          15m
knative-build    build-webhook-78d5b65766-v8jqv              1/1     Running   0          15m
knative-eventing   eventing-controller-5bd99c655d-xpzlp      1/1     Running   0          15m
knative-eventing   stub-clusterbus-dispatcher-7b9cd785ff-26qjd   2/2     Running   0          12m
knative-eventing   webhook-7cc49cdd87-7svwv                  1/1     Running   0          15m
knative-serving   activator-7fb99457fd-bv5ps                 2/2     Running   0          15m
knative-serving   activator-7fb99457fd-jcm7l                 2/2     Running   0          15m
knative-serving   activator-7fb99457fd-rsk59                 2/2     Running   0          15m
knative-serving   autoscaler-8674d7bf9-sxxcd                 2/2     Running   0          15m
knative-serving   controller-8585468b4c-z8hz7                1/1     Running   0          15m
knative-serving   webhook-75ccbc6b57-2bp88                   1/1     Running   0          15m
kube-system      coredns-576cbf47c7-xqnkd                    1/1     Running   0          1h
kube-system      etcd-minikube                               1/1     Running   0          1h
kube-system      kube-addon-manager-minikube                 1/1     Running   0          1h
kube-system      kube-apiserver-minikube                     1/1     Running   0          1h
kube-system      kube-controller-manager-minikube            1/1     Running   0          1h
kube-system      kube-proxy-djb7j                            1/1     Running   0          1h
kube-system      kube-scheduler-minikube                     1/1     Running   0          1h
kube-system      kubernetes-dashboard-5bb6f7c8c6-t6vkt       1/1     Running   0          1h
kube-system      registry-dvb89                              1/1     Running   0          1h
kube-system      storage-provisioner                         1/1     Running   0          1h
```

PFS is now installed. Next you need to prepare one or more namespaces for your functions.

Prepare a Namespace

Use the `pfs` CLI to initialize PFS resources in a Kubernetes namespace. Pass the path to the previously relocated manifest file using the `-m` flag. The command below initializes the `default` namespace.

```
pfs namespace init default -m pfs-relocated/manifest.yaml --no-secret
```

You can now create your first function.

Build from local-path

You can use PFS to build functions from source in a local directory, instead of first committing the code to a repo on GitHub. Currently this requires that you use a non-private registry for the relocated images. This is the case when using minikube with the registry addon.

> This capability will be extended to work with private registries in a future release.

Building from a local directory requires a couple of environment variables to be set.

Run the following using the path to the previously relocated manifest file:

```
export PFS_BUILDER_IMAGE=`grep -o "$REGISTRY/$REGISTRY_USER/projectriff-builder.*" \
  pfs-relocated/image-manifest.yaml | awk -F": " '{print $1}'`

export PFS_PACKS_RUN_IMAGE=`grep -o "$REGISTRY/$REGISTRY_USER/packs-run.*" \
  pfs-relocated/image-manifest.yaml | awk -F": " '{print $1}'`
```

If you like, you can capture the values of the environment variables `PFS_BUILDER_IMAGE` and `PFS_PACKS_RUN_IMAGE` in a file as follows:

```
echo "export PFS_BUILDER_IMAGE=$PFS_BUILDER_IMAGE" > pfs-local-path-env
echo "export PFS_PACKS_RUN_IMAGE=$PFS_PACKS_RUN_IMAGE" >> pfs-local-path-env
```

You can set the environment variables whenever you need them by sourcing the file:

```
source pfs-local-path-env
```

To build a function from a local directory, use `--local-path` instead of `--git-repo` .

The output from the build will log a message saying “Pulling builder image” and “(use −−no−pull flag to skip this step)”. This is logged by the pack library that the pfs CLI is using and is not exposed to be set directly by the user doing the build with the pfs CLI. The advantage of always pulling the builder image is that you get the latest version that could include important fixes.

For example, assuming you are in a directory with a file called `square.js` :

```
// square.js
module.exports = x => x ** 2;
```

then you can run this to create the function:

```
pfs function create square \
  --local-path . \
  --artifact square.js \
  --image registry.kube-system.svc.cluster.local/testuser/square \
  --verbose
```

and run this to update the function and deploy a new revision:

```
pfs function update square \
  --local-path .
```

Uninstalling PFS

Page last updated:

This topic describes how to uninstall Pivotal Function Service (PFS).

Requirements

- The `pfs` CLI has been downloaded and installed.
- The `kubectl` CLI has been installed and configured to point your Kubernetes cluster where PFS is installed.

Using the pfs CLI to uninstall PFS

To remove Istio and Knative from your Kubernetes cluster:

```
pfs system uninstall
```

```
Are you sure you want to uninstall the pfs system? [y/N]: y
Removing Knative for pfs components
Deleting CRDs for knative.dev
Deleting clusterrolebindings prefixed with knative-
Deleting clusterrolebindings prefixed with build-controller-
Deleting clusterrolebindings prefixed with eventing-controller-
Deleting clusterrolebindings prefixed with clusterbus-controller-
Deleting clusterroles prefixed with knative-
Deleting resources defined in: knative-eventing
Deleting resources defined in: knative-serving
Deleting resources defined in: knative-build
Deleting resources defined in: knative-monitoring
Namespace "knative-monitoring" was not found
Do you also want to uninstall Istio components? [y/N]: y
Removing Istio components
Deleting CRDs for istio.io
Deleting clusterrolebindings prefixed with istio-
Deleting clusterroles prefixed with istio-
Deleting resources defined in: istio-system
Deleting horizontalpodautoscaler.autoscaling/istio-pilot resource

pfs system uninstall completed successfully
```

To force an uninstall of both Istio and Knative without prompting:

```
pfs system uninstall --force --istio
```

See `riff system uninstall --help` for additional options and information.

The pfs Command Line Interface (CLI)

Page last updated:

This topic describes how to use the `pfs` command line interface (CLI).

To install the `pfs` CLI, see Download PFS from Pivotal Network.

Command Help

Most `pfs` commands take the following form:

```
pfs COMMAND ACTION NAME [--option option-value]
...
```

For example:

```
pfs function create square \
  --git-repo https://github.com/projectriff-samples/node-square \
  --artifact square.js \
  --image gcr.io/$GCP_PROJECT/square:v1 \
  --namespace default
```

```
pfs service delete square
```

Online help is available with the flag `--help` or `-h`. Run `pfs -h` to get started and `pfs COMMAND -h` for command-specific help.

The `pfs` CLI provides commands for interacting with functions, eventing, image relocation, installation, and more.

Functions and Knative Services

This section describes the `pfs function` and `pfs service` commands.

- `pfs function create NAME`
  Create a new Knative service by building a function from source.
- `pfs function update NAME`
  Trigger a function build to generate a new revision for the service.

Note: After you create a function, you can continue to interact with the Knative service using `pfs service` commands.

- `pfs service create NAME`
  Create a new Knative service using a specified container image.
- `pfs service delete NAME`
  Delete an existing Knative service (including those built from functions).
- `pfs service invoke NAME`
  Invoke a Knative service.
- `pfs service list`
  List Knative services.
- `pfs service status NAME`
  Display the status of a Knative service.
- `pfs service update NAME`
  Create a new revision for a Knative service, with updated attributes.

Eventing

This section describes the `pfs channel` and `pfs subscription` commands.

- `pfs channel create NAME`
  Create a new channel on a bus or a cluster bus.
- `pfs channel delete NAME`
  Delete an existing channel.
- `pfs channel list`
  List channels.
- `pfs subscription create NAME`
  Create a new subscription, binding a service to an input channel.
- `pfs subscription delete NAME`
  Delete an existing subscription.
- `pfs subscription list`
  List existing subscriptions.

For more information, see Eventing Channels.

Image Relocation

This section describes the `pfs image` commands.

- `pfs image relocate`
  Relocate a PFS distribution to use images from a specified registry.
- `pfs image push`
  Push docker images from files in a distribution to a registry.

For more information, see Relocate Container Images.

Install

This section describes PFS installation commands.

- `pfs system install`
  Install PFS and Knative system components using the manifest from a distribution.
- `pfs system uninstall`
  Remove PFS and Knative system components.
- `pfs namespace init`
  Initialize a Kubernetes namespace using the manifest from a distribution.

For more information, see Installing PFS.

Miscellaneous

This section describes miscellaneous PFS commands.

- `pfs version`
  Print version information about `pfs`.
- `pfs completion`
  Generate shell completion scripts for `pfs`.

Java Functions

Page last updated:

This topic provides an overview of how to write Java functions for Pivotal Function Service (PFS).

Requirements

- PFS has been installed and you have prepared a namespace.
- The `pfs` CLI has been installed.
- You have set `REGISTRY` and `REGISTRY_USER` environment variables.
  For GCR use `gcr.io` and your GCP Project ID.

```
export REGISTRY=gcr.io
export REGISTRY_USER=$(gcloud config get-value core/project)
```

For the registry in Minikube, use `registry.kube-system.svc.cluster.local` and a dummy username.

```
export REGISTRY=registry.kube-system.svc.cluster.local
export REGISTRY_USER=testuser
```

How the Java Function Invoker Works

The Java function invoker is a Spring Boot application which will locate your function based on configuration settings, and invoke the function for each request.

PFS function support for the Java language relies on function code being written using interfaces like `Function<T,R>`, `Supplier<T>`, or `Consumer<T>` from the `java.util.function` package in the Java SE platform.

The implementation can be provided as a plain Java class or as part of a Spring Boot app.

For more in-depth coverage see the riff java-function-invoker.

A Simple Spring Boot Based Function

We recommend using the Spring Initializr to bootstrap your project. Select the `Web` and `Cloud Function` dependencies and generate your project with either Maven or Gradle for the build configuration. After you unzip the generated project you can modify the app and add a `@Bean` definition for your function.

Here we are adding the following `uppercase` function bean:

```
@Bean
public Function<String, String> uppercase() {
    return s -> s.toUpperCase();
}
```

to our Spring Boot app:

src/main/java/functions/UppercaseApplication.java

```
package functions;

import java.util.function.Function;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
public class UppercaseApplication {

  @Bean
  public Function<String, String> uppercase() {
    return s -> s.toUpperCase();
  }

  public static void main(String[] args) {
    SpringApplication.run(UppercaseApplication.class, args);
  }
}
```

Commit your source to a Git repo once the function compiles and tests pass. The source for this function is also available in this sample repo on GitHub.

Use the pfs CLI to Run the Boot Function

Create the `uppercase` function by running the pfs CLI command below. You can replace the sample `git-repo` URL with your own.

```
pfs function create uppercase \
  --git-repo https://github.com/projectriff-samples/java-boot-uppercase.git \
  --image $REGISTRY/$REGISTRY_USER/uppercase \
  --verbose
```

The invoker attempts to detect the function from the function source. Since you have a single function declared with `@Bean` in the Spring Boot app, that is the function that is used. If you have multiple functions in the source then you must specify which one you want to use by providing a `--handler` flag for the `pfs function create` command.

You should now be able to invoke the function service with the `pfs service invoke` command:

```
pfs service invoke uppercase --text -- -w '\n' -d 'welcome to pfs'
```

```
curl 35.239.12.146/ -H 'Host: uppercase.default.example.com' -H 'Content-Type: text/plain' -w '\n' -d 'welcome to pfs'
WELCOME TO PFS
```

Finally, delete the function using the `pfs service delete` command:

```
pfs service delete uppercase
```

A Plain Java Function

For plain Java functions you have to provide your own Maven or Gradle build configuration. After you configure the build you can add a Java class that implements `Function<String, String>`:

src/main/java/functions/Hello.java

```
package functions;

import java.util.function.Function;

public class Hello implements Function<String, String> {

    public String apply(String name) {
        return "Hello " + name;
    }
}
```

Commit your source to a Git repo once the function compiles and tests pass. The source for this function is also available in this sample repo on GitHub.

Using the pfs CLI to Run the Plain Java Function

Create the `hello` function by running the `pfs function create` command below. You can replace the sample `git-repo` URL with your own. When using plain Java functions, you have to specify the function class name by providing a `--handler` flag.

```
pfs function create hello \
  --git-repo https://github.com/projectriff-samples/java-hello.git \
  --handler functions.Hello \
  --image $REGISTRY/$REGISTRY_USER/hello \
  --verbose
```

You should now be able to invoke the function service with the `pfs service invoke` command:

```
pfs service invoke hello --text -- -w '\n' -d 'PFS'
```

```
curl 35.232.242.167/ -H 'Host: hello.default.example.com' -H 'Content-Type: text/plain' -w '\n' -d PFS
Hello PFS
```

To clean up, for example after a failure, delete the function using the `pfs service delete` command:

```
pfs service delete hello
```

Creating a function from local source

> This capability is currently only available when using Minikube with the registry addon. It will be extended to work in other environments in a future release.

First you must prepare your environment for local builds by setting the `PFS_BUILDER_IMAGE` and `PFS_PACKS_RUN_IMAGE` environment variables. See the Installing PFS on Minikube page for detailed instructions.

Here we assume that you have created the `pfs-local-path-env` file in your download directory. Source the file with the environment settings using (if it is in a different location just change the path):

```
source ~/Downloads/pfs-local-path-env
```

To build a function from a local directory, use `--local-path` instead of `--git-repo`.

E.g. if you are in a directory that is a Java project like the hello example above:

```
pfs function create hello \
  --local-path .
  --handler functions.Hello \
  --image $REGISTRY/$REGISTRY_USER/hello \
  --verbose
```

To update the function and deploy a new revision:

```
pfs function update hello \
  --local-path .
```

JavaScript Functions

Page last updated:

This topic provides an overview of how to write JavaScript functions for Pivotal Function Service (PFS). For more in-depth coverage see the riff node-function-invoker on GitHub.

Requirements

- PFS has been installed and you have prepared a namespace.
- The `pfs` CLI has been installed.
- You have set `REGISTRY` and `REGISTRY_USER` environment variables.
  For GCR use `gcr.io` and your GCP Project ID.

```
export REGISTRY=gcr.io
export REGISTRY_USER=$(gcloud config get-value core/project)
```

  For the registry in Minikube, use `registry.kube-system.svc.cluster.local` and a dummy username.

```
export REGISTRY=registry.kube-system.svc.cluster.local
export REGISTRY_USER=testuser
```

Using the pfs CLI to Create a JavaScript Function

This example uses the sample `hello.js` function from GitHub. It consists of a single JavaScript file named `hello.js` with the following code:

```
module.exports = x => `hello ${x}`
```

Create a `hello` function by running the CLI command below:

```
pfs function create hello \
--git-repo https://github.com/projectriff-samples/hello.js \
--artifact hello.js \
--image $REGISTRY/$REGISTRY_USER/hello \
--verbose
```

The CLI output should show that the node invoker was selected (excerpt below), based on the value of the `--artifact` flag above.

```
-----> Process types:
      web:      node /workspace/io.projectriff.riff/riff-invoker-node/server.js
      function: node /workspace/io.projectriff.riff/riff-invoker-node/server.js
```

You should now be able to invoke the function service with the `pfs service invoke` command:

```
pfs service invoke hello --text -- -w '\n' -d 'PFS'
```

```
curl 35.205.116.145/ -H 'Host: hello.default.example.com' -H 'Content-Type: text/plain' -w '\n' -d PFS
hello PFS
```

Creating a function from local source

> This capability is currently only available when using Minikube with the registry addon. It will be extended to work in other environments in a future release.

First you must prepare your environment for local builds by setting the `PFS_BUILDER_IMAGE` and `PFS_PACKS_RUN_IMAGE` environment variables. See the Installing PFS on Minikube page for detailed instructions.

Here we assume that you have created the `pfs-local-path-env` file in your download directory. Source the file with the environment settings using (if it is in a different location just change the path):

```
source ~/Downloads/pfs-local-path-env
```

To build a function from a local directory, use `--local-path` instead of `--git-repo`.

E.g. if you are in a directory with a file called `hello.js` just like the one above:

```
pfs function create hello \
--local-path . \
--artifact hello.js \
--image $REGISTRY/$REGISTRY_USER/hello \
--verbose
```

To update the function and deploy a new revision:

```
pfs function update wordcount \
  --local-path .
```

How the Node Function Invoker Works

At runtime, the node function invoker will `require()` the target function module. This module must export the function to invoke.

```
// square
module.exports = x => x ** 2;
```

The first argument is the triggering message's payload and the returned value is the resulting message's payload.

Async

Asynchronous work can be completed by defining either an `async function` or by returning a `Promise`.

```
// async
module.exports = async x => x ** 2;

// promise
module.exports = x => Promise.resolve(x ** 2);
```

Streams (Experimental)

Streaming functions can be created by setting the `$interactionModel` property on the function to `node-streams`. The function is invoked with two arguments: an `input` Readable Stream and an `output` Writeable Stream. Both streams are object streams. Any value returned by the function is ignored, and new messages must be written to the output stream.

```
// echo.js
module.exports = (input, output) => {
  input.pipe(output);
};
module.exports.$interactionModel = "node-streams";
```

Any npm package that works with Node Streams can be used.

```
// upperCase.js
const miss = require("mississippi");

const upperCaser = miss.through.obj((chunk, enc, cb) => {
  cb(null, chunk.toUpperCase());
});

module.exports = (input, output) => {
  input.pipe(upperCaser).pipe(output);
};
module.exports.$interactionModel = "node-streams";
```

The `Content-Type` for output messages can be set with the `$defaultContentType` property. By default, `text/plain` is used. For request-reply function, the `Accept` header is used, but there is no `Accept` header in a stream.

```
// greeter.js
const miss = require("mississippi");

const greeter = miss.through.obj((chunk, enc, cb) => {
  cb(null, {
    greeting: `Hello ${chunk}!`
  });
});

module.exports = (input, output) => {
  input.pipe(greeter).pipe(output);
};
module.exports.$interactionModel = "node-streams";
module.exports.$defaultContentType = "application/json";
```

Messages Versus Payloads

By default, functions accept and produce payloads. Functions that need to interact with headers can instead opt to receive and/or produce messages. A message is an object that contains both headers and a payload. Message headers are a map with case-insensitive keys and multiple string values.

Since JavaScript and Node have no built-in type for messages or headers, PFS uses the @projectriff/message npm module. To use messages, functions install the `@projectriff/message` package:

```
npm install --save @projectriff/message
```

Receiving Messages

```
const { Message } = require('@projectriff/message');

// a function that accepts a message, which is an instance of Message
module.exports = message => {
    const authorization = message.headers.getValue('Authorization');
    ...
};

// tell the invoker the function wants to receive messages
module.exports.$argumentType = 'message';

// tell the invoker to produce this particular type of message
Message.install();
```

Producing Messages

```
const { Message } = require("@projectriff/message");

const instanceId = Math.round(Math.random() * 10000);
let invocationCount = 0;

// a function that produces a Message
module.exports = name => {
  return Message.builder()
   .addHeader("X-PFS-Instance", instanceId)
   .addHeader("X-PFS-Count", invocationCount++)
   .payload(`Hello ${name}!`)
   .build();
};

// even if the function receives payloads, it can still produce a message
module.exports.$argumentType = "payload";
```

Lifecycle

Functions that communicate with external services, like a database, can use the `$init` and `$destroy` lifecycle hooks on the function. These methods are invoked once per function invoker instance, whereas the target function may be invoked multiple times within a single function invoker instance.

The `$init` method is guaranteed to finish before the main function is invoked. The `$destroy` method is guaranteed to be invoked after all of the main functions are finished.

```
let client;

// function
module.exports = async ({ key, amount }) => {
  return await client.incrby(key, amount);
};

// setup
module.exports.$init = async () => {
  const Redis = require("redis-promise");
  client = new Redis();
  await client.connect();
};

// cleanup
module.exports.$destroy = async () => {
  await client.quit();
};
```

The lifecycle methods are optional and should only be implemented when needed. The hooks may be either traditional or async functions. Lifecycle functions have up to 10 seconds to complete their work or the function invoker aborts.

Command Functions

Page last updated:

This topic introduces Pivotal Function Service (PFS) command functions and provides an example built using the pfs CLI.

Command functions are functions that can be executed by themselves in PFS. They are either shell scripts, with the executable permission set, or a compiled binary program that doesn't require any particular runtime or dynamic library to be present.

Command functions are handled by the riff command-function-invoker on GitHub.

Requirements

- PFS has been installed and you have prepared a namespace.
- The `pfs` CLI has been installed.
- You have set `REGISTRY` and `REGISTRY_USER` environment variables.
  For GCR use `gcr.io` and your GCP Project ID.

  ```
  export REGISTRY=gcr.io
  export REGISTRY_USER=$(gcloud config get-value core/project)
  ```

  For the registry in Minikube, use `registry.kube-system.svc.cluster.local` and a dummy username.

  ```
  export REGISTRY=registry.kube-system.svc.cluster.local
  export REGISTRY_USER=testuser
  ```

How Command Functions Work

Command functions use standard input (stdin) and standard output (stdout) streams.

For each invocation, functions are expected to read stdin until the end of the stream (EOF) and provide a result on stdout.

Correct function execution is assumed if the exit code is zero. Any other value indicates an error.

Using the pfs CLI to Create a Command Function

This example uses the sample command-wordcount function from GitHub. It consists of a single executable file named `wordcount.sh` with the following content:

```
#!/bin/bash

tr ' ' '\n' | sort | uniq -c | sort -n
```

Create a `wordcount` function by running the CLI command below:

```
pfs function create wordcount \
--git-repo https://github.com/projectriff-samples/command-wordcount \
--artifact wordcount.sh \
--image $REGISTRY/$REGISTRY_USER/wordcount \
--verbose
```

The CLI output should show that the command invoker was selected.

For example:

```
-----> Process types:
web:      /workspace/io.projectriff.riff/riff-invoker-command/command-function-invoker
function: /workspace/io.projectriff.riff/riff-invoker-command/command-function-invoker
```

Now, let's count the occurrences of words in the US Declaration of Independence by posting the contents of the document to the function using `pfs service invoke`:

```
curl -s https://www.constitution.org/usdeclar.txt \
    | pfs service invoke wordcount --text -- -d @-
```

```
...
20 in
20 their
26 our
56 and
64 to
76 the
80 of
372
```

Creating a function from local source

This capability is currently only available when using Minikube with the registry addon. It will be extended to work in other environments in a future release.

First you must prepare your environment for local builds by setting the `PFS_BUILDER_IMAGE` and `PFS_PACKS_RUN_IMAGE` environment variables. See the Installing PFS on Minikube page for detailed instructions.

Here we assume that you have created the `pfs-local-path-env` file in your download directory. Source the file with the environment settings using (if it is in a different location just change the path):

```
source ~/Downloads/pfs-local-path-env
```

To build a function from a local directory, use `--local-path` instead of `--git-repo`.

E.g. if you are in a directory with an executable file called `wordcount.sh` just like the one above:

```
pfs function create wordcount \
--local-path . \
--artifact wordcount.sh \
--image $REGISTRY/$REGISTRY_USER/wordcount \
--verbose
```

To update the function and deploy a new revision:

```
pfs function update wordcount \
  --local-path .
```

Using Eventing Channels

Page last updated:

This topic describes how to interact with eventing concepts using the pfs CLI for Pivotal Function Service (PFS).

For more information about eventing, see PFS Concepts - Eventing.

WARNING: The eventing features in this version of PFS, and more generally in Knative, are still under development.

Eventing and Buses

Knative and PFS use the concept of a *bus* to describe the middleware responsible for propagating events. Buses can be made available to all namespaces in a *cluster-bus* or be scoped to each namepace in a regular bus.

During installation PFS provided a default, non-durable cluster bus implementation called `stub`. Developing and installing other bus implementations is beyond the scope of this documentation, so this guide uses the `stub` cluster bus.

Dealing with Channels and Subscriptions

You can create a new channel by using the `pfs channel create` command:

```
pfs channel create numbers --cluster-bus stub
```

Note: What the `pfs channel create` command actually does is bus-specific. In a messaging middleware, it typically creates a topic. In the case of the stub bus, this doesn't do much as a stub is a very direct implementation that dispatches events as they are received, with no persistence.

After a channel is created, you can subscribe one or several functions or services to it using *subscriptions*.

For example:

```
pfs subscription create --channel numbers --subscriber square
```

Unlike most other `pfs` commands, the name of the subscription is optional. The previous example creates a `square` subscription, wiring the input of the `square` function or service to the `numbers` channel.

One of the advantages of channels and eventing is that you can subscribe several functions to a single channel. Here is how you would do it, continuing from the previous examples:

```
pfs subscription create --channel numbers --subscriber cube
```

From there on, every time a number is posted on the `numbers` channel, both the `square` and `cube` functions are invoked.

Chaining

Eventing permits chaining of functions so that the result of one function invocation becomes the input to the next function. This is achieved at subscription time by providing the `reply-to` flag.

For example:

```
pfs subscription create --channel numbers --subscriber square --reply-to squares
```

Now, if another function is subscribed to the `squares` channel, you get a chain:

```
pfs subscription create --channel squares --subscriber greet
```

Posting to Channels

For each channel, there is a private Kubernetes service created that can be used to ingest events. Its DNS name is in the format `<channel-name>-channel.<namespace>.svc.cluster.local` .

Any Knative service can perform an HTTP POST on port 80 of that service, and the bus backing that channel dispatches the event to each subscriber, possibly with retry and catchup for after-the-fact subscriptions.

> **Note**: As stated in the introduction of this page, the eventing part of the PFS API is under active development. The channel and subscription concepts are lower-level primitives and one can expect higher constructs to become available in later versions of PFS.

Listing and Deleting

Both the `pfs subscription` and `pfs channel` commands provide support for listing and deleting created resources.

For example:

```
pfs [channel|subscription] list
```

The above command lists all the channels or subscriptions of a given namespace, while:

```
pfs [channel|subscription] delete NAME
```

deletes a channel or a subscription by its name.

Troubleshooting PFS

Page last updated:

This topic describes how to troubleshoot Pivotal Function Service (PFS).

For general Kubernetes troubleshooting, the kubectl Cheat Sheet is handy.

Watch Pods with the watch Utility

It is useful to monitor your cluster using a utility like `watch` that is provided with many Linux distributions.

For more information, see the `watch` Linux manual page.

To install `watch` on a Mac:

```
brew install watch
```

Run the following command to watch all pods:

```
watch -n 1 kubectl get pod --all-namespaces
```

The command above runs `kubectl get pod --all-namespaces` once per second.

View Function Container Logs

To see a function's log entries use the `kubectl logs` command targeting the `user-container` container of the function pod. The command below will print logs from the `user-container` container of a pod named `hello-00001-deployment-5c64894dbd-pk5k2`

```
kubectl logs hello-00001-deployment-5c64894dbd-pk5k2 -c user-container
```

To find the identity of pods running a PFS function use the `kubectl get pods` command in the appropriate namespace with a label query of the form `riff.projectriff.io/function={name of function}`. The example below shows how the pods running a PFS function created with the name `hello` can be found in the default namespace.

```
kubectl get pods -l riff.projectriff.io/function=hello
```

```
NAME                                    READY   STATUS    RESTARTS   AGE
hello-00001-deployment-5c64894dbd-pk5k2   3/3     Running   0          8m
```

Alternatively, just run `kubectl get pods` in the appropriate namespace and look for pods with a name of the form `{name of function}-{revision number}-{guid}` such as `hello-00001-deployment-5c64894dbd-pk5k2`

Examine Function Pod Details

To see the status and other details of a function's pod, use `kubectl describe po` specifying the pod name and namespace. The example below shows the details of a failed build due to a failure in the analysis step.

```
kubectl describe po square-00001-7g74x
```

```
Name:         square-00001-7g74x
Namespace:      default
...
Init Containers:
  build-step-credential-initializer:
    ...
    Ready:        True
    ...
  build-step-git-source:
    ...
    Ready:        True
    ...
  build-step-prepare:
    ...
    Ready:        True
    ...
  build-step-write-riff-toml:
    ...
    Ready:        True
    ...
  build-step-detect:
    ...
    Ready:        True
    ...
  build-step-analyze:
    ...
    State:       Terminated
      Reason:      Error
      Exit Code:    1
      Started:     Thu, 06 Dec 2018 08:59:21 +0000
      Finished:     Thu, 06 Dec 2018 08:59:22 +0000
    Ready:        False
  build-step-build:
    ...
    State:       Waiting
      Reason:      PodInitializing
    Ready:        False
  build-step-export:
    ...
    State:       Waiting
      Reason:      PodInitializing
    Ready:        False
Containers:
    ...
Conditions:
  Type          Status
  Initialized     False
  Ready          False
  ContainersReady   False
  PodScheduled     True
...
```

To dig deeper, you can then grab the logs for the failed container.

```
kubectl logs square-00001-7g74x -c build-step-analyze
```

```
2018/12/06 08:59:22 Error: failed to access image metadata from image : failed to access manifest gcr.io/testproj/square: DENIED: "Permission denied for \"latest\" from request \"/v2/testproj/square/manifests/late
```

An alternative way of viewing a function pod's details is to use `kubectl get po` with the `-oyaml` flag to produce a YAML dump. The example below again shows the details of the failed build, but with somewhat different details.

```
kubectl get po square-00001-7g74x -oyaml
```

Examine Knative Service Details

To see the status and other details of a Knative service, use `kubectl describe services.serving.knative.dev` specifying the pod name and namespace. The example below shows the details of Knative service named `greet`.

```
kubectl describe services.serving.knative.dev greet
```

Alternatively, use `kubctl get` as in the example below.

```
kubectl get services.serving.knative.dev greet -oyaml
```

Restart a Controller

Sometimes Knative issues are solved by restarting a controller pod. This is as simple as issuing `kubectl delete` specifying the name of the pod and the relevant Knative namespace. Once deleted, the pod should restart automatically.

The example below restarts the Knative serving controller.

```
kubectl delete po controller-f4c59f474-6rwwh -n knative-serving
```

Pivotal

Configuring Google Cloud

Page last updated:

This topic describes how to configure your Google Cloud environment.

Create a Google Cloud Project

A project is required to consume any Google Cloud services, including GKE clusters. When you log into the console you can select or create a project from the dropdown at the top.

Create an environment variable, replacing ??? with your project name.

```
export GCP_PROJECT=???
```

Install the gcloud SDK

Follow the quickstart instructions to install the Google Cloud SDK which includes the `gcloud` CLI. You may need to add the `google-cloud-sdk/bin` directory to your path. Once installed, `gcloud init` will open a browser to start an oauth flow and configure gcloud to use your project.

```
gcloud init
```

Install kubectl

Kubectl is the Kubernetes CLI. It is used to manage minikube as well as hosted Kubernetes clusters like GKE. If you don't already have kubectl on your machine, you can use gcloud to install it.

```
gcloud components install kubectl
```

Configure gcloud

Check your default project, region, and zone, and list your available projects

```
gcloud config list
gcloud projects list
```

If necessary change the default project.

```
gcloud config set project $GCP_PROJECT
```

To list the available compute regions and zones:

```
gcloud compute zones list
```

To change the default region and zone:

```
gcloud config set compute/region us-east4
gcloud config set compute/zone us-east4-c
gcloud config list compute/
```

Enable the necessary APIs for gcloud. The following are required for GCR and GKE. You may also need to enable billing for your project.

```
gcloud services enable \
  cloudapis.googleapis.com \
  container.googleapis.com \
  containerregistry.googleapis.com
```

Configure Docker Credential Helper for gcr

Configure your docker environment to push images to Google Container Registry:

```
gcloud components install docker-credential-gcr
docker-credential-gcr configure-docker
```

This is only needed once for a new install of docker, see setting-up-a-docker-registry.
