Pivotal

PRODUCT DOCUMENTATION

PCF Tile Developer Guide

v2.0

Published: 8 April 2019

Table of Contents

PCF Tile Developer Guide

Note: PCF Tile Developer Guide v2.0 is not designed for use with the current versions of Ops Manager. For your tiles to stay up to date with the latest software, features, and security updates, use the latest version of the PCF Tile Developer Guide.

Page last updated:

This topic exists to help Pivotal Cloud Foundry (PCF) Partners learn the high-level process of building and publishing a tile on Pivotal Network.

For advanced developers with previous experience building tiles, see Product Template Reference and Development Workflow Reference.

What is a Tile?

Tiles are packaged software that can be integrated into PCF. PCF operators can install tiles on PCF. PCF developers can use these services once they are installed.

Tile developers can publish tiles on Pivotal Network, where services and tiles are available for download.

Tile Structure

Tiles are packaged as compressed files with a `.pivotal` file extension. These compressed files require three subdirectories: `metadata`, `migrations`, and `releases`.

When you package your software with Tile Generator, it generates these subdirectories for you. You can perform different actions within each subdirectory:

Directory	Description
`metadata`	Configure settings for your software in a YAML file.
`migrations`	Track changes across different releases in a `.js` file. Only tiles with multiple releases use this subdirectory. Do not modify the files in this subdirectory during your first tile release.
`releases`	Deploy your service source code and other inputs for your build, such as a BOSH release.

Why Build a Tile?

There are multiple reasons to build and publish a tile on Pivotal Network. Tiles can help you:

- Find the widest possible audience for your service.
- Join a growing ecosystem that can easily integrate your service.
- Enable operators and app developers to interact with your service in an accessible and standardized way.

Building Your First Tile

There are two options for building your first tile. You can either attend partner days or develop independently. However, Pivotal strongly recommends attending Partner Days for hands-on guidance.

Attending Partner Days

Partner Days are the single best resource to introduce you to PCF and tile development. During these three-day workshops, Pivotal and partner Independent Software Vendor (ISV) engineers collaborate to prototype and build a software integration with PCF.

These events streamline your development process by providing hands-on guidance, giving you a head start for publishing a tile on Pivotal Network. The workshop is free for all Pivotal partners.

Pivotal recommends any interested partner to register for Partner Days. If you are not a Pivotal partner yet, you can sign up for the partner program.

You can see footage of previous Partner Days in this brief YouTube video.

Developing Independently

If you want to build a tile without attending Partner Days, follow the procedure below to minimize the learning curve for tile development.

Creating a tile is a complex process and can be time consuming to complete on your own. You can message the Pivotal Partners Slack channel with questions if you register for the Pivotal Partner program.

1. Decide What to Build

If you use Tile Generator to package your software you also need to determine the inputs you need to build before development. Inputs for your tile also depend on the service you are providing.

Before starting tile development, see How PCF and PCF Services Work.

Depending on what you build, you might need to install the following tools:

- Tile Generator: Used to package your software into a tile.
- BOSH Command Line Interface (CLI): A CLI for running BOSH commands. You need BOSH commands to run Tile Generator.
- Cloud Foundry Command Line Interface (cf CLI): A CLI for deploying and managing apps on Cloud Foundry. If you are developing on Cloud Foundry, you use cf CLI when building your tile.
- Kubernetes Command Line Tool (kubectl): A command line interface for deploying and managing apps on Kubernetes. If you are developing on Kubernetes, you use kubectl when building your tile.
- CF Dev (optional): A lightweight PCF installation for deploying and debugging apps locally. You can use CF Dev if you want to run PCF on your local workstation.

2. Generate a Tile

Tile Generator is a tool that simplifies the building process for tiles. To use Tile Generator, upload your software components, such as the service broker, buildpack, and Docker image, and the tool generates a base tile.

For information on setting up Tile Generator and building a base tile, see Tile Generator.

3. Test Your Tile

Before you publish your tile, you can test it manually using a Partner Integration Environment (PIE). In PIE you can see how the tile functions on an IaaS, such as Amazon Web Services (AWS) or Google Cloud Platform (GCP). You can upload, configure, and install your tile in PIE just like an operator would.

To gain access to your PIE, reach out to your contact at Pivotal or register as a partner.

If you already have access to your PIE, for information on how to log in, see Shared PCF Development Environments.

4. Document Your Tile

When you are ready to publish your tile, write documentation. Documentation is valuable for operators who use your tile.

For more information on how to write and publish documentation for your tile, see Tile Documentation.

5. Publish Your Tile on Pivotal Network

Contact your Pivotal representative who can guide you through the process of uploading your tile to Pivotal Network. When you upload your tile to Pivotal Network, it becomes available for operators and developers to do the following:

Audience	Benefits
	• Download and install your service as a tile.

Operators	• Configure your service using a UI. • Update your service with a single click.
Developers	• See your service on Pivotal Network. • Select service plans to which they would like to subscribe. • Create instances of your service and call them from their apps. • Support a continuous and fast development cycle.

For information on the release cycle for Partner tiles, see Partner Software Release Cycle.

Contact Us

If you want to learn more about the Pivotal ISV Partner Program or request assistance with your integration project, see Contact Us.

PCF v2.0 Partners Release Notice

Page last updated:

This topic describes the changes that Pivotal Cloud Foundry (PCF) v2.0 introduces which may be relevant to partner service tiles.

Colocated Errands

Tile authors can configure the errands defined in their product tile to run on existing virtual machines (VMs) in a deployment. Colocated errands run faster than traditional errands and use fewer resources, including disk and IP space.

See Tile Errands for more information.

Runtime Configs

Tile authors can include `runtime_configs` as a top-level key in tile metadata to define global deployment configurations. Named runtime config settings apply to all VMs in a deployment.

Ops Manager v2.0.0 supports defining any number of runtime configs in an existing tile. Tile authors can also create a tile that only includes a runtime config and does not define any job types or errands.

See Managing Runtime Configs for more information.

On-Demand Disk and VM Type Defaults

On-demand service tiles have a configuration pane for each service plan. Operators use drop-down menus on the plan configuration pane to set the VM type and persistent disk type for each instance of that plan.

Ops Manager v2.0.0 allows tile authors to specify the default values for VM types and persistent disk types in their tile's plan configuration pane.

See Configuring Disk and VM Type Defaults for On-Demand Service Tiles for more information.

BOSH DNS

Ops Manager v2.0.0 introduces BOSH DNS as a runtime config colocated on every VM in a deployment. Since BOSH DNS is a beta feature in PCF v2.0, operators can opt out of the feature in this release.

Tile authors can use the new `$director.dns_release_present` accessor in tile metadata to expose the `disable_dns_release` setting on the BOSH Director. If an operator chooses to opt out of BOSH DNS, `disable_dns_release` is set to `true`.

See Property Reference for more information.

Network Name Accessors

Ops Manager v2.0.0 adds new accessors to return network information, including the network name for a product and the top-level domain (TLD) of the BOSH Director. Ops Manager uses these values when constructing BOSH DNS aliases.

The following manifest snippet returns the names of the networks where the products are installed:

```
my_network_name: (( .network_name ))
other_network_name: (( ..other_product.network_name ))
```

See the `network_name` section in Property Blueprint Reference for more information.

The following manifest snippet returns the BOSH Director TLD:

```
bosh_tld: (( $director.tld ))
```

The snippet above returns the string `bosh`.

See Dollar Contexts in the *Property Blueprint Reference* topic for more information.

BOSH Metrics Server UAA Credentials

PCF now forwards BOSH health metrics generated for all VMs in a deployment to the Loggregator Firehose by default.

To support this feature, Ops Manager v2.0.0 colocates the new BOSH Metrics Server on the BOSH Director and includes a UAA client with the correct authorities and scopes.

To access BOSH Metrics Server UAA credentials, tile authors can use the following two accessors:

- `(( $director.bosh_metrics_forwarder_client_name ))` returns the name of the client.
- `(( $director.bosh_metrics_forwarder_client_secret ))` returns the value of the auto-generated client secret.

Named Manifests for Collection

Breaking Change: The `current_record` property is now reserved. You can no longer create a new property named `current_record`.

Tile authors can specify a property for collection within the `named_manifest` section of tile metadata. Use the `current_record` property within a collection record to refer to other properties in the same record. For example:

```
- name: collection-job
  type: collection
  configurable: true
  property_blueprints:
    - name: blueprint-name
      type: string
  named_manifests:
  - name: example-manifest
    manifest: |
      name:  (( current_record.blueprint-name.value ))
```

See the `named_manifest` section of the Product Template Reference topic for more information.

Pivotal Application Service Tile Property Changes

Note: Elastic Runtime has been renamed Pivotal Application Service.

Properties in the Pivotal Application Service (PAS) tile have changed. Tile developers must change any `(( ..cf.PROPERTY.NAME ))` calls accordingly if their tiles access PAS property values.

The following tables list the properties that Pivotal removed, added, renamed, and retyped between PAS v1.12 and v2.0:

Removed Properties
`.diego_cell.dns_servers`
`.doppler.shared_secret_credentials`
`.properties.networking_point_of_entry`
`.properties.secure_diego_communication`

Added Properties
`.properties.cf_networking_enable_space_developer_self_service`
`.properties.container_networking_interface_plugin`

.properties.credhub_database
.properties.credhub_database.external.host
.properties.credhub_database.external.password
.properties.credhub_database.external.port
.properties.credhub_database.external.tls_ca
.properties.credhub_database.external.username
.properties.credhub_database_name
.properties.credhub_key_encryption_passwords
.properties.credhub_tls
.properties.haproxy_client_certificate
.properties.routing_custom_ca_certificates
.properties.secure_service_instance_credentials
.properties.syslog_rule
.uaa.cc_service_key_credentials
.uaa.container_networking_interface_client_credentials
.uaa.services_credhub_credentials

Renamed Properties	
v1.12 Name	**v2.0 Name**
.diego_cell.garden_network_mtu	.properties.container_networking_interface_plugin.silk.network_mtu
.properties.container_networking_log_traffic	.properties.container_networking_interface_plugin.silk.enable_log_traffic
.properties.container_networking_log_traffic.enable.iptables_accepted_udp_logs_per_sec	.properties.container_networking_interface_plugin.silk.iptables_accepted_udp_logs_per_sec
.properties.container_networking_log_traffic.enable.iptables_denied_logs_per_sec	.properties.container_networking_interface_plugin.silk.iptables_denied_logs_per_sec
.properties.container_networking_network_cidr	.properties.container_networking_interface_plugin.silk.network_cidr
.properties.container_networking_vtep_port	.properties.container_networking_interface_plugin.silk.vtep_port
.properties.router_forward_client_cert	.properties.routing_tls_termination
.properties.routing_frontend_idle_timeout	.router.frontend_idle_timeout
.push-apps-manager.accent_color	.properties.push_apps_manager_accent_color
.push-apps-manager.company_name	.properties.push_apps_manager_company_name
.push-apps-manager.currency_lookup	.properties.push_apps_manager_currency_lookup
.push-apps-manager.display_plan_prices	.properties.push_apps_manager_display_plan_prices
.push-apps-manager.enable_invitations	.properties.push_apps_manager_enable_invitations
.push-apps-manager.favicon	.properties.push_apps_manager_favicon
.push-apps-manager.footer_links	.properties.push_apps_manager_footer_links
.push-apps-manager.footer_text	.properties.push_apps_manager_footer_text
.push-apps-manager.global_wrapper_bg_color	.properties.push_apps_manager_global_wrapper_bg_color
.push-apps-manager.global_wrapper_footer_content	.properties.push_apps_manager_global_wrapper_footer_content
.push-apps-manager.global_wrapper_header_content	.properties.push_apps_manager_global_wrapper_header_content
.push-apps-manager.global_wrapper_text_color	.properties.push_apps_manager_global_wrapper_text_color
.push-apps-manager.logo	.properties.push_apps_manager_logo
.push-apps-manager.marketplace_name	.properties.push_apps_manager_marketplace_name
.push-apps-manager.nav_links	.properties.push_apps_manager_nav_links
.push-apps-manager.product_name	.properties.push_apps_manager_product_name
.push-apps-manager.square_logo	.properties.push_apps_manager_square_logo

Properties Moved to CredHub	
PAS 1.12 Name	CredHub Name
`.autoscaling.broker_credentials`	`deploy-autoscaling-broker-credentials`
`.autoscaling.encryption_key`	`deploy-autoscaling-encryption-key`
`.backup-prepare.backup_encryption_key`	`backup-prepare-backup-encryption-key`
`.diego_database.bbs_encryption_passphrase`	`diego-db-bbs-encryption-passphrase`
`.nats.credentials`	`nats-credentials`
`.nfs_server.blobstore_secret`	`nfs-server-blobstore-secret`
`.notifications.encryption_key`	`deploy-notifications-encryption-key`
`.properties.consul_encrypt_key`	`consul-encryption-key`
`.push-pivotal-account.encryption_key`	`push-pivotal-account-encryption-key`
`.push-usage-service.secret_token`	`push-usage-service-secret-token`
`.router.route_services_secret`	`router-route-services-secret`

Product Dependency Syntax

Tile authors can specify product version dependencies in tile metadata using `~>`. Ops Manager interprets this operator based on the context in the metadata. For example:

```
- name: cf
  version: "~> 1.8"
- name: example-product
  version: "~> 1.12.1"
```

If the version number contains only two segments, Ops Manager interprets `~>` as `>=`. In the example above, this includes all versions of `cf` later than `1.8`.

If the version number contains more than two segments, Ops Manager evaluates `~>` for the final segment. In the example above, this includes only versions `1.12.x` of `example-product`.

Consul Version Requirement

To ensure compatibility with PCF v2.0, tiles using consul must update to consul agent v174 or later. This change supports the effort to transition from consul to BOSH DNS for service discovery.

Syslog Formatting Requirement

Pivotal requires that PCF v2.0 compatible service tile components emit syslog messages according to the standard documented in Log Format for PCF Components.

Requirement to Use BOSH Links for Credentials and IP Addresses

To ensure compatibility with PCF v2.0, tiles must use BOSH links to retrieve IP addresses and credentials from other components.

- For credentials, BOSH links allows your service to receive credentials without the security risk of them being exposed in the BOSH deployment manifest.
- For IP addresses, BOSH links allows your service to receive IP addresses assigned by BOSH instead of Ops Manager. This enables PCF users to do more automation with Ops Manger-generated manifests because IP address management (IPAM) will not be done by Ops Manager, removing the potential conflict from changes made through automation.

Procedure for Using BOSH Links

1. If you use tile-generator to build your tile, update to the latest version and rebuild.
2. If you define BOSH jobs in your tile, use `dynamic_ips: 1` and `static_ips: 0` for each job. This uses BOSH for IPAM instead of Ops Manager.

   Note: Despite the property name *dynamic*, BOSH keeps your job at the same IP address unless that is not possible, such as when the operator changes the IP address range and that IP address is no longer available.
3. If a BOSH release in your tile needs the IP address of another component, consume its BOSH link.
4. If other components need the IP address of your BOSH job, provide a BOSH link.
5. The following properties are not present in PCF v2.0:

   - `..cf.doppler.shared_secret_credentials`
   - `..cf.nats.credentials.identity`
   - `..cf.nats.credentials.password`
   - `..cf.properties.consul_encrypt_key`

   If your tile uses any of these properties, you can get them from a BOSH link provided by its respective job. See the following table:

Property	BOSH Link
`.cf.doppler.shared_secret_credentials`	No longer needed
`..cf.nats.credentials.identity`	`nats`
`..cf.nats.credentials.password`	`nats`
`..cf.properties.consul_encrypt_key`	`consul_common`

   For implementation details, refer to the with-link examples in our pcf-examples repository and the Tile Generator documentation. For more background and context on BOSH links, see BOSH Links: Why and How and the official BOSH links documentation.

UAA Endpoint Changes

If your tile uses the `/oauth/token` and `/check_token` endpoints of the UAA API, you must ensure you are using HTTP POST with body instead of HTTP GET requests. Using HTTP GET is no longer supported as it presents a security risk due to the access logs recording query parameters and exposing the UAA token.

BOSH Releases: Use SHA-2 Hash

You must ensure that your tile signs its components using SHA-2, as SHA-1 has been proven insecure. Follow these steps:

1. If you use tile-generator to build your tile, update to the latest version and rebuild.
2. If you create a BOSH release for your tile, use the `--sha2` flag of the `bosh create-release` command.
3. If you include third-party BOSH releases in your tile, update those to newer versions that are signed with SHA-2 hash.

Tile Basics

Page last updated:

This section gives a high-level overview of how tiles, Pivotal Cloud Foundry (PCF), and PCF service brokers work together.

Cloud Foundry Service Brokers and PCF Tiles

Service brokers let developers create service instances in their development spaces that they can call from their code. To do this, the brokers provide an interface between the Cloud Controller and the add-on software service that they represent. The service can run internal or external to a CF deployment, but the service broker always runs inside the cloud.

The service broker works by providing an API which the Cloud Controller calls to create service instances, bind them to apps, and perform other operations. Cloud Foundry service brokers are implemented as HTTP servers that conform to the service broker API.

In addition to providing an API, a service broker publishes a service catalog that may include multiple service plans, such as a free tier and a metered tier. Brokers register their service plans with the Cloud Controller to populate the Marketplace, which developers access with `cf marketplace` or through the Pivotal Cloud Foundry (PCF) Apps Manager.

On PCF, cloud operators make software services available to developers by finding them on Pivotal Network and then installing and configuring them through a tile interface in the Ops Manager **Installation Dashboard**. Installing a service tile creates a service broker, registers it with the Cloud Controller, and publishes the service plans that the broker offers. Developers can then create service instances in their spaces and bind them to their apps.

See the following topics:

- How PCF and PCF Services Work
- How Tiles Work

How PCF and PCF Services Work

Page last updated:

There are many ways to integrate services with Pivotal Cloud Foundry (PCF). The right one for each service depends on what the service does, and how customer applications consume it. To determine the best way to integrate your service, you'll need a good understanding of PCF concepts like applications, containers, services, brokers, and buildpacks.

This page provides a collection of links to documentation for the most relevant concepts. If you prefer to learn through guided training, ask us about available training options.

General Overview

For general overview of PCF, and the various ways to interact with it, use the following links:

- Cloud Foundry Subsystems provides high-level descriptions of internal functions performed by different PCF components.
- Cloud Foundry Command Line Interface (cf CLI) links to topics that explain how to direct PCF deployment from your local command line.
- Pivotal Ops Manager describes the Ops Manager and Installation Dashboard interfaces, where cloud operators see, install, configure, and deploy service tiles.
- Pivotal Apps Manager describes the Apps Manager interface, where app developers create and configure service instances and bind them to their apps.

Applications

Cloud Foundry is primarily a cloud native application platform. To understand how to integrate your services with Cloud Foundry, you should understand how your customers are using the platform to develop, deploy, and operate their applications.

- Developer Guide explains how to push an app to run on PCF and enable it to use services.
- Logging and Monitoring describes how PCF aggregates and streams logs and metrics from the apps it hosts and from internal system components.

Services

Most value-add integrations are done by exposing your software to customer applications as services. To understand the service concepts, and what a service integration looks like, read the following documentation:

- Services Overview explains how developers provision and use existing services in their apps.
- Cloud Foundry Service Brokers and PCF Tiles briefly describes the two main elements of PCF service integration: the service broker API, which connects the service to PCF internally by taking commands from the Cloud Controller; and the tile, a packaged interface that cloud operators use to install and configure a service within PCF.
- Custom Services explains how service authors package their service as a Managed Service that is available for use by PCF operators and developers, and which runs locally on PCF rather than running remotely.

Buildpacks

When application code is deployed to Cloud Foundry, it is processed by a language-specific buildpack. Language buildpacks provide a convenient integration hook for any service that needs to inspect or embellish application code. Supplying buildpacks also provides a language-agnostic way to inject your code into the application container image.

- Application Staging Process explains how PCF packages and deploys apps in containers with buildpacks so that they can run on multiple VMs interchangeably.
- Language Buildpacks describes the language-specific buildpacks support PCF apps.
- Custom Buildpacks describes how to use supply buildpacks to add dependencies or code without having to change (multiple) language-sepcific buildpacks.

Embedded Agents

Some integrations depend on the ability to inject code into the application container. We refer to these injected components as “container-embedded agents”. Buildpacks provide a mechanism to inject components into the application container image, and the `.profile.d` directory provides a way to start agents before or alongside the customer application.

- Agent Injection with a supply buildpack
- Using .profile.d

Nozzles

Cloud Foundry’s logging system, Loggregator, has a feature named **firehose**. The firehose includes the combined stream of logs from all apps, plus metrics data from Cloud Foundry components, and is intended to be used by operators and administrators.

A nozzle takes this data and forwards it to an external logging and/or metrics solution.

- Loggregator system

How Tiles Work

Page last updated:

Product tiles make it easy for cloud operators to offer new and upgraded software services to developers in a Pivotal Cloud Foundry (PCF) deployment. Pivotal Network distributes these tiles as zipped code directories, with filename extension `.pivotal`, that contain or point to all of the software elements that perform the tile's functions.

This topic explains what each functional element of a tile does and how you create or specify it as input to the Tile Generator tool that creates `.pivotal` files.

This topic also describes the typical structure of a tile directory. This is useful information for modifying generated tiles or legacy tiles that were created without the Tile Generator.

Tile Functions

PCF service tiles perform multiple functions that streamline the use of software services on PCF, including:

- Deploy a service broker that interfaces between the Cloud Controller, PCF's main executive component, and the service.
- Publish a catalog of available service plans to the **Services Marketplace**.
- Define an interface for configuring service properties in Ops Manager.
- Generate a BOSH manifest for deploying instances of the service, populating it with both user-configured and fixed properties.
- Run BOSH errands: deploy errands that set PCF up to run the service when an operator first deploys the service, and delete errands that clean up when an operator deletes the service.
- Define dependencies for the tile, to prevent Ops Manager from installing the service when its dependencies are missing.
- Support one-click installation and upgrading from previous versions.

These functions are described in more detail below.

Service Broker

Service brokers integrate services with PCF by providing an API for the Cloud Controller to create service instances, bind them to apps, and perform other operations. The Service Broker API v2.10 topic specifies requirements for this API.

Each service tile acts as a wrapper for a service broker. Installing the tile creates its service broker, registers it with the Cloud Controller, and publishes the service plans that the broker offers.

You can write a service broker in any language, and it can run anywhere, inside your PCF installation or external. See Example Service Brokers for sample code in Ruby, Java, and Go.

Specify the service broker for a tile in the tile directory's `tile.yml` file, as a package with `type:` set to `app-broker`, `docker-app-broker`, or `external-broker`. The `external-broker` type requires a `uri` value, for the service broker location.

Catalog

Service brokers include catalog metadata that list their service plans. This information publishes to the Marketplace that app developers use to browse and select services.

Developers on either PCF or open-source Cloud Foundry see a plain-text version of the Marketplace by running `cf marketplace`. But PCF also features a graphical Marketplace, and PCF service brokers support this Marketplace with additional catalog metadata fields for display names, logo images, and links to more information and documentation.

Define this catalog metadata for your service by writing your service broker to return the API calls listed in the Catalog Metadata topic.

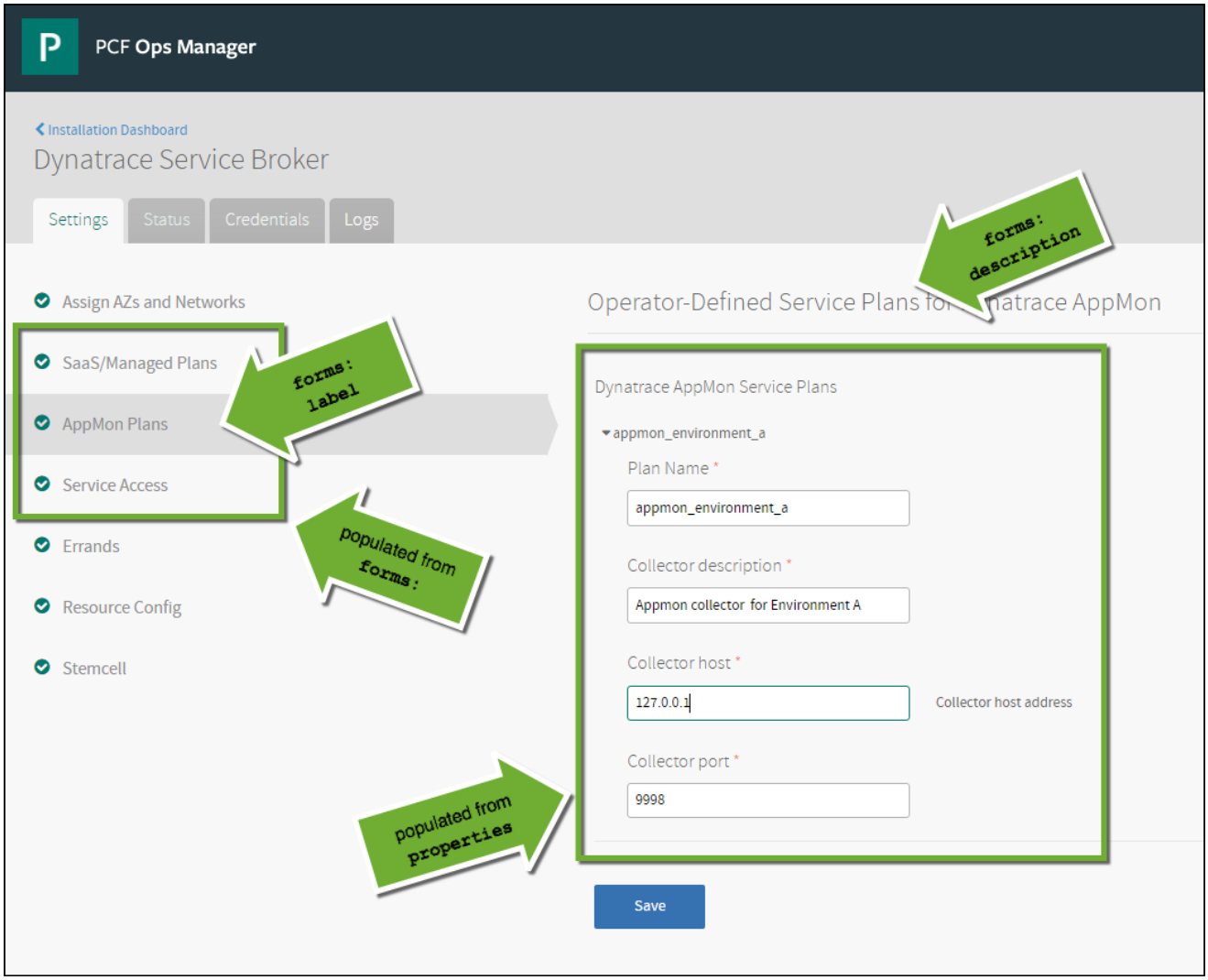
Configuration

In the Ops Manager Installation Dashboard, service tiles present a form-based interface that cloud operators use to configure the service. These

configured properties become part of the BOSH manifest that PCF uses to deploy instances of the service.

You define this configuration interface in the `forms:` section of the `tile.yml` configuration file that you pass to the Tile Generator. Each named form element defines a configuration pane accessible under the tile's **Settings** tab.

A left-side menu lists all configuration panes and indicates with check marks which ones have been configured. The menu lists service-specific panes, defined by the tile developer, between system-level panes like **Assign AZs and Networks** and **Resource Config** that all PCF products and services use.



Each form, or configuration pane, has `label` for the menu text, a `description` to appear up top, and `property_inputs` that define the configuration fields themselves. Construct your `forms` by following the Product Template Reference topic and the Property Blueprint Reference section of the About PCF Tiles topic.

For each property, you can combine specifications for `name`, `type`, `default`, `configurable`, `options`, and `constraints`, under both the Form Properties and Property Blueprints sections of the topic.

Note: In the tile installer `.yml` that Tile Generator creates, form properties appear in two locations: a `form_types` section that defines the contents and layout of the configuration interface, and a `property_blueprints` section that defines the corresponding field value types and constraints.

Tile Appearance

In the Ops Manager Installation Dashboard, your service tile bears an identifying label, description, and logo icon. Specify these at the top of your `tile.yml` configuration file as `label`, `description`, and `icon_file`. The value of `icon_file` should be the name of a 128×128 pixel PNG image.

Fixed Properties

A tile also writes fixed, unconfigurable properties into the BOSH manifest that it creates. You specify these properties in your `tile.yml` configuration file using Double-Paren Expressions format.

Credentials

Include credentials to pass into a BOSH manifest as `salted_credentials` in your `tile.yml` file. But you need not include credentials that already exist in other tiles, such as Elastic Runtime. BOSH automatically generates these for any packages that require them.

Errands

Tile Generator automatically generates `deploy` and `delete` lifecycle errands for packages that deploy to PCF. These errand scripts deploy the service to PCF and publish its plans in the Marketplace, and remove the service from PCF and the Marketplace.

You can also define additional `post_deploy` and `pre_delete` errand scripts in `tile.yml` that prepare PCF to host the service or clean up before deleting it. You can configure these errands to run on their own dedicated VMs or co-locate them on existing errand VMs.

For `bosh-release` and `docker-bosh` packages, which run jobs directly on BOSH rather than on the PCF layer, you need to include `post_deploy` and `pre_delete` errands with their package definitions in `tile.yml`. Label them as lifecycle errands using `lifecycle: errand` and either `post_deploy: true` or `pre_delete: true`.

Tile Generator writes the `bosh-release` errands into the main BOSH release that it creates for the service, and adds `docker-bosh` errands into a separate Docker BOSH release that the main release depends on.

Dependencies

Include product dependencies under `requires_product_versions` at the top of your `tile.yml` file.

Update Rules

Tile Generator automatically generates the JavaScript migration file that enables one-click updates from Ops Manager. This file describes how to change existing tile property names and values in order to match the new version of the tile.

A mature tile may contain several of these `.js` files, from previous versions and the current one, to enable tile updates to automatically chain together in sequence.

You can add custom update code in the `tile.yml` Tile Generator configuration file, following the properties documented in the Migrating Tile Versions topic.

Tile File Format and Structure

Tile directories contain the following components, which include each other as shown:

- BOSH release
  - Service source code
  - Service broker
  - Language-specific buildpack(s)
  - Errands (service start and stop scripts)
  - BOSH manifest (deployment properties for service)
    - Packages
    - Dependencies
- Tile manifest template (adds properties into BOSH manifest)
  - Configuration forms and properties
  - Catalog metadata (for the Marketplace)
- Migrations

The three required top-level subdirectories in a `.pivotal` tile directory are:

- `metadata` - high-level information for configuring and publishing your service.
- `migrations` - rules that govern tile upgrades.
- `releases` - the BOSH releases that deploy your service.

The tile manifest template defines these subdirectory locations, so they can reside anywhere in the directory, but the typical structure looks like this:

```
.
├── example-product
│   ├── metadata
│   │   └── example-product.yml
│   ├── migrations
│   │   └── v1
│   │       ├── 201512301616_convert_14_transmogrifier_rules.js
│   │       ├── 201512301631_convert_15_16_transmogrifier_rules.js
│   │       └── 201611060205_example_migration.js
│   └── releases
│       └── example-release-18.tgz
```

.pivotal File Format

Within the tile directory, the BOSH release exists as a gzipped tarfile.

The entire tile directory is also a gzipped tarfile, with the `.tgz` extension renamed to `.pivotal`.

You can use any zip utility to create a `.pivotal` file. Ensure that the top-level subfolders as seen above in the `example-product` folder remain.

Example Workflow

```
$ cd example-product
$ zip -r example-product.pivotal metadata/ migrations/ releases/
$ unzip -l example-product.pivotal
Archive:  example-product.pivotal
  Length     Date   Time    Name
 --------    ----   ----    ----
        0  08-09-16 16:10   metadata/
    89458  08-09-16 16:10   metadata/example-product.yml
        0  07-08-16 09:32   migrations/
        0  07-08-16 09:32   migrations/v1/
      423  07-08-16 09:32   migrations/v1/201512301616_convert_14_transmogrifier_rules.js
     1228  07-08-16 09:32   migrations/v1/201512301631_convert_15_16_transmogrifier_rules.js
      582  07-08-16 09:32   migrations/v1/201611060205_example_migration.js
        0  08-09-16 16:11   releases/
        0  07-12-16 17:19   releases/example-release-18.tgz
```

GitHub Repository Structure

Tile developers typically develop and archive their code on GitHub, and their Concourse build pipeline pulls from GitHub to perform continuous integration.

Tile Generator does not dictate any directory structure for a GitHub repository, but by convention your tile repository might look like this:

```
/tile.yml
/src       # source code for all components deployed by the tile
/resources  # other resources, such as icon images and imported Docker images or bosh releases
/release    # generated bosh release(s)
/product    # generated tile
```

Packages

PCF services typically require multiple component job processes to run concurrently, such as a main app, a helper app, and a service broker. They also

require buildpacks that run as one-time compilation tasks. Services also require components such as external brokers or storage, which do not run as jobs, but nevertheless need to remain available.

The `tile yml` file that you pass to Tile Generator defines these service components it its `packages:` section. Each package has a name and a package type. The list of possible package types to pass to Tile Generator is in the Tile Generator code. It includes:

- app - `cf push` ed to PCF
- docker-app - `cf push` ed to PCF (image will not be embedded so requires Docker registry access)
- app-broker - `cf push` ed to PCF and registered as a broker
- docker-app-broker - `cf push` ed to PCF and registered as a broker (image is not embedded, so requires Docker registry access)
- external-broker - Registered as a broker
- buildpack - installed with `cf create-buildpack`; runs as a one-time task rather than a long-running process
- docker-bosh - describes a collection of Docker images that embed in the tile and run on BOSH-managed VMs, not PCF
- bosh-release - a pre-existing BOSH release wrapped in a tile, to run on BOSH-managed VMs, not PCF; requires you to describe all jobs (long-running processes and errands)

Packages typically contain a single process, but can include more than one, packaged to run in the same location.

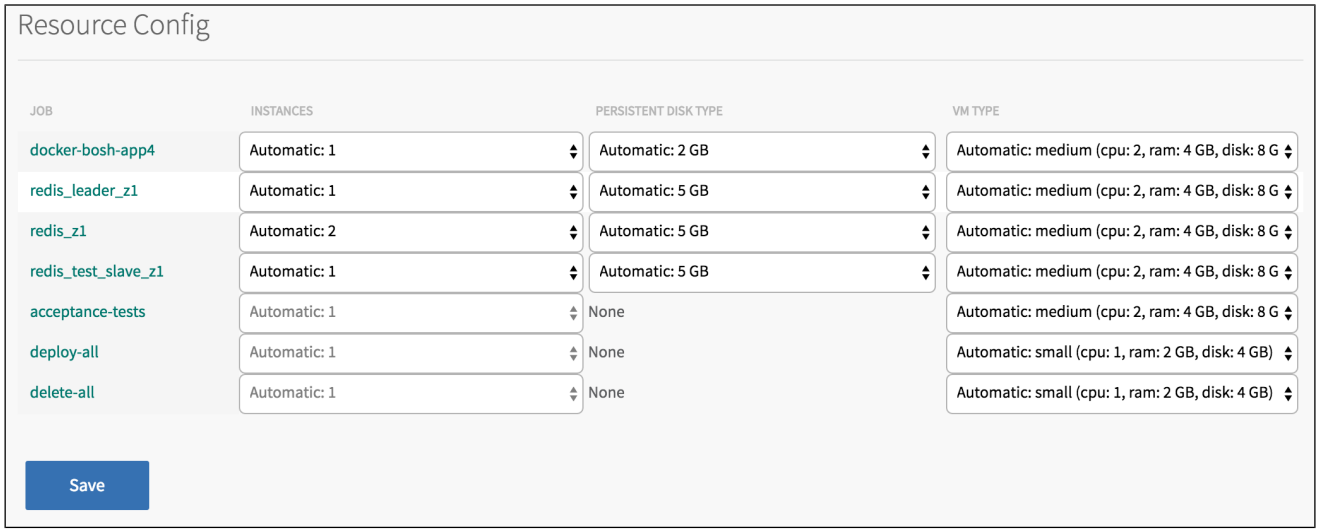
Where Package Processes Run

Where packaged processes run depends on their package type, as follows:

- `app`, `docker-app`, `app-broker`, and `docker-app-broker` packages call `cf push` to run processes in containers on a Diego cell.
- `docker-bosh` and `bosh-release` packages run their processes on VMs in the underlying BOSH layer.
- `external-broker` and `buildpack` packages run one-time tasks, not long-running processes, on Diego cells.

Package VM Resources

The service tile's **Resource Config** pane lets the operator configure resources individually for each package. This pane also lets operators provision resources for VMs that handle one-time tasks, with the `acceptance-tests`, `deploy-all`, and `delete-all` rows.



Configuring Disk and VM Type Defaults for On-Demand Service Tiles

Page last updated:

Note: Ops Manager 2.0 and later supports defining VM and disk type defaults and constraints.

This topic describes how tile authors can configure the dropdown menu items for VM types and persistent disk types in their tile.

On-demand service tiles have a configuration pane for each service plan. Operators use dropdown menus on the plan configuration pane to set the VM type and persistent disk type for each instance of that plan.

Ops Manager populates the menus with options based on the VM and disk options available on the current IaaS. Setting default values for VMs and disk types helps operators to choose the right resources for on-demand service broker (ODB) services when using on-demand plans.

VM and Persistent Disk Types

The property that defines the VM type options is `vm_type_dropdown`, and the menu options for disk type come from the `disk_type_dropdown` property. Tile authors do not specify the menu items in the product template.

Because VM and disk options differ by IaaS, Ops Manager uses a best-fit algorithm to match defaults to their closest equivalents on the IaaS, similar to how the **Resource Config** pane handles its **VM Type** and **Persistent Disk Type** options.

If a tile developer does not include a default value for a VM or disk resource, and then an operator configuring the tile does not choose a value from the dropdown, Ops Manager by default sets the resource to the smallest option available on the IaaS.

Set VM Type Defaults

For `vm_type_dropdown` the resources are `ram`, `ephemeral_disk`, and `cpu`. Tile authors can also apply `constraints` to any of these resources. Constraints can include `min` or `power_of_two`. For example:

```
- name: example_vm_type
  type: vm_type_dropdown
  configurable: true
  resource_definitions:
  - name: ram
    default: 1024
    constraints:
      min: 1024
      power_of_two: true
  - name: ephemeral_disk
    default: 1024
  - name: cpu
    default: 1
```

Set Persistent Disk Type Defaults

For `disk_type_dropdown` the resource is `persistent_disk`. Tile authors can also apply `constraints` to this resource. Constraints can include `min` or `power_of_two`. For example:

```
- name: example_disk_type_dropdown
  type: disk_type_dropdown
  configurable: true
  resource_definitions:
    - name: persistent_disk
      default: 2000
      constraints:
        min: 50
        power_of_two: false
```

Managing Runtime Configs

Page last updated:

This topic explains how to define and manage named runtime configs with your service tile for Pivotal Cloud Foundry (PCF).

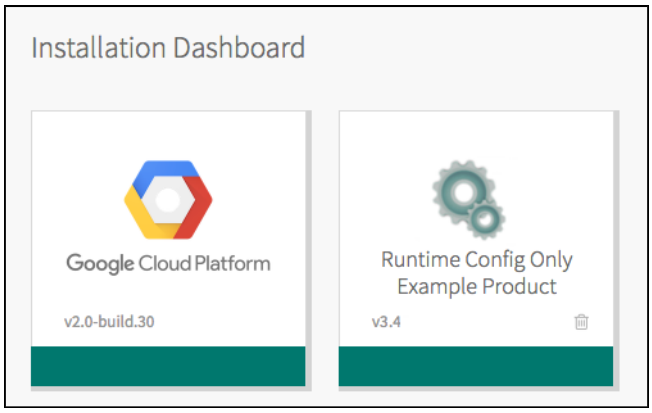
Tile authors can create a new runtime config in an existing product tile, delete a runtime config from a tile, or add a tile that contains a runtime config only.

See the BOSH documentation for more information about runtime configs.

Overview

A runtime config is a section of the tile metadata that can define global deployment configurations. When a tile author includes a runtime config as a top-level key in the tile metadata, BOSH applies the runtime config to every VM in the deployment.

To the operator, a runtime config appears in Ops Manager as a tile with minimal configuration options. Runtime config tiles contain no stemcell, network, availability zone (AZ), or resource config information.



When you click **Apply Changes**, Ops Manager combines the runtime config information from every tile in the deployment and assigns each named runtime config a unique identifier. Ops Manager creates the name using the tile name, a generated GUID, and the runtime config name defined in the metadata in the following format:

```
TILE_NAME-GUID-RUNTIME_CONFIG_NAME
```

Create a Runtime Config

Tile authors can add `runtime_configs` as a top-level key in tile metadata. In this key, the tile author defines configuration properties that Ops Manager applies to all deployments. A tile can support any number of runtime configs.

A named runtime config, such as `MY-RUNTIME-CONFIG` in the example below, can contain any number of addons. Each addon can contain any number of jobs.

To add a runtime config to a tile, add the following section to the tile metadata:

```
runtime_configs:
  - name: MY-RUNTIME-CONFIG
    runtime_config: |
      releases:
      - name: os-conf
        version: 15
      addons:
      - name: MY-ADDON-NAME
        jobs:
        - name: MY-RUNTIME-CONFIG-JOB
          release: os-conf
        properties:
          MY-ADDON-NAME:
          ...
```

Replace the text in the example above with the following:

- `MY-RUNTIME-CONFIG`: Choose a name for the runtime config.
- `MY-ADDON-NAME`: Choose a name for the addon that contains the runtime config job.
- `MY-RUNTIME-CONFIG-JOB`: Choose a name for the job the runtime config describes.

Important: The names you choose must be unique across a deployment. Pivotal recommends appending your product name or another unique identifier to each of the named items in the `runtime_configs` section.

Define the runtime config job properties in the `properties` section.

Delete a Runtime Config

Tile authors can remove an existing runtime config from a tile by removing the reference from the metadata. When the operator upgrades the tile, Ops Manager detects the missing reference and deletes the runtime config.

Create a Runtime Config-Only Tile

Tile authors can create a tile that only contains a runtime config. The only release that a tile author must include in a runtime config tile is `os-conf`. When creating a runtime config-only tile, a tile author is not required to define the following top-level keys:

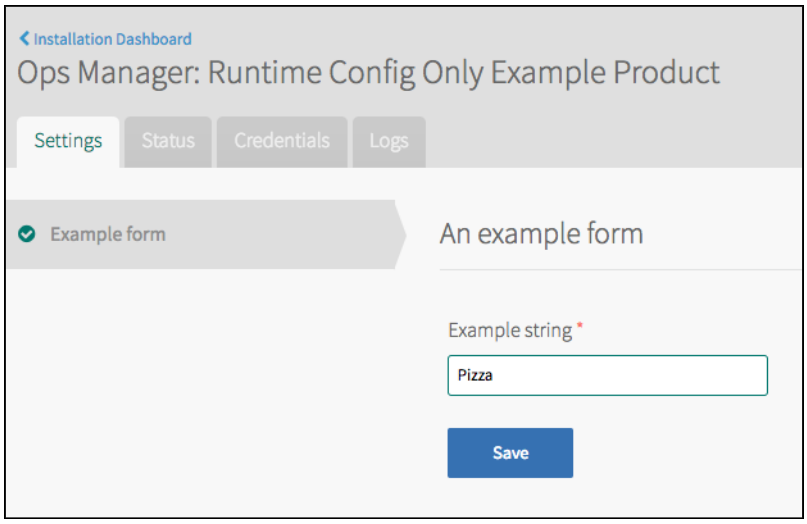
- `post_deploy_errands`
- `pre_delete_errands`
- `job_types`

Example Runtime Config-Only Tile

The following example shows a runtime config-only tile with minimal configuration:

```
---
name: runtime-config-only-example-product
product_version: "3.4"
minimum_version_for_upgrade: "2.0"
metadata_version: "2.0"
label: 'Runtime Config Only Example Product'
description: An example product to demonstrate runtime config features
rank: 1
service_broker: false # Default value
stemcell_criteria:
  os: ubuntu-trusty
  version: STEMCELL-VERSION

releases:
  - name: os-conf
    file: os-conf
    version: '15'

post_deploy_errands: []

pre_delete_errands: []

form_types:
  - name: example_form
    label: 'Example form'
    description: 'An example form'
    property_inputs:
      - reference: .properties.example_string
        label: 'Example string'

property_blueprints:
  - name: example_string
    type: string
    configurable: true
    default: Pizza

job_types: []

runtime_configs:
  - name: example-runtime-config
    runtime_config: |
      releases:
      - name: os-conf
        version: 15
      addons:
      - name: login
        jobs:
        - name: login-banner
          release: os-conf
        properties:
          login_banner:
            text: |
              (( .properties.example_string.value )).
```

In the example runtime config above, the `login-banner` job prints a banner when a user logs into any VM in the deployment. The operator can use the default value defined in the `form_types` section of the metadata or configure the banner by editing the **Example string** value in Ops Manager.



Testing Tiles

Page last updated:

This topic explains recommended testing practices for tile developers.

Tile Testing

Good testing assures tile developers that their product installs and runs properly on diverse platforms and assures PCF platform operators that the tile they install can provide its service successfully on their platform.

Pivotal recommends a pyramid structure for testing, starting with unit tests and stepping up to successively broader and more automated levels of integration. Pivotal uses and recommends Concourse for creating build pipelines that follow this test structure. Other continuous integration tools should also support a pyramid testing approach.

Tile Test Pyramid

For PCF tiles, a typical test pyramid progresses as follows:

1. Unit tests for each tile **component** (e.g. service components, broker, adapter, and metrics emitter), manual by developer and in automated pipeline.
2. System tests of the tile's **BOSH release**, including:
    - **Functional tests** covering the main features of the service. The main features typically interact with almost all important external integration points, so these tests confirm product functionality.
    - **Smoke tests** (lifecycle tests) for service instances that create and bind a service instance, call it from a test app, check the logs it generates, and delete it. For a typical end-to-end test sequence, see Smoke Tests below.
3. System tests of **tile** operation within Ops Manager.
    - These include:
        - **Configuration checks** that test every external configurable integration point and connection to remote servers using configured credentials
        - **Default checks** that confirm "happy path" functionality.
    - Use the Ops Manager API to verify that property blueprints in the tile metadata are correct and that they translate correctly to the BOSH manifest that Ops Manager generates.
    - Use the Om tool to call the Ops Manager API programmatically from Go. Avoid the unsupported opsmgr gem that called the Ops Manager API from Ruby.
    - Confirm manually that the tile wires property blueprints to the expected pane and form controls in the UI.
    - Test your environment using one of the environments described in Development Environments

Note: System tests might incur costs from using third party services, IaaS resources, etc.

Smoke Tests

Smoke tests are end-to-end lifecycle tests for service instances that you can include as post-deploy errands within a tile and also automate in Concourse or other integration platforms.

A typical smoke test runs as follows:

1. Create an org and space for the test to run in.
2. Register the tile's service broker.
3. Enable service access for the created org.
4. Iterate through all service plans (or a subset of them) to do the following:
    a. Create a service instance for the plan.

   b. Push a test app.
   c. Bind the service instance to the app.
   d. Use the app in a way that exercises the service instance. For a data service, for example, write and read from the service instance.
   e. Unbind the service instance.
   f. Delete the service instance.
   g. Delete the test app.

5. Delete the service broker.

6. Delete the test org and space.

General Recommendations

The following are general recommendations for designing and running tests on PCF tiles:

- Clean up after yourself. Leave the environment exactly as it was before the test was run.
- Generate verbose logging with lots of contextual data to make troubleshooting easier.
- Design test suites for re-usability by making them highly parameterizable. Important parameters include:
  - External settings such as domains, creds, and certs
  - Plans to test against. For example, the Redis for PCF smoke tests use identical code for two different service plans, pre-provisioned and on-demand.
  - Timeouts, numbers of retries, and other things that you need to adjust for different environments
  - Switches to include or exclude portions of the tests such as generating metrics or backups
- Re-use tests that exist already, for example in Concourse.
- Use an example CF app that uses your service. This app can serve for testing, demoing your tile capabilities, and as a code code example. See the MySQL Test App an example.
- When testing manually, using the UI is better than calling the underlying API directly. Use UIs and APIs the way a customer would.

Types of Integration

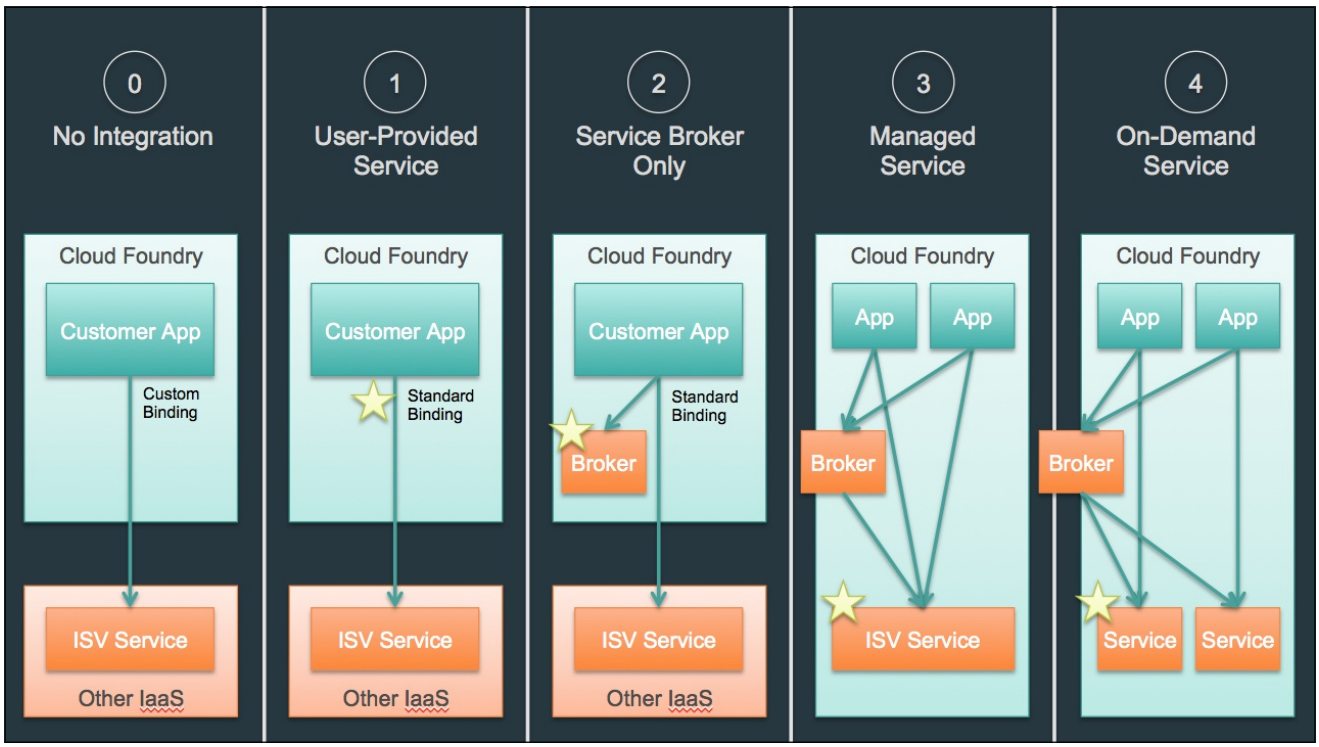
Page last updated:

Integration Levels

A service can integrate with PCF at four levels, shown here in order of increasing integration. In general, user-experience and production-readiness improves as the integration level increases. But none of the higher levels is required. You can stop service integration and declare it complete (enough) after any of these:

When integrating third-party software with Cloud Foundry, the effort typically progresses through increasing levels of integration. We recommend this staged approach because it enables early feedback on the value and the design of the integration, which helps make better decisions about future stages.

For non-service integrations (such as applications or buildpacks), a similar staged integration approach is often possible and desirable.



Level 1. User-Provided Service

The service runs external to PCF and has no service broker or tile. To use a service with an app, the developer creates a service broker by running `cf create-user-provided-service` from the Cloud Foundry Command-Line Interface (cf CLI).

Configuring, running, upgrading, and paying for a user-provided service are all up to the developer.

Level 2. Brokered Service

A brokered service runs external to PCF, but has a tile on Pivotal Network (PivNet).

PivNet designates brokered services by including “Service Broker for PCF” in the name.

Operators install, configure, and upgrade the tile through the Ops Manager Installation Dashboard. Developers can then see your service plans and create service instances in Apps Manager, or by running `cf marketplace` and `cf create-service` from the command-line.

The Brokered Service topic has more information about brokered service tiles and how to create them.

Level 3. Managed Service

With a managed service, both the service broker and the service itself run within PCF. This enables PCF to manage, monitor, and increase service performance.

As with the brokered service, the service has a service broker and a tile listed on PivNet. PivNet lists managed services as “for PCF,” without “Service Broker” in the name.

When the operator installs the tile, they allocate a block of VMs to run service instances and provisions their CPU and memory resources uniformly.

The Managed Service topic has more information about managed service tiles and how to create them.

Level 4. On-Demand (Dynamic) Service

As with a managed service, an on-demand service and broker both run within PCF, and PivNet lists the service tile without “Service Broker” in the name. But unlike a managed service, an on-demand service does not limit the number of service instance VMs. The operator does not have to pre-allocate and provision VM resources for the service.

When a developer creates an instance of an on-demand service, they provision its resources (within an allowed range) and BOSH dynamically creates a new, dedicated VM for the instance.

The On-Demand Service topic has more information about On-Demand service tiles and how to create them.

User-Provided Service

Page last updated:

This topic explains how to create a user-provided service for PCF.

Overview

A PCF developer can call your service from their app code, even if the service runs outside of PCF and has no service broker. Use cases for this include:

- Your software is available as a SaaS.
- You already have a way to install your software on-premises at a customer site.
- Your customer already uses your software, is now adopting PCF, and wants to consume your software from applications that they deploy on PCF.

This do-it-yourself solution represents the lowest level of PCF service integration. It works only for services running external to PCF, and does not publish the services to the Services Marketplace or make them available to anyone outside the space of the developer who runs these commands. See the User-Provided Service Instances topic for more information.

Running apps with a user-provided service is a great way to determine what information needs to be passed in the credential structure (useful in higher integration levels), verify that the integration works, and develop a test app that can continue to be used at higher levels. From the app developer perspective, once a user-provided service works, later integrations of the service will not require any further code changes. User-provided service bindings are fully forward-compatible with brokered service bindings.

Using a User-Provided Service

To use an external service that has no tile, they do the following from the Cloud Foundry Command-Line Interface (cf CLI).

1. Run `cf create-user-provided-service MY-SERVICE-NAME -p CREDENTIALS` (or `cf cups`) to create a service instance. The `CREDENTIALS` argument should be a valid JSON string that contains the URL and credentials necessary to connect to your externally-deployed service.
2. Run `cf bind-service` to bind the service instance to their app.

By doing this, app developers can bind their apps to your service and write all code necessary to access it through a Cloud Foundry service binding.

Brokered Service

Page last updated:

The topics in this subsection explain how to integrate your software service with Pivotal Cloud Foundry (PCF) to create a brokered service and service tile for PCF.

Overview

You can achieve the first real improvement in your PCF customers user experience by creating a Service Broker for your service.

A brokered service runs external to PCF, but it has a tile on Pivotal Network (PivNet). Operators install, configure, and upgrade the tile through the Ops Manager Installation Dashboard.

The service broker eliminates the need for your customers to know the URLs and credentials for your services; they are managed automatically by the broker.

Building a broker for a (still) externally deployed service is generally a good way to publish a first tile that adds real value for customers who have both your software and PCF.

Create a Brokered Service

- A brokered service requires a service broker, which publishes an API to the Cloud Controller.
  Service Brokers explains how to create one.
- Route Services explains how to create a route service, for use in the routing layer of PCF rather than by hosted PCF apps.
- Catalog explains how to design the part of your service broker API that publishes service plan information to the Services Marketplace.
- You can write your service broker in the language of your choice.
  Buildpacks explains how to create a language-specific buildpack that compiles and packages your service broker to run on PCF.
- Once you have the individual components for your brokered service integration, you can work through Building Your First Tile to create your tile.

At any level of integration, Pivotal recommends and supports using Concourse for continuous integration during development.

Service Brokers

Page last updated:

This topic provides resources for building service brokers and routing services.

Service Broker Resources

- The Custom Services Overview topic gives a high-level description of how service brokers work in Pivotal Cloud Foundry (PCF).
- Service Broker API gives a more detailed explanation of PCF service brokers, and provides a full specification for the endpoints, requests, responses, and status codes that a service broker must support.
- Example Service Brokers offers example brokers written in Ruby, Java, and Go.

Route Services Resources

- Route Services explains how route services work, and what are the different architectures for using them in a Cloud Foundry deployment.
- Example Route Services gives examples of a logging route service, a rate-limiting route service, and another logging service written in Spring Boot. It also offers a tutorial on setting up the logging route service.

Catalog Resources

- Catalog Metadata explains how to publish service plan information to the Services Marketplace, including the icons, display names, and links that appear in the PCF Apps Manager UI but not the plain text output of `cf marketplace`.

Managed Service

Page last updated:

The topics in this subsection explain how to integrate your brokered service more closely with Pivotal Cloud Foundry (PCF) to create a managed service and service tile for PCF.

Overview

The next level of integration is to get your service to be deployed on PCF rather than externally, on the same IaaS that your particular Cloud Foundry instance is deployed on, and by the same orchestration tool, BOSH.

This is usually one of the more involved integrations, as you will have to change your packaging to allow your service components to be deployed by BOSH onto the PCF infrastructure.

Offering your software as a managed service means that your PCF customers will not have to learn different ways to deploy, manage, and monitor different components of their application platform.

As with the brokered service, the service has a service broker and a tile listed on PivNet. PivNet lists managed services as “for PCF,” without “Service Broker” in the name.

To integrate your service at this level, you will have to learn about stemcells, BOSH releases, and manifests. You will also have to decide how your service maps to virtual machines and how persistent storage is managed.

Minimal Viable Product

For a Minimal Viable Product (MVP) version of a managed service, we typically recommend that you aim for a single, shared service instance, and don’t yet worry too much about High Availability of this instance. This integration level is mostly about getting the BOSH packaging, deployment, and monitoring working correctly.

High Availability

Once you have a managed service, you may decide to prioritize either on-demand provisioning of service instances, or making your single shared service instance more highly available.

When properly configured, BOSH monitors and restarts any failing processes and virtual machines that are part of your service deployment. But to further increase availability, you will have to think about spreading your resources across multiple availability zones or even regions, and replicating your persistent storage across those as well.

Create a Managed Service

- For BOSH to manage your service, you need to create a BOSH release for it.
  BOSH Releases explains how to do this, and how to use your already-existing Docker image as a shortcut.
- Once you have created a BOSH release for your managed service integration, you can work through Building Your First Tile to create your tile.
- The Tile Generator tool automatically creates the lifecycle errands that can run after a PCF tile is deployed or before it is removed. PCF operators control which errands run the next time they click **Apply Changes** to redeploy. See the Errands topic for how PCF operators control when errands run, and how to set default errand run rules in the tile.

At any level of integration, Pivotal recommends and supports using Concourse for continuous integration during development.

BOSH Releases

Page last updated:

This topic provides resources for creating a BOSH release that integrates a software service with Pivotal Cloud Foundry (PCF) at the managed service level.

Overview

A BOSH release is a directory that contains the source code for your service along with everything else that BOSH needs to deploy it reproducibly to cloud VMs running a specified operating system (stemcell). These contents include but are not limited to buildpacks, start up scripts, binary artifacts, and a BOSH manifest containing configuration and deployment properties.

The BOSH manifest specifies the following major components:

- **Packages** that can be installed on PCF stemcells to create virtual machine images
- **Jobs** that describe how to install, run, and remove your software
- A **Monitor** script, that describes how to monitor the health of your service components and stop or restart them

BOSH Resources

These topics give more details on BOSH and BOSH releases:

- BOSH Documentation is the top-level contents page for BOSH documentation.
- BOSH Problem Statement explains what BOSH does.
- BOSH Basic Workflow lists the high-level steps for creating a BOSH deployment.

Creating a BOSH Release

These topics explain how to create a BOSH release:

- Creating a Release
- Defining your Jobs
- Defining your VMs
- Defining your Runtime Configs
- Monitoring the Health of your Service

Shortcut: Start with Docker Images

If you have already packaged your service as Docker images, you can emulate a managed service deployment using the Tile Generator's support for `docker-bosh` packages. This feature lets you deploy pre-existing Docker images into BOSH managed virtual machines on the PCF infrastructure.

While this is a great, easy way to deploy your service on PCF, we don't recommend this as a long-term, production-ready solution. There is really no benefit of running your service in containers on the VMs, and it does have a number of operational ("day 2") drawbacks:

- You introduce more software (Docker) which needs to be kept up-to-date, and has the potential for bugs, downtime, and security vulnerabilities.
- You can no longer take advantage of the patching capabilities of PCF for stemcells and application dependencies, like frameworks and libraries. Instead, you become directly responsible for managing all software that is in the Docker images you deploy.

Enhancing the BOSH Release

After the basic BOSH release is in place, additional features for logging help operators run the service. For logging information, see syslog-migration-release.

Logs written under the expected BOSH location `/var/vcap/sys/log` are forwarded to the configured syslog server by the release. Integrating syslog forwarding into a tile should not require code changes; it only requires including the release and configuration forms in the `tile.yml`. For an example, see pcf-examples/tile-for-bosh-with-syslog.

Errands

Page last updated:

Lifecycle errands are BOSH errands (scripts) that run at the beginning and end of an installed product's availability time. Product teams create errands as part of a product package, and a product can only run errands it includes.

For more information about BOSH errands, see BOSH documentation, and for more information about errands in Pivotal Cloud Foundry (PCF), see Managing Errands in Ops Manager.

In Ops Manager 2.0 and later, tile authors can choose to colocate errands on existing VMs. When errands are not colocated, BOSH deploys a new VM for each errand defined in the tile metadata. Colocated errands can run alongside other jobs or errands on existing VMs in an operator's deployment.

Products can have two kinds of errands. Post-deploy errands run after a product installs but before Ops Manager displays makes it available for use. Pre-delete errands run after an operator chooses to delete a product, but before Ops Manager finishes removing it from use.

To save deployment time, operators can set errand run rules that dictate whether or not errands run. Tile authors can set defaults for these run rules.

Define a Colocated Errand

Note: Ops Manager 2.0 and later supports colocated errands.

Instead of deploying a new VM for each errand, colocated errands run on an existing VM. Errands can run alongside other jobs on a VM, and multiple errands can be colocated on the same VM. Colocated errands run faster than traditional errands and use fewer resources, including disk and IP space.

To configure a colocated errand, define the following properties in the `pre_delete_errands` and `post_deploy_errands` sections of the tile metadata:

Property	Description
`name: MY-ERRAND`	Provide the name of the errand job. The example manifest in the following section uses `example_colocated_errand`.
`colocated: true`	Set this value to `true` to enable colocated errands. If you do not set this value, Ops Manager ignores all other errand attributes in this section.
`run_default: on`	(Optional) You can set the run rules to `on`, `off`, or `when-changed`. See Errand Run Rules for more information. If you do not define this property, Ops Manager sets the run default to `on`. The operator can override this setting using the Ops Manager API or the tile's **Errand Config** tab.
`instances: []`	(Optional) Provide an array that tells BOSH where to run the errand. Use the name of an instance group, such as `web_server`, or a single instance, such as `web_server/first`. If you do not define this property or you provide an empty array, the errand runs on every instance of the job in the operator's deployment.
`label: ERRAND-LABEL`	Define the errand name to be shown in the tile's **Errand Config** page and above **Apply Changes**. The example manifest in the following section uses `colocated errand on web_server`.
`description: TEXT`	(Optional) Provide a description for the errand that appears in the tile's **Errand Config** page.

After defining the errand in the sections above, add the errand to the job properties in the `job_types` section.

Colocated Errand Example Manifest

The following example shows colocated `post_deploy_errands` and `pre_delete_errands` sections in the tile metadata:

```
post_deploy_errands:
  - name: example-errand
    colocated: false
  - name: example_colocated_errand
    colocated: true
    run_default: on
    instances:
      - web_server/first
    label: colocated errand on web_server
    description: This errand does little more than print a message in order to prove colocated errands work.

pre_delete_errands:
  - name: example-errand
```

The following example shows the colocated errands referenced within the `job_type`:

```
job_types:
 - name: web_server
    resource_label: Web Server
    templates:
      - name: web_server
        release: example-release
        provides: |
          web_server_info: (( .properties.example_selector.selected_option.parsed_manifest(provides_section) ))
        consumes: |
          web_server_info: (( .properties.example_selector.selected_option.parsed_manifest(consumes_section) ))
      - name: time_logger
        release: example-release
      - name: example_colocated_errand
        release: example-release
    release: example-release
    static_ip: 1
    dynamic_ip: 0
    max_in_flight: 1
```

Backward Compatibility for Colocated Errands

Colocated errand support is available in Ops Manager 2.0 and later. If your tile uses colocated errands, use the instructions in this section to ensure your tile is also compatible with Ops Manager 1.12 and earlier.

When your tile no longer requires Ops Manager 1.12 support, configure your errands as either colocated or non-colocated. Future versions of Ops Manager will not support the workaround described in this section.

The following example manifest shows an `example_colocated_errand` configured as a colocated errand in Ops Manager 2.0 and as an instance group errand in Ops Manager 1.12:

```
post_deploy_errands:
  - name: example_colocated_errand
    colocated: true
    run_default: on
    instances:
      - web_server/first
    label: colocated errand on web_server
    description: This errand does little more than print a message in order to prove colocated errands work.
...
job_types:
  - name: example_colocated_errand
    description: The very best illustrative errand that prints all the properties, including secrets.
    templates:
      - name: dummy
        release: dummy
    errand: true
...
  - name: web_server
    resource_label: Web Server
    templates:
      - name: example_colocated_errand
        release: example-release
```

To make your tile compatible with both colocated and non-colocated errands, perform the following steps:

1. Configure your colocated errand for Ops Manager 2.0, as shown in the Colocated Errand Example Manifest. Ops Manager versions 1.12 and earlier ignore this property in the manifest.

2. In the `job_types` section, define the same errand in the `web_server` instance group, as shown in the example above. Ops Manager 1.12 and earlier runs the errand on every VM in the `web_server` instance group. If you want the errand to run only once, configure the errand to run on an instance group with only one instance.

3. Configure the instance group that corresponds to your errand:

    - Set `instance_definition.configurable: false`
    - Set `instance_definition.default: 0`
    - Configure at least one non-errand job in the instance group. Ops Manager requires each instance group to contain at least one job.

    > **Note**: The example manifest above uses the `dummy` job from the Dummy BOSH release. You can use any no-op job.

4. Ops Manager 1.12 and earlier displays the following warning, but runs the errand on the specified instance group:

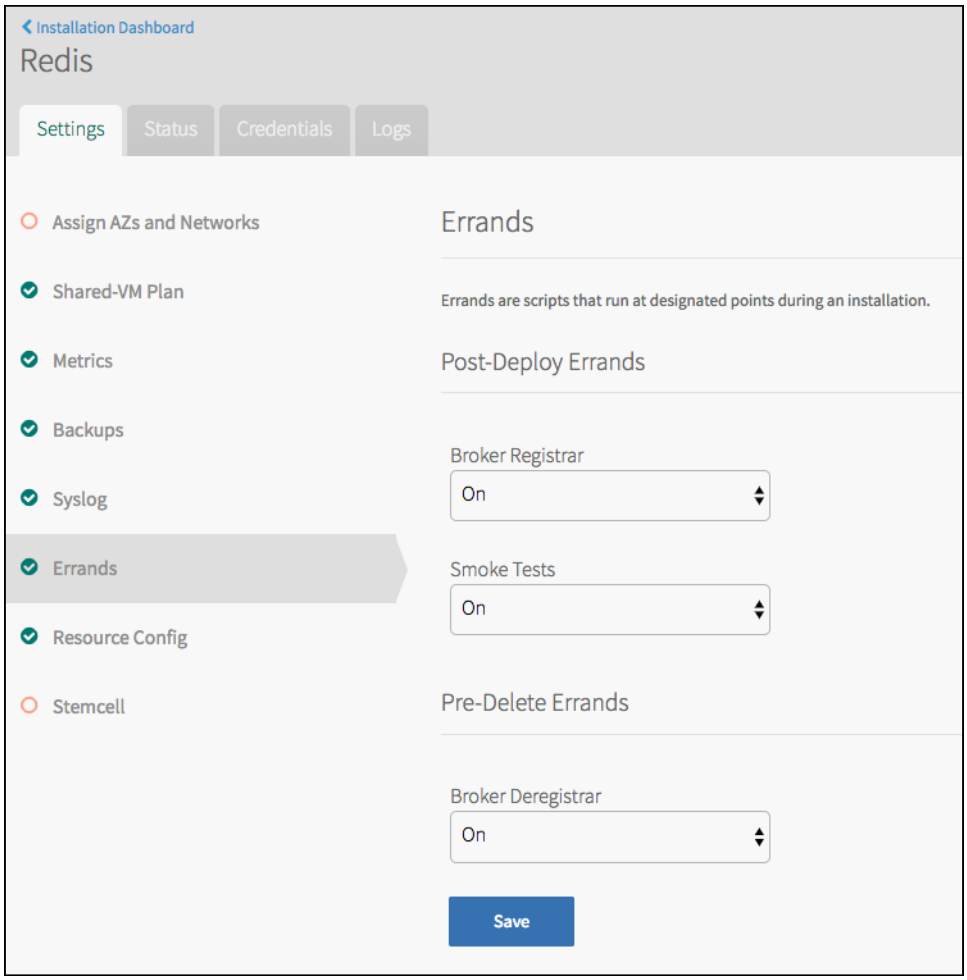
    ```
    Warning: Ambiguous request: the requested errand name 'example_colocated_errand' matches both a job name and an errand instance group
    ```

Post-Deploy Errands

Post-deploy errands run after a product installs, but before Ops Manager makes it available for use.

Typical post-install errands include smoke or acceptance tests, database initialization or database migration, and service broker registration.

Post-deploy errands run by default. An operator can prevent a post-deploy errand from running by setting its run rule to **Off** under **Pending Changes** in the Ops Manager Installation Dashboard or on the product tile's **Settings** tab **Errands** pane, before installing the product.



For example, Redis has a **Broker Registrar** post-deploy errand that the Elastic Runtime tile uses to register its service broker with the Cloud Controller and publish its service plans.

If an operator chooses **Off** in the drop-down menu for Elastic Runtime's **Broker Registrar** errand before installation, Elastic Runtime's service broker is not registered with the Cloud Controller and its service plans are not made public.

Pre-Delete Errands

Pre-delete errands run after an operator chooses to delete a product, but before Ops Manager actually finishes deleting it.

Typical pre-delete errands include clean up of application artifacts and service broker de-registration. For example, Pivotal MySQL has a **Broker Deregistrar** pre-delete errand that:

- Purges the service offering
- Purges all service instances
- Purges all application bindings
- Deletes the service broker from the Cloud Controller

When an operator chooses to delete the Pivotal MySQL product, Ops Manager first runs the **Broker Deregistrar** pre-delete errand, then deletes the product.

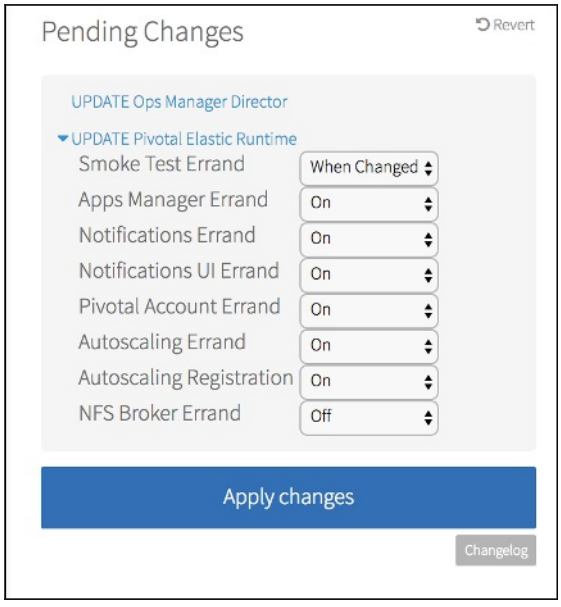
Pre-delete errands run by default. An operator can prevent a pre-delete errand from running by setting its run rule to **Off** under **Pending Changes** in the Ops Manager Installation Dashboard or on the product tile's **Settings** tab **Errands** pane, before installing the product.

Errand Run Rules

warning: In Ops Manager v1.10.0 and later, errands set to the **When Changed** rule do not always run when the tile has relevant changes. Instead of using **When Changed**, Pivotal recommends that tile developers leave the default run rule for errands as **On** and let operators use one-time rules to turn errands off and save deploy time.

Some errands do not always need to run. For example, installing a minor patch to a existing service might not require re-registering its broker. Ops Manager lets operators save installation time by turning errands off or on. They set these errand run rules in two places:

- **One-Time Rules** under **Pending Changes** in the Ops Manager Installation Dashboard. These rules only apply to the next time you run **Apply Changes** and do not persist after the next successful installation.



- **Persistent Rules** in the tile's **Errands** pane. These rules persist through subsequent installations, until changed in the **Errands** pane.

For more information, see Configure Run Rules in Ops Manager.

On-Demand Service

Page last updated:

This topic explains how to integrate your software as an on-demand service and service tile for PCF.

Overview

Brokered service and managed service integrations assume that you have a single VM instance deployed for your software deployed, or a limited number of VMs.

These VMs can be multi-tenant, and you can possibly scale them manually to accommodate many concurrent applications. But for real production deployments, most of your customers will want dedicated VM instances of your service for each application.

On-demand (dynamic) services enable this flexibility in a scalable way. When an operator deploys the service, do not pre-allocate VM resources for service instances. Instead, they define an allowable range of VM memory and CPU sizes and create a dedicated network on the IaaS to host any required number of service instance VMs.

When a developer creates an instance of an on-demand service, they provision its resources within the allowed range, and BOSH dynamically creates a new, dedicated VM for the instance.

Create an On-Demand Service

The best way to create an on-demand service is to use the On-Demand Services SDK.

The on-demand services SDK provides a generic on-demand service broker (ODB) that Tile Generator can consume like any other service broker.

The on-demand service author does not write a service broker. Instead, they write a service adapter component that takes requests from the ODB and interfaces with their service software to fulfill requests from the ODB.

To create their tile, the tile author then feeds their service adapter and the BOSH release of the ODB to Tile Generator.

- On-Demand Services SDK documentation explains how to write a service adapter for an on-demand service that uses the ODB.
- Once you have the individual components for your brokered service integration, you can work through Building Your First Tile to create your tile.

At any level of integration, Pivotal recommends and supports using Concourse for continuous integration during development.

High Availability

If you had not already configured your service for High Availability as a managed service, the final step would be to consider how you can make each of your dynamically-provisioned service instances more highly available.

Buildpacks

Page last updated:

Buildpacks compile and package apps to run on Pivotal Cloud Foundry (PCF). This topic lists resources for using and deploying buildpacks with PCF apps, and for creating your own custom buildpack.

Official Buildpacks

- Java buildpack (by far the most complicated!)
- Go buildpack
- Ruby buildpack
- Node.js buildpack
- Python buildpack
- PHP buildpack
- Static file buildpack (for static web content)
- Binary buildpack

Other Buildpacks

Buildpacks can also be used to inject additional code into the application container. For more information, see the following:

- The PCF documentation topic Creating Custom Buildpacks
- The github repo Eureka Registrar Sidecar
- The github repo Spring Config Injection

Custom Buildpacks

- Creating a Custom Buildpack

CredHub

Page last updated:

BOSH CredHub is a secure credential management component that runs on the BOSH VM to minimize the surface area where credentials can be compromised. This topic provides resources for configuring service tiles to store their internal credentials in BOSH CredHub, instead of encoding them in product template and job template files.

Credentials that service tiles store in BOSH CredHub for their own internal use are distinct from secure service instance credentials that Pivotal Application Service (PAS) stores in runtime CredHub to enable PAS apps to securely access services.

Both BOSH CredHub and runtime CredHub are instances of the CredHub credential management component. See the CredHub documentation for more information.

Overview

Many PCF components use credentials to authenticate connections, and PCF installations often have hundreds of active credentials. Secure credential management is essential to prevent data and security breaches.

In Pivotal Cloud Foundry (PCF) v1.11.0, CredHub runs on the BOSH VM, alongside the BOSH Director and UAA. Ops Manager v1.11 stores its credentials in CredHub, and users can retrieve them using the CredHub API or the **Credentials** tab of the Ops Manager Director tile. Tile developers can embed CredHub calls in manifest snippets and PCF apps can retrieve credentials using the CredHub API.

See Fetching Variable Names and Values for how to fetch variable names and values using the CredHub API.

CredHub Credential Types

CredHub stores and retrieves the following types of credentials:

- `value` — single string value
- `json` — arbitrary JSON object
- `user` - username
- `password` — password string
- `certificate` — object containing certificate authority (CA), certificate, and private key
- `ssh` — object containing SSH public key and private key
- `rsa` — object containing RSA public key and private key

For more information, read CredHub Credential Types.

For BOSH variable types, read BOSH Variable Types.

Creating New Variables

To use CredHub in your deployment, you must create new variables and store them in CredHub. By default, variable namespaces are written to prevent collision across deployments, but you can type variable names precisely if you wish.

For more information, read Creating New Variables in CredHub.

Migrating Credentials

To migrate existing non-configurable credentials to CredHub, such as blobstore secrets and backup encryption keys, use the JavaScript migration process. After a successful migration, Ops Manager deletes the migrated credentials from installation.yml.

For more information, read Migrating Existing Credentials to CredHub.

Fetching Variable Names and Values

API endpoints are available to help you find variable names and values for products known to the Ops Manager Director.

For more information, read Fetching Variable Names and Values.

CredHub in Manifest Snippets

Tile developers can embed CredHub in product template and job template manifest snippets using triple-parenthesis notation:

```
manifest: |
  credhub:
    concatenated_password: prefix-((( credhub-password )))-suffix
    password: ((( credhub-password )))
```

PCF v1.11.0 Limitations

PCF v1.11.0 supports CredHub for credential storage, but it does not support the following:

- Automatic backup and restore for CredHub, along with other PCF system components.
- Automatic tile upgrades that migrate all types of credentials defined in property blueprints in previous tile versions, to storage in CredHub.
- Using CredHub to generate new credentials.

Tile authors may choose to wait until PCF supports some or all of these features before incorporating CredHub into their service.

Creating New Variables in CredHub

Page last updated:

This topic explains how CredHub manages variables in the context of a larger deployment, and how to create new variables for use in CredHub.

Background

When a tile author defines a top-level `variables` section in the product template, Ops Manager passes the `variables` section to the product manifest. tile authors can define variables in the product template as follows:

```
variables:
 - name: EXAMPLE-CREDHUB-PASSWORD
   type: password
```

You can reference these variables in the manifest snippets in their tile metadata using a triple parentheses syntax:

```
((( EXAMPLE-CREDHUB-PASSWORD )))
```

Using triple parentheses lets Ops Manager identify CredHub variables while still supporting the BOSH double parentheses syntax. A variable referenced within triple parentheses is replaced by double parentheses in the generated manifest. After contacting CredHub, BOSH populates that variable value internally.

The benefit of this approach is that the Ops Manager YAML file does not contain sensitive credentials when the metadata manifest snippets have triple parentheses. The resulting manifest file contains variables within double parentheses, rather than unobscured credentials.

For example, a tile author adds credentials to a manifest snippet in the following format:

```
key: ((( EXAMPLE-CREDHUB-PASSWORD )))
key: prefix-((( ANOTHER-CREDHUB-PASSWORD )))-suffix
```

Ops Manager evaluates the above example to generate the following section in the product manifest:

```
(( EXAMPLE-CREDHUB-PASSWORD ))
prefix-(( ANOTHER-CREDHUB-PASSWORD ))-suffix
```

How CredHub Works Within a Deployment

CredHub is distributed as a BOSH release. As part of this installation, Ops Manager co-locates the CredHub release on the Ops Manager Director, including the CredHub job configurations, and the Director is configured to point to the CredHub API.

Once CredHub has been deployed and configured on the Director, any Director deployment can use CredHub variables in place of credential values. Using variables, rather than values, provides an extra layer of security when transmitting credentials within your deployment.

Changing Your Deployment Manifest to Include CredHub Variables

The Ops Manager Director interpolates credential values into manifests that use the `((variables))` syntax. When the Director encounters a variable using this syntax, it requests the credential value from CredHub. If the credential does not exist and the release or manifest contains generation properties, the credential value is generated automatically.

The manifest excerpt below includes references to two credentials, `EXAMPLE-PASSWORD` and `EXAMPLE-TLS`.

When this manifest is deployed, the Ops Manager Director retrieves the stored variables and replaces them with the credential values associated with each variable. The `EXAMPLE-TLS` variables include property accessors, so only the `certificate` and `private_key` components are interpolated.

```
name: demo-deploy

instance_groups:
 jobs:
 - name: demo
   release: demo
   properties:
     demo:
       password: ((EXAMPLE-PASSWORD))
       tls:
         certificate: ((EXAMPLE-TLS.certificate))
         private_key: ((EXAMPLE-TLS.private_key))
```

Ops Manager configures the Director to generate a credential if it does not exist. The manifest includes generation parameters that define how the credential should be generated. These generation parameters are defined in the variables section as shown below.

```
---
name: demo deploy

variables:
- name: EXAMPLE-PASSWORD
  type: password
- name: EXAMPLE-CA
  type: certificate
  options:
    is_ca: true
    common_name: 'Example Certificate Authority'
- name: EXAMPLE-TLS
  type: certificate
  options:
    ca: EXAMPLE-CA
    common_name: example.com

instance_groups:
 jobs:
 - name: demo
   release: demo
   properties:
     demo:
       password: (( EXAMPLE-PASSWORD ))
       tls:
         certificate: (( EXAMPLE-TLS.certificate ))
         private_key: (( EXAMPLE-TLS.private_key ))
```

Variable Namespacing

Deployment manifests often use common variable names; for example, `(( PASSWORD ))`. To avoid variable name collisions between deployments, the Ops Manager Director automatically stores variables with the Ops Manager Director name and deployment name. For example, the variable `(( EXAMPLE-PASSWORD ))` is stored in CredHub as /Ops-Manager-Director-name/deployment-name/example-password.

Other Namespacing Options

Use a BOSH link to share credentials across deployments. You can read about BOSH links in the v1.11 Release Notice. Alternatively, if you want to use an exact name, prefixing the variable with a forward slash (/) will cause the Director to use the exact name you type. An example of a precisely typed variable follows.

```
((/EXAMPLE-PASSWORD))
```

Migrating Existing Credentials to CredHub

Page last updated:

This topic explains how to migrate non-configurable secrets from Ops Manager into CredHub.

CredHub Credential Types

CredHub uses BOSH credential types, which may have different names from Ops Manager credential types. The following table lists the Ops Manager credential types you can migrate to CredHub and the corresponding CredHub credential types.

Ops Manager Credential Type	CredHub Credential Type	Supported Ops Manager Version
`secret`	`password`	1.11.1
`simple_credential`	`user`	1.12 Alpha 1
`salted_credential`	`user`	1.12 Beta 1
`rsa_pkey_credential`	`rsa`	1.12 Alpha 1

Note: CredHub does not retain the salt when migrating `salted_credentials`.

See Property Reference for more information about credential types.

Use the JavaScript Migration Process

Tile authors can write a JavaScript migration to move their existing non-configurable secrets into CredHub. After a successful migration, Ops Manager deletes credentials from installation.yml.

1. Use the following example to write the JavaScript migration. Save the JavaScript file to the PRODUCT/migrations/v1 directory of your .pivotal tile, following the naming conventions discussed in the Update Values or Property Names Using JavaScript topic.

   ```
   exports.migrate = function(input) {
     input.variable_migrations.push({
       from: input.properties['.PROPERTY-REFERENCE.EXAMPLE-SECRET'],
       to_variable: 'SECRET-VARIABLE'
     });
     return input;
   };
   ```

   In the code block above, replace the example text as follows:

   - `PROPERTY-REFERENCE`: Replace with the property reference that corresponds to the metadata file, such as `properties`. See Tile Upgrades for more information about migrating properties.
   - `EXAMPLE-SECRET`: Replace with the name of the key.
   - `SECRET-VARIABLE`: Choose a variable name for the migrated secret.

2. Remove the property blueprint for the secret and replace it with a CredHub variable.

   - In your metadata, remove the block that includes the credential. For example, remove the block that includes `-name: EXAMPLE-SECRET` and `type: secret`:

     ```
     property_blueprints:
       - name: EXAMPLE-SECRET
         type: secret
       - name: generated_uuid
         type: uuid
       - name: configured_secret
         type: secret
         configurable: true
         optional: true
       - name: configured_simple_credentials
         type: simple_credentials
         configurable: true
         optional: true
     ```

- In handcraft.yml, add a variables section and include the variable name and type:

```
variables:
 - name: SECRET-VARIABLE
   type: password
```

> **Note**: While the property blueprint refers to the above type as `secret`, BOSH refers to the type as `password`. See the CredHub Credential Types table at the beginning of this topic for more information about credential types.

3. In your manifest snippet, replace the existing secret value with the new triple-parenthesis syntax.
   - Remove the existing secret from the manifest snippet:

```
secret: (( .PROPERTY-REFERENCE.SECRET-VARIABLE.SECRET-VALUE ))
```

   - Add the new CredHub variable to the manifest snippet:

```
secret: ((( SECRET-VARIABLE )))
```

4. Run a test deploy of your tile.
5. Use an API endpoint to confirm that the credential is stored in the variable. For more information about the endpoint, see Fetching Variable Names and Values.

Fetching Variable Names and Values

Page last updated:

Overview

CredHub has two API endpoints to identify and re-use variables. Operators who want to see all the credentials associated with their product, or support engineers who want to troubleshoot issues specific to one virtual machine (VM), can use these APIs for those purposes.

The API endpoints perform these functions:

- Identifying and printing the name of a variable
- Using the name of the variable to identify and print the value of the variable

Using the API Endpoints

Use these endpoints to view variables for any product in Ops Manager, except the Ops Manager Director. These endpoints are read-only. You cannot use them to add, remove, or rotate variables.

Fetching Variables

This endpoint returns the list of variables associated with a product that are stored in CredHub. Not all variables are stored in CredHub. If you call a variable that is not stored in CredHub, the call returns an empty value.

```
$ curl "https://OPS-MAN-FQDN/api/v0/deployed/products/product-guid/variables" \
    -X GET \
    -H "Authorization: Bearer EXAMPLE_UAA_ACCESS_TOKEN"
```

Example Response

```
HTTP/1.1 200 OK

{
 "variables": ["FIRST-EXAMPLE-VARIABLE", "SECOND-EXAMPLE-VARIABLE", "THIRD-EXAMPLE-VARIABLE"]
}
```

Query Parameters

Parameter	Description
product_guid	The unique product identifier, formatted as a text string

This endpoint returns a variable's name. Use the name in the next endpoint to return the variable's value.

Fetching Variable Values

This endpoint returns the value of a variable stored in CredHub. Not all variables are stored in CredHub, so if you call a variable that isn't in CredHub, the call will return an empty value.

```
$ curl "https://OPS-MAN-FQDN/api/v0/deployed/products/product-guid/variables?name=EXAMPLE-VARIABLE-NAME" \
    -X GET \
    -H "Authorization: Bearer UAA_ACCESS_TOKEN"
```

Example Response

```
HTTP/1.1 200 OK

{
  "credhub-password": "EXAMPLE-PASSWORD"
}
```

Query Parameters

Parameter	Description
variable_name	The name of the variable, formatted as a text string
product_guid	The unique product identifier, formatted as a text string

Securing Service Credentials with Runtime CredHub

Page last updated:

This topic describes how to develop your Pivotal Cloud Foundry (PCF) service tile to support secure service instance (SSI) credentials using runtime CredHub.

Background

When developers bind an app to a service instance, the binding typically includes *binding credentials* required to access the service.

In PCF v2.0 and later, service brokers can store binding credentials as SSI credentials in runtime CredHub and apps can retrieve these credentials from CredHub. This secures service instance credential management by avoiding the following:

- Leaking environment variables to logs, which increases risk of disclosure.
- Sending credentials between components, which increases risk of disclosure.
- Requiring users to rotate credentials through the environment, which requires container recreation.

To store binding credentials in runtime CredHub, your service tile needs to support the following:

- Discover the location of runtime CredHub.
- Provide this CredHub location to the broker app. The service broker uses the provided location to store binding credentials in CredHub.
- Enable operators to select the SSI credentials option in the tile UI.

Difference between SSI and Internal Service Credentials

SSI credentials, which let apps access services through service instances, are distinct from the credentials that service tiles store in BOSH CredHub for their own internal use.

When a service uses SSI credentials, its service broker stores the binding credentials in runtime CredHub. Then, when PAS binds an app to an instance of the service, the broker retrieves the credentials from runtime CredHub and delivers them to the Cloud Controller (CC) to enable the app to access the service.

These SSI credentials are different from credentials that the tile uses internally, for example, to give the service broker access to an internal database. PAS generates the internal tile credentials for a service when the service is first installed and stores them in BOSH CredHub, not runtime CredHub.

For more information on the CredHub credential management component, see the CredHub documentation topic.

The sections below describe an example implementation of how to add SSI credentials functionality to a service tile.

Step 1: Modify Your BOSH Release

To use runtime CredHub, your service tile needs to retrieve the location of the CredHub server, which is published in the Pivotal Application Service (PAS) tile, through a BOSH link.

Note: BOSH Links let multiple jobs share deployment-time configuration properties. This helps to avoid redundant configurations in BOSH releases and deployment manifests. For more information about BOSH Links, see BOSH Links.

Update Spec File and Templates

The location of runtime CredHub is stored in the `credhub.internal_url` and `credhub.port` properties of the PAS tile. To enable your service tile to retrieve these CredHub-provided properties, add a `consumes:` section with the BOSH link from the PAS tile to the spec file of the BOSH job that will use them and edit the job's templates to access the values in the link:

```
consumes:
- {name: credhub, type: credhub}
```

For information about using BOSH Links in the spec file and templates of a job and consuming shared properties provided by other jobs, see Links in Spec Files and Links in Templates.

Save the Runtime CredHub Location

To use the runtime CredHub location retrieved from the PAS tile, you must write a `post_deploy` tile errand that saves the value out in some way and enables the service broker to access it.

Depending on how your tile deploys the service broker app, the service instance errand can save the CredHub location in different ways. If the tile pushes the broker as a Cloud Foundry app, the errand can store the location in an environment variable such as `CREDHUB_URL` for the service broker to call. If BOSH deploys the service broker outside of of PAS, the errand could write the CredHub location out to a templated configuration file that the service broker reads.

Update Deployment Manifest

In the BOSH release for your tile, edit the deployment manifest `.yml` file so that it contains the BOSH link to CredHub:

```
- name: broker
  release: my-broker-release
  consumes:
    credhub:
      from: credhub
      deployment: cf-XXXXXXXXX
```

For more information about using BOSH links in deployment manifests, see Links in Manifests

Step 2: Enable Your Tile to Find Runtime CredHub

To enable your service tile to discover runtime CredHub, edit your product template so that it consumes the location of CredHub. See the following example:

```
job_types:
- name: JOB-NAME
  resource_label: LABEL-NAME
  templates:
  - name: TEMPLATE-NAME
    release: RELEASE-NAME
    consumes: |
      credhub: {from: credhub, deployment: "(( ..cf.deployment_name ))"}
```

You can also use the address from the BOSH link to verify that the CredHub server is available at that address during tile installation. See the following example:

```
properties:
    aliases:
      (( dig credhub.service.cf.internal @169.256.0.2 )):
      - '*.credhub.(( ..cf.credhub.network )).(( ..cf.deployment_name )).bosh'
```

In the example, the runtime CredHub instance can be accessed at `credhub.service.cf.internal`. If your broker runs as an app, you can resolve this address with BOSH DNS. If your broker runs on a VM with a Consul agent, you can resolve the address with Consul. Alternatively, from a VM, you can resolve the address with `dig credhub.service.cf.internal @169.256.0.2`. This command uses the PAS BOSH DNS server to do lookup.

Step 3: Provide Operators with the Choice to Use CredHub

To provide operators with the choice to select the SSI credentials option, edit your product template. See the following example:

```
form_types:
- name: FORM-NAME
  label: LABEL-NAME
  description: DESCRIPTION
  property_inputs:
  - reference: .JOB-NAME.secure_credentials
    label: Secure service instance credentials
    description: "When checked, service instance credentials are stored in CredHub. Enable only when installing with PCF v2.0 or later and this feature is also enabled in the PAS tile."

property_blueprints:
  - name: hidden_credhub_selector
    type: selector
    configurable: false
    default: "default"
    option_templates:
      - name: default_option
        select_value: "default"
        named_manifests:
          - name: consumes_section_credhub_disabled
            manifest: |
              credhub: nil
          - name: consumes_section_credhub_enabled
            manifest: |
              credhub: {from: credhub, deployment: "(( ..cf.deployment_name ))"}

job_types:
- name: JOB-NAME
  resource_label: LABEL-NAME
  templates:
  - name: TEMPLATE-NAME
    release: RELEASE-NAME
    consumes: |
      "(( secure_credentials.value ? .properties.hidden_credhub_selector.selected_option.parsed_manifest(consumes_section_credhub_enabled) : .properties.hidden_credhub_selector.selected_option.parsed_manifest
  errand: true
  resource_definitions:
    ...
  property_blueprints:
  ...
  - name: secure_credentials
    type: boolean
    configurable: true
    default: false
```

Step 4: Store Binding Credentials in Runtime CredHub

When the CC receives a request to bind a service instance to an app, it forwards the request to the service broker. The service broker then returns the binding credentials that allow access to the service.

To enable your service broker to store binding credentials in runtime CredHub and make them SSI credentials, do the following:

1. In your service broker code, locate where your broker handles binding requests from the CC.
2. Add code that authenticates your service broker to CredHub using OAuth2 tokens from UAA. Each call to the CredHub API must include an authorization header. For more information about CredHub authentication, see the Authentication section of the CredHub API documentation.
3. Update your code to store your binding credentials in CredHub using the CredHub API endpoint for setting the `json` credential type with a user-provided value. See the following example for how to format your API call:

   ```
   curl "https://CREDHUB.INTERNAL_URL:CREDHUB.PORT/api/v1/data" \
   -X PUT \
   -d '{
     "name": "/c/CLIENT-IDENTIFIER/SERVICE-IDENTIFIER/BINDING-GUID/CREDENTIAL-NAME",
     "type": "json",
     "value": {
       "uri": "SERVICE-URL",
       "username": "USERNAME",
       "password": "PASSWORD"
     }
   }' \
   -H 'Content-type: application/json'
   ```

   Where:

   - `CREDHUB.INTERNAL_URL` and `CREDHUB.PORT` are the address and port of CredHub.
   - `CLIENT-IDENTIFIER` is a value provided by the service broker to uniquely identify the broker.

- `SERVICE-IDENTIFIER` is the name of the service offering as shown in the services catalog.
- `BINDING-GUID` is the GUID created by the CC and passed to the service broker in the service binding request.
- `CREDENTIAL-NAME` is a value provided by the service broker to name the credential.
- `SERVICE-URL` is the URL of your service.
- `USERNAME` and `PASSWORD` are your binding credentials.

For further reference, see the Set Credentials section of the CredHub API documentation.

4. Modify your service broker so that it returns a reference to the stored credentials in response to the binding request from the CC. Return the credentials as a single key `credhub-ref` with its value formatted as `/c/CLIENT-IDENTIFIER/SERVICE-IDENTIFIER/BINDING-GUID/CREDENTIAL-NAME`. For example, the binding response might look like the following:

```
{
"credentials": {
  "credhub-ref": "/c/example-service-broker/example-service/faa677f5-25cd-4f1e-8921-14a9d5ab48b8/credentials"
}
}
```

Note: Java Virtual Machine (JVM) apps can use Spring CredHub to access the CredHub API.

Embedded Agents

Page last updated:

This topic provides resources for configuring services that use software agents embedded in application containers.

Overview

Some service integrations depend on the ability to inject code into application containers. Examples include:

- Application Performance Monitoring (APM) agents for monitoring services
- Container-embedded API gateways
- Client-side routers

We refer to these injected components as “container-embedded agents.”

Embedded Agents Resources

- Buildpacks provide a mechanism to inject components into the application container image, and the `.profile.d` directory provides a way to start agents before or alongside the customer application.
- Using .profile.d

Logs, Metrics, and Nozzles

Page last updated:

This topic explains how to integrate PCF services with Cloud Foundry's logging system, the *Loggregator*, by writing to and reading from its *Firehose* endpoint.

Overview

Cloud Foundry's *Loggregator* logging system collects logs and metrics from PCF apps and platform components and streams them to a single endpoint, the *Firehose*. Your tile can integrate its service with the Loggregator system in two ways:

- By sending your service component logs and metrics to the Firehose, to be streamed along with PCF core platform component logs and metrics.
- By installing a *nozzle* on the Firehose that directs Firehose data to be consumed by external services or apps. A built-in nozzle can enable a service to:
  - Drain metrics to an external dashboard product, for system operators
  - Send HTTP request details to search or analysis tools
  - Drain app logs to an external system
  - Auto-scale itself based on Firehose metrics

Firehose-to-syslog is a real world, production example of a nozzle.

Firehose Communication

PCF components publish logs and metrics to the Firehose through Metron agent processes that run locally on the component VMs. Metron agents input the data to the Loggregator system by writing it to Loggregator's etcd key-value store via a gRPC proxy. The topic Overview of the Loggregator System shows how logs and metrics travel from PCF system components to the Firehose.

Component VMs running PCF services can publish logs and metrics the same way, by including a Metron agent that writes to etcd. In PCF v1.10 and later, components only communicate with `etcd` via secure, encrypted `https` protocol. Earlier versions of PCF allow both encrypted `https` and unencrypted `http` communications with etcd.

Secure HTTPS Protocol: PCF 1.10+

To enable a service component to supply logs and metrics to the Firehose through encrypted communications, you need to include a Metron agent and a Consul agent in its template definitions.

The Metron definition includes double-paren properties defining a keypair for accessing etcd. The Consul definition includes double-paren properties for securely looking up the internal IP addresses of the etcd nodes at `cf-etcd.service.cf.internal`. This avoids hard-coding any etcd server addresses.

For example:

```
name: service
label: Service
templates:
  - name: consul
    release: consul
  - name: metron_agent
    release: loggregator
  - name: service
    release: service
manifest: |
  metron_agent:
    deployment: cf-my-service
    etcd:
      client_cert: (( ..cf.properties.cf_etcd_client_cert.cert_pem ))
      client_key: (( ..cf.properties.cf_etcd_client_cert.private_key_pem ))
  metron_endpoint:
    shared_secret: (( ..cf.doppler.shared_secret_credentials.password ))
  loggregator:
    etcd:
      require_ssl: true
      machines: ['cf-etcd.service.cf.internal']
      ca_cert: (( $ops_manager.ca_certificate ))
  consul:
    encrypt_keys:
    - (( ..cf.properties.consul_encrypt_key.value ))
    ca_cert: (( $ops_manager.ca_certificate ))
    agent_cert: (( ..cf.properties.consul_agent_cert.cert_pem ))
    agent_key: (( ..cf.properties.consul_agent_cert.private_key_pem ))
    agent:
      domain: cf.internal
      servers:
        lan: (( ..cf.consul_server.ips ))
```

Metron versions v72 and later do not use etcd to communicate with Loggregator, but the configuration above works with any version of Metron. If the Metron agent does not need values for etcd, it safely ignores them.

HTTP Protocol: PCF 1.9 and Earlier

In PCF v1.9, service components can send logs and metrics to the Firehose encrypted or unencrypted. In v1.8 and earlier releases, components only communicate their log and metrics data unencrypted.

To enable unencrypted communications with etcd, define a Metron agent and list the addresses of the etcd servers in the template definitions as follows:

```
name: service
label: Service
templates:
  - name: metron_agent
    release: loggregator
  - name: service
    release: service
manifest: |
  metron_agent:
    deployment: cf-my-service
  metron_endpoint:
    shared_secret: (( ..cf.doppler.shared_secret_credentials.password ))
  loggregator:
    etcd:
      machines: (( ..cf.etcd_server.ips ))
```

Nozzles

A nozzle is a component dedicated to reading and processing data that streams from the Firehose. A service tile can install a nozzle as either a managed service, with package type `bosh-release`; or as an app pushed to Elastic Runtime, with the package type `app`.

Develop a Nozzle

Pivotal recommends developing a nozzle in Go, to leverage the NOAA library. NOAA does the heavy lifting of establishing an authenticated websocket connection to the logging system as well as de-serializing the protocol buffers.

Draining the logs consists of:

1. Authenticating
2. Establishing a connection to the logging system
3. Forwarding events on to their ultimate destination

Authenticate against the API (https://github.com/cloudfoundry-community/go-cfclient) with a user in the `doppler.firehose` group:

```
import "github.com/cloudfoundry-community/go-cfclient"

...

config := &cfclient.Config{
  ApiAddress:       apiUrl,
  Username:         username,
  Password:         password,
  SkipSslValidation: sslSkipVerify,
}

client, err := cfclient.NewClient(config)
```

Using the client's token, create a consumer and connect to the Firehose with a subscription id. The id is important, since the Firehose looks for connections having the same id and only sends an event to one of those connections. This is how a nozzle developer can prevent message loss during upgrades an other deployments: run at least two instances.

```
token, err := client.GetToken()

consumer := consumer.New(config.TrafficControllerURL, &tls.Config{
  InsecureSkipVerify: config.SkipSSL,
}, nil)
events, errors := consumer.Firehose(firehoseSubscriptionID, token)
```

`Firehose` will give back two channels: one for events and a second for errors.

The events channel receives six different types of events.

- ValueMetric: Some platform metric at a point in time, emitted by platform components. For example, how many `2xx` responses the router has sent out.
- CounterEvent: An incrementing counter, emitted by platform components. For example, a Diego cell's remaining memory capacity.
- Error: An error.
- HttpStartStop: HTTP request details, including both app and platform requests.
- LogMessage: A log message for an individual app.
- ContainerMetric: Application container information. For example, memory used.

For the full details on events, see the dropsonde protocol .

The above events show how this data targets two different personae: platform operators and app developers. Keep this in mind when designing an integration.

Having `doppler.firehose` scope gets a nozzle data for*every* app as well as the platform. Any filtering based on the event payload is the nozzle implementor's responsibility. An advanced integration could do something like combine a service broker with a nozzle to:

- Let app developers opt-in to logging (implementing filtering in the nozzle)
- Establish SSO exchange for authentication such that developers only can access logs for their space's apps

For a full working example (suitable as an integration starting point), see firehose-nozzle .

Deploy a Nozzle

Once you've build a nozzle, you can deploy it as either a managed service or as an app.

As a Managed Service

Visit managed service for more details on what it means to be a managed service.

See also this example nozzle BOSH release.

As an App

You can also deploy the nozzle as an app on Elastic Runtime. Visit the Tile Generator's section on pushed apps for more details.

Example Nozzles

There are several open source examples you could use as a reference for building your nozzle

firehose-nozzle

- Example that simply writes to standard out
- Useful starting point: scaffolding, tests, etc are in place

example-nozzle

- A single file implementation with no tests: as minimal as things can get

gcp-tools-release

- In addition to Nozzle data, it drains component syslogs and health data
- Shows how to do a bosh-addon (for additional data outside a nozzle)
- Nozzle is managed through BOSH
- Raw logs and metrics data take different paths in the source

firehose-to-syslog

- Includes implementation code that adds additional metadata, which might be needed for an access control list (ACL)
  - App name
  - Space UUID and name
  - Org UUID and name
- logsearch-for-cloudfoundry packages this nozzle as a BOSH release

splunk-firehose-nozzle

- Source code based on `firehose-to-syslog`
- Packaged to run an app on PCF

datadog-firehose-nozzle

- Another real world implementation

Log Format for PCF Components

Pivotal's standard log format adheres to the RFC-5424 syslog protocol, with log messages formatted as follows:

```
<${PRI}>${VERSION} ${TIMESTAMP} ${HOST_IP} ${APP_NAME} ${PROD_ID} ${MSG_ID} ${SD-ELEMENT-instance}
${MESSAGE}
```

The Syslog Message Elements table immediately below describes each element of the log, and the Structured Instance Data Format table describes the contents of the structured data element that carries Cloud Foundry VM instance information.

Syslog Message Elements

This table describes each element of a standard PCF syslog message.

Syslog Message Element	Meaning or Value
`${PRI}`	Priority value (PRI), calculated as `8 × Facility Code + Severity Code` Pivotal uses a Facility Code value of `1`, indicating a user-level facility. This adds `8` to the RFC-5424 Severity Codes, resulting in the numbers listed in the table below. If in doubt, default to `13`, to indicate Notice-level severity.
`${VERSION}`	`1`
`${TIMESTAMP}`	The timestamp of when the log message is forwarded; typically slightly after it was generated. Example: `2017-07-24T05:14:15.000003Z`
`${HOST_IP}`	Internal IP address of origin server
`${APP_NAME}`	Process name of the program the generated the message. Prefixed with `vcap`. For example: • `vcap.rep` • `vcap.garden` • `vcap.cloud_controller_ng` You can derive this process name from either the program name configured for the local Metron agent or the `:progname` that blackbox derives from the folder that syslog-release forwards logs into.
`${PROD_ID}`	The Process ID of the syslog process doing the forwarding. If this is not easily available, default to `-` (hyphen) to indicate unknown.
`${MSG_ID}`	The type of log message. If this is not easily available, default to `-` (hyphen) to indicate unknown.
`${SD-ELEMENT-instance}`	Structured data (SD) relevant to PCF about the source instance (VM) that originates the log message. See the Structured Instance Data Format table below for content and format.
`${MESSAGE}`	The log message itself, ideally in JSON

RFC-5424 Severity Codes

PCF components generate log messages with the following severity levels. The most common severity level is `13`.

Severity Code	Meaning
`8`	Emergency: system is unusable
`9`	Alert: action must be taken immediately
`10`	Critical: critical conditions
`11`	Error: error conditions
`12`	Warning: warning conditions
`13`	Notice: normal but significant condition
`14`	Informational: informational messages
`15`	Debug: debug-level messages

Structured Instance Data Format

The RFC-5424 syslog protocol includes a structured data element that people can use as they see fit. Pivotal uses this element to carry VM instance information as follows:

`SD-ELEMENT-instance` element	Meaning
`${ENTERPRISE_ID}`	Your Enterprise Number, as listed by the Internet Assigned Numbers Authority (IANA)

`${DIRECTOR}`	The BOSH director managing the deployment.
`${DEPLOYMENT}`	BOSH `spec.deployment` value
`${INSTANCE_GROUP}`	BOSH `instance_group`, currently `spec.job.name`
`${AVAILABILITY_ZONE}`	BOSH `spec.az` value
`${ID}`	BOSH `spec.id` value. This is a GUID, not an index. Necessary because BOSH Availability Zone index values are not always unique or sequential.

Making Sense of Metrics

Monitoring Pivotal Cloud Foundry has a great rundown of the various metrics and how to make them useful.

Other Resources

- CF Summit Video Monitoring Cloud Foundry: Learning about the Firehose
- Loggregator GitHub repository
- Overview of the Loggregator System
- Loggregator's Slack Channel

Development Tools

Page last updated:

The topics in this section describe tools that Pivotal uses and recommends for tile development.

- Tile Generator takes a service software, a service broker, optional other components, and a simple configuration file and creates a tile and everything else required to deploy your software into PCF.
- The pcf Command Line Utility provides a command line interface for deploying and testing PCF tiles, to avoid the longer process of going through the Ops Manager GUI.
- Concourse is a continuous integration (CI) platform where you can create build pipelines that automate and streamline your tile development and integration with PCF.
- The Services SDK is a suite of tools designed to help you build enterprise-ready service offerings for the Marketplace. The SDK includes the On Demand Service Broker, Service Metrics for PCF, and Service Backups for PCF.

Development Environments

Page last updated:

This topic explains how to set up tile development environments, from simple standalone tools to a full PCF development environment. As you progress through the stages of tile development, you will likely also progress through these environments.

PCF Dev and BOSH Lite

Pivotal provides a lightweight (vagrant packaged) instance of PCF with some basic services as a free product named PCF Dev. This is a great environment to develop and test everything that runs in the Cloud Foundry Elastic Runtime.

Either of these environments allow you to develop the first three levels of service for Pivotal Cloud Foundry (PCF): a User-Provided Service, a Brokered Service, and a Managed Service.

If your integration includes managed services, you will also need an instance of BOSH that can manage virtual machines and BOSH releases for you. BOSH-Lite works well for that purpose.

Between these two components, you will have everything you need to develop tiles, except for Pivotal's Ops Manager. But if you followed the recommended steps in Building Your First Tile you will not need an actual full PCF environment until the later phases of your development.

Setting up BOSH-Lite

- Install BOSH-Lite

> **Note**: For this type of development environment, you only need BOSH-Lite itself to deploy managed service releases. You do not need to follow the instructions to Deploy Cloud Foundry in BOSH-Lite, as Cloud Foundry is provided by the PCF Dev installation above.

Setting up PCF Dev

- Try PCF on your Local Workstation

PWS or Other Supported CF Infrastructure

Pivotal Web Services (PWS) is a highly-available, production-scale PCF environment hosted by Pivotal. You can use it to develop and run PCF apps, but a PWS account does not give access to Ops Manager and its Installation Dashboard, which is where PCF operators install and configure tiles.

- Set Up Your PWS Account and Download the cf CLI explains how to get started with Pivotal Web Services (PWS).

PCF with Ops Manager

Shared PCF Development Environments for Pivotal Partners

Pivotal operates and manages a number of shared PCF development environments, called Pivotal Integration Environments (PIEs), for Pivotal Technical Partnership Program (PTPP) program members to develop their tiles on.

To use your assigned PIE environment:

1. Log in to the Pivotal Tile Dashboard using the credentials that you use for Pivotal Partners Slack.
2. Click the `pie-xx` environment assigned to you.
3. Log in to Ops Manager with the given Ops Manager URL and credentials.
4. Log in to Apps Manager or the cf CLI with the Cloud Foundry information provided on the same page.

If you are not in the PTPP or cannot access Pivotal Partners Slack, email isv@pivotal.io.

Install Your Own PCF Environment

If you need an isolated or dedicated PCF development environment, or you need to work offline, you can install your own environment that includes Pivotal's Ops Manager:

- Installing Pivotal Cloud Foundry
- Operating Pivotal Cloud Foundry
- Upgrading Pivotal Cloud Foundry

The PTPP program does not troubleshoot partner installations of PCF development environments.

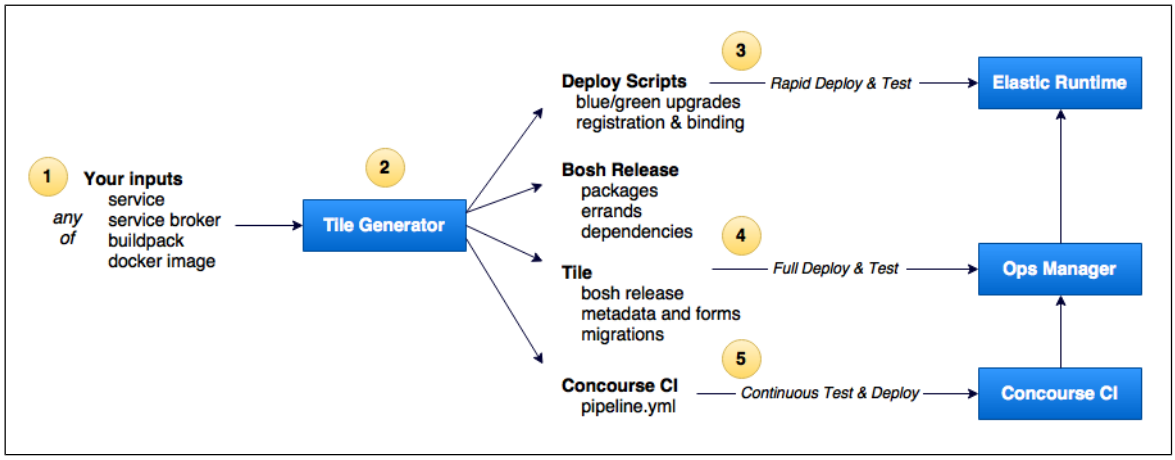
Tile Generator

Page last updated:

This topic describes the Tile Generator tool, which helps tile authors develop, package, test, and deploy services and other add-ons to Pivotal Cloud Foundry (PCF).

Overview

Tiles are the installation package format used by Pivotal Ops Manager to deploy services and other add-ons to both public and private cloud deployments. Tile Generator uses templates and patterns that are based on years of experience integrating third-party services into Cloud Foundry and eliminates much of the need for you to have intimate knowledge of all the tools involved.



Tile Generator takes your software components and a simple configuration file that provides the minimal amount of information to describe and customize your tile. It then creates everything that's required to deploy your software into PCF:

- **BOSH errands** to deploy and delete your software, including blue/green deployments for zero-downtime upgrades
- A **BOSH release** suitable for deploying your software to the Elastic Runtime or open-source Cloud Foundry
- A **Pivotal Ops Manager Tile** that can be imported into Ops Manager, installed, configured, and deployed, including UI forms and automatic upgrades from previous versions
- A **Concourse pipeline configuration** to enable Continuous Integration of your software with the latest versions of PCF

Use Tile Generator in combination with the pcf utility to enable rapid deploy and test cycles of your software.

The current release of Tile Generator supports tiles that have any combination of the following package types:

- Cloud Foundry Applications
- Cloud Foundry Buildpacks
- Cloud Foundry Service Brokers (both inside and outside the Elastic Runtime)
- Docker images (both inside and outside the Elastic Runtime)

Legacy Tiles and OSS-Compatible Service Brokers

Many tile authors, in both Pivotal-internal teams and at external partner companies, built their PCF tiles before Tile Generator existed.

Many other tile authors serve two markets with their service integrations, offering both a Cloud Foundry-compatible service broker to open-source users and corresponding PCF tile for PCF users. They want to continue serving both sets of users.

All of these tile authors can now use Tile Generator to simplify and speed up their development. Tile Generator can generate an OSS-compatible BOSH release service broker BOSH release in addition to a PivNet-ready PCF tile.

Screencast

For a 7-minute introduction into what Tile Generator is and does, see this screencast.

How to Use

1. Install the Tile Generator by doing one of the following:

    - Download the Tile Generator binary for your platform from GitHub, and then make it executable and available by running the following commands:

      ```
      chmod +x TILE-BINARY
      mv TILE-BINARY /usr/local/bin/tile
      ```

      Where:
      `TILE-BINARY` is the name of the tile binary file.

      For example:

      ```
      chmod +x tile_darwin-64bit
      mv tile_darwin-64bit /usr/local/bin/tile
      ```

    - Use Python 2 and Virtualenv. Pivotal recommends using a Virtualenv environment to avoid conflicts with other Python packages.

      A virtualenv is a directory containing dependencies for a project. When a virtual environment is active, packages install into the virtualenv instead of the system-wide Python installation.

      To use this method run the following commands:

      ```
      virtualenv -p python2 tile-generator-env
      source tile-generator-env/bin/activate
      pip install tile-generator
      ```

      This puts the `tile` and `pcf` commands in your `PATH` when the virtualenv is active. To deactivate the virtualenv, run the command `deactivate`.

      Note: To upgrade Tile Generator, run `pip install tile-generator --upgrade` with the virtualenv activated.

2. Install the BOSH CLI.

3. From within the root directory of the project for which you want to create a tile, initialize the directory as a tile repository by running the following commands:

    Note: Pivotal recommends that you use a git repository.

    ```
    cd YOUR-PROD-DIRECTORY
    tile init
    ```

4. Edit the generated `tile.yml` file to define your tile.

5. Build your tile by running:

    ```
    tile build
    ```

The generator first creates a BOSH release in the `release` subdirectory, then wraps that release into a Pivotal tile, in the `product` subdirectory. If required for the installation, it automatically pulls down the latest release version of the Cloud Foundry CLI.

Tile Generator is also available pre-installed in a Docker image on Docker Hub. This image contains the tile-generator `tile` and `pcf` commands, the necessary Python dependencies and the BOSH CLI.

You can use this in Concourse pipelines by specifying it as the base image for your tasks:

```
- task: tile-build
  config:
    platform: linux
    image: cfplatformeng/tile-generator
```

Or, you can derive your own Docker images from this one by using it as the base image in your Dockerfile:

```
FROM cfplatformeng/tile-generator
```

Build the Sample

The tile-generator repository includes a sample tile that exercises most of the features of Tile Generator. This sample tile is used by Tile Generator's CI pipeline to verify that things work correctly. You can build this sample using the following steps:

1. Download the Redis BOSH release and save it to `sample/resources/redis-13.1.2.tgz`.
2. Run the following commands:

```
cd sample
src/build.sh
tile build
```

Note: The sample tile includes a Python app that is re-used in several packages, sometimes as an app, sometimes as a service broker. One of the deployments (app3) uses the sample app inside a Docker image that is currently only modified by the CI pipeline. If you modify the sample app, you have to build your own Docker image using the provided `Dockerfile` and change the image name in `sample/tile.yml` to include the modified code in app3.

Define your Tile in *tile.yml*

All required configuration for your tile is in the file called `tile.yml`. `tile init` creates an initial version for you that can serve as a template. The first section in the file describes the general properties of your tile:

```
name: tile-name # Match Pivotal Network product name, lowercase with dashes
icon_file: resources/icon.png
label: Brief Text for the Tile Icon
description: Longer description of the tile's purpose
```

The `name` should be informative, for example, your company name followed by the product name, e.g., `acme-anvil`. The name should match your product slug on Pivotal Network, which enables update notifications for customers. Coordinate with your product team to agree upon a name; marketing teams often care about the name because it shows up in Pivotal Network URLs.

The `icon_file` should be a 128x128 pixel image that appears on your tile in the Ops Manager GUI. By convention, any resources used by the tile should be placed in the `resources` sub-directory of your repository, although this is not mandatory. The `label` text appears on the tile under your icon.

Packages

Next you can specify the packages to be included in your tile. The format of each package entry depends on the type of package you are adding.

Pushed Apps

Apps (including service brokers) that are being `cf push`ed into the Elastic Runtime use the following format:

```
- name: my-application
  type: app # or app-broker
  manifest:
    # any options that you would normally specify in a cf manifest.yml, including</i>
    buildpack: # required
    command:
    domain:
    host:
    instances:
    memory:
    path:
    env:
    services:
  health_check: none              # optional
  configurable_persistence: true  # optional
  needs_cf_credentials: true      # optional
  auto_services:                  # optional
  - name: p-mysql
    plan: 100MB
  - name: p-redis
    plan: shared-vm
  consumes:                       # optional
    redis:
      from: redis
```

For apps that are normally pushed as multiple files (node.js for example) zip up the project files plus all dependencies into a single ZIP file, then edit `tile.yml` to point to the zipped file:

```
cd <your project dir>
zip -r resources/<your project name>.zip <list of file and dirs to include in the zip>
```

If your app is a service broker, use `app-broker` as the type instead of just `app`. The app is then automatically registered as a broker on install, and deleted on uninstall.

`health_check` lets you configure the value of the cf cli `--health_check_type` option. Expect this option to move into the manifest as soon as CF supports it there. Currently, the only valid options are `none` and `port`.

`configurable_persistence: true` results in the user being able to select a backing service for data persistence. If there is a specific broker you want to use, you can use the `auto-services` feature described below. If you want to bind to an already existing service instance, use the `services` property of the `manifest` instead.

`needs_cf_credentials` causes the app to receive two additional environment variables named `CF_ADMIN_USER` and `CF_ADMIN_PASSWORD` with the admin credentials for the Elastic Runtime into which they are being deployed. This allows apps and services to interact with the Cloud Controller.

The `auto_services` feature is described in more detail below.

`consumes` specifies the BOSH links to consume and presents the hosts and properties from the links as environment variables on the app:

- `<LINK>_HOST`: The address of the first instance of the link.
- `<LINK>_HOSTS`: A JSON array of the addresses of all instances of the link.
- `<LINK>_PROPERTIES`: A JSON object of the properties on the link.

Service Brokers

Most modern service brokers are pushed into the Elastic Runtime as normal CF apps. For these types of brokers, use the Pushed Application format specified above, but set the type to `app-broker` or `docker-app-broker` instead of just `app` or `docker-app`:

```
- name: my-broker
  type: app-broker
  manifest:
    buildpack: # required
    command:
    domain:
    path:
    # ...
  needs_cf_credentials: true          # optional
  auto_services:                      # optional
  - name: p-mysql
    plan: 100MB
  - name: p-redis
    plan: shared-vm
  enable_global_access_to_plans: true  # optional
```

Note: Unless you specify the `enable_global_access_to_plans: true` option, your broker's services do not appear in the user's Marketplaces. Operators have to use the `cf enable-service-access` command to allow specific users, orgs, and spaces to access your services.

Your broker is automatically registered with the Cloud Controller. The Cloud Controller invokes your broker's endpoints, and it uses basic authentication to secure those API calls. The credentials it uses are passed to your broker in two environment variables:

```
SECURITY_USER_NAME
SECURITY_USER_PASSWORD
```

Your broker is expected to accept those credentials. If it doesn't, automatic broker registration fails.

Some service brokers support operator-defined service plans, for instance when the plans reflect customer license keys. To allow operators to add plans from the tile configuration, add the following section at the top level of your `tile.yml`:

```
service_plan_forms:
- name: service_plans_1
  label: Service 1 Plans
  description: Specify the plans you want Service 1 to offer
  properties:
  - name: description
    type: string
    description: "Some Description"
    configurable: true
  - name: license_key1
    type: string
    configurable: true
    description: The license key for this plan
  - name: num_seats1
    type: integer
    configurable: true
    description: The number of available seats for this license
    default: 1
    constraints:
      min: 1
      max: 500
```

Name and GUID fields are supplied by default for each plan, but all other fields are optional and customizable. Multiple forms are supported. The operator-configured plans are passed to your service broker in JSON format in an environment variable named after your form but in ALL CAPS (in this case `SERVICE_PLANS_1`).

For an external service broker, use:

```
- name: my-application
  type: external-broker
  uri: http://broker3.example.com
  username: user
  password: #secret
  internal_service_names: 'service1,service2'
```

BOSH Releases

You can include BOSH releases in your tile with the `bosh-release` package type. For example, here is a package definition to include a Redis BOSH release:

```
- name: redis
  type: bosh-release
  path: resources/redis-13.1.2.tgz
  jobs:
  - name: redis
    templates:
    - name: redis
      release: redis
    memory: 512
    ephemeral_disk: 4096
    persistent_disk: 4096
    instances: 2
    cpu: 2
    static_ip: 0
    dynamic_ip: 1
    default_internet_connected: false
    max_in_flight: 1
    properties:
      password: red!s
  - name: sanity-tests
    templates:
    - name: sanity-tests
      release: redis
    lifecycle: errand
    post_deploy: true
    run_post_deploy_errand_default: when-changed
    memory: 512
    ephemeral_disk: 4096
    persistent_disk: 0
    cpu: 2
    dynamic_ip: 1
```

To include BOSH links in your bosh-release package's deployment manifest, you can include the `consumes` and/or `provides` declarations *as strings* in the job's `templates` section, e.g.:

```
# ...
  jobs:
  - name: job_name
    templates:
    - name: template_name
      consumes:
        consumed_link: {from: foo}
      provides:
        provided_link: {as: bar}
```

Buildpacks

```
- name: my-buildpack
  type: buildpack
  path: resources/buildpack.zip
  buildpack_order: 99     # optional, 99 means end of the list
```

Docker Images

Apps packages as Docker images can be deployed inside or outside the Elastic Runtime. To push a Docker image as a CF app, use the *Pushed Application* format specified above, but use the `docker-app` or `docker-app-broker` type instead of just `app` or `app-broker`. The Docker image to be used is then specified using the `image` property:

```
- name: app1
  type: docker-app
  image: test/dockerimage
  manifest:
    ...
```

If this app is also a service broker, use `docker-app-broker` instead of just `docker-app`. This option is appropriate for Docker-wrapped 12-factor apps that delegate their persistence to bound services.

Docker apps that require persistent storage can not be deployed into the Elastic Runtime. These can be deployed to separate BOSH-managed VMs instead by using the `docker-bosh` type:

```
- name: docker-bosh1
  type: docker-bosh
  cpu: 5
  memory: 4096
  ephemeral_disk: 4096
  persistent_disk: 2048
  instances: 1
  manifest: |
    containers:
    - name: redis
      image: "redis"
      command: "--dir /var/lib/redis/ --appendonly yes"
      bind_ports:
      - "6379:6379"
      bind_volumes:
      - "/var/lib/redis"
      entrypoint: "redis-server"
      memory: "256m"
      env_vars:
      - "EXAMPLE_VAR=1"
    - name: mysql
      image: "google/mysql"
      bind_ports:
      - "3306:3306"
      bind_volumes:
      - "/mysql"
    - name: elasticsearch
      image: "bosh/elasticsearch"
      links:
      - mysql:db
      depends_on:
      - mysql
      bind_ports:
      - "9200:9200"
```

If a Docker image cannot be downloaded by BOSH dynamically, provide a ready-made Docker image and package it as part of the BOSH release. In that case, specify the image as a local file.

```
- name: docker-bosh2
  type: docker-bosh
  files:
  - path: resources/cfplatformeng-docker-tile-example.tgz
  cpu: 5
  memory: 4096
  ephemeral_disk: 4096
  persistent_disk: 2048
  instances: 1
  manifest: |
    containers:
    - name: test_docker_image
      image: "cfplatformeng/docker-tile-example"
      env_vars:
      - "EXAMPLE_VAR=1"
      # See below on custom forms/variables and binding it to the Docker env variable
      - "custom_variable_name=((.properties.customer_name.value))"
```

To expose a container via gorouter, for example, one of the Docker containers hosts an admin webapp interface, use `routes` to choose a port and prefix. The external URL is `[prefix]-[package.name].[system-domain]`. In this case, the URL is `https://admin-docker-bosh3.sys.example.com`, where `sys.example.com` is the PCF system domain. `routes` is a list, so multiple containers can be exposed.

```
- name: docker-bosh3
  type: docker-bosh
  docker_images:
  - "cfplatformeng/database"
  - "cfplatformeng/admin_ui"
  routes:
    - prefix: admin
      port: 8080
  cpu: 5
  memory: 4096
  ephemeral_disk: 4096
  instances: 1
  manifest: |
    containers:
    - name: database
      image: "cfplatformeng/database"
      bind_ports:
      - "5432:5432"
    - name: admin_ui
      image: "cfplatformeng/admin_ui"
      bind_ports:
      - "8080:8080"
```

Custom Forms and Properties

You can pass custom properties to all apps deployed by your tile by adding the to the properties section of `tile.yml` :

```
properties:
- name: author
  type: string
  label: Author
  value: Tile Ninja
```

If you want the properties to be configurable by the tile installer, place them on a custom form instead:

```
forms:
- name: custom-form1
  label: Test Tile
  description: Custom Properties for Test Tile
  properties:
  - name: customer_name
    type: string
    label: Full Name
  - name: street_address
    type: string
    label: Street Address
    description: Address to use for junk mail
  - name: city
    type: string
    label: City
  - name: zip_code
    type: string
    label: ZIP+4
    default: '90310'
  - name: country
    type: dropdown_select
    label: Country
    options:
    - name: country_us
      label: US
      default: true
    - name: country_elsewhere
      label: Elsewhere
- name: account-info-1
  label: Account Info
  description: Example Account Information Form
  properties:
  - name: username
    type: string
    label: Username
  - name: password
    type: secret
    label: Password
```

Properties defined in either section are passed to all pushed apps as environment variables (the name of the environment variable is the same as the

property name but in ALL_CAPS). They can also be referenced in other parts of the configuration file by using `(( .properties.<property-name> ))` instead of a hardcoded value.

All properties supported by Ops Manager may be used. The syntax is the same as used by Ops Manager, except that for simplicity property blueprints for form fields do not need to be declared separately. Instead, the declaration is included in the form itself. For a complete list of supported property types and syntax, see the Ops Manager Product Template Reference.

Properties of type `secret` have their value hidden on the forms and obfuscated in the installation logs (all but the first two characters are replaced by `*****`). But their value is passed to your apps in plain text as all other value types.

Automatic Provisioning of Services

Tile Generator automates the provisioning of services. Any app (including service brokers and Docker-based apps) that are being pushed into the Elastic Runtime can automatically be bound to services through the `auto_services` feature:

```
- name: app1
  type: app
  auto_services:
  - name: p-mysql
    plan: 100mb-dev
  - name: p-redis
```

You can specify any number of service names, optionally specifying a specific plan. During deployment, the generated tile creates an instance of each service if one does not already exist and then bind that instance to your package.

Service instances provisioned this way survive updates, but are deleted when the tile is uninstalled.

Note: The name is the name of the provided *service, not the broker*. In many cases these are not the same, and a single broker may even offer multiple services. Use `cf service-access` to see the services and plans offered by installed service brokers.

If you do not specify a plan, Tile Generator uses the first plan listed for the service in the broker catalog. It is a good idea to always specify a service plan. If you *change* the plan between versions of your tile, Tile Generator attempts to update the plan while preserving the service (thus not causing data loss during upgrade). If the service does not support plan changes, this causes the upgrade to fail.

`configurable_persistence` is really just a special case of `auto_services`, letting the user choose between some standard brokers.

Declaring Product Dependencies

When your product has dependencies on others, you can have Ops Manager enforce that dependency by declaring it in your `tile.yml` file as follows:

```
requires_product_versions:
- name: p-mysql
  version: '~> 1.7'
```

If the required product is not present in the PCF installation, Ops Manager displays a message saying `<your-tile> requires 'p-mysql' version '~> 1.7' as a dependency` and refuses to install your tile until that dependency is satisfied.

When using automatic provisioning of services as described above, it is often appropriate to add those products as a dependency. Tile Generator can not do this automatically as it can't always determine which product provides the requested service.

Orgs and Spaces

By default, Tile Generator creates a single new org and space for any packages that install into the Elastic Runtime, using the name of the tile and appending `-org` and `-space`, respectively. The default memory quota for a newly created org is 1024 (1 G). You can change any of these defaults by specifying the following properties in `tile.yml`:

```
org: test-org
org_quota: 4096
space: test-space
```

Security

If your cf packages need outbound access (including access to other packages within the same tile), you need to apply an appropriate security group. The following option removes all constraints on outbound traffic:

```
apply_open_security_group: true
```

Stemcells

Tile Generator defaults to a recent stemcell supported by Ops Manager. In most cases the default is fine, because the stemcell is only used to execute CF command lines and/or the Docker daemon. But if you have specific stemcell requirements, you can override the defaults in your `tile.yml` file by including a `stemcell-criteria` section and replacing the appopriate values:

```
stemcell_criteria:
  os: 'ubuntu-trusty'
  version: '3146.5'    #NOTE: You must quote the version to force the type to be string
```

Custom Errands

Tile Generator supplies standard errands to deploy and delete CF type packages. You can replace or augment those errands by specifying errand shell commands in your tile.yml file. Here is an example of a custom deploy errand to install a buildpack only if a newer version of that same buildpack is not already present:

```
packages:
- name: my-buildpack
  type: buildpack
  buildpack_order: 0 # Go to head of list
  path: my_buildpack.zip
  deploy: |
    cp my_buildpack.zip my_buildpack-v{{context.version}}.zip
    existing=`cf buildpacks | grep '^my_buildpack'`
    if [ -z "$existing" ]; then
      cf create-buildpack my_buildpack my_buildpack-v{{context.version}}.zip 0
    else
      semver=`echo "$existing" | sed 's/.* my_buildpack-v\(.*\)\.zip/\1/'`
      if is_newer "{{context.version}}" "$semver"; then
        cf update-buildpack my_buildpack -p my_buildpack-v{{context.version}}.zip
      else
        echo "Newer version ($semver) of my_buildpack is already present"
      fi
      cf update-buildpack my_buildpack -i 0
    fi
  delete: |
    # Intentional no-op, as others may have a dependency on this
```

`deploy` and `delete` completely replace the standard errand commands for the package in which you include them. If you want to keep the standard commands, but add additional commands to execute before or after the standard errand, use `pre_deploy`, `post_deploy`, `pre_delete`, and/or `post_delete` instead.

Versioning

Tile Generator uses semver versioning. By default, `tile build` generates the next patch release. Major and minor releases can be generated by explicitly specifying `tile build major` or `tile build minor`. Or to override the version number completely, specify a valid semver version on the build command, e.g. `tile build 3.4.5`.

No-op content migration rules are generated for every prior release to the current release, so that Ops Manager allows tile upgrades from any version to any newer version. This depends on the existence of the file `tile-history.yml`. In a pinch, if you need to be able to upgrade from a random old version to a new one, you can edit that file, or do:

```
tile build <old-version>
tile build <new-version>
```

The new tile then supports upgrades from `old-version`.

Upgrades

By default, Tile Generator produces all code necessary to do a blue/green, zero-downtime deployment of all tile components when installing a newer version over an older one. For most tile versions this is all that is needed.

Ops Manager has support for performing upgrade actions, like database migrations, during a tile upgrade, but this capability is not yet exposed through tile generator.

Example

```
$ tile build
name: tibco-bwce
icon: icon.png
label: TIBCO BusinessWorks Container Edition
description: BusinessWorks edition that supports deploying to Cloud Foundry
version: 0.0.2

bosh init-release --dir=cf
bosh generate-package cf_cli
bosh generate-package bwce_buildpack
bosh generate-job install_bwce_buildpack
bosh generate-job remove_bwce_buildpack
bosh create-release --final --tarball=cf_incubator --version 0.0.2

tile generate release
tile generate metadata
tile generate errand install_bwce_buildpack
tile generate errand remove_bwce_buildpack
tile generate content-migrations

created tile tibco-bwce-0.0.2.pivotal
```

This tile includes a single large buildpack and takes less than 15 seconds to build including the CF CLI download and the BOSH release generation.

Supported Commands

```
tile init [<tile-name>]
tile build [patch|minor|major|<version>]
```

Credits

- sparameswaran supplied most of the actual template content, originally built as part of cf-platform-eng/bosh-generic-sb-release
- frodenas contributed most of the Docker content through cloudfoundry-community/docker-boshrelease
- joshuamckenty suggested the jinja template approach he employed in opencontrol

pcf Command Line Utility

Page last updated:

The `pcf` utility provides a command line interface to Pivotal Cloud Foundry for the purpose of deploying and testing tiles. Its primary reason for existence is to enable Ops Manager access from CI pipelines, but developers also find it convenient to use this CLI rather than the Ops manager GUI.

The `pcf` utility also allows you to test your tile's BOSH errands directly from your CLI, without going through Ops Manager and BOSH. This greatly reduces the time it takes to deploy/test each iteration of your software components.

Installation

The `pcf` utility comes bundled with the Tile Generator tool. To install the `pcf` utility, follow the Tile Generator installation instructions.

Authentication

The `pcf` utility looks for a file called `metadata` in the current directory. This file is expected to provide the URL and credentials to connect to Ops Manager, in the following format:

```
---
opsmgr:
  url: https://opsmgr.example.com
  username: admin
  password: <redacted>
```

The reason for this file naming is because this is how Concourse passes credentials of a "claimed" PCF pool resource to the CI pipeline scripts. For interactive use, this means that you will have to create a `metadata` file in the directory where you run the `pcf` command.

Pivotal recommends that you do **not** create this file inside your git or other version control system repository, as you do not want to accidentally commit these credentials to version control.

Commands

The `pcf` utility implements many different commands. To see available commands:

```
$ pcf --help
Usage: pcf [OPTIONS] COMMAND [ARGS]...

Options:
  --help  Show this message and exit.

Commands:
  apply-changes
  cf-info
  changes
  configure
  delete-unused-products
  import
  install
  is-available
  is-installed
  logs
  products
  settings
  target
  test-errand
  uninstall
```

Checking Ops Manager Settings

To see which products are currently available and installed in Ops Manager:

```
$ pcf products
- p-bosh 1.7.0.0 (installed)
- cf 1.7.0-build.258 (installed)
- test-tile 0.3.95
```

To test if a specific product is available or installed from within a script:

```
$ pcf is-available test-tile && echo "Product test-tile is available"
$ pcf is-installed test-tile && echo "Product test-tile is installed"
```

You can retrieve the settings for a specific product (this will give you a *lot* of json):

```
$ pcf settings test-tile
{
  "network_reference": "669e213111ab5aa1008a",
  "guid": "test-tile-be3e50cf26c530acca6e",
  "jobs": [
    {
      "instance": {
        "identifier": "instances"
      },
      "identifier": "compilation",
      "guid": "compilation-066a85d82fbcd936f9d7",
      "installation_name": "compilation",
      "vm_credentials": {
        "password": <redacted>,
        "salt": <redacted>,
        "identity": "vcap"
      }
    },
    {
      "guid": "deploy-all-b83a7cb7be00ebfd26d6",
      "vm_credentials": {
  ...
```

Deploying Tiles

After your software works and correctly deploys using `test-errand`, you can go through the real Ops Manager deployment process from the CLI, as you would normally do through the Ops Manager GUI.

Import your .pivotal file into Ops Manager:

```
$ pcf import sample/product/test-tile-0.0.2.pivotal
```

Install the uploaded version of your product:

```
$ pcf install test-tile 0.0.2
```

Where you would normally configure the tile settings in the GUI, the `configure` command lets you pass in any user-specified properties as a .yml file. This command also sets the stemcell for the tile to the same one used by your Elastic Runtime, to avoid the need to upload a tile-specific stemcell.

```
$ pcf configure test-tile sample/missing-properties.yml
- Using stemcell bosh-vsphere-esxi-ubuntu-trusty-go_agent version 3215
```

The property file looks like this:

```
---
customer_name: Jimmy's Johnnys
street_address: Cartaway Alley
city: New Jersey
country: US
username: SpongeBob
password: { 'secret': Square'Pants }
app2:
  persistence_store_type: none
# In PCF 1.8+, BOSH-job-specific configuration is supported:
jobs:
  a_job:
    # Job resource configuration:
    resource_config:
      persistent_disk:
        size_mb: "10240"
    # Job-specific property configuration:
    job_property: property_value
```

You must define any `secret` type property value as a hash, in curly brackets. Specifying a simple string value for a field of this type results in a `500 System Error` being returned from `pcf configure`. The `secret` type property values can contain special characters.

To see what changes are ready to be applied:

```
$ pcf changes
install: test-tile-207b165fcb7dc8b2597b
delete:
```

To apply these changes:

```
$ pcf apply-changes
  ===== 2016-04-21 18:45:05 UTC Running "bosh-init deploy /var/tempest/workspaces/default/deployments/bosh.yml"
  Deployment manifest: '/var/tempest/workspaces/default/deployments/bosh.yml'
  Deployment state: '/var/tempest/workspaces/default/deployments/bosh-state.json'

  Started validating
    Validating release 'bosh'... Finished (00:00:08)
    Validating release 'bosh-vsphere-cpi'... Finished (00:00:00)
    Validating release 'uaa'... Finished (00:00:06)
    Validating cpi release... Finished (00:00:00)
    Validating deployment manifest... Finished (00:00:00)
```

`pcf apply-changes` automatically tails the logs for the installation process it started. If this gets aborted for any reason, you can always tail the logs of the most recent installation:

```
$ pcf logs
```

Removing Tiles

To uninstall a tile:

```
$ pcf uninstall test-tile
```

If you accumulate a lot of uninstalled tiles or old versions, you can clean up Ops Manager's available products (and disk space):

```
$ pcf delete-unused-products
```

Accessing Elastic Runtime

To see details about the Elastic Runtime of your PCF environment:

```
$ pcf cf-info
- admin_password: <redacted>
- admin_username: admin
- apps_domain: cfapps-04.example.com
- system_domain: run-04.example.com
- system_services_password: <redacted>
- system_services_username: system_services
```

To target your `cf` command line at this PCF environment:

```
$ pcf target
Setting api endpoint to api.example.com...
OK

API endpoint:   https://api.example.com (API version: 2.52.0)
User:           admin
Org:            my-org
Space:          my-space
API endpoint: https://api.example.com
Authenticating...
OK

...
```

Continuous Integration Testing

Page last updated:

This topic explains how to use the Tile Dashboard continuous integration (CI) system and its underlying Concourse platform to help develop and integrate software services for Pivotal Cloud Foundry (PCF).

Tile Dashboard CI

With your tile in our Tile Dashboard continuous integration testing system, we all win. You stay on top of changes to PCF that may require changes in your tile. Our field representatives gain a clear understanding of your tile's compatibility across PCF versions, underlying IaaS, and different flavors of environments. This also relieves you from maintaining your own CI system, keeping up with latest PCF versions, etc. Further, the Tile Dashboard CI is part of our Enterprise Readiness criteria, which is used to inform the field of the quality and capabilities of your tile, so it is important to get your tile performing well.

Tile Dashboard Steps

Tile Dashboard runs your tile through a series of steps, which include:

- Download your tile from PivNet and check hash integrity.
- Scan your tile for known issues or potential problems, like:
  - Use of deprecated properties.
  - Use of properties whose values/meanings have changed.
  - Use of features that are no longer supported.
- Configure, install, test, and uninstall your tile in several PCF environments:
  - A patch release of every supported ERT/PAS minor version.
  - Every supported IaaS.
  - Environments with extra configuration (e.g., multiple availability zones, IPsec).

Tile Dashboard reports the results of each step. The results report for each step includes a general pass/fail status, the execution log, and output. If a test failed for a reason unrelated to a tile (e.g., a network glitch), you can retry the step from Tile Dashboard.

What Pivotal Needs from You

To integrate your tile with Tile Dashboard CI, Pivotal needs you to upload or send the following:

- Your pre-release tile
- Your tile's configuration parameters
- One or more test configurations
- Any backing services that the tile requires

These requirements are discussed below.

Uploaded Tile

If this is your first tile, Marina or Jake can upload pre-releases of your product. Tile Dashboard will then pick up those pre-releases and run them through CI.

After the first release of your tile, the admin for your tile can continue to upload new pre-releases for future versions to network.pivotal.io.

Tile Configuration Parameters

To automate the installation of your tile, we need the configuration parameters the operator would enter into Ops Manager forms. Tile Dashboard includes an interface for you to enter this configuration directly for properties, in the format used by the om tool. Click on your tile's slug, then click the

“Configure” link near the top of the screen, and you can enter the following information:

- **Properties**: Configure your tile’s properties, if necessary, using the JSON product properties format used by om. (Note: this is the same format used by the Operations Manager product properties API.)
- **UAA Users**: Include a list of UAA users to use for testing your tile. The format is a JSON array, with the specific format described on the configuration page.

Test Configuration

After your tile is installed, Tile Dashboard will run any post-deploy errands your tile has defined, including tests. Ideally your tile will include tests that exercise all of its functionality. We have some ideas for expanding the Tile Dashboard testing capabilities; if you’re interested in other ways of defining tests, please reach out to us on Pivotal Partners Slack.

Backing Services

If your tile requires a backing service outside of the existing PCF environment (e.g., your tile is a service broker to a SaaS offering), you are responsible for maintaining the backing service in an environment that the Tile Dashboard can reach (i.e., it must be internet-facing).

Concourse

The Tile Dashboard CI that Pivotal runs for its technical partnership program members uses the CI tool Concourse to make sure that partner products continue to work with every new release of the platform.

With more effort, you can also follow the pointers below to set up your own Concourse CI pipeline that integrates and tests your tile on your own deployment of the latest PCF.

While you are of course also free to use any other CI system you are familiar with, Pivotal’s tools and documentation are built to make Concourse CI as easy as possible.

Set Up a Concourse Server

You need a Concourse server to host your pipeline.

If you partner with Pivotal, the Tile Dashboard CI servers can host your pipeline and provide S3 storage to exchange artifacts with your own servers.

If you choose to set up your own Concourse server, see the instructions Concourse: Setup & Operations.

Create a Concourse Pipeline for Your Tile

A typical CI pipeline for a tile consists of the following jobs:

- Build the tile
- Deploy it to PCF
- Run a set of deployment tests to verify that it deployed and works correctly
- Remove it from PCF

You describe this pipeline in a `pipeline.yml` file that is then uploaded to the Concourse server. Tile Generator contains a sample pipeline that you can clone for your own tile. We are working on automating the process of generating a pipeline template for you.

Set Up PCF for Your CI Pipeline

Pivotal partners who have us host their pipeline have access to a pool of PCF instances that are managed by us and are regularly updated with the latest (pre-)release versions of PCF. If you set up your own concourse server, you will have to target your pipeline at a PCF instance you have setup.

Concourse has a resource type to manage a pool of resources that are shared between pipelines, which is what we use to serialize PCF access between the partner pipelines that run on our concourse server.

Pivotal Cloud Foundry Services SDK

Page last updated:

Dynamic Provisioning, Metrics, and Backups

The Pivotal Cloud Foundry (PCF) Services SDK is designed to help you build enterprise-ready service offerings for the Marketplace. The SDK includes the following components:

- The On Demand Service Broker enables dynamic provisioning of your service using BOSH 2.0.
- Service Metrics for PCF integrates your service into the PCF Logging and Metrics system, empowering platform operators to gain immediate insight into system health based on live service metrics.
- Service Backups for PCF runs regular backups for your service, triggering and uploading backup artifacts to a range of destinations, including S3 and Azure.

Active Pivotal partners and customers can use the PCF Services SDK by agreeing to the Pivotal SDK EULA when downloading the products on https://network.pivotal.io/.

Publish and Update

Page last updated:

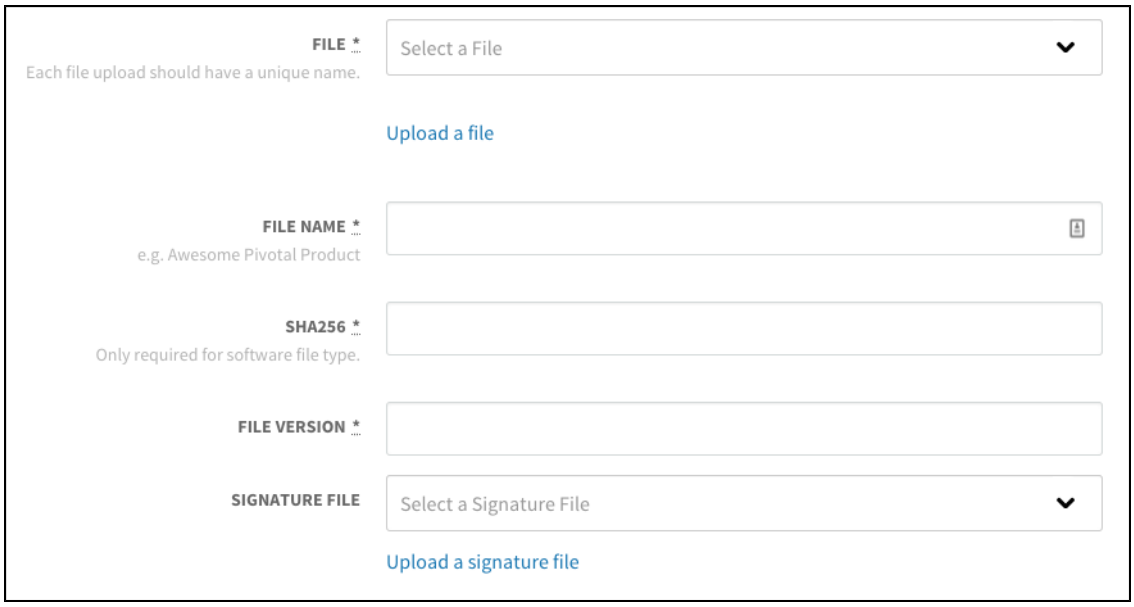
This topic provides resources to help you publish and update your service tile for Pivotal Cloud Foundry (PCF).

Publish Your Tile

The Pivotal Partner Software Product Release Cycle explains how Pivotal works with partners to release PCF products, from the private alpha and closed beta phases, to general availability and publication on Pivotal Network.

After you've packaged your product's BOSH releases, stemcell, metadata, and other tile components into a single zipped download file, post it to Pivotal Network in one of two ways:

- Use the Pivotal Network API command `POST /api/v2/products/:product_slug/product_files`.
- Use the Pivotal Network product upload form.



Update Your Tile

Most tile updates originate with the tile developer, but new releases of PCF can also necessitate tile changes to maintain compatibility with the current version of the platform.

- Tile Generator automates tile versioning and upgrades.
  For more information, see Versioning in the Tile Generator documentation.
- Tile Upgrades explains how to write and include a JavaScript file that automates tile upgrades by migrating property names and values from one tile version to another.
- When changes to PCF require tile changes, Pivotal distributes instructions to all of its partners:
  - Pivotal Cloud Foundry v2.2 Partners Release Notice
  - Pivotal Cloud Foundry v2.1 Partners Release Notice
  - Pivotal Cloud Foundry v2.0 Partners Release Notice
  - Pivotal Cloud Foundry v1.12 Partners Release Notice

Tile Documentation

Page last updated:

This topic explains how to document your service tile for Pivotal Cloud Foundry (PCF).

Overview

When a PCF service tile launches on Pivotal Network, Pivotal publishes corresponding documentation at https://docs.pivotal.io under **Partner Services for Pivotal Cloud Foundry**.

This documentation is formatted in Markdown, stored in a GitHub repository that Pivotal creates, and is published with the bookbinder platform.

Partner Documentation Template

The PCF Partner Documentation Template is a GitHub repository that you can clone to create documentation for your service tile that follows Pivotal's format and works with its documentation publishing platform, bookbinder.

Documentation content resides in the `/docs-content` folder of the repository, as skeleton pages with embedded prompts for content that you should fill in, approximately following the content descriptions below.

See the repository README.md for how to use the template with bookbinder to develop your documentation.

Documentation Content

While the specifics of your documentation will vary depending on the product, we have provided a basic blueprint below. At minimum, documentation should include #1 (Overview) and #2 (Installing/Configuring).

For a good example of a partner service document, see the JFrog Artifactory documentation.

If you have questions or want to collaborate on drafting the documentation, feel free to hop on our Slack channel #pcf-docs. We're always happy to help!

Index/Landing Page

General overview of Partner Product. What does it do? What are its features?

Key Features

- Feature one
- Feature two
- Feature three

Partner Service Broker

A Service Broker allows Cloud Foundry applications to bind to services and consume the services easily from App Manager UI or command line. The Partner Service Broker will enable you to use one or more Partner accounts and is deployed as a Java Application on Cloud Foundry. The Broker exposes the Partner service on the Cloud Foundry Marketplace and allows users to directly create a service instance and bind it to their applications either from the Pivotal Apps Manager Console or from the command line.

The Pivotal Cloud Foundry (PCF) Tile for Partner installs the Partner Service Broker as an application and registers it as a Service Broker on Cloud Foundry and exposes its service plans on the Marketplace. This makes the installation and subsequent use of Partner on your Cloud Foundry applications simple and easy.

If a trial license available, customers interested in using Partner can obtain a 60 day free trial license from edit link here.

Product Snapshot

Current Partner Tile for Pivotal Cloud Foundry Details:

- Version:
- Release Date:
- Software components versions: Partner product version
- Compatible Ops Manager Version(s): 1.5.x, 1.6.x
- Compatible Elastic Runtime Version(s): 1.4.x, 1.5.x, 1.6.x

Requirements (or Prerequisites, Packaging Dependencies for Offline Buildpacks, etc.)

Provide any general or specific requirements here. A general requirement might be something like, “An AppDynamics account.” A specific requirement might be something like, “Packaging Dependencies for Offline Buildpacks.”

Limitations

Any known limitations.

Feedback

Please provide any bugs, feature requests, or questions to the Pivotal Cloud Foundry Feedback list.

Installing/Configuring the Tile

This topic provides instructions for how to install and configure the tile. Typically this includes procedures for how to download the tile from Pivotal Network, install it on Ops Manager, configure the tile, and do any required third-party configuration. Screenshots should be provided where necessary. Consult the following format:

Install Using the Pivotal Ops Manager

- Download the product file from Pivotal Network.
- Upload the product file to your Ops Manager installation.
- Click Add next to the uploaded product description in the Ops Manager Available Products view to add this product to your staging area.
- Click the newly added tile to review any configurable options.
- Click **Apply Changes** to install the service.

Upgrading to the Latest Version

If there are any specific instructions for upgrading the tile, you can include those here. If the procedures are complicated, create a new Upgrading topic.

Configuring the Partner Tile

Add snapshots for each step when possible or add details as required.

- Log in to Pivotal Ops Manager.
- Click **Import a Product** and import the Partner Tile.
- Select the Partner option.
- Click **Add** on the Partner Tile.
- Select the Partner Tile.
- Configure the Partner Tile.
- Apply your changes.

On completion of Partner Tile install, check Services Marketplace in Apps Manager:

- View Partner Service Plans.
- Bind the Partner Service to an Application.
- Check the service or dashboard for the partner for more data.

Other Configurations/Third-Party Configurations

Provide information for specific configurations like configuring for HTTP proxy, or doing any necessary configurations on a third-party service portal.

Using the Tile

This topic provides instructions for how to use the tile. Typically this includes procedures for how to perform the different functions offered by the service. Screenshots should be provided where necessary. You can also include information about Architecture here if necessary.

Troubleshooting

This topic provides troubleshooting information for known errors, following the Symptom/Explanation format used here: https://docs.pivotal.io/p-identity/okta/troubleshooting.html

Release Notes

Include the release notes as the final topic, following the format in the docs-partners-template.

Partner Software Product Release Cycle

Page last updated:

This topic describes the four phases of product release to Pivotal Cloud Foundry (PCF).

Phase 1: Alpha

A product begins development in the **Alpha** phase. The product undergoes constant churn and refactoring, and may not be feature-complete.

Customers do not have exposure to a product during Alpha, and there are no quality requirements in this phase. Instead, developers use this stage for internal testing.

Phase 2: Closed Beta

During **Closed Beta**, a limited pool of users gains access and provides feedback to a product. This feedback drives further development. A status of Closed (Private) Beta informs users that the product may be unstable and should not be used in production.

A product should remain in Closed Beta while:

- Changes may break product function or cause loss of data.
- Users may experience major bugs.
- Users may need to delete and reinstall tiles rather than upgrading them.

Developers make products in Closed Beta available to specific groups or individual customers on Pivotal Network.

Requirements

To enter Closed Beta, a product must meet the following requirements:

- The product must run properly on at least one IaaS, so that customers can install and try it out. Supported infrastructures are AWS, vSphere and OpenStack.
- Customers must be able to install the product error-free through a tile in Pivotal Ops Manager, and delete the product there without any traces remaining.
- The product tile must target the latest released stemcell version, as listed on Pivotal Network.
- The release notes must make clear the following constraints:
    - Potential data loss and lack of support make the beta version of the product unsuitable for use in production.
    - Users will need to delete the old tile and install a new one in order to move to the next version of the product. No upgrade path exists.
- The product must fulfill its promised feature set, and perform as desired.

Pivotal also recommends that any Closed Beta product include an easy way for users to provide feedback to the product developer.

Steps to Release

The following steps create a new Closed Beta release for your product:

1. Log into Pivotal Network.
2. Create a new release for your product and populate all of the required fields.
3. Check that the release version states **BETA**.
4. Clearly state in the release description that the product cannot be upgraded, and that users may suffer data loss.
5. Email your Pivotal contact to request product validation and Closed Beta release. Please provide basic instructions on how to validate the new feature set. Pivotal will verify that the release meets all requirements, then make it accessible to invited customers.

Phase 3: Public Beta

Your product will be made available to the general public in **Public Beta**. The wider pool of users increases public awareness and feedback and facilitates marketing and advertising. As development continues, you may publish a series of product versions in Public (Open) Beta.

Your product is a good candidate for the **Public Beta** stage if:

- You have high confidence that further development will not break the product or incur data loss for users.
- The tile can be upgraded.
- You still want user feedback to discover minor bugs and evaluate existing features.
- The product does not contain the full set of features intended for the final release.
- You feel comfortable supporting this tile for customers.

Products in Public Beta are available on Pivotal Network to any user with a free Pivotal Network account.

Requirements

Products in Public Beta must meet the following requirements:

- The product meets all requirements for **Closed Beta**.
- The tile can be upgraded to subsequent versions without requiring the customer to uninstall the previous version.
- The product supports upgrade paths from any minor version or patch to the next minor version and any patches.
- Tile version upgrades result in no data or configuration loss, and maintain service functionality and availability.
- Where appropriate, PCF integrations work properly, including:
  - Registered routes
  - UAA
  - Service brokers
- You can respond to discovery of a security flaw on the Common Vulnerabilities and Exposures (CVE) list within a reasonable time frame. Security flaws include vulnerabilities in your stemcell or within one of the components of your tile.

> **Note**: Pivotal attempts to respond to all critical CVEs within 48 hours.

Steps to Release

1. Log into Pivotal Network.
2. Create a new release for your product and populate all of the required fields.
3. Check that the release version states **BETA**.
4. Email your Pivotal contact to request product validation and Public Beta release. Please provide basic instructions on how to validate the new feature set. Pivotal will also validate the upgrade scenario and data persistence. After verifying that the release meets all requirements, Pivotal will make it visible to customers.

Phase 4: General Availability

A product qualifies for **General Availability** when:

- It is production-ready.
- You can charge money for this product and provide support guarantees to your customers.
- The product's full set of features meets the standards of quality that you wish to uphold.

Requirements

Products must meet the following requirements for **General Availability**:

- The product meets all requirements for **Public Beta**.
- You consider the product production-ready, and you have adequate unit and functional tests to ensure high quality.
- You can provide customer support.
- Your business team can “Go to market.”
- The product can scale vertically, by increasing the amount of RAM or CPU. Vertical scaling improves performance and does not result in data loss.
- If appropriate, the product can scale horizontally for high availability.
    - Scaled-out nodes (application VMs) function correctly.
    - Removing a node does not result in downtime.
- If appropriate, the product supports zero downtime deployment.
- Product installation does not require an internet connection, after initial product download.

Steps to Release

1. Log into Pivotal Network.
2. Create a new release for your product and populate all of the required fields.
3. Email your Pivotal contact to request product validation and General Availability release. Please provide basic instructions on how to validate the new feature set. Pivotal will also validate the upgrade scenario and data persistence.

Upgrading Tiles

Page last updated:

This topic discusses product tile migrations, which refers to changing the name and values of properties when a customer upgrades tile versions. Tile authors supply a JavaScript file to trigger chaining migrations. Chaining migrations allows for multiple migrations to run sequentially.

Note: In order to use JS migrations, ensure you are using Ops Manager 1.7 or later.

Note: Changing the value of `single_az_only` for jobs launched by your tile can cause data loss for customers who upgrade to Ops Manager v1.7 versions older than v1.7.20, or v1.8 versions older than v1.8.12. Contact Pivotal Support for help avoiding this.

Update Values or Property Names Using JavaScript

To update a product tile, tile authors must complete the following steps:

1. In a single `.js` file, write JavaScript functions which return a hash of the tile's properties.
2. Name the file in the format `TIMESTAMP_NAME.js`. TIMESTAMP must be in the form "YYYYMMDDHHMM" to indicate when the author created the migration. NAME is a human-readable name for the migration, for example, `201606150900_example-product.js`.
3. Copy the `TIMESTAMP_NAME.js` file to the PRODUCT/migrations/v1 directory.

Example JavaScript Migration File

The functions below display an example migration file:

```
exports.migrate = function(input) {
  // Append text to a string

  input.properties['.web_server.example_string']['value'] += '!';

  // Delete property 'legacy_property' that's removed in new tile version
  delete input.properties['.properties.legacy_property'];

  // Rename property 'example_port' to 'example_port_renamed',
  // retaining the previous value.
  input.properties['.properties.example_port_renamed'] =
    input.properties['.properties.example_port'];
  delete input.properties['.properties.example_port'];

  // Append text to a string list
  input.properties['.properties.example_string_list']['value'].push(
    'new-string-append-by-migration');

  return input;
};
```

The properties object passed to your anonymous JavaScript migration functions are composed of properties at the job-level and product-level. Review the property names in the example metadata file in Tutorial Tile V3 for more information about job-level and product-level properties. The tile author must update migrations to match the corresponding product metadata file.

Each property's key in the properties object is its property reference from the metadata file. Property references use one of the following forms:

- `.properties.{property_name}` for product-level properties
- `.{job_name}.{property_name}` for job-level properties
- `.properties.{property_name}.options.{option_name}` or `.{job_name}.{property_name}.options.{option_name}` for selector option properties

The object accessed through the property reference contains a value key whose structure is specific to the type of the property. Objects may be a string, an array, or a hash. Review the reference below for the structure of each type of property.

JavaScript Migrations API

Inside a JavaScript migration function, the system provides the following functions for your code:

```
console.log(string)
Arguments:  string
Return value:  none
Description:  Prints the string to the Rails log
Example:
 console.log("Hello World");
```

```
getCurrentProductVersion()
Arguments:  none
Return value:  string (example:  1.7.1.0)
Description:  Returns the version of the product that is currently installed
Example:
 console.log(getCurrentProductVersion());
```

```
generateGuid()
Arguments:  none
Return value:  string (example:  115f9ced-3167-4c7c-959b-d52c07f32cbf)
Description:  Returns a globally unique identifier (GUID) that can be used as the unique identifier for each element of a Collections property.  When updating a Collection property blueprint, you as the migration a
Notes:  This function can be called a maximum of 100 times per `.js` file.  If you need more than 100 GUIDs, break your migration into two `.js` files.
Example:
 console.log("Here's a GUID:  "+generateGuid())
```

```
abortMigration(string)
Arguments:  string containing error message
Return value:  none (never returns)
Description:  Causes the migration to fail immediately.  Rolls back all migrations in the current chain, i.e, no changes will be committed.
Example:
 if (something > 5) {
   abortMigration("Can't upgrade tile when the value of something is more than 5")
 }
```

Property Type	Value Structure	Example
`single-value properties`	Single value, but type-specific	`properties['.properties.my-prop'].value = 'my-string'; properties['.properties.other-prop'].value = true`
`dropdown`	Array of options	`properties['.properties.my-prop'].value = ['option1', 'option2']`
`rsa_cert_credentials`	Object	`properties['.properties.my-prop'].value = {'private_key_pem' => 'a-private-key', 'cert_pem' => 'a-cert-pem'}`
`rsa_pkey_credentials`	Object	`properties['.properties.my-prop'].value = {'private_key_pem' => 'a-private-key'}`
`salted_credentials`	Object	`properties['.properties.my-prop'].value = {'identity' => 'an-identity', 'salt' => 'mortons', 'password' => 'books'}`
`simple_credentials`	Object	`properties['.properties.my-prop'].value = {'identity' => 'an-identity', 'password' => 'secret'}`
`collections`	Array of objects	`properties['.properties.my-prop'].value = [{name: {value: 'foo'}, record_id: {value: 1}}, {name: {value: 'bar'}, record_id: {value: 2}}]`
`selectors` *Selected value*	String	`properties['.properties.my-prop'].value = 'selected option label'`

`selectors` *{selector option name.property name}*	Value object specific to property type	`properties['.properties.selector.option1.prop1'].value = 'foo'`
		`properties['.properties.selector.option1.prop2'].value = 2`
		`properties['.properties.selector.option2.prop3'].value = ['bar', 'baz']`

Single value properties refer to properties whose type are any of the following: boolean, ca_certificate, domain, dropdown_select, email, http_url, integer, ip_address, ip_ranges, ldap_url, multi_select_options, network_address, network_address_list, port, smtp_authentication, string, string_list, text, uuid.

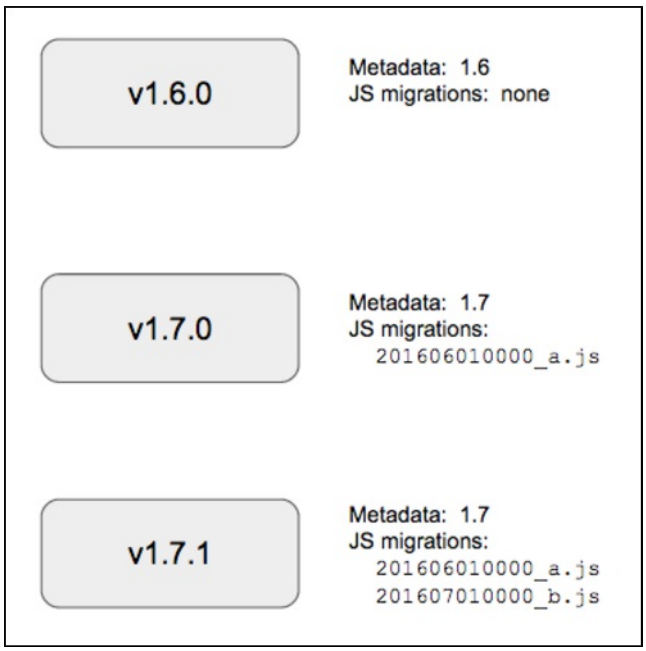
Refer to the example properties below when writing your own tile migration JS file:

```
{
  properties: {
    '.properties.example_boolean': { value: false } ,
    '.properties.example_ca_certificate': { value: 'simple-typed-value'},
    '.properties.example_domain': { value: 'simple-typed-value'} ,
    '.properties.example_dropdown_select': { value: 'simple-typed-value'},
    '.properties.example_email': { value: 'simple-typed-value'},
    '.properties.example_http_url': { value: 'simple-typed-value'},
    '.properties.example_integer': { value: 111},
    '.properties.example_ip_address': { value: 'simple-typed-value'},
    '.properties.example_ip_ranges': { value: 'simple-typed-value'},
    '.properties.example_ldap_url': { value: 'simple-typed-value'},
    '.properties.example_multi_select_options': { value: ['simple-typed-value']},
    '.properties.example_network_address': { value: 'simple-typed-value'},
    '.properties.example_network_address_list': { value: 'simple-typed-value'},
    '.properties.example_port': { value: 22},
    '.properties.example_smtp_authentication': { value: 'simple-typed-value'},
    '.properties.example_string': { value: 'simple-typed-value'},
    '.properties.example_string_list': { value: 'simple-typed-value'},
    '.properties.example_text': { value: 'simple-typed-value'},
    '.properties.example_uuid': { value: 'simple-typed-value'},
    '.properties.example_rsa_cert_credentials': {
      value: {'private_key_pem': 'a-private-key', 'cert_pem':'a-cert-pem'},
    },
    '.properties.example_rsa_pkey_credentials': {
      value: {'private_key_pem':'a-private-key'},
    },
    '.properties.example_salted_credentials': {
      value: {'identity':'an-identity', 'salt':'mortons', 'password':'books'},
    },
    '.properties.example_simple_credentials': {
      value: {'identity':'an-identity', 'password':'secret'},
    },
    '.properties.example_collection': [
      {name: {value: 'foo'}, record_id: {value: 1}},
      {name: {value: 'bar'}, record_id: {value: 2}}
    ],
    '.properties.example_selector': {value: 'option1'},
    '.properties.selector.option1.prop1': {value: 'foo'},
    '.properties.selector.option1.prop2': {value: 2},
    '.properties.selector.option2.prop3': {value: 'bar,baz'}
  }
}
```

Note: If your product uses Ops Manager 1.6 or earlier metadata, you need to write a transmogrifier content migration for customers using your product on 1.6, and a JavaScript migration for those on Ops Manager 1.7 or later. Review the transmogrifier example in the Tile Tutorial V1.

Examples Demonstrating Chaining Migrations

Migration chaining allows for multiple migrations to run sequentially when an upgrade is performed that skips an intermediate version. For example, suppose you have three versions of your product: 1.6.0, 1.7.0, and 1.7.1. The 1.6.0 product contains 1.6 metadata, so it does not contain any JavaScript migrations.



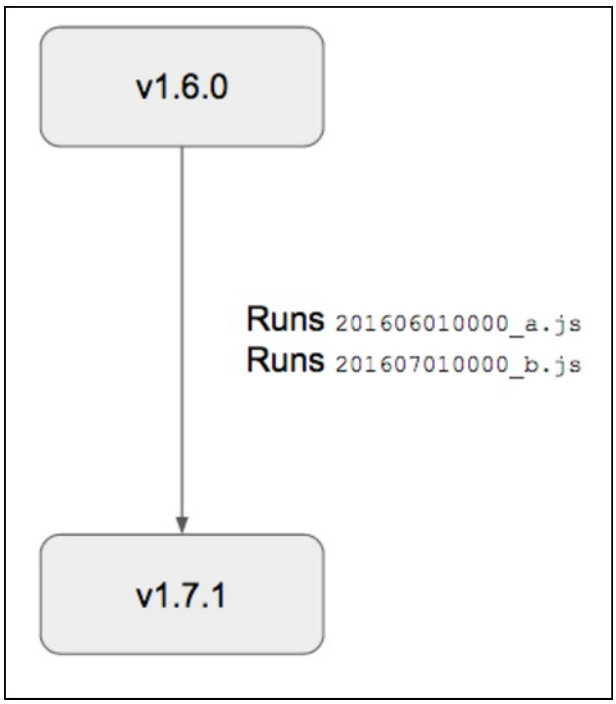
The following customer upgrade scenarios illustrate chaining migrations in more detail, and use the example product versions described above.

Scenario A: Upgrading from 1.6.0 -> 1.7.0 -> 1.7.1

In this scenario, the customer starts with the 1.6.0 product installed. After upgrading to Ops Manager 1.7 or higher, they decide to upgrade the product to 1.7.0. This causes the `migration 201606010000_a.js` to run. Several weeks later, the customer decides to upgrade from 1.7.0 to 1.7.1. Now the `201607010000_b.js` migration runs. Even though the 1.7.1 product includes both migrations, Ops Manager does not re-run `201606010000_a.js`, because it maintains a record of migrations.

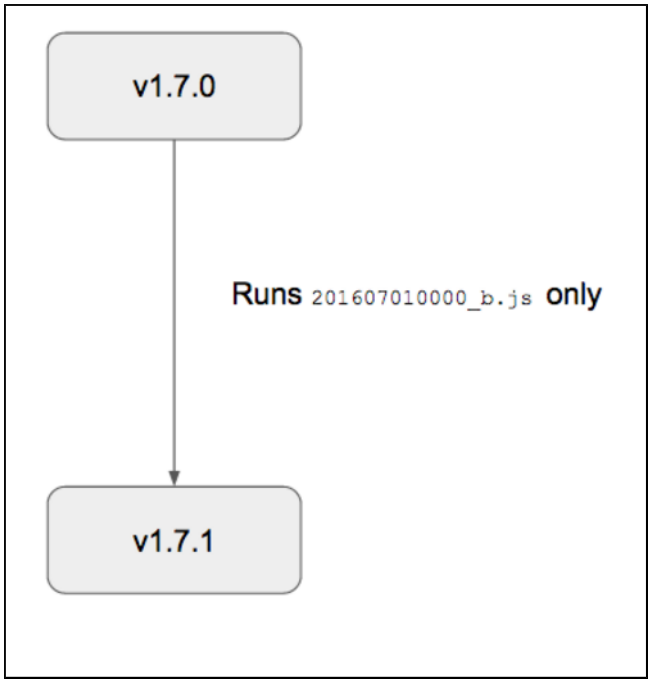
Scenario B: Upgrading Directly from 1.6.0 -> 1.7.1

In this scenario, the customer also starts with 1.6.0 installed, but they decide to upgrade directly to 1.7.1, skipping the 1.7.0 version. Both migrations run in lexicographical order.



Scenario C: Installing 1.7.0, Then Upgrading to 1.7.1

In this scenario, the customer starts with nothing installed. They perform a clean install of version 1.7.0 of the product. On install of 1.7.0, no migrations run because migrations only run on upgrades. Later, the customer decides to upgrade to 1.7.1 of the product. Because 1.7.1 contains both migrations, and because no migrations have run on this system, only the second migration `201607010000_b.js` runs. The system recorded the fact that 1.7.0 includes `201606010000_a.js`, so that migration does not run.



Scenario D: Installing 1.7.1

In this scenario, the customer performs a clean install of 1.7.1, with no previous versions of the product installed. Since migrations are only triggered by upgrade events, no migrations run.

Note: Do not omit a migration from a later version of your tile. This breaks the “chaining” nature of migrations. Using the example above, if you

release a 1.7.1 tile without the `201606010000_a.js` migration, the system could not detect that `201606010000_b.js` is the same migration that was present in the clean install in Scenario C.

References

Page last updated:

This topic collects API, configuration property, and other references for building Pivotal Cloud Foundry (PCF) tiles.

Troubleshooting

Sooner or later you will run into problems that require digging a little bit deeper. Here are some great resources on how to best troubleshoot more complex issues:

- Troubleshooting PCF
- Troubleshooting Applications
- Advanced Troubleshooting with BOSH

API

- Service Broker API v2.10 lists the requests, responses, and status codes required for a service broker.
- Catalog Metadata lists the metadata fields that a service broker must publish to create listings in the Services Marketplace.
- Subcommands from the On-Demand Services SDK documentation lists the subcommands that ODB service adapter must respond to.

Configuration Properties

- Product Template Reference catalogs how top-level properties, form properties, property blueprints, configurable properties, and job types are defined in tile installer `.yml` files, such as those generated by the Tile Installer or hand-coded legacy tiles.
- Property Blueprint Reference compiles another list of accessors and values for configuration properties in the `property_blueprints` section of a tile installer `.yml` file.
- Referencing Properties explains how to specify the locations of tile configuration properties in a tile installer `.yml` file.

Command Line Tools

- Cloud Foundry CLI Reference Guide catalogs the cf CLI.
- pcf Command Line Utility catalogs the `pcf` utility, which you can use to bypass Ops Manager.
- The Fly CLI catalogs the `fly` command-line interface to Concourse.

Partners Release Notices

- Pivotal Cloud Foundry v2.2 Partners Release Notice
- Pivotal Cloud Foundry v2.1 Partners Release Notice
- Pivotal Cloud Foundry v2.0 Partners Release Notice
- Pivotal Cloud Foundry v1.12 Partners Release Notice

Development Workflow Reference

Page last updated:

This document references topics that follow Pivotal's recommended tile development workflow in Building Your First Tile.

Development Workflow

The following topics can help you learn the necessary background information to publish and maintain a finished tile product:

- PCF Tile Developer Guide
- Tile Basics describes how PCF, service brokers, and tiles work together, and how tiles are structured.
- Types of Intergrations gives a high-level view of a staged tile development process that iterates through increasing levels of integration:
  - User-Provided Service
  - Brokered Service
  - Managed Service
  - On-Demand Service
- Development Environments describes how to set up development environments for different stages and levels in the tile development process.
- Development Tools describes three tools that streamline the tile development process: Tile Generator, the pcf Command Line utility, and Concourse continuous integration (CI).
- Tile Documentation explains how to document your tile as part of PCF documentation.
- Publish and Update explains how to publish your tile on Pivotal Network (PivNet) and package upgrade information into your new versions.
- Reference provides language references for tile elements such as the Service Broker API and the Properties list for tile configuration.
- Contact Us lists contacts to learn more about the Pivotal ISV Partner Program or request our assistance with your integration project, and explains where you can contribute to this documentation.

Product Template Reference

Page last updated:

This document defines the separate pieces of a product template. For the purpose of explanation we use the PCF example tile, a functional tile provided by the Ops Manager engineering team that deploys the NGINX web server.

The product template, a `.yml` file in the tile's `metadata` subdirectory, includes or points to the following:

- Metadata: high level information about your tile
- Dependencies: how to specify product dependencies
- Property Blueprints: the building blocks of representing values
- Form Types: exposing property blueprints into generated forms
- Jobs

Top Level Properties

The following is an example of the properties that appear at the top of a product template. Following this example are definitions of each property.

```
---
  name: example-product
  product_version: <%= version.inspect %>
  minimum_version_for_upgrade: "1.7.0"
  pivnet_filename_regex: "/product-.*\.pivotal$"
  metadata_version: "1.11"
  label: 'Ops Manager: Example Product'
  description: An example product to demonstrate Ops Manager product-author features
  rank: 1
  service_broker: false # Default value
  stemcell_criteria:
    os: ubuntu-trusty
    version: <%= stemcell_version.inspect %>

    enable_patch_security_updates: true
  releases:
    - name: example-release
      file: <%= release_file_name.inspect %>
      version: <%= release_file_name.match(/^example-release-(.*)\.tgz$/)[1].inspect %>

  variables:
    - name: credhub-password
      type: password

  post_deploy_errands:
    - name: example-errand
  pre_delete_errands:
    - name: example-errand
```

name

String. Required. The internal name of the product. You must keep the name of your product consistent for migrations to function properly. Changing the name indicates the installation of a completely different product.

product_version

String. Required. The version of the product. At present you can only import this version into Ops Manager once. If you intend to import the same product / version, you must delete the existing one from the `/metadata` folder and delete the installation files from Ops Manager's disk. The version number is important for migrations.

minimum_version_for_upgrade

String. Required. You must set a minimum version for upgrading to your current product version. This example shows a current product version of v1.7

that only upgrades from a v1.6.x version of the same product:

```
- product_version:  1.7.0.0
  minimum_version_for_upgrade:  1.6.0.0
```

metadata_version

String. Required. The versioned structure of the product template (the file you are editing). Changing the version number can unlock new properties, and also break properties that changed from previous versions. The metadata version does not always correlate to Ops Manager version number and depends on what, or if, new metadata properties were introduced.

label

String. Optional. The label that appears in the product tile when it displays in the Ops Manager Dashboard.

description

String. Optional. A description of the product. This is not currently used but may be displayed in a future version of Ops Manager.

rank

Integer. Required. The order in which a product tile appears on the dashboard. The Ops Manager Director always appears at rank 100. For your product to appear to the right of Ops Manager Director (preferable), you must set this value to an integer less than 100. Pivotal recommends that you set it to 1. Ops Manager sorts tiles alphabetically if all tiles have the same rank. This is a known weak point.

pivnet_filename_regex

String. Optional. This regular expression allows Ops Manager's Pivotal Network integration to pull a specific product file. You must do this when there are multiple products within the same product slug.

service_broker

Boolean. Optional, default `false`. Set `service_broker` to `true` for on-demand service brokers. Setting `service_broker` to `true` does the following:

- Enables the service network selector property type
- Requires the operator to select a service network during tile configuration. Tile authors can reference the selected service network with `(( $self.service_network ))`.
- Includes a UAA client for the service to use. Tile authors can reference the UAA client credentials with `(( $self.uaa_client_name ))` and `(( $self.uaa_client_secret ))`.

stemcell_criteria

Hash. Required. For a list of stemcells, including OS and version, see the BOSH hub. You do not specify which IaaS the Stemcell targets. This keeps your product template IaaS agnostic so that one product template can be deployed on any IaaS. At the time of this writing, none of the BOSH stemcells require a Cloud Provider Interface (CPI). This is expected to change in a future release of BOSH.

`enable_patch_security_updates` allows you to automatically use the latest patched version of a stemcell. This is by default set to `true`. For products using static compilations, you can disable this feature. If you set the property to `false`, your product does not receive security patches through automatic stemcell updates.

```
stemcell_criteria
os: ubuntu-trusty
version: <%= stemcell_version.inspect %>
enable_patch_security_updates: true
```

This feature increases security by automatically using the latest patched version of a stemcell. However, operators may experience longer than expected upgrade times. For more information, see Understanding Floating Stemcells.

releases

Array of Hashes. Required. The list of releases contained in your product's releases directory. The version of the release must be exactly the same as the version contained in the release (BOSH releases are versioned and signed by BOSH).

Each release requires the following keys:

- `name`
- `file`
- `version`

variables

Array of Hashes. Optional. A list of variables, that are generated after a deploy succeeds. You can reference variables in a manifest snippet using triple-parentheses expressions.

Each variable requires a `name` and a `type`.

post_deploy_errands

Array of Hashes. Optional. A list of errands that run after a deploy succeeds.

Set the `run_post_deploy_errand_default:` property to `on` or `off` to set the default for the errand's run rule selector in Ops Manager. See Lifecycle Errands. If this property is not supplied, the selector defaults to `On`.

pre_delete_errands

Array of Hashes. Optional. A list of errands that run before a deployment is deleted.

Set the `run_pre_delete_errand_default:` property to `on` or `off` to set the default for the errand's run rule selector in Ops Manager. See Lifecycle Errands. If this property is not supplied, the selector defaults to `On`.

icon_image

Base64 Image. Required. This is the icon that displays on the tile in the Ops Manager Installation Dashboard.

Form Properties

Each form type you write is composed of form properties. Form properties represent the outline to the form fields that appear in the Ops Manager UI. The `name` of each form appears on the left-hand side as navigational tabs.

Form properties reference `property_blueprints`. Property blueprints define each field's data type. For a corresponding example to the `form_types` example below, see property_blueprints.

The following is an example of the properties that appear in the `form_types` section of a product template:

```
form_types:
  - name: example-form
    label: Configurable Properties
    description: All the properties that you can configure!
    markdown: |
      ## Example markdown text

      ![Alt text](http://placekitten.com/g/400/200)

      Things to do:

      1. Learn [markdown](https://daringfireball.net/projects/markdown/).
      1. ...
      1. Profit!
    property_inputs:
      - reference: .web_server.example_string
        label: Example string
        description: 'Configure a property of type string'
      - reference: .web_server.example_string_with_placeholder
        label: Example string containing Placeholder text
        description: 'Optional field. Configuration not necessary'
        placeholder: 'Ghost text. Spooky!'
      - reference: .web_server.example_migrated_integer
        label: Example integer
        description: 'Configure a property of type integer'
      - reference: .web_server.example_boolean
        label: Example boolean
```

name

String. Required. The internal name of the form.

label

String. Required. The label of the form as it appears as a link on the left hand side of each form.

description

String. Optional. The description of the form. Appears at the top of the form as a header.

markdown

Markdown. Optional. Provide a block of markdown to display at the top of the form. Includes image support. You can use this property to document the tile and provide explanations or references.

property_inputs

Array of Hashes. Required. References to properties defined in the property_blueprints section of the product template.

verifiers

Verifiers reach out and find objects in the world. For example, given an IP, a verifier can ping the IP to see that it responds.

Verifiers are separate from validators, which check whether a string is formatted properly. For an example of a validator, see must_match_regex.

See the following for a list of available verifiers you can use:

- `BlobstoreVerifier`
- `LDAPBindVerifier`
- `MysqlDatabaseVerifier`
- `SmtpAuthenticationVerifier`

- `SsoUrlVerifier`
- `StaticIpsVerifier`
- `WildcardDomainVerifier`

placeholder

String. Optional. Specify placeholder text for a field. The text appears in light gray to show an example value for the user. The text disappears when the user types in the field and reappears if the user leaves the field empty.

The `placeholder` attribute displays for the following form types:

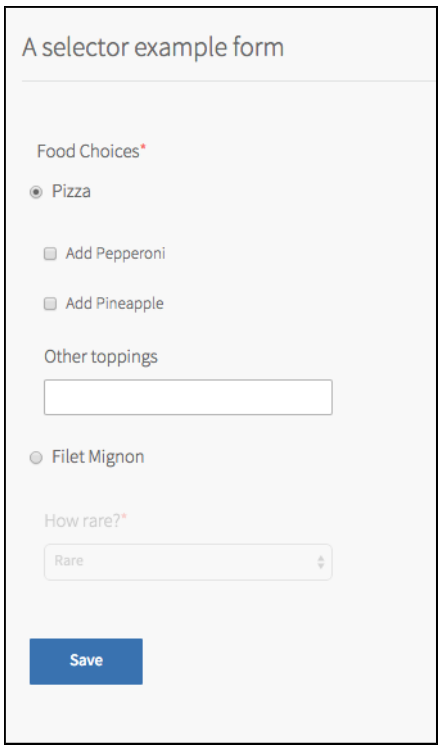
- string
- integer
- domain
- wildcard_domain
- string_list
- text
- ldap_url
- email
- http_url
- ip_address
- ip_ranges
- network_address_list
- network_address
- port

Simple vs. Complex Inputs (Selectors and Collections)

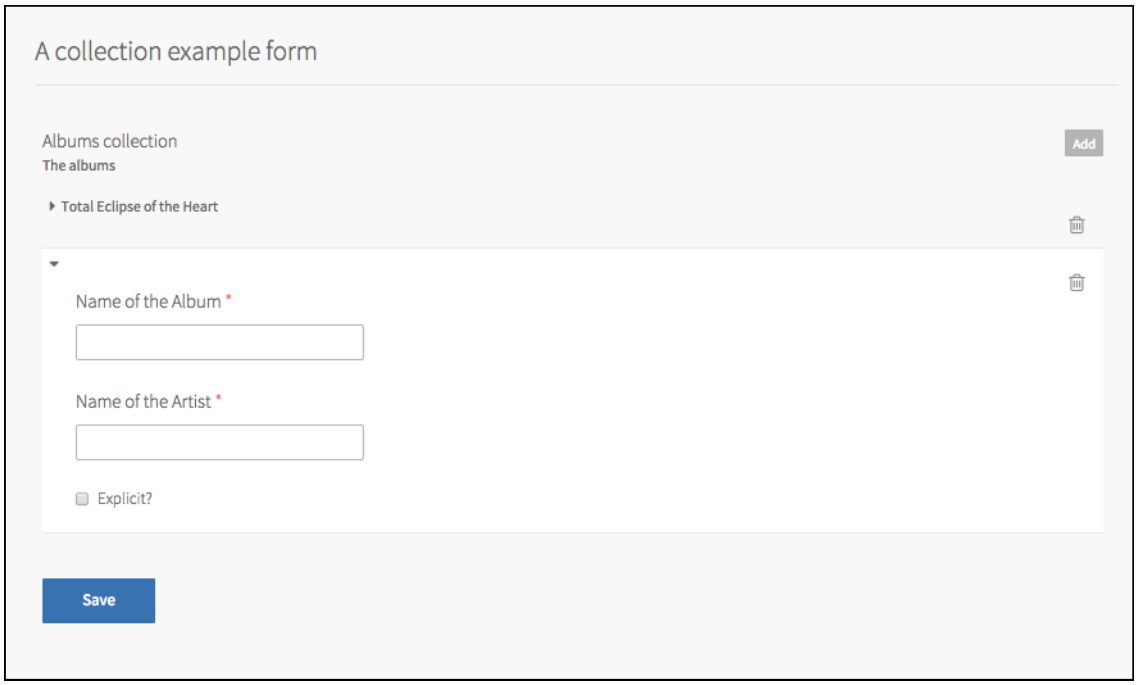
Most properties are simple values such as strings, integers, URL addresses, or IP addresses. Others are complex, such as selectors or collections.

Selectors are a means of giving the user a choice of a set of inputs. Collections are a means of giving the user the ability to enter an array of values to create a hash.

Selectors appear as follows:



Collections appear as follows:



Property Blueprints

The following is an example of the `property_blueprints` that appear in a product template.

The example is referenced by the form properties example above. See Form Properties.

```
- name: web_server
  ...
  property_blueprints:
    - name: property_with_nil_value
      type: string
    - name: property_with_false_value
      type: boolean
      configurable: false
      default: false
    - name: property_with_true_value
      type: boolean
      configurable: false
      default: true
    - name: static_ips
      configurable: true
      optional: true
    - name: generated_secret
      type: secret
    - name: generated_uuid
      type: uuid
    - name: configured_secret
      type: secret
      configurable: true
      optional: true
    - name: configured_simple_credentials
      type: simple_credentials
      configurable: true
```

configurable

No property will be viewable in a form unless `configurable` is set to `true`. Rather than giving the user the ability to enter a value, the value is generated by Ops Manager.

must_match_regex

Regular Expression. Optional. Create a validator that runs on the form save event. If the user input does not match the `must_match_regex` constraint, the form displays the specified `error_message`. Multiple `must_match_regex` constraints for a single property blueprint are evaluated in the order listed.

Configurable Properties

Many of these properties are strings, but can be used with validators in order to check that the user typed in the correct format for a URL, IP, address, domain, etc.

string

A string.

integer

An integer.

boolean

A boolean. Viewed as a checkbox.

dropdown_select

A list of options. The user chooses one viewed as an HTML select box.

multi_select_options

A list of options. The user chooses zero or more, viewed as HTML checkboxes.

domain

A second, third, fourth, etc level domain.

wildcard_domain

A domain with a wildcard in front of it. Example: `*.domain.com`

text

A string. Appears as an HTML textarea.

ldap_url

A URL prefaced by `ldap://`.

email

An email address.

ip_ranges

A range of IP addresses, with dashes and commas allowed. Example: `1.1.1.1-1.1.1.4,2.2.2.1-2.2.2.4`

port

An integer representing a network port.

network_address

A single IP address or domain. Example: `1.1.1.1`

network_address_list

A list of IP addresses or domains. Example: `1.1.1.1,example.com,2.2.2.2`

Generated Properties (Also Configurable)

The following properties are configurable, but can also be generated by Ops Manager if configurable is false or the configurable key is omitted. The exceptions are the uuid and salted credentials properties, which are never configurable.

rsa_cert_credentials

An RSA certificate.

rsa_pkey_credentials

An RSA private key.

salted_credentials

Username and password created using a non-reversible hash algorithm.

simple_credentials

Username and password.

secret

A random string or password.

uuid

A universal unique identifier.

Complex Properties (Selectors and Collections)

The selector and collections inputs are referenced by their selector and collection property blueprints. These are more complicated than simple properties in that they contain manifest snippets, which are further referenced in other manifest snippets. We will learn about manifest snippets in the next section.

Job Types

The following is an example of the `job_types` section that appears in a product template. This section defines the jobs that end up in a BOSH manifest. Those jobs are defined in your BOSH release. Jobs require many different settings in order to function properly, and that is the crux of what Ops Manager does for you: it asks a user for values to those settings and generates a manifest based on what was entered.

Ops Manager does not require product authors to provide `vm_credentials` in the `property_blueprints` for each `job_type`. This is because `vm_credentials` are generated automatically, and you can find them in the release manifest.

Note: Starting in PCF v2.1, Ops Manager ignores `static_ip` and `dynamic_ip` keys.

```
job_types:
  - name: web_server
    resource_label: Web Server
    templates:
      - name: web_server
        release: example-release
      - name: time_logger
        release: example-release
    release: example-release
    static_ip: 1
    dynamic_ip: 0
    max_in_flight: 1
    single_az_only: true:
    instance_definition:
      name: instances
      type: integer
      configurable: true
      default: 1
      constraints:
        max: 1
      zero_if:
        property_reference: '.web_server.example_text'
        property_value: 'magic value'
    resource_definitions:
      - name: ram
        type: integer
        configurable: true
```

name

String. Required. The name of the job as it will be created in the Ops Manager generated BOSH manifest.

resource_label

String. Required. The label of the job as it will appear in the resources page of the tile.

templates

Array of Hashes. Required. Each element has the following fields:

name

The name of the job template to use. Required.

release

The name of the release the template is from. Required.

consumes

A YAML string defining BOSH links this job consumes. Optional.

provides

A YAML string defining BOSH links this job provides. Optional.

This is a BOSH feature (creating jobs from different releases). See the BOSH documentation for more information.

release

String. Required. The name of the BOSH release contained in your product archive (.pivotal file).

single_az_only

Boolean. Required. You can give users control of balancing jobs across availability zones (AZs) by setting `single_az_only` to `false`. To limit a job to a single AZ, set this to `true`.

warning: If you change the `single_az_only` setting, your VMs may switch AZs. This change can cause an orphaned disk.

max_in_flight

Integer. Required. A BOSH setting that controls the number of instances of this job that BOSH will deploy in parallel.

resource_definitions

Array of Hashes. Required. A set of resource settings for the job along with max and min constraints, defaults, and whether or not the user can configure (change) the setting. The resources that can be set are:

- ram
- ephemeral_disk
- persistent_disk
- cpu

Note: If you set the `default` property for `persistent_disk` to `0`, users cannot edit this value and the **Resource Config** page in **Ops Mananger** displays **None** under the persistent disk field.

instance_definition

Hash. Required. The number of default instances for a job along with max, min, odd, and the ability to decrease sizing after deploy constraints.

If your product uses an external service that performs the same job as a service in Elastic Runtime, you can reduce resource usage by setting the instance count of a job to `0` with the `zero_if` property. For example, your product uses Amazon Relational Database Service (RDS) instead of MySQL, which is the default system database for Elastic Runtime. Set `property reference` to `.properties.system.database` and `property value` to `magic value` to change the instance counts of all MySQL jobs to `0`.

manifest

Text snippet, prefaced by pipe symbol: `|`. Optional. Ops Manager generates a BOSH manifest that defines properties for each job that the manifest deploys. Some of these properties are not set until the user clicks **Apply Changes**, because the user configures them in the tile or because Ops Manager has to generate them.

To include these properties in a manifest snippet, use “double-parens” syntax, which consists of a variable name surrounded by two sets of parentheses:

```
manifest: |
  pizza_toppings:
    peppers: (( .properties.example_selector.pizza_option.peppers.value ))
```

When Ops Manager parses a product template and BOSH parses a manifest, they both fill in properties designated by double-parens syntax. Some property values in a product template, such as CredHub credentials, must be filled in by BOSH on the BOSH Director VM, rather than by Ops Manager. To include these BOSH deploy-time properties in a manifest snippet, use “triple-parens” notation:

```
manifest: |
  credhub:
    concatenated_password: prefix-((( credhub-password )))-suffix
    password: ((( credhub-password )))
```

Ops Manager strips the outer parentheses from these expressions and includes the resulting double-parens expressions in the manifest it generates, for BOSH to evaluate at deploy time.

named_manifest

Specify a property for collection within the `named_manifest` section of the metadata. See the Simple vs. Complex Inputs section for more information about collections.

The following example uses a named manifest called `for_routing` that belongs to the `certificate_collection` job:

```
- name: certificate_collection
  type: collection
  configurable: true
  property_blueprints:
    - name: some_cert_name
      type: string
    - name: some_cert
      type: rsa_cert_credentials
  named_manifests:
  - name: for_routing
    manifest: |
      name:  (( current_record.some_cert_name.value ))
      private_key:  (( current_record.some_cert.private_key_pem ))
      public_key:  (( current_record.some_cert.public_key_pem ))
      certificate: (( current_record.some_cert.cert_pem ))
```

Use the `current_record` property within a collection record to refer to other properties in the same record. For example, the properties in the `for_routing` named manifest refer to the values for `name`, `private_key`, `public_key`, and `certificate` within this record only.

Note: The `current_record` property is reserved. You cannot create a new property named `current_record`.

After defining a named manifest, you can reference it using a manifest snippet in the following format:

```
routing_certificates: (( .properties.certificate_collection.parsed_manifest(for_routing) ))
```

Ops Manager renders the following manifest from this example:

```
routing_certificates:
  - name: foo_cert
    private_key: PRIVATE-KEY
    public_key: PUBLIC-KEY
    certificate: CERTIFICATE
  - name: bar_cert
    private_key: PRIVATE-KEY
    public_key: PUBLIC-KEY
    certificate: CERTIFICATE
```

Selector Manifest Snippets

Selector snippets are evaluated twice. As you saw in the `property_blueprint`, the selector has a manifest snippet for both sets of inputs that the user might choose. Only one of these sets is evaluated and inserted into the job's manifest.

Ops Manager Provided Snippets

The following double-parens accessors retrieve your job properties:

- name: `(( name ))`
- ram: `(( ram ))`
- ephemeral_disk: `(( ephemeral_disk ))`
- persistent_disk: `(( persistent_disk ))`

- instances: `(( instances ))`
- availability_zone: `(( availability_zone ))` (deprecated)
- bosh_job_partition_stats: `(( bosh_job_partition_stats ))` (deprecated)
- first_network_deprecated: `(( first_network_deprecated ))` (deprecated)
- subnet_cidrs: `(( subnet_cidrs ))`

Note: As of PCF v2.1, IP accessors are no longer supported.

The following is a list of all typed values with the accessor “value”:

- collection
- ldap_url
- domain
- wildcard_domain
- ip_ranges
- ip_address
- email
- port
- integer
- string
- boolean
- text
- smtp_authentication
- network_address
- network_address_list
- string_list
- ca_certificate
- multi_select_options
- dropdown_select
- vm_type_dropdown
- disk_type_dropdown
- uuid
- service_network_az_multi_select
- service_network_az_single_select
- secret

The following list shows typed values with multiple accessors:

- simple_credentials: identity, password
- rsa_cert_credentials: private_key_pem, cert_pem, public_key_pem, cert_and_private_key_pems
- rsa_pkey_credentials: private_key_pem, public_key_pem, public_key_openssh, public_key_fingerprint
- salted_credentials: salt, identity, password
- selector: value, selected_option, nested context

In addition, Ops Manager supports accessors that are global to the entire installation rather than job specific.

- $ops_manager.ca_certificate: The internal SSL CA certificate used to sign all SSL certificates generated by this Ops Manager instance, such as when the user clicks a **Generate Self-Signed RSA Certificate** link
- $ops_manager.trusted_certificates
- $ops_manager.http_proxy
- $ops_manager.https_proxy
- $ops_manager.no_proxy
- $director.deployment_ip

- $director.hostname
- $director.username
- $director.password
- $director.ntp_servers
- $director.ca_public_key
- $director.tld
- $director.bosh_metrics_forwarder_client_name
- $director.bosh_metrics_forwarder_client_secret
- $self.uaa_client_name
- $self.uaa_client_secret
- $self.service_network
- $self.stemcell_version
- ..PRODUCT-NAME.properties
- ..PRODUCT-NAME.deployment_name

Property Reference

Page last updated:

This topic explains how PCF Tiles describe properties.

Double-Parentheses Expressions

The product template `.yml` file in a tile's `metadata` subdirectory defines how the tile interface collects configurable properties from the user, and how Ops Manager incorporates these properties into the deployment manifest that it creates.

The product template contains `manifest` snippets in both the `form_types` section that defines the tile interface, and the `job_types` section describing the jobs that the manifest deploys. Within these snippets, you can use special expressions to include property values that are otherwise not known ahead of time, such as configurable properties or system properties:

- Double-parentheses expressions designate property values that Ops Manager fills in when it generates the deployment manifest, after the user clicks **Apply Changes**. These values include configurable properties and properties supplied by Ops Manager.
- Triple-parentheses expressions designate property values that BOSH supplies when it deploys instances of the tile service, such as CredHub credentials.

Referencing Properties

Evaluating a property can be represented by piecing two segments together:

- The location of the property
- What information from the property you are looking to access, or *accessors*

Together, the double-parentheses expression can be written as:

```
(( LOCATION_OF_PROPERTY.ACCESSOR ))
```

The method of referencing the location of the property varies. Here is a complete list of ways to reference a property with some help text to indicate the situation.

.properties.top_level_property	Refers to the property blueprint whose name is "top_level_property" found in the global list of properties of the same product
.job_one.job_level_property	Refers to the property blueprint whose name is "job_level_property" found in the list of properties of the job "job_one" of the same product
job_level_property	Refers to the property blueprint whose name is "top_level_property" found in the same product and job whose manifest is currently being evaluated
..other_product.properties.top_level_property	Refers to the property blueprint whose name is "top_level_property" found in the global list of properties of the product "other_product"
..other_product.job_two.job_level_property	Refers to the property blueprint whose name is "job_level_property" found in the list of properties of the job "job_one" of the product "other_product"

Accessors vary between property blueprint types. See the Property Blueprint Reference for available properties and their accessors.

The following example uses the property blueprint type `string` with its one accessor, `value`. A valid double-parentheses expression to access the value of this property (assuming it is top-level, and has the name `example-string`) would look like:

```
(( .properties.example-string.value ))
```

Ops Manager allows empty arrays in double-parentheses expressions. For example:

```
(( .properties.example-string.value || [] ))
```

Dollar Contexts

Outside of properties, you can also retrieve information about various configuration details of your product and Ops Manager.

- $ops_manager: used by any product to obtain information about specific OpsManager
- $director: used by any product to obtain information about the Director
- $self: used by your own product to obtain information about your product's configuration

$ops_manager

ca_certificate	Provides the root CA cert that is used to sign the Director VM
trusted_certificates	Provides a list of certificates that are applied by the Director to all VMs
http_proxy	Provides the comma separated values that are entered if Ops Manager traffic is directed to an HTTP Proxy
https_proxy	Provides the comma separated values that are entered if Ops Manager traffic is directed to an HTTPS Proxy
no_proxy	Provides the comma separated values that should not go through a proxy

$director

deployment_ip	Provides the IP address that the BOSH Director is deployed on
username	Provides the username for the Director VM
password	Provides the password for the Director VM
ntp_servers	Provides a list of ntp servers that are deployed by the Director
ca_public_key	Provides the public key that is used to sign the Director VM
hostname	Provides the hostname for the Director VM
tld	Returns the string `bosh` as the top-level domain (TLD) of the BOSH Director
bosh_metrics_forwarder_client_name	Provides the BOSH Metrics Forwarder client name
bosh_metrics_forwarder_client_secret	Provides the BOSH Metrics Forwarder client secret
dns_release_present	Exposes the Director configuration for `disable_dns_release`

Note: Support for the `$director.username` and `$director.password` accessors will be removed in future versions of Ops Manager.

$self

uaa_client_name	Provides the UAA client name created for your Product to communicate with the BOSH Director
uaa_client_secret	Provides the UAA client secret created for your Product to communicate with the BOSH Director
service_network	Provides the name of the service network that has been assigned to your product
stemcell_version	Provides the stemcell version that is being used by your product

Property Blueprint Reference

string

Holds a single string value

Accessors:

value	Returns the string value

Product template example:

```
- name: example_string
  type: string
  configurable: true
  default: 'Hello world'
  constraints:
  - must_match_regex: '\A[^!@#$%^&*()]*\z'
    error_message: 'This name cannot contain special characters.'
  - must_match_regex: '\A[^0-9]*\z'
    error_message: 'This name cannot contain digits.'
```

boolean

Holds a single boolean value

Accessors:

value	Returns the boolean value

Example:

```
- name: example_boolean
  type: boolean
  configurable: true
  default: false
```

collection

Collections represent the ability to hold multi-property entries. Each “record” will contain values for the configured set of property blueprints.

Accessors:

value	An array of hashes whose key are the property name. Example: `[{album: 'my-album', artist: 'some-artist', explicit: true, genre: 'rock'}]`

Example:

```
- name: example_collection
  type: collection
  configurable: true
  property_blueprints:
    - name: album
      type: string
      freeze_on_deploy: true
    - name: artist
      type: string
      freeze_on_deploy: true
    - name: explicit
      type: boolean
    - name: genre
      type: dropdown_select
      configurable: true
      optional: true
      options:
        - name: rock
          label: 'Rock'
        - name: country
          label: 'Country'
        - name: edm
          label: 'Beep Boop PSH'
  default:
    - album: Christmas Carols
      artist: Ops Manatee
      explicit: true
      genre: edm
```

Selector

Provides the ability to switch between groups of properties.

Selectors are unique in the way that property information is accessed. Ops Manager provides accessors available at the top-level selector property, accessors for retrieving a specific property in an option group, and the ability to provide manifest snippets for a selector option group.

Each selector group may provide manifest snippets. This is because Ops Manager does not support conditionally adding manifest snippets. Therefore, it's difficult to be able to write manifest sections for a selector. A manifest snippet should be present within all option groups, and can

Accessors on Selector Property:

value	Returns a string of the currently selected option group. Example: "Filet Mignon"
selected_option	Scopes the accessor to the currently selected option group. Does not return meaningful information alone. Must be chained with an accessor available to a Selector Option Group.
SPECIFIC_SELECTOR_OPTION_GROUP	Scopes the accessor to a specific selector option group. Does not return meaningful information alone. Must be followed with the name and accessor of a specific property in the option group.

Example, `value`:

```
.properties.example_selector.filet_mignon_option.review.value
```

Accessors on Selector Option Group:

parsed_manifest(manifest_snippet_name)	Returns a hash of the specific manifest snippet

Example, `selected_option`:

```
.properties.example_selector.selected_option.parsed_manifest(my_snippet)
```

Here, `my_snippet` corresponds to the name of an entry within each option_template's named_manifests section.

Example, option group:

```
- name: example_selector
  type: selector
  configurable: true
  default: Pizza
  freeze_on_deploy: true
  option_templates:
    - name: pizza_option
      select_value: Pizza
      named_manifests:
        - name: my_snippet
          manifest: |
            pizza_toppings:
              pepperoni: (( .properties.example_selector.pizza_option.pepperoni.value ))
              pineapple: (( .properties.example_selector.pizza_option.pineapple.value ))
              other: (( .properties.example_selector.pizza_option.other_toppings.value ))
      property_blueprints:
        - name: pepperoni
          type: boolean
          configurable: true
          freeze_on_deploy: true
        - name: other_toppings
          type: string
          configurable: true
          optional: true
          constraints:
          - must_match_regex: '\A[^!@#$%^&*()]*\z'
            error_message: 'This name cannot contain special characters.'
    - name: filet_mignon_option
      select_value: Filet Mignon
      named_manifests:
        - name: my_snippet
          manifest: |
            rarity: (( .properties.example_selector.filet_mignon_option.rarity_dropdown.value ))
            review: (( .properties.example_selector.filet_mignon_option.review.value ))
            secret_sauce: (( .properties.example_selector.filet_mignon_option.secret_sauce.value ))
      property_blueprints:
        - name: rarity_dropdown
          type: dropdown_select
          configurable: true
          default: rare
          options:
            - name: rare
              label: 'Rare'
            - name: medium
              label: 'Medium'
            - name: well-done
              label: 'Well done'
        - name: secret_sauce
          type: secret
          configurable: true
          optional: true
```

ldap_url

Ensures the inputted string matches a URL of the LDAP protocol

Accessors:

value	Returns a string

Example:

```
- name: example_ldap_url
  type: ldap_url
  configurable: true
  default: 'ldap://example.com'
```

domain

Ensures the string value is a domain

Accessors:

value	Returns a string

Example:

```
- name: example_domain
  type: domain
  configurable: true
  default: 'example.com'
```

wildcard_domain

Ensures the string value is a domain prefixed with “*.”

Accessors:

value	Returns a string
to_wildcard	Returns a string of the value prefixed with “*.” if not present

Example:

```
- name: example_wildcard_domain
  type: wildcard_domain
  configurable: true
  default: '*.example.com'
```

ip_ranges

Holds an array of strings and ensure the values are IP ranges

Accessors:

value	Returns a string containing a comma-separated list of IP ranges
parsed_ip_ranges	Returns an array of strings for each IP range

Example:

```
- name: example_ip_ranges
  type: ip_ranges
  configurable: true
  default: '1.1.1.1-1.1.14,2.2.2.1-2.2.2.4'
```

ip_address

Ensures the string value is an IP address

Accessors:

value	Returns a string

Example:

```
- name: example_ip_address
  type: ip_address
  configurable: true
  default: '192.168.0.1'
```

email

Ensures the string value is formatted as an email address

Accessors:

value	Returns a string

Example:

```
- name: example_string
  type: email
  configurable: true
  default: 'john@example.com'
```

port

Holds a single integer value

Accessors:

value	Returns an integer

Example:

```
- name: example_port
  type: port
  configurable: true
  default: 3000
```

integer

Holds a single integer value

Accessors:

value	Returns an integer

Example:

```
- name: example_integer
  type: integer
  configurable: true
  default: 100
```

text

Holds a single string value

Accessors:

value	Returns a string

Example:

```
- name: example_text
  type: text
  configurable: true
  default: |
    Example
    Text
```

smtp_authentication

Holds string with a possible value of plain, login, or cram_md5

Accessors:

value	Returns a string with possible value of `plain`, `login`, `cram_md5`

Example:

```
- name: example_smtp_authentication
  type: smtp_authentication
  configurable: true
  default: plain
```

network_name

Ensure the string is a network name

Accessors:

value	Returns a string

Example:

```
- name: example_network_name
  type: network_name
  configurable: true
  default: 'ExampleNetwork'
```

network_address

Ensure the string is a network address

Accessors:

value	Returns a string

Example:

```
- name: example_network_address
  type: network_address
  configurable: true
  default: 'localhost'
```

network_address_list

Holds an array of new addresses

Accessors:

value	Returns a string containing a comma separated list of network addresses
parsed_network_addresses	Returns an array of strings for each network address

Example:

```
- name: example_network_address_list
  type: network_address_list
  configurable: true
  default: 'localhost,1.1.1.1'
```

string_list

Holds an array of strings

Accessors:

value	Returns a string
parsed_strings	Returns an array of strings for each string entry
parsed_regex	Returns a string containing a regex of the format “^(string1|string2|string3)$” where the value of this property is“string1,string2,string3”

Example:

```
- name: example_string_list
  type: string_list
  configurable: true
  default: 'foo,bar,baz'
```

ca_certificate

Holds a string value

Accessors:

value	Returns a string

Example:

```
- name: example_ca_certificate
  type: ca-certificate
  configurable: true
  default: |
    -- BEGIN FAKE CERT --
    -- END FAKE CERT --
```

multi_select_options

Holds an array of selected string values

Accessors:

value	Returns an array of strings for the selected options

Example:

```
- name: example_multi_select_options
  type: multi_select_options
  configurable: true
  default: ['earth', 'mercury']
        options:
        - name: mercury
          label: 'label for mercury'
        - name: venus
          label: 'label for venus'
        - name: earth
          label: 'label for earth'
```

dropdown_select

Holds an array of strings selected string values

Accessors:

value	Returns a string

Example:

```
- name: example_dropdown
  type: dropdown_select
  configurable: true
  default: kiwi
  options:
    - name: kiwi
      label: 'label for kiwi'
    - name: lime
      label: 'label for lime'
    - name: avocado
      label: 'label for avocado'
```

vm_type_dropdown

Holds single string value selected from allowed vm_types

Accessors:

value	Returns a string

Example:

```
- name: example_vm_type_dropdown
  type: vm_type_dropdown
  configurable: true
```

disk_type_dropdown

Holds single string value selected from allowed disk_types

Accessors:

value	Returns a string

Example:

```
- name: example_disk_type_dropdown
  type: disk_type_dropdown
  configurable: true
```

uuid

Holds a string uuid value

Accessors:

value	Returns a string

Example:

```
- name: example_uuid
  type: uuid
  configurable: true
```

service_network_az_multi_select

Holds an arrays of string value selected from allowed azs

Accessors:

value	Returns an array of strings for the selected options

Example:

```
- name: example_service_network_az_multi_select
  type: service_network_az_multi_select
  configurable: true
```

service_network_az_single_select

Holds a single string value selected from allowed azs

Accessors:

value	Returns a string

Example:

```
- name: example_service_network_az_single_select
  type: service_network_az_single_select
  configurable: true
```

secret

Holds a single string value

Accessors:

value	Returns a string

Example:

```
- name: example_secret
  type: secret
  configurable: true
```

Contact Us

Page last updated:

To learn more about the Pivotal ISV Partner Program, or to request our assistance with your integration project, please contact us at one of the following addresses:

- Program Manager: Marina Joseph
- Business Development: Nima Badiey
- Platform Engineering: Guido Westenberg

Contributions

The source code for this site is in a public GitHub repository.

We greatly appreciate contributions to the content in the form of pull requests, as well as GitHub issues with corrections, comments, or suggestions.