Creating your first schema

JSON Schema is a vocabulary that you can use to annotate and validate JSON documents. This tutorial guides you through the process of creating a JSON Schema document.

After you create the JSON Schema document, you can validate the example data against your schema using a validator in a language of your choice. See Tools for a current list of supported validators.

Overview

The example we use in this guide is a product catalog that stores its data using JSON objects, like the following:

data

```
{
  "productId": 1,
  "productName": "A green door",
```

Each product in the catalog has:

- `productId` : an identifier for the product
- `productName` : the product name
- `price` : the cost to the consumer
- `tags` : an optional array of identifying tags

The JSON object is human-readable, but it doesn't include any context or metadata. There's no way to tell from looking at the object what the keys mean or what the possible inputs are. JSON Schema is a standard for providing answers to these questions. In this guide, you will create a JSON Schema document that describes the structure, constraints, and data types for a set of JSON data.

Introduction to JSON Schema

The *instance* is the JSON document that is being validated or described, and the *schema* is the document that contains the description.

The most basic schema is a blank JSON object, which constrains nothing, allows anything, and describes nothing:

```data
{}
```

By adding validation keywords to the schema, you can apply constraints to an instance. For example, you can use the `type` keyword to constrain an instance to an object, array, string, number, boolean, or null:

JSON Schema is hypermedia-ready and ideal for annotating your existing JSON-based HTTP API. JSON Schema documents are identified by URIs, which can be used in HTTP link headers and within JSON Schema documents to allow for recursive definitions.

Create a schema definition

To create a basic schema definition, define the following keywords:

- `$schema` : specifies which draft of the JSON Schema standard the schema adheres to.
- `$id` : sets a URI for the schema. You can use this unique URI to refer to elements of the schema from inside the same document or from external JSON documents.
- `title` and `description` : state the intent of the schema. These keywords don't add any constraints to the data being validated.
- `type` : defines the first constraint on the JSON data. In the product catalog example below, this keyword specifies that the data must be a JSON object.

For example:

schema

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://example.com/product.schema.json",
  "title": "Product",
  "description": "A product in the catalog",
  "type": "object"
}
```

The keywords are defined using JSON keys. Typically, the data being validated is contained in a JSON data document, but JSON Schema can also validate JSON data

`description` are schema annotations, and `type` is a validation keyword.

Define properties

This section adds the `properties` keyword. In JSON Schema terms, `properties` is a validation keyword. When you define `properties`, you create an object where each property represents a key in the JSON data that's being validated. You can also specify which properties defined in the object are required.

Add the properties object

Using the product catalog example, `productId` is a numeric value that uniquely identifies a product. Since this is the canonical identifier for the product, it's required.

To add the `properties` object to the schema:

1. Add the `properties` validation keyword to the end of the schema:

```
...
  "title": "Product",
  "description": "A product from Acme's catalog",
  "type": "object",
  "properties": {
    "productId": {}
  }
```

2. Add the `productId` keyword, along with the following schema annotations:

- `description`: describes what `productId` is. In this case, it's the product's unique identifier.

```
...
  "properties": {
    "productId": {
      "description": "The unique identifier for a product",
      "type": "integer"
    }
  }
```

With the new `properties` validation keyword, the overall schema looks like this:

schema

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://example.com/product.schema.json",
  "title": "Product",
  "description": "A product from Acme's catalog",
  "type": "object",
  "properties": {
    "productId": {
      "description": "The unique identifier for a product",
      "type": "integer"
    }
  }
}
```

The following example adds another required key, `productName` . This value is a string:

schema

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://example.com/product.schema.json",
  "title": "Product",
```

```
      "productId": {
        "description": "The unique identifier for a product",
        "type": "integer"
      },
      "productName": {
        "description": "Name of the product",
        "type": "string"
      }
    }
  }
```

The `properties` object now includes two keys, `productId` and `productName`. When JSON data is validated against this schema, validation fails for any documents that contain invalid data in either of these fields.

Define required properties

This section describes how to specify that certain properties are required. This example makes the two existing keys required and adds another required key named `price`. The `price` key has a `description` and `type` just like the other keys, but it also specifies a minimum value. Because nothing in the store is free, each product requires a price value that's above zero. Define this using the `exclusiveMinimum` validation keyword.

To define a required property:

1. Inside the `properties` object, add the `price` key. Include the usual schema annotations `description` and `type`, where `type` is a number:

```
    "properties": {
      ...
      "price": {
        "description": "The price of the product",
```

2. Add the `exclusiveMinimum` validation keyword and set the value to zero:

```
    "price": {
      "description": "The price of the product",
      "type": "number",
      "exclusiveMinimum": 0
    }
```

3. Add the `required` validation keyword to the end of the schema, after the `properties` object. Add `productID`, `productName`, and the new `price` key to the array:

```
  ...
    "properties": {
      ...
      "price": {
        "description": "The price of the product",
        "type": "number",
        "exclusiveMinimum": 0
      },
    },
    "required": [ "productId", "productName", "price" ]
```

With the new `required` keyword and `price` key, the overall schema looks like this:

schema

```
{
```

```
    "description": "A product from Acme's catalog",
    "type": "object",
    "properties": {
      "productId": {
        "description": "The unique identifier for a product",
        "type": "integer"
      },
      "productName": {
        "description": "Name of the product",
        "type": "string"
      },
      "price": {
        "description": "The price of the product",
        "type": "number",
        "exclusiveMinimum": 0
      }
    },
    "required": [ "productId", "productName", "price" ]
  }
```

The `exclusiveMinimum` validation keyword is set to zero, which means that only values above zero are considered valid. To include zero as a valid option, you could use the `minimum` validation keyword instead.

Define optional properties

This section describes how to define an optional property. For this example, define a keyword named `tags` using the following criteria:

- The `tags` keyword is optional.
- If `tags` is included, it must contain at least one item.
- All tags must be unique.
- All tags must be text.

annotations `description` and `type`, and define `type` as an array:

```
...
  "properties": {
    ...
    "tags": {
      "description": "Tags for the product",
      "type": "array"
    }
  }
```

2. Add a new validation keyword for `items` to define what appears in the array. For example, `string`:

```
...
    "tags": {
      "description": "Tags for the product",
      "type": "array",
      "items": {
        "type": "string"
      }
    }
```

3. To make sure there is at least one item in the array, use the `minItems` validation keyword:

```
...
    "tags": {
      "description": "Tags for the product",
```

```
      },
      "minItems": 1
    }
```

4. To make sure that every item in the array is unique, use the `uniqueItems` validation keyword and set it to `true`:

```
...
    "tags": {
      "description": "Tags for the product",
      "type": "array",
      "items": {
        "type": "string"
      },
      "minItems": 1,
      "uniqueItems": true
    }
```

With the new `tags` keyword, the overall schema looks like this:

schema

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://example.com/product.schema.json",
  "title": "Product",
  "description": "A product from Acme's catalog",
  "type": "object",
  "properties": {
    "productId": {
      "description": "The unique identifier for a product",
      "type": "integer"
```

```
14          "type": "string"
15        },
16        "price": {
17          "description": "The price of the product",
18          "type": "number",
19          "exclusiveMinimum": 0
20        },
21        "tags": {
22          "description": "Tags for the product",
23          "type": "array",
24          "items": {
25            "type": "string"
26          },
27          "minItems": 1,
28          "uniqueItems": true
29        }
30      },
31      "required": [ "productId", "productName", "price" ]
32    }
```

Because the new keyword is not required, there are no changes to the `required` section.

Create a nested data structure

The earlier examples describe a flat schema with only one level. This section describes how to use nested data structures in JSON Schema.

To create a nested data structure:

1. Inside the `properties` object, create a new key called `dimensions` :

```
1    ...
```

```
  }
```

2. Define the `type` validation keyword as `object`:

```
...
  "dimensions": {
    "type": "object",
  }
```

3. Add the `properties` validation keyword to contain the nested data structure. Inside the new `properties` keyword, add keywords for `length`, `width`, and `height` that all use the `number` type:

```
...
  "dimensions": {
    "type": "object",
    "properties": {
      "length": {
        "type": "number"
      },
      "width": {
        "type": "number"
      },
      "height": {
        "type": "number"
      }
    }
  }
```

inside the `dimensions` object:

```
  ...
    "dimensions": {
      "type": "object",
      "properties": {
        "length": {
          "type": "number"
        },
        "width": {
          "type": "number"
        },
        "height": {
          "type": "number"
        }
      },
      "required": [ "length", "width", "height" ]
  }
```

Using the new nested data structures, the overall schema looks like this:

schema

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://example.com/product.schema.json",
  "title": "Product",
  "description": "A product from Acme's catalog",
  "type": "object",
  "properties": {
    "productId": {
      "description": "The unique identifier for a product",
      "type": "integer"
```

```
14          "type": "string"
15        },
16        "price": {
17          "description": "The price of the product",
18          "type": "number",
19          "exclusiveMinimum": 0
20        },
21        "tags": {
22          "description": "Tags for the product",
23          "type": "array",
24          "items": {
25            "type": "string"
26          },
27          "minItems": 1,
28          "uniqueItems": true
29        },
30        "dimensions": {
31          "type": "object",
32          "properties": {
33            "length": {
34              "type": "number"
35            },
36            "width": {
37              "type": "number"
38            },
39            "height": {
40              "type": "number"
41            }
42          },
43          "required": [ "length", "width", "height" ]
44        }
45      },
46      "required": [ "productId", "productName", "price" ]
47    }
```

Add an external reference

This section describes how to reference resources outside of the schema. Sharing schemas across many data structures is a common way to make them easier to use, read, and keep up-to-date. So far, the product catalog schema is self-contained. This section creates a new schema and then references it in the product catalog schema.

The following schema validates a geographical location:

schema

```
{
  "$id": "https://example.com/geographical-location.schema.json",
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "title": "Longitude and Latitude",
  "description": "A geographical coordinate on a planet (most commonly Earth).",
  "required": [ "latitude", "longitude" ],
  "type": "object",
  "properties": {
    "latitude": {
      "type": "number",
      "minimum": -90,
      "maximum": 90
    },
    "longitude": {
      "type": "number",
      "minimum": -180,
      "maximum": 180
    }
  }
}
```

```
...
  "properties": {
  ...
    "warehouseLocation": {}
  }
```

2. To link to the external geographical location schema, add the `$ref` schema keyword and the schema URL:

```
...
  "warehouseLocation": {
   "description": "Coordinates of the warehouse where the product is located."
    "$ref": "https://example.com/geographical-location.schema.json"
  }
```

With the external schema reference, the overall schema looks like this:

schema

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://example.com/product.schema.json",
  "title": "Product",
  "description": "A product from Acme's catalog",
  "type": "object",
  "properties": {
    "productId": {
      "description": "The unique identifier for a product",
      "type": "integer"
```

```
      "type": "string"
    },
    "price": {
      "description": "The price of the product",
      "type": "number",
      "exclusiveMinimum": 0
    },
    "tags": {
      "description": "Tags for the product",
      "type": "array",
      "items": {
        "type": "string"
      },
      "minItems": 1,
      "uniqueItems": true
    },
    "dimensions": {
      "type": "object",
      "properties": {
        "length": {
          "type": "number"
        },
        "width": {
          "type": "number"
        },
        "height": {
          "type": "number"
        }
      },
      "required": [ "length", "width", "height" ]
    },
    "warehouseLocation": {
      "description": "Coordinates of the warehouse where the product is located.",
      "$ref": "https://example.com/geographical-location.schema.json"
    }
```

Validate JSON data against the schema

This section describes how to validate JSON data against the product catalog schema.

This example JSON data matches the product catalog schema:

data

```
{
  "productId": 1,
  "productName": "An ice sculpture",
  "price": 12.50,
  "tags": [ "cold", "ice" ],
  "dimensions": {
    "length": 7.0,
    "width": 12.0,
    "height": 9.5
  },
  "warehouseLocation": {
    "latitude": -78.75,
    "longitude": 20.4
  }
}
```

To validate this JSON data against the product catalog JSON Schema, you can use any validator of your choice. In addition to command-line and browser tools, validation tools are available in a wide range of languages, including Java, Python, .NET, and many others. To find a validator that's right for your project, see Tools.

Use the example JSON data as the input data and the product catalog JSON Schema as the schema. Your validation tool compares the data against the schema, and if the data meets all the requirements defined in the schema, validation is successful.

JSON Schema

Did you find these docs helpful?

Help us make our docs great!

At JSON Schema, we value docs contributions as much as every other type of contribution!

Edit this page on Github

Learn how to contribute

Still Need Help?

Learning JSON Schema is often confusing, but don't worry, we are here to help!.

Ask the community on GitHub

Ask the community on Slack

Open Collective

Code of Conduct

Slack Twitter LinkedIn

Youtube GitHub

Load Balancers in PKS

In this topic

Page last updated: November 12, 2019

Warning: Pivotal Container Service (PKS) v1.3 is no longer supported because it has reached the End of General Support (EOGS) phase as defined by the **Support Lifecycle Policy**.
To stay up to date with the latest software and security updates, upgrade to a supported version.

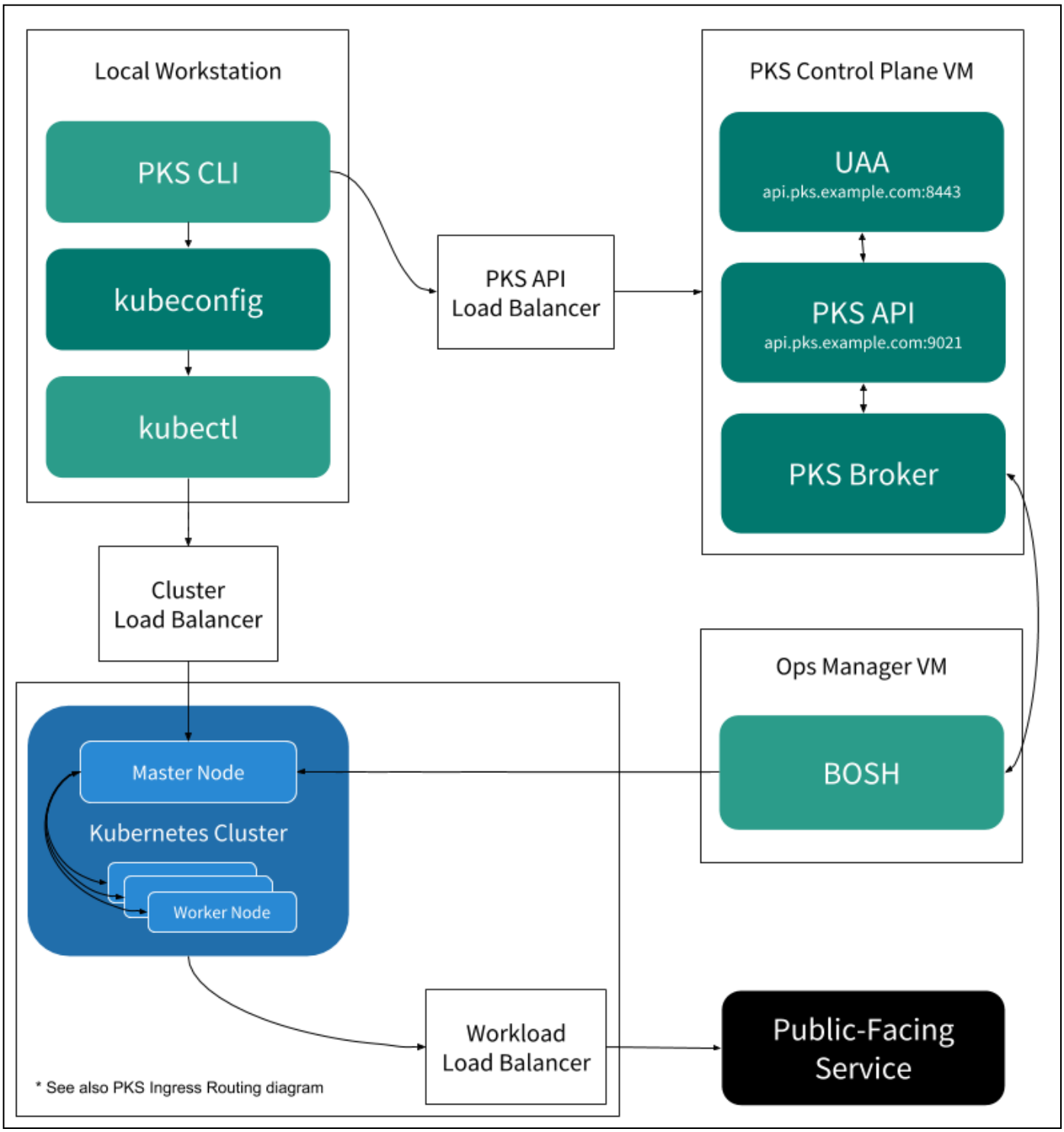
This topic describes the types of load balancers that are used in Pivotal Container Service (PKS) deployments. Load balancers differ by the type of deployment.

Load Balancers in PKS Deployments without NSX-T

For PKS deployments on GCP, AWS, or vSphere without NSX-T, you can configure load balancers for the following:

- **PKS API**: Configuring this load balancer allows you to run PKS Command Line Interface (PKS CLI) commands from your local workstation.
- **Kubernetes Clusters**: Configuring a load balancer for each new cluster allows you to run Kubernetes CLI (kubectl) commands on the cluster.
- **Workloads**: Configuring a load balancer for your application workloads allows external access to the services that run on your cluster.

The following diagram shows where each of the above load balancers can be used within your PKS deployment on GCP, AWS, or on vSphere without NSX-T:



If you use either vSphere without NSX-T or GCP, you are expected to create your own load balancers within your cloud provider console. If your cloud provider does not offer load balancing, you can use any external TCP or HTTPS load balancer of your choice.

About the PKS API Load Balancer

For PKS deployments on GCP, AWS, and on vSphere without NSX-T, the load balancer for the PKS API allows you to access the PKS API from outside the network. For example, configuring a load balancer for

the PKS API allows you to run PKS CLI commands from your local workstation.

For information about configuring the PKS API load balancer, see the **Configure External Load Balancer** section of *Installing PKS* for your IaaS.

About Kubernetes Cluster Load Balancers

For PKS deployments on GCP, AWS, and on vSphere without NSX-T, when you create a cluster, you must configure external access to the cluster by creating an external TCP or HTTPS load balancer. The load balancer allows the Kubernetes CLI to communicate with the cluster.

If you create a cluster in a non-production environment, you can choose not to use a load balancer. To allow kubectl to access the cluster without a load balancer, you can do one of the following:

- Create a DNS entry that points to the cluster's master VM. For example:

```
my-cluster.example.com    A    10.0.0.5
```

- On the workstation where you run kubectl commands, add the master IP address of your cluster and `kubo.internal` to the `/etc/hosts` file. For example:

```
10.0.0.5 kubo.internal
```

For more information about configuring a cluster load balancer, see the following:

- **Creating and Configuring a GCP Load Balancer for PKS Clusters**
- **Creating and Configuring an AWS Load Balancer for PKS Clusters**
- **Creating and Configuring an Azure Load Balancer for PKS Clusters**

About Workload Load Balancers

For PKS deployments on GCP, AWS, and on vSphere without NSX-T, to allow external access to your app, you can either create a load balancer or expose a static port on your workload.

For information about configuring a load balancer for your app workload, see **Deploying and Exposing Basic Workloads**.

If you use AWS, you must configure routing in the AWS console before you can create a load balancer for your workload. You must create a public subnet in each availability zone (AZ) where you are deploying the workload and tag the public subnet with your cluster's unique identifier.

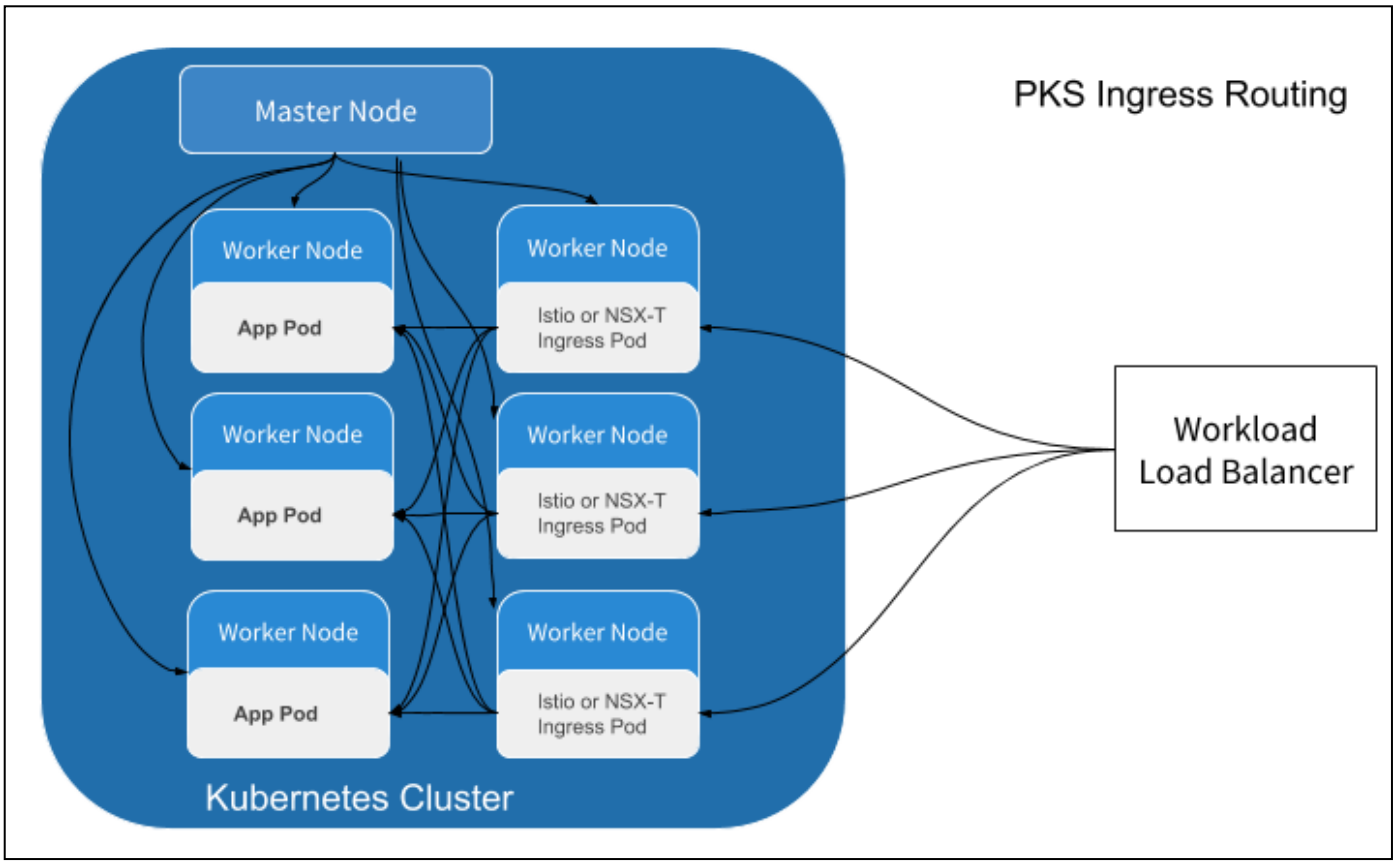
See the **AWS Prerequisites** section of *Deploying and Exposing Basic Workloads* before you create a workload load balancer.

Deploy Your Workload Load Balancer with an Ingress Controller

A Kubernetes ingress controller sits behind a load balancer, routing HTTP and HTTPS requests from outside the cluster to services within the cluster. Kubernetes ingress resources can be configured to load balance traffic, provide externally reachable URLs to services, and manage other aspects of network traffic.

If you add an ingress controller to your PKS deployment, traffic routing is controlled by the ingress resource rules you define. Pivotal recommends configuring PKS deployments with both a workload load balancer and an ingress controller.

The following diagram shows how the ingress routing can be used within your PKS deployment.



The load balancer on PKS on vSphere with NSX-T is automatically provisioned with Kubernetes ingress resources without the need to deploy and configure an additional ingress controller.

For information about deploying a load balancer configured with ingress routing on GCP, AWS, Azure, and vSphere without NSX-T, see **Configuring Ingress Routing**. For information about ingress routing on vSphere with NSX-T, see **Configuring Ingress Resources and Load Balancer Services**.

Load Balancers in PKS Deployments on vSphere with NSX-T

PKS deployments on vSphere with NSX-T do not require a load balancer configured to access the PKS API. They require only a DNAT rule configured so that the PKS API host is accessible. For more information, see **Share the PKS Endpoint** in *Installing PKS on vSphere with NSX-T Integration*.

NSX-T handles load balancer creation, configuration, and deletion automatically as part of the Kubernetes cluster create, update, and delete process. When a new Kubernetes cluster is created, NSX-T creates and configures a dedicated load balancer tied to it. The load balancer is a shared resource designed to provide efficient traffic distribution to master nodes as well as services deployed on worker nodes. Each application service is mapped to a virtual server instance, carved out from the same load balancer. For more information, see **Load Balancing** in the NSX-T documentation.

Virtual server instances are created on the load balancer to provide access to the following:

- **Kubernetes API and UI services on a Kubernetes cluster**. This allows requests to be load balanced across multiple master nodes.
- **Ingress controller**. This allows the virtual server instance to dispatch HTTP and HTTPS requests to services associated with Ingress rules.
- `type:loadbalancer` **services**. This allows the server to handle TCP connections or UDP flows toward exposed services.

Load balancers are deployed in high-availability mode so that they are resilient to potential failures and able to recover quickly from critical conditions.

Note: The `NodePort` Service type is not supported for PKS deployments on vSphere with NSX-T. Only `type:LoadBalancer` Services and Services associated with Ingress rules are supported on vSphere with NSX-T.

Resizing Load Balancers

When a new Kubernetes cluster is provisioned using the PKS API, NSX-T creates a dedicated load balancer for that new cluster. By default, the size of the load balancer is set to Small.

With network profiles, you can change the size of the load balancer deployed by NSX-T at the time of cluster creation. For information about network profiles, see **Using Network Profiles (NSX-T Only)**.

For more information about the types of load balancers NSX-T provisions and their capacities, see **Scaling Load Balancer Resources** in the NSX-T documentation.

PKS API Authentication

In this topic

Page last updated: November 12, 2019

Warning: Pivotal Container Service (PKS) v1.3 is no longer supported because it has reached the End of General Support (EOGS) phase as defined by the **Support Lifecycle Policy**.
To stay up to date with the latest software and security updates, upgrade to a supported version.

This topic describes how the Pivotal Container Service (PKS) API works with User Account and Authentication (UAA) to manage authentication and authorization in your PKS deployment.

Authenticating PKS API Requests

Before users can log in and use the PKS CLI, you must configure PKS API access with UAA. For more information, see **Configuring PKS API Access** with UAA.

You use the UAA Command Line Interface (UAAC) to target the UAA server and request an access token for the UAA admin user. If your request is successful, the UAA server returns the access token. The UAA admin access token authorizes you to make requests to the PKS API using the PKS CLI and grant cluster access to new or existing users. For more information, see **Grant Cluster Access** in *Managing Users in PKS with UAA*.

When a user with cluster access logs in to the PKS CLI, the CLI requests an access token for the user from the UAA server. If the request is successful, the UAA server returns an access token to the PKS CLI. When the user runs PKS CLI commands, for example, `pks clusters`, the CLI sends the request to the PKS API server and includes the user's UAA token.

The PKS API sends a request to the UAA server to validate the user's token. If the UAA server confirms that the token is valid, the PKS API uses the cluster information from the PKS broker to respond to the request. For example, if the user runs `pks clusters`, the CLI returns a list of the clusters that the user is authorized to manage.

Routing to the PKS API Control Plane VM

The PKS API server and the UAA server use different port numbers on the control plane VM. For example, if your PKS API domain is `api.pks.example.com`, you can reach your PKS API and UAA servers at the following URLs:

Server	URL
PKS API	api.pks.example.com:9021
UAA	api.pks.example.com:8443

Refer to **Ops Manager > Pivotal Container Service > PKS API > API Hostname (FQDN)** for your PKS API domain.

Load balancer implementations differ by deployment environment. For PKS deployments on GCP, AWS, or vSphere without NSX-T, you configure a load balancer to access the PKS API when you install the PKS tile. For more information, see the **Configure External Load Balancer** section of *Installing PKS* for your IaaS.

For procedures that describe routing to the PKS control plane VM, see the **Configure External Load Balancer** section of *Installing PKS* for your IaaS.

For overview information about load balancers in PKS, see **Load Balancers in PKS Deployments without NSX-T**.

Managing Runtime Configs

In this topic

Page last updated: October 28, 2019

> **Warning:** Pivotal Operations Manager v2.5 is no longer supported because it has reached the End of General Support (EOGS) phase as defined by the **Support Lifecycle Policy**. To stay up to date with the latest software and security updates, upgrade to a supported version.

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

Tile authors can manage runtime configs in the following ways:

- Add a new runtime config in an existing product tile. See **Add a Runtime Config**.
- Delete a runtime config from a tile. See **Delete a Runtime Config**.
- Create a tile that only contains a runtime config. See **Create a Runtime Config-Only Tile**.

For more information about runtime configs, see **Director Runtime Config** in Cloud Foundry BOSH documentation.

Overview

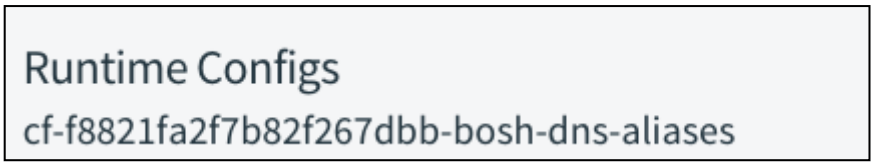
A runtime config is a section of the tile metadata that defines global deployment configurations. Tile authors can create tiles that only contain a runtime config or add a runtime config to an existing product tile. When a tile author includes a runtime config as a top-level property in the tile metadata, BOSH applies the runtime config to every VM in the deployment.

For more information about top-level properties, see **Top-Level Properties**.

When operators apply changes to deployments, 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 identifier using the tile name, a generated GUID, and the runtime config name defined in the metadata.

The identifier follows the following format:

```
TILE_NAME-GUID-RUNTIME_CONFIG_NAME
```



Add a Runtime Config

Tile authors can add `runtime_configs` as a top-level property in tile metadata. In the runtime config section, 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, can contain any number of add-ons. Each add-on 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: YOUR-RUNTIME-CONFIG
    runtime_config: |
      releases:
      - name: RELEASE
        version: RELEASE-VERSION
      addons:
      - name: YOUR-ADDON-NAME
        jobs:
        - name:YOUR-RUNTIME-CONFIG-JOB
          release: RELEASE
        properties:
          YOUR-PROPERTY:
        ...
```

Where:

- `YOUR-RUNTIME-CONFIG` is the name of the runtime config.

- `RELEASE` is the release used for the runtime config.
- `RELEASE-VERSION` is the version of the release.
- `YOUR-ADDON-NAME` is the name of the add-on that contains the runtime config job.
- `YOUR-RUNTIME-CONFIG-JOB` is the name of the job the runtime config describes.
- `YOUR-PROPERTY` are the properties used in the job.

For more information about runtime config keys, see **Director Runtime Config** in Cloud Foundry BOSH documentation.

For more information about top-level properties in the tile manifest, see **Top-Level Properties**.

Note: 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.

Delete a Runtime Config

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

To delete a runtime config from a tile, remove the `runtime_configs` section from the tile metadata.

For example, to delete a runtime config to a tile you remove the following section:

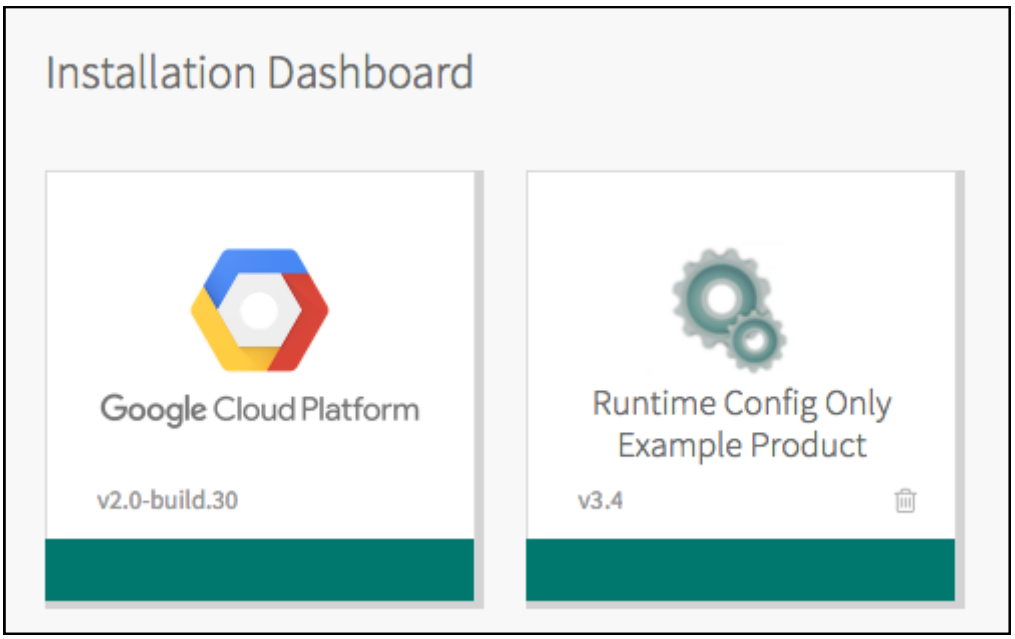
```
runtime_configs:
 - name: YOUR-RUNTIME-CONFIG
   runtime_config: |
     releases:
     - name: RELEASE
       version: RELEASE-VERSION
     addons:
     - name: YOUR-ADDON-NAME
       jobs:
       - name:YOUR-RUNTIME-CONFIG-JOB
         release: RELEASE
       properties:
         YOUR-PROPERTY:
       ...
```

Create a Runtime Config Tile

In Ops Manager, a runtime config tile appears as a tile with minimal configuration options. Runtime config tiles contain no stemcell, network, availability zone (AZ), or resource config information. You might create a runtime config tile if you want global deployment configurations to be applied to all VMs in deployment and do not any other functionality.

For a runtime config tile, tile authors are not required to define the following top-level properties:

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

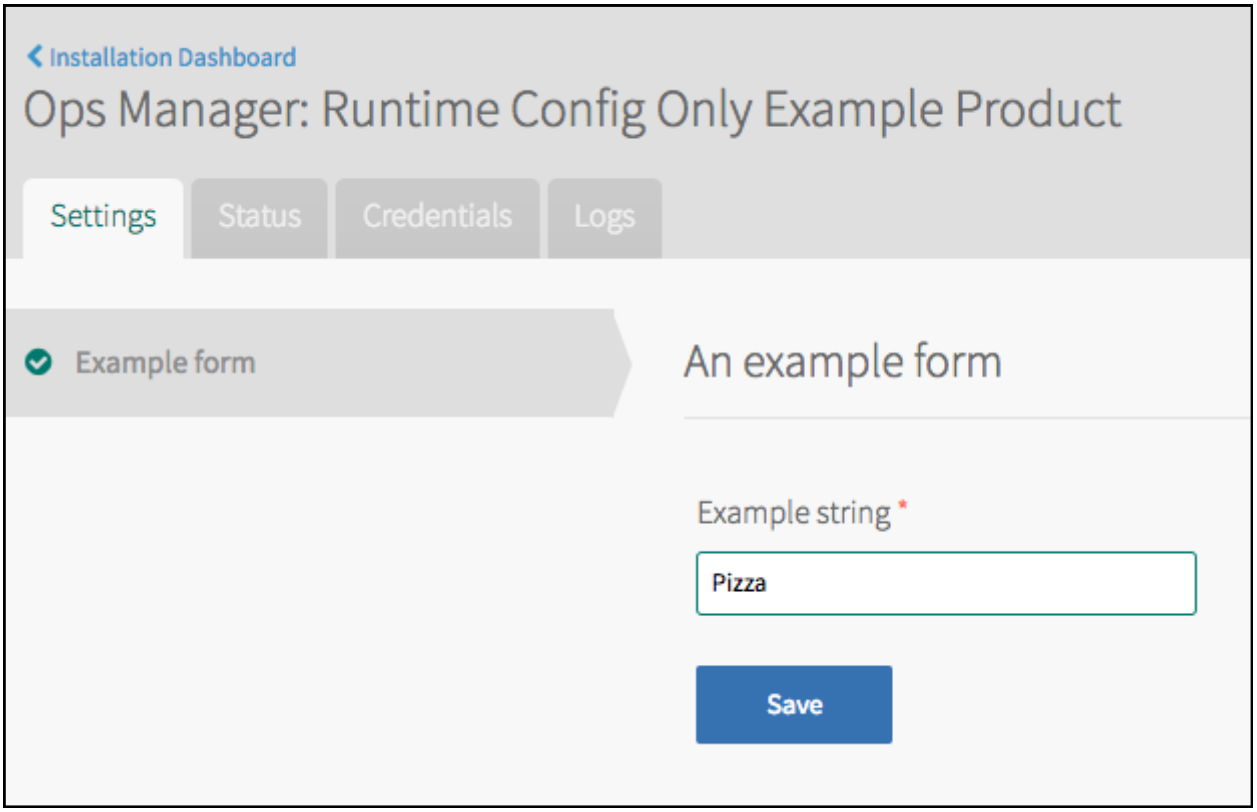


Example Runtime Config-Only Tile

The following example shows a runtime config 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: xenial 97
  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 runtime config example 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.



vmware Tanzu Docs

side navigation overhaul (before)

Pivotal Cloud Foundry v2.6

PCF v2.6 Feature Highlights

Warning: Pivotal Cloud Foundry (PCF) v2.6 is no longer supp
of General Support (EOGS) phase as defined by the **Support**
with the latest software and security updates, upgrade to a s

This topic highlights important new features included in Pivotal Cl

PCF Ops Manager Highlights

Ops Manager v2.6 includes the following important major features
these and other features included in Ops Manager v2.6, see **PCF Op**

Ops Manager Supports Multiple Stemcells for Products

Ops Manager supports products that require multiple stemcells. Th
deploy separate tiles for products that require different stemcells.

Product Deployment Times from API and Change Log Pag

Ops Manager publishes deployment times for individual products
`/api/v0/installations` endpoint and **Change Log** page.

Compiled Release Assets Included in BOSH Deployment

vmware Tanzu Docs

side navigation overhaul (after)

Pivotal Platform v2.7

PAS Overview

Page last updated: July 14, 2021

This topic describes Pivotal Application S

Overview

Cloud platforms let anyone deploy netwo
few minutes. When an app becomes pop
and migration efforts that once took mor
exclusively on your apps and data withou

The following diagram shows the layers o
model to the cloud platform model: