



Course developed by
Pivotal Academy

Pivotal Cloud Foundry Developer v1.7

ADVANCED TOPICS

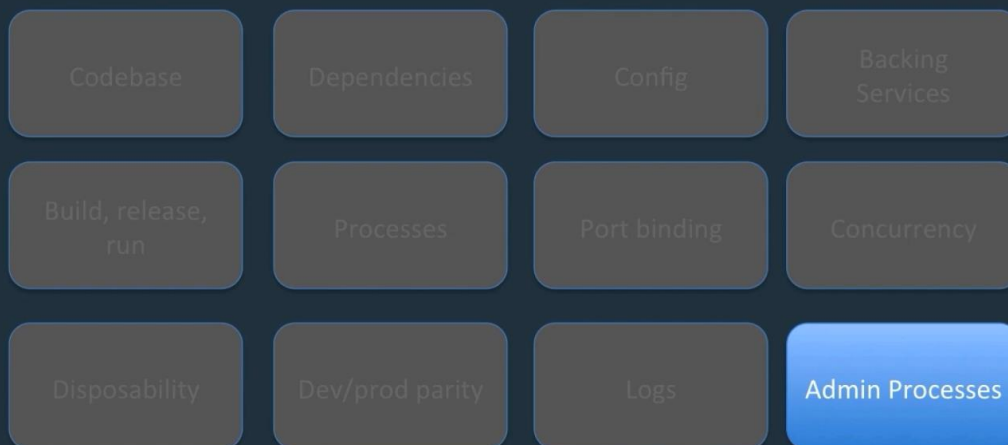
Advanced Topics

Tasks, Volumes, Routing

Agenda

1. Tasks
2. File System as a Service
3. Container to Container Routing
4. TCP Routing

The 12 Factors



Being able to run tasks in PCF helps to guarantee that the tasks run in the same environments as our applications.

What is a Task?

Unlike a long-running process (LRP), a task runs once and then terminates

eg: application staging

Examples of Tasks

application staging
database migration
sending an email
running a batch job
processing images
uploading data

These tasks are all processes that can be kicked off by an event or performed on a schedule, they do their duty and terminate on completion.

Running a Task

Tasks that run in PCF are associated with an app

```
cf run-task <app-name> <command>
```

We simply deploy the application that contains the process to the PCF platform using the standard **\$ cf push** command. Once the application containing the task process is successfully deployed, we can use the **\$ cf run-task** command to trigger the task execution. This requires the application name and the command to launch the task, a Java application will require a Java executable to specified as part of the <command>. We can also specify a name for the task and the memory required for the task.

Running a Task

```
$> cf run-task my-app "bin/rails db:migrate" --name my-task
Creating task for app my-app in org myorg / space development
as user@my.com...
OK
Task 1 has been submitted successfully for execution.

$> cf logs my-app --recent
2017-01-03T15:58:06.57-0800 [APP/TASK/my-task/0]OUT Creating
container
2017-01-03T15:58:08.45-0800 [APP/TASK/my-task/0]OUT
Successfully created container
2017-01-03T15:58:13.32-0800 ... CREATE TABLE ...
...
```

This is an example of the output for a task being run

Task Lifecycle



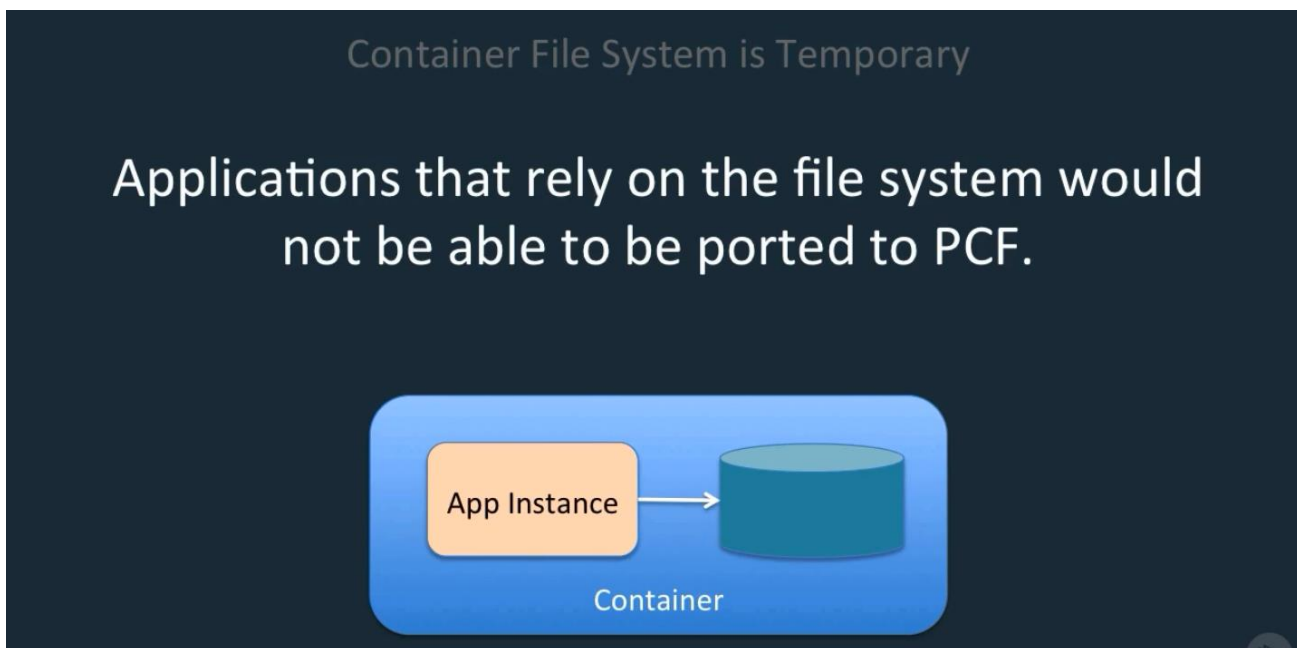
Once a task has been launched, we can monitor its status by using the **`$ cf tasks`** command, each task runs in its own container which only exists for the duration of the task execution. You can use the **`$ cf terminate task`** command, the task status will be shown as failed. All task execution history is retained for a period of one month.

Agenda

1. Tasks
2. File System as a Service
3. Container to Container Routing
4. TCP Routing



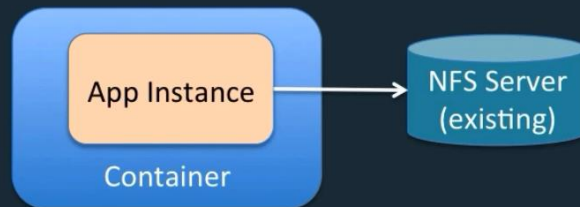
Recall that cloud-native apps should be stateless and should share nothing, any data that needs to be shared must be stored in a stateful backing service.



Since each application runs inside a separate container on PCF, those containers with their file systems are ephemeral. This makes it difficult to port existing apps that rely on their local file system. The **PCF file system service** feature specifically addresses these concerns by exposing a fixed mount point NFS Server that applications can write to.

File System as a Service

independent of the container
out-lives the container
NFS mounted storage



Marketplace Service: nfs

Administrator must enable Volume Services

```
$ cf marketplace
service  plans      description
nfs      Existing   Service for connecting to NFS volumes
```

An existing NFS Server should be available and running, then the PCF admin must enable volume services from the **OpsManager** UI to make this service available to developers via the marketplace.

NFS Service

Usage pattern is the same as for other services

create: use `-c` option to specify volume

```
$> cf create-service nfs Existing nfs_service instance
-c '{"share": "10.10.10.10/export/myshare"}'
```

Once a service is available in the marketplace, developers can create an instance of it for use in their applications. The additional **share:...** value is required and **MUST** be set to the IP address and path of the existing NFS Server.

NFS Service

Usage pattern is the same as for other services

bind: use `-c` option to specify GID and UID

```
$> cf bind-service my-app nfs_service_instance
      -c '{"uid":"1000","gid":"1000","mount":"/var/volume1"}'
...
$> cf restage my-app
...
```

The volume service instance can be bound to your application using the **uid** and **gid** values for the group and user id to be set for any files the application might create. The mount parameter is optional, if not specified; the system will create a default mount of **container_dir** as shown below

VCAP_SERVICES

```
"VCAP_SERVICES": {
  "nfs": [ {
    "credentials": {},
    "label": "nfs",
    "name": "nfs_service_instance",
    "plan": "Existing",
    "provider": null,
    "syslog_drain_url": null,
    "tags": [ "nfs" ],
    "volume_mounts": [
      {
        "container_dir": "/var/vcap/data/153e3c4b-1151-4cf7-b311-948dd77fce64",
        "device_type": "shared",
        "mode": "rw"
      }
    ]
  }
]...
```

Once the volume service instance has been created and bound, the application can look up the connection information from the **VCAP_SERVICES** environment variable just like for any other service in the platform.

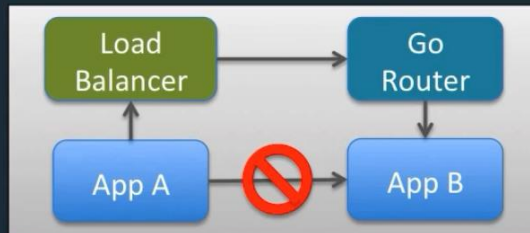
Agenda

1. Tasks
2. File System as a Service
3. Container to Container Routing
4. TCP Routing

Without Container-to-Container (C-C) Routing

Apps in PCF (eg. microservices) may need to communicate

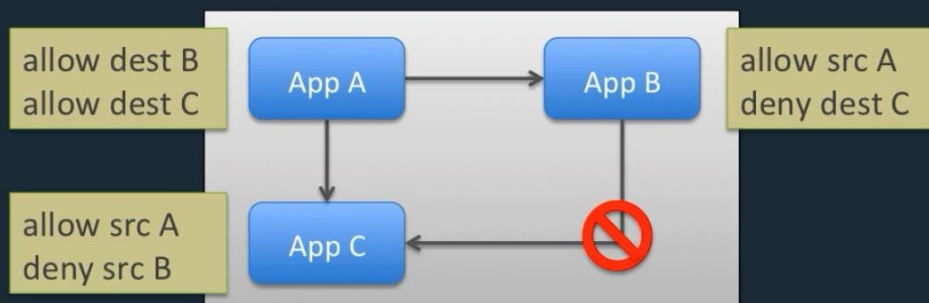
URL's are public
all requests pass through GoRouter



Suppose we have a lot of microservices deployed to PCF and one microservice needs to call another microservice. Previously, the request will have to go through the LB and the Router in order to get to the intended microservice.

With Container-to-Container (C-C) Routing

Once permitted, apps can communicate directly



With the new container-to-container networking, those extra network hops are eliminated. The developer can explicitly specify which apps are allowed to call which apps on the platform.

Benefits

Application URL's no longer public
Fewer network hops → better performance

Caveat

Client-side load balancing is required

Java applications can use Netflix Ribbon
Node.js applications can use Resilient*

*<https://www.npmjs.com/package/resilient>



Clients must do client-side load balancing by themselves

Enabling Container to Container Routing

Must be enabled by your PCF Administrator

Install `network-policy-plugin` for CF CLI
Specify desired access

```
$> cf install-plugin ~/Downloads/network-policy-plugin
...
$> cf allow-access frontend backend -protocol tcp -port 8080
...
```



Agenda

1. Tasks
2. File System as a Service
3. Container to Container Routing
4. TCP Routing

TCP Routing

Support for non-HTTP protocols

`internet of things (MQTT, AMQP)`
`legacy workloads`
`BYO container!`

By default, PCF exposes all applications over ports 80 and 443, but there are many other network protocols outside of HTTP that leverage their own custom ports.

TCP Routing

New use-cases for existing workloads

`TLS termination at your app`

The TCP routing feature can also help with mutual-TLS between applications on the platform.

TCP Routing - Domains

PCF Administrator will assign a separate domain

Look for domain type of tcp

```
>$ cf domains
Getting domains in org pivotaledu as user@my.com...
name      status  type
cfapps.io  shared
cf-tcpapps.io  shared  tcp ←
```

Admins are responsible for enabling mTLS on the platform, developers can check if mTLS is enabled by examining the `$ cf domains` command to see the domains available to them and their type.

Assigning a TCP Route to an Application

Option 1: Specify app domain on push

```
>$ cf domains
Getting domains in org pivotaledu as user@my.com...
name      status  type
cfapps.io  shared
cf-tcpapps.io  shared  tcp

>$ cf push myapp -d cf-tcpapps.io --random-route
Binding ...cf-tcpapps.io:60010 to myapp...
OK                                ← random port
```

Apps set to use TCP routing must always be set to listen on port 8080, this is the port that gets mapped at the container level when each application instance runs. When pushing their apps, developers must specify which port has been assigned exclusively for TCP routing, the platform then select a random port within a pre-defined range specified by the admin, the application will then be made publicly available on that port.

Assigning a TCP Route to an Application

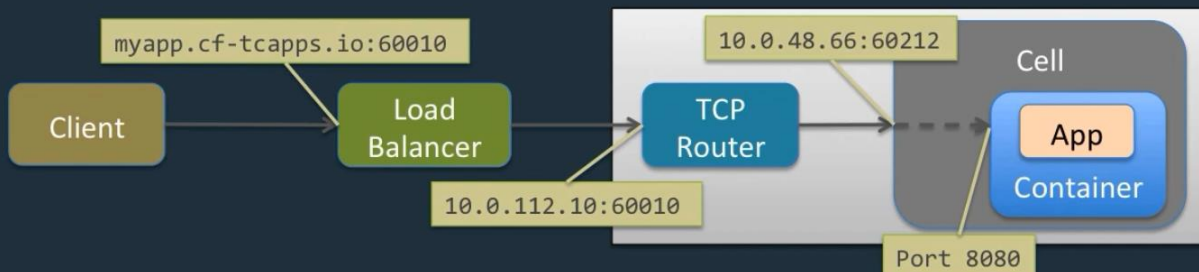
Option 2: Use *map-route*

```
$> cf map-route myapp cf-tcpapps.io --port 60010
Creating route cf-tcpapps.io:60600 for org...
```

Alternatively, the CF CLI **map-route** command with the **-port** option can be used to create new TCP routes for accessing applications as above. The **random-route** flag can also be used to assign the public-facing port for an application.

TCP Route Traffic Flow

TCP Router routes based on inbound port
your app must listen on port 8080



Here, the TCP Router has been assigned an IP address of 10.0.112.10 with inbound requests being received on port :60010, this is mapped to the random port 60212 that was allocated at deployment time on the app's Cell. Within the cell, the platform automatically configures IP tables to forward requests for port 60212 to the container's internal interface that is always port 8080 that the app listens on. The client must use the same protocol format that the app is expecting to receive.

PivotalTM

Course developed by
Pivotal Academy

Pivotal Cloud Foundry
Developer v1.7

CONCLUSION

Send feedback to:

pcf-developer-student-feedback@pivotal.io