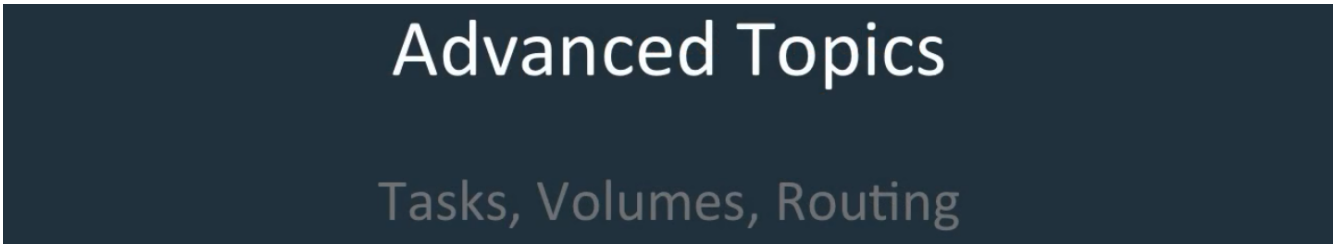
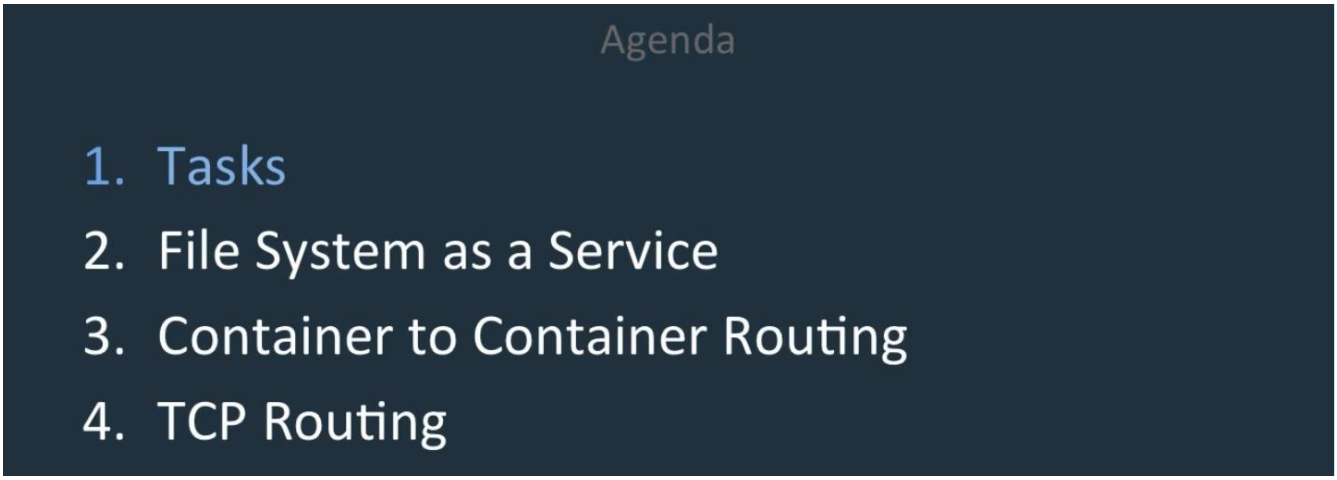
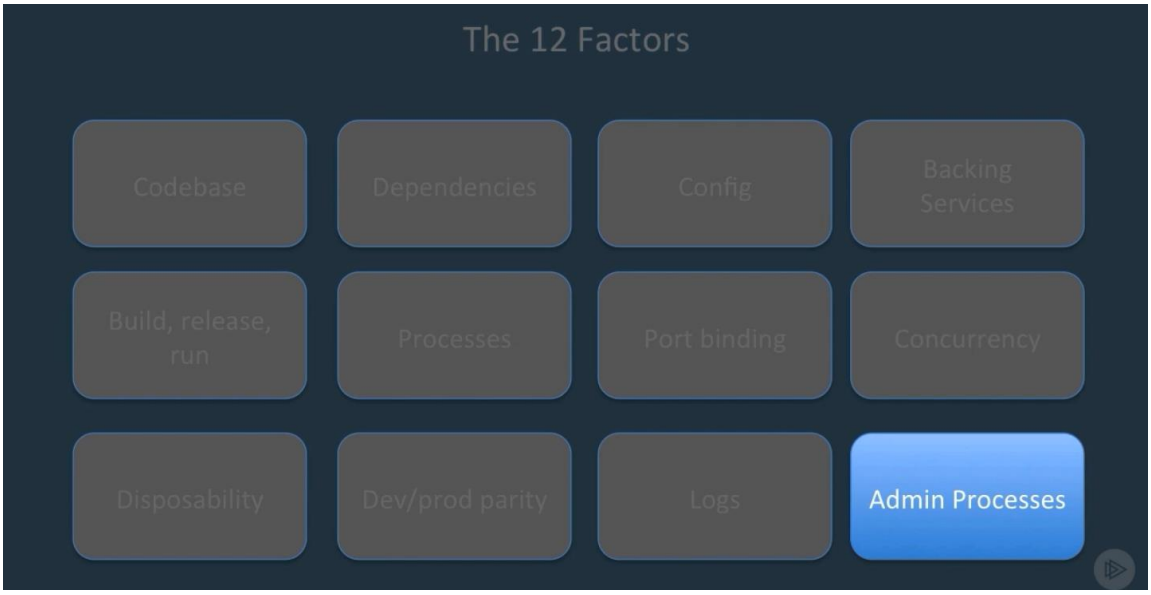
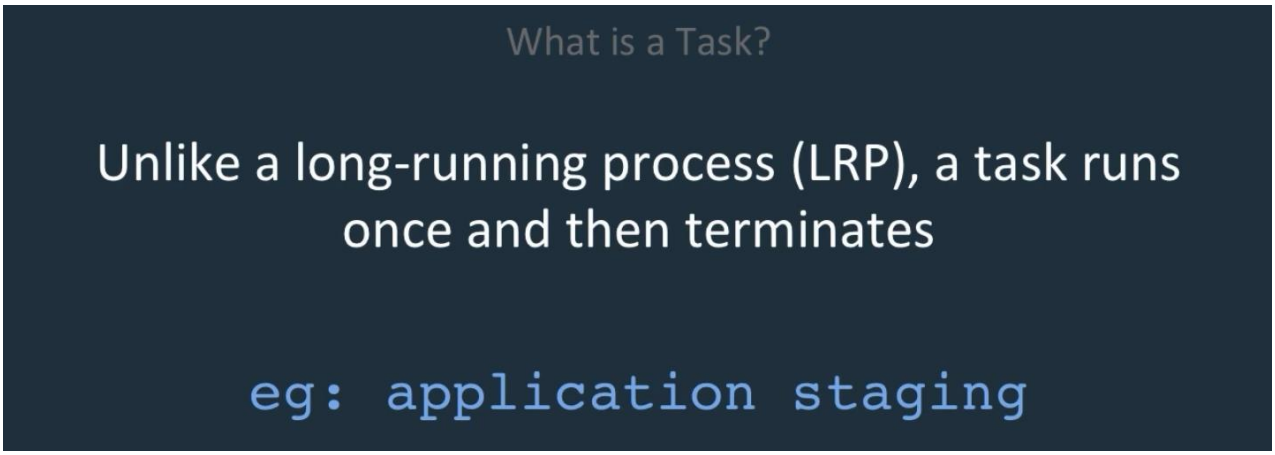
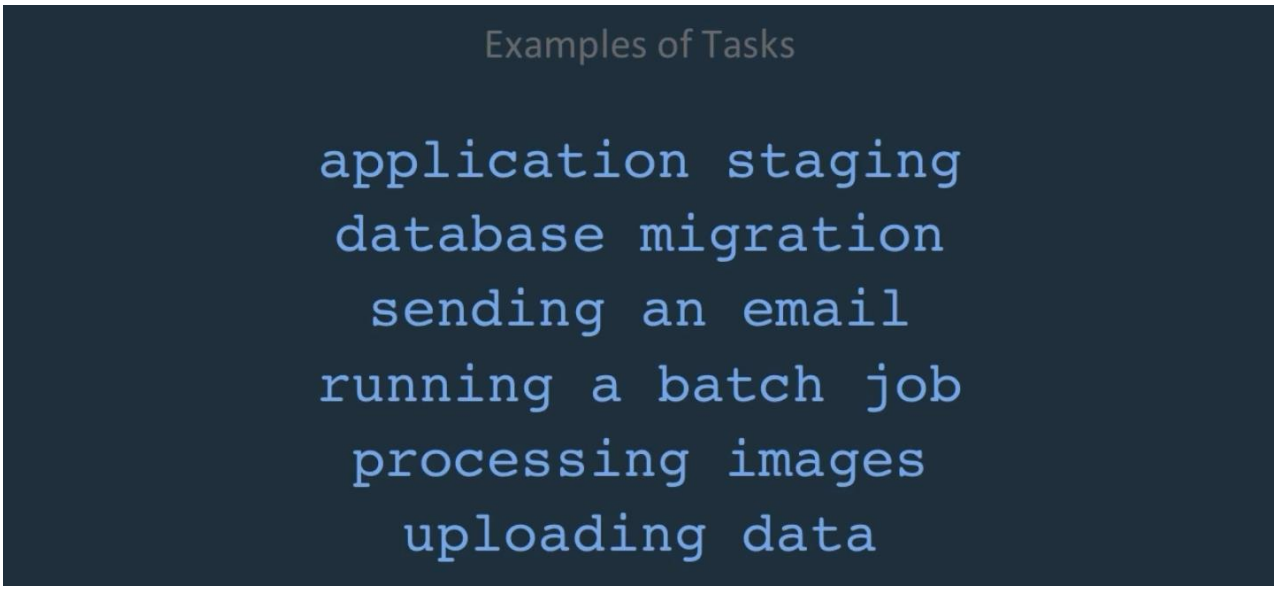




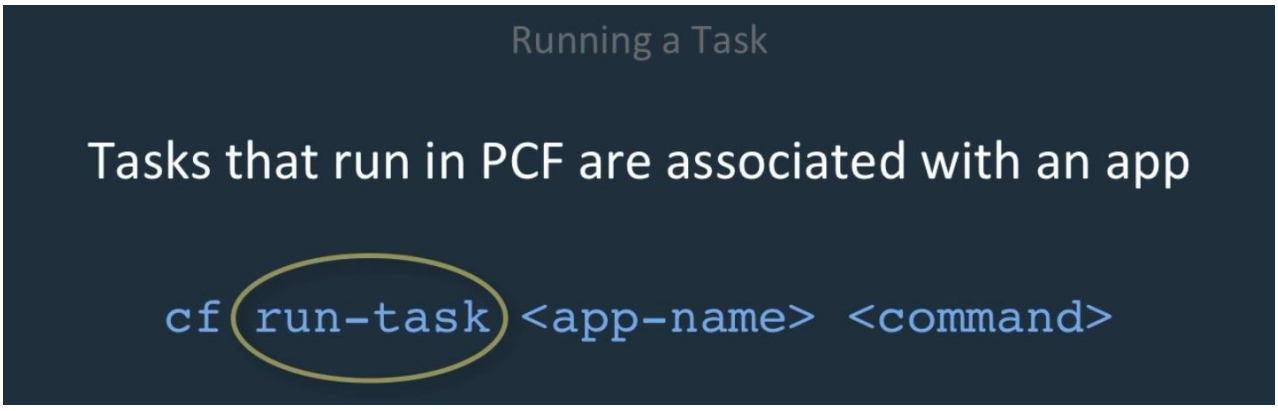


Being able to run tasks in PCF helps to guarantee that the tasks run in the same environments as our applications.

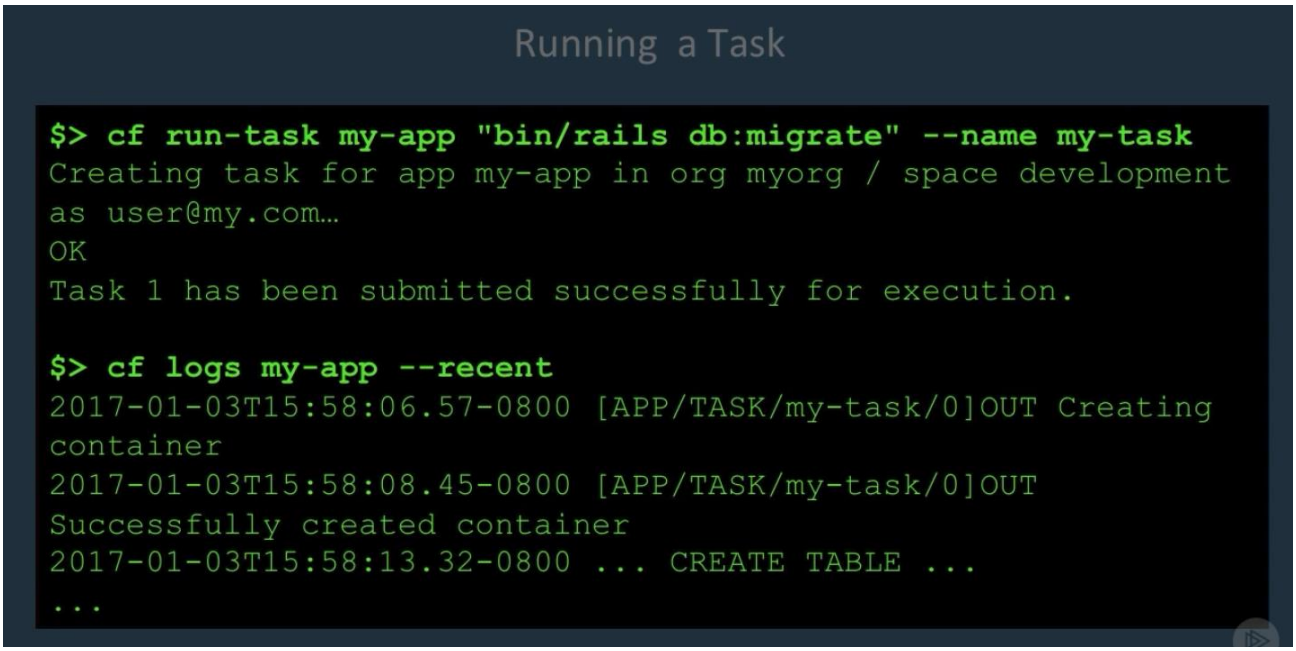




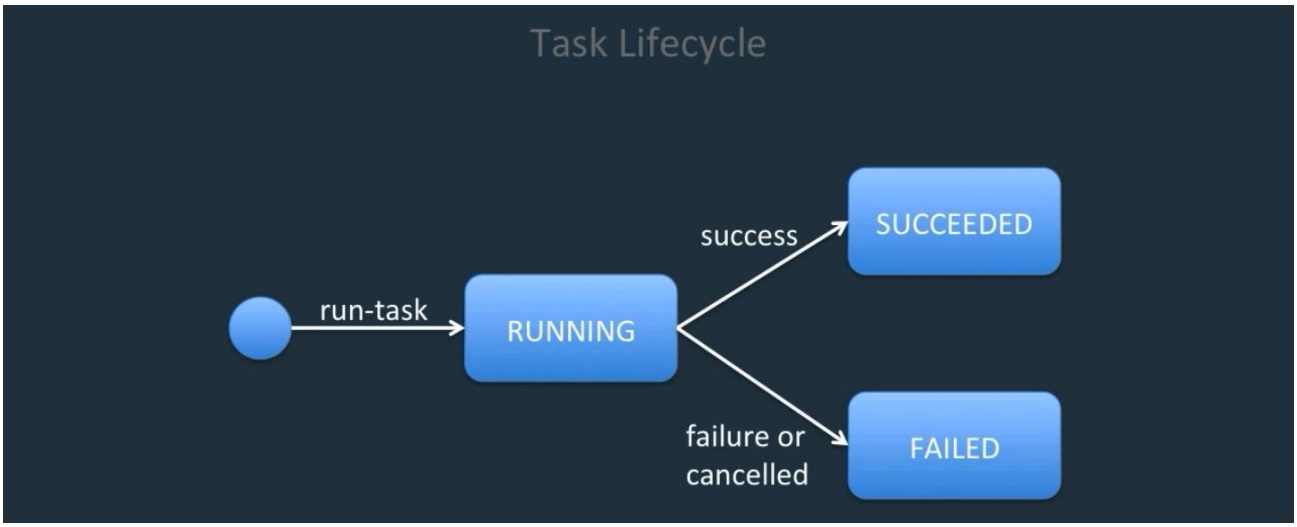
These tasks are all processes that can be kicked off by an event or performed on a schedule, they do their duty and terminate on completion.



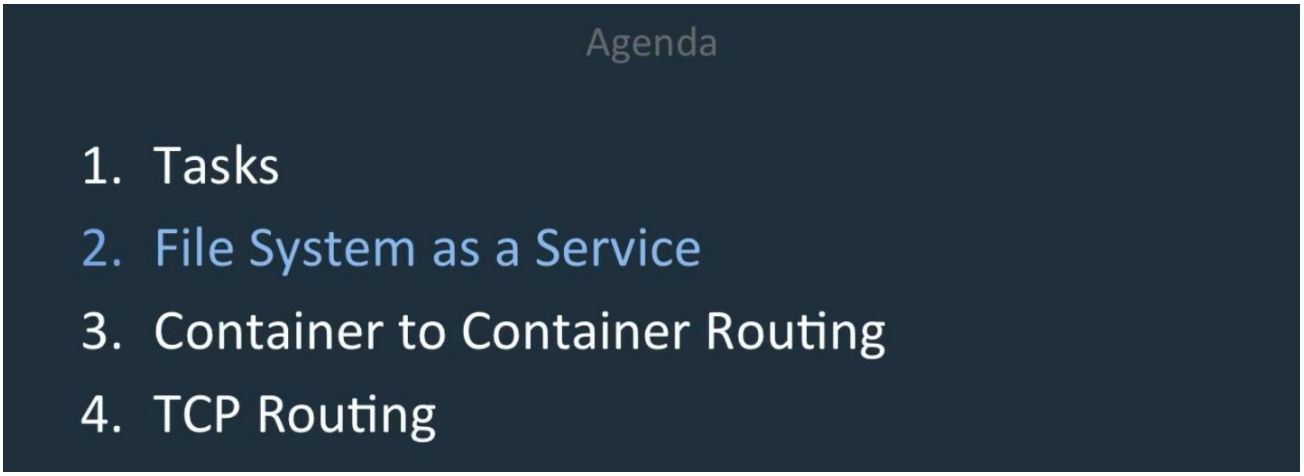
We simply deploy the application that contains the process to the PCF platform using the standard ***$ cf push*** command. Once the application containing the task process is successfully deployed, we can use the ***$ cf run-task*** command to trigger the task execution. This requires the application 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.



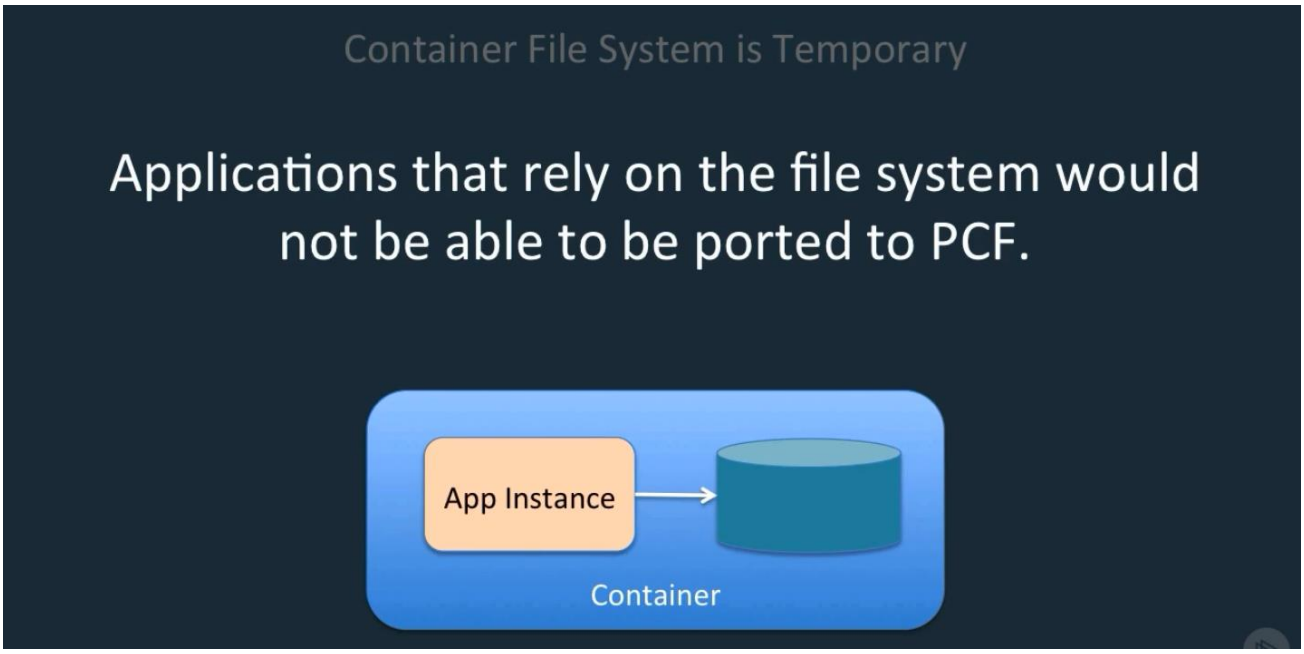
This is an example of the output for a task being run



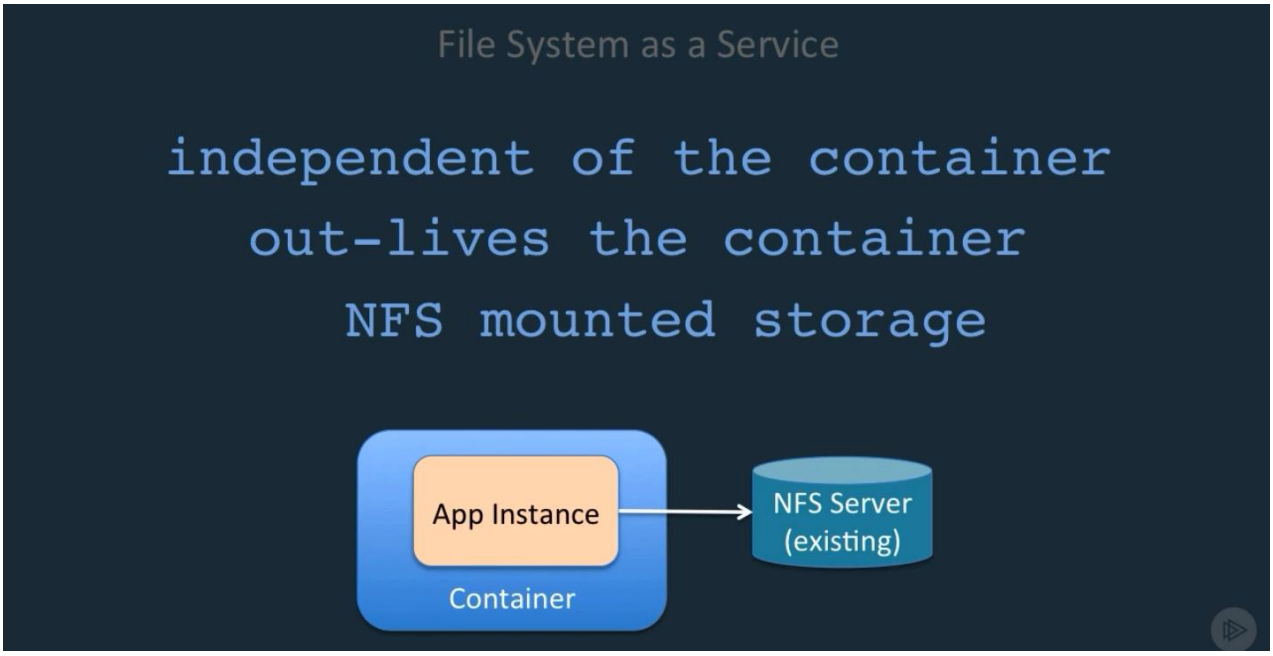
Once a task has been launched, we can monitor its status by using the ***$ cf tasks*** command, each task runs in its own container which only exists for the duration of the task execution. You can use the ***$ cf terminate task*** command, the task status will be shown as failed. All task execution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

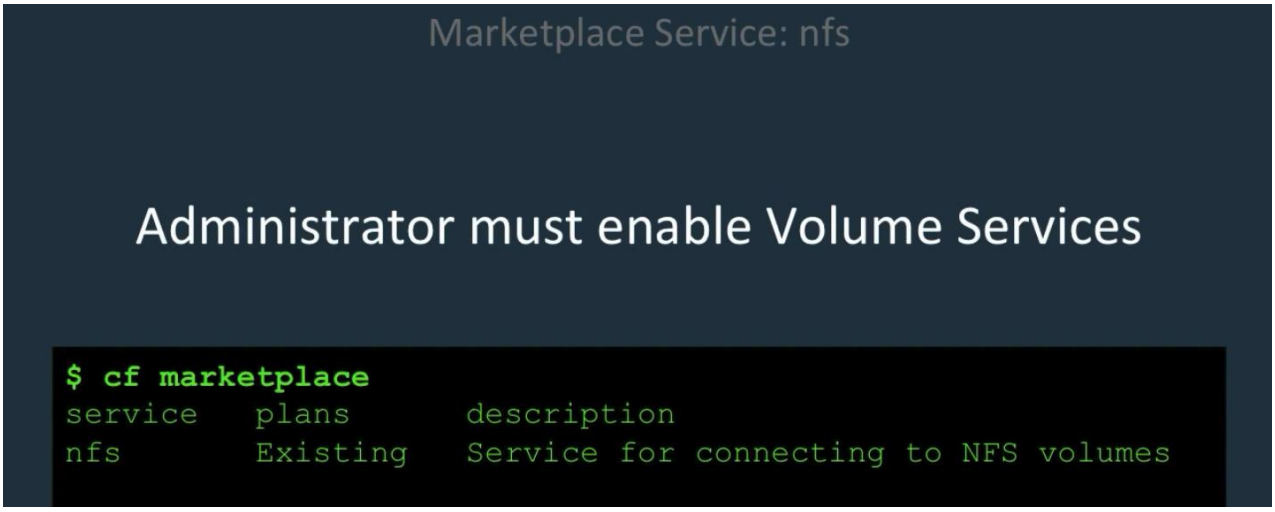


Recall that cloud-native apps should be stateless and should share nothing, any data that needs to be shared must be stored in a stateful backing service.

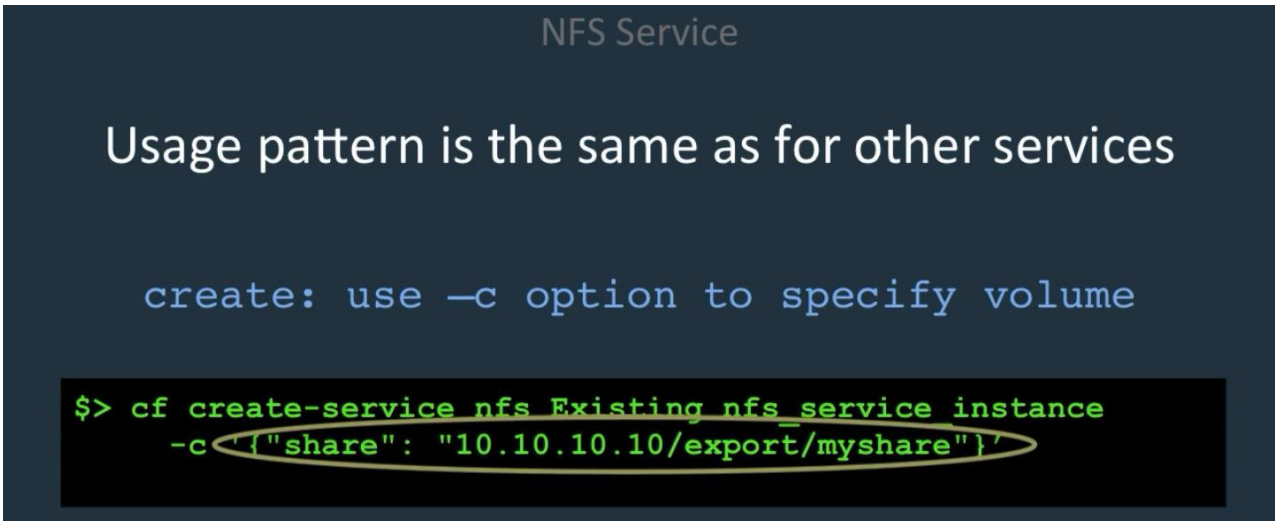


Since each application runs inside a separate container on PCF, those containers with their file systems are ephemeral. This makes it difficult to port existing apps that rely on their local file system. The **PCF file system service** feature specifically addresses these concerns by exposing a fixed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

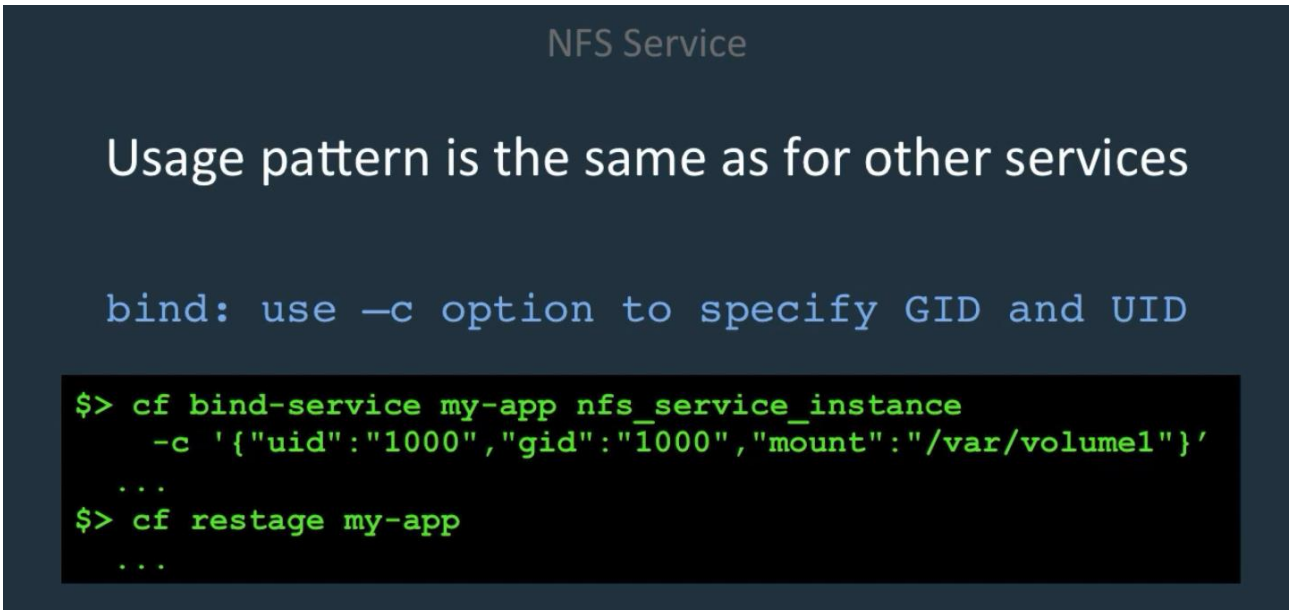




An existing NFS Server should be available and running, then the PCF admin must enable volume services from the **OpsManager** UI to make this service available to developers via the marketplace.

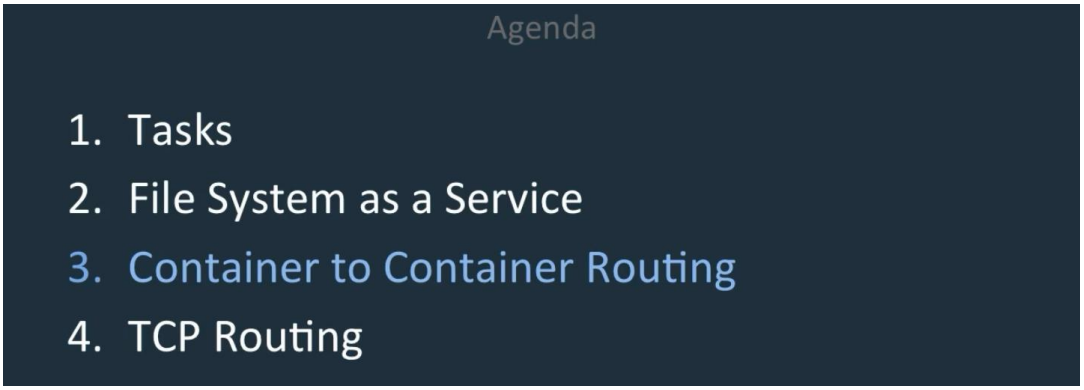


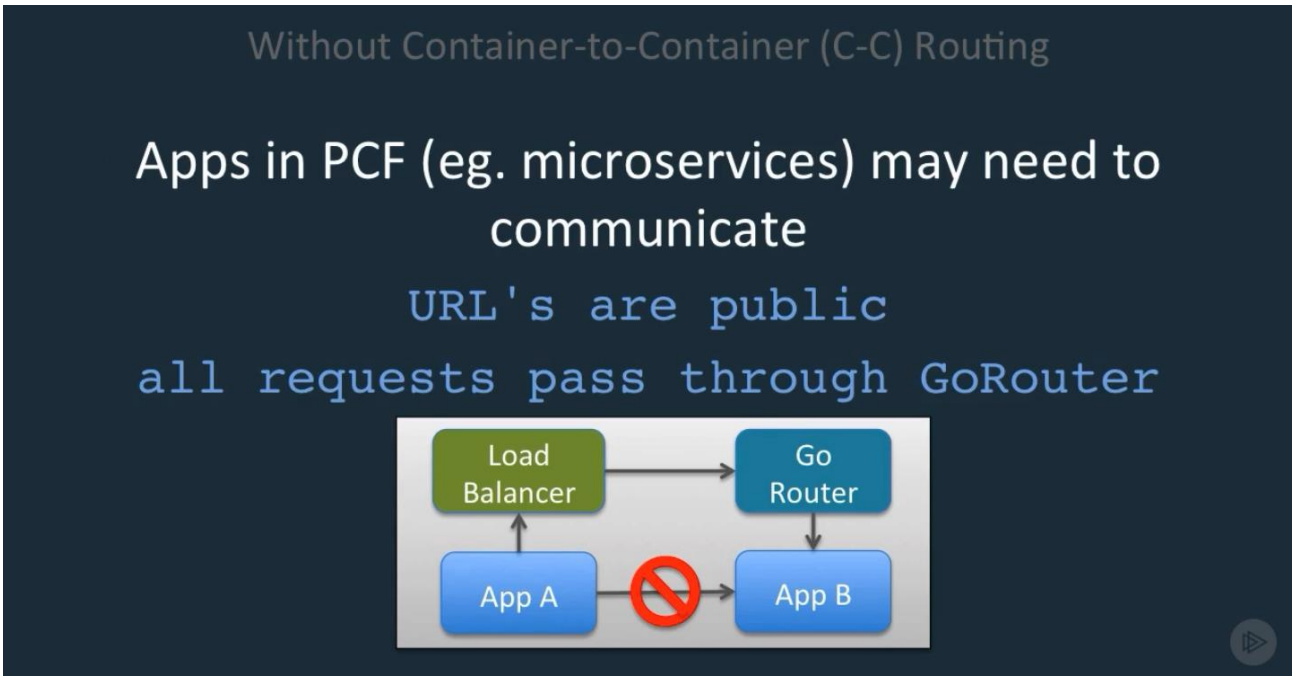
Once a service is available in the marketplace, developers can create an instance of it for use in their applications. The additional **share:…** value is required and MUST be set to the IP address and path of the existing NFS Server.



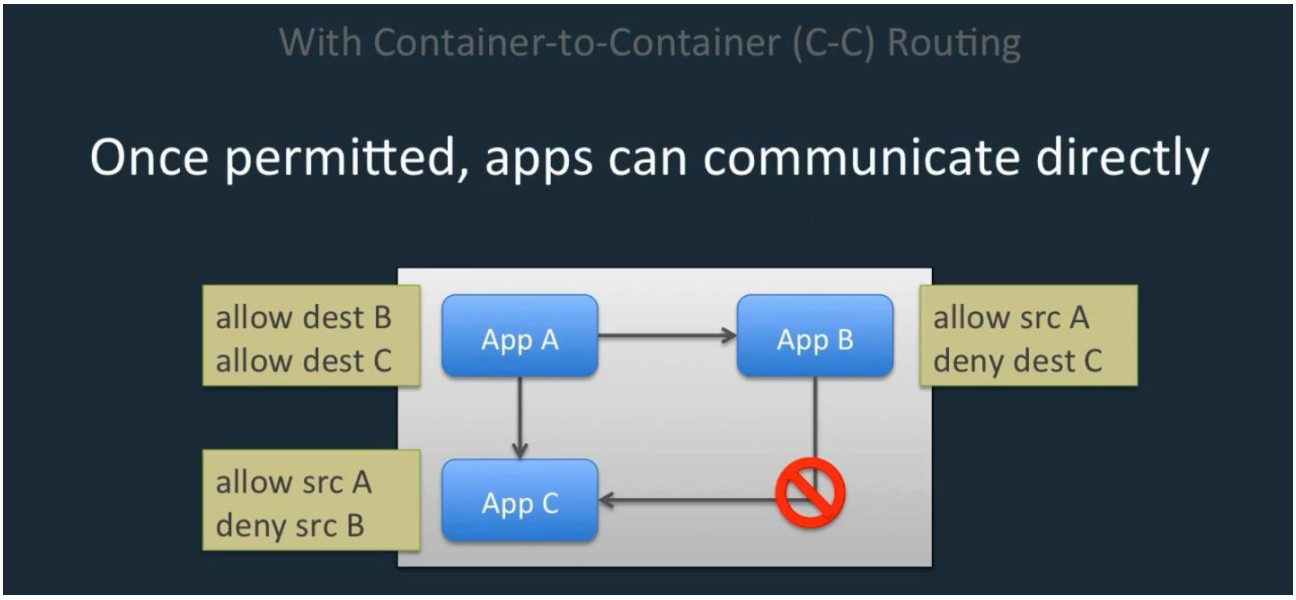
The volume service instance can be bound to your application using the **uid** and **gid** values for the group and user id to be set for any files the application might create. The mount parameter is optional, if not specified; the system will create a default mount of **container_dir** as shown below

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.

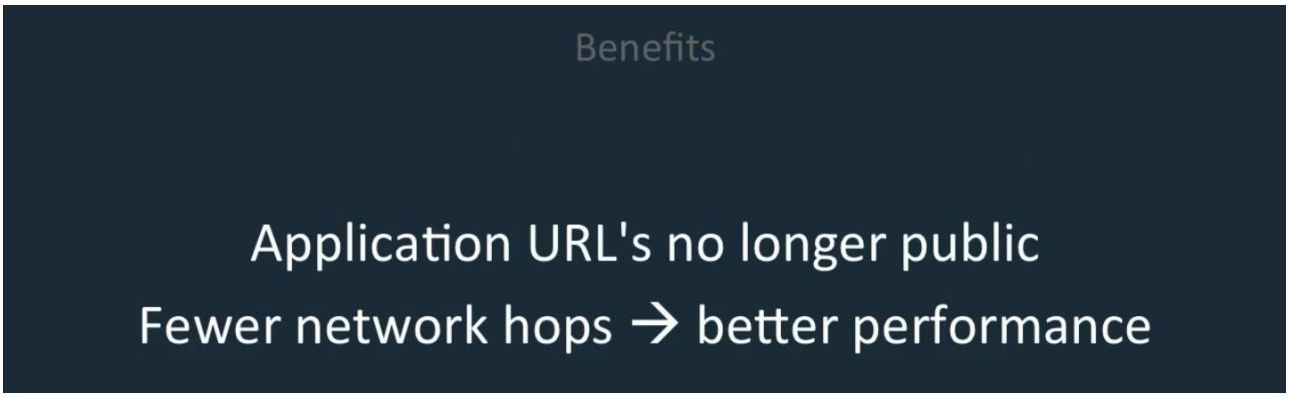


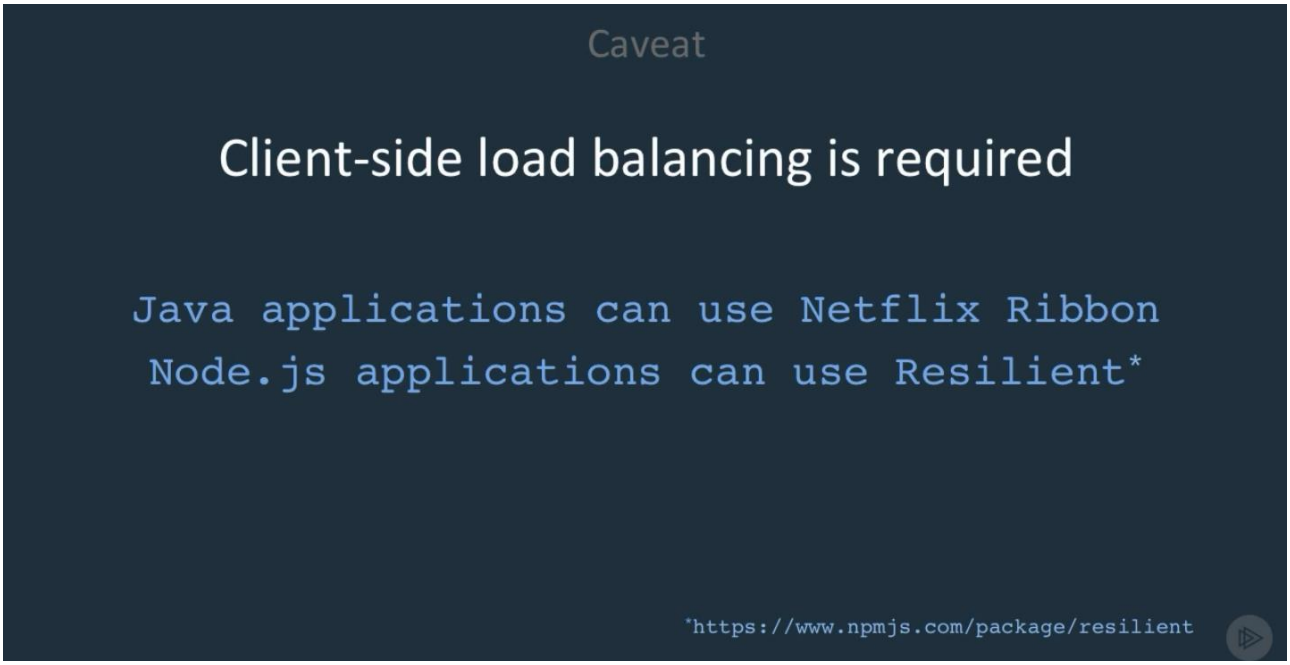


Suppose we have a lot of microservices deployed to PCF and one microservice needs to call another microservice. Previously, the request will have to go through the LB and the Router in order to get to the intended microservice.

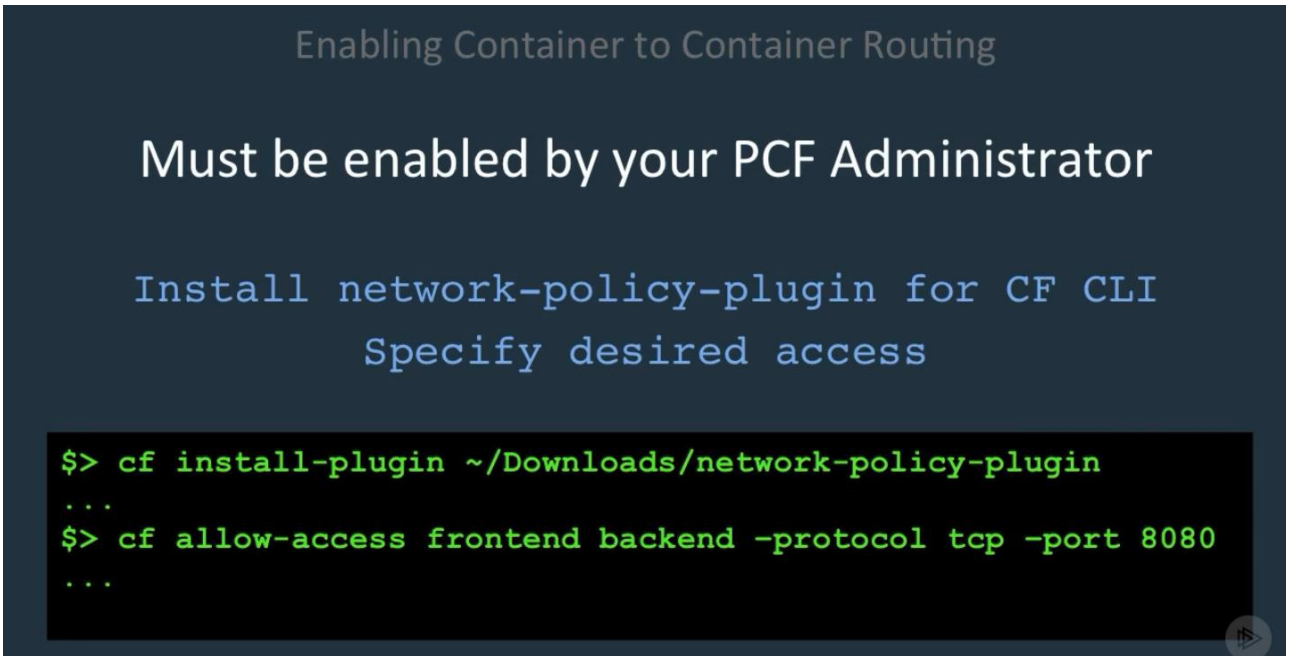


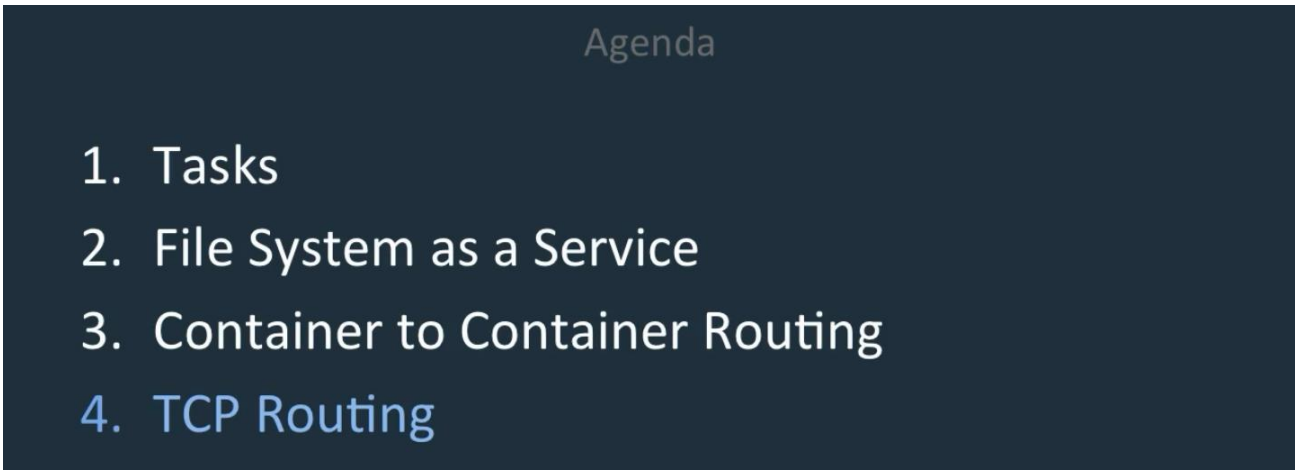
With the new container-to-container networking, those extra network hops are eliminated. The developer can explicitly specify which apps are allowed to call which apps on the platform.



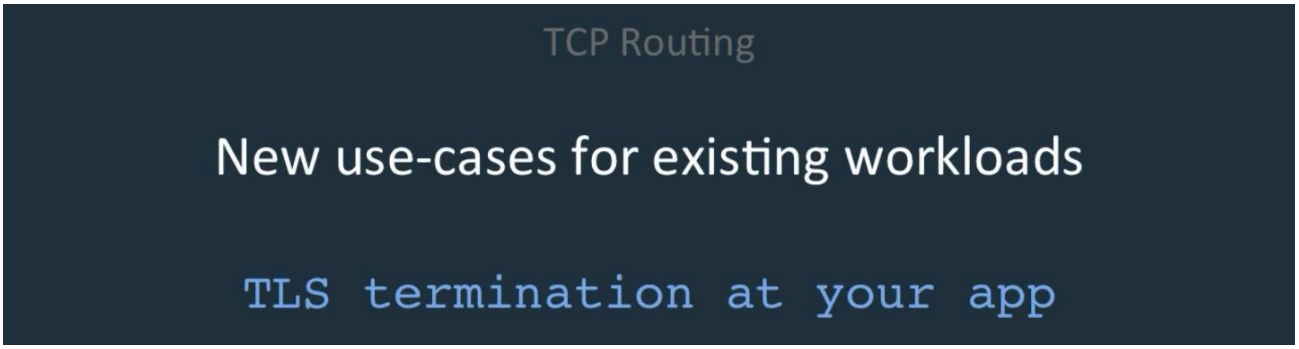


Clients must do client-side load balancing by themselves

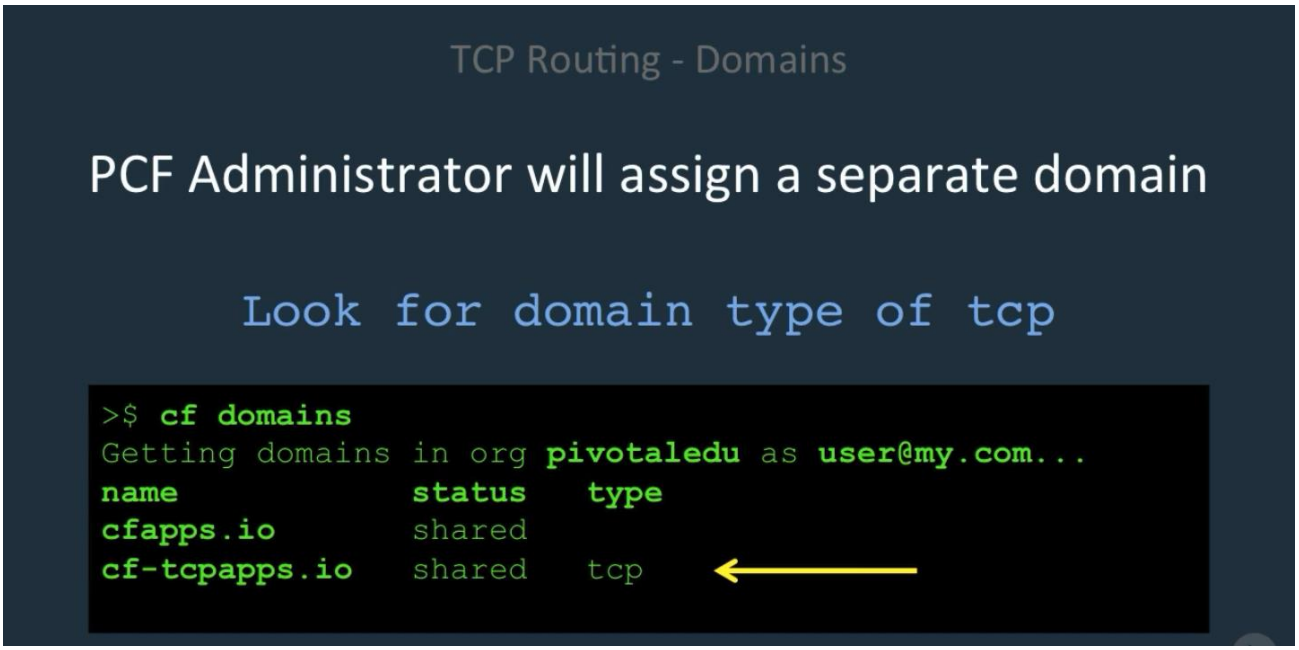




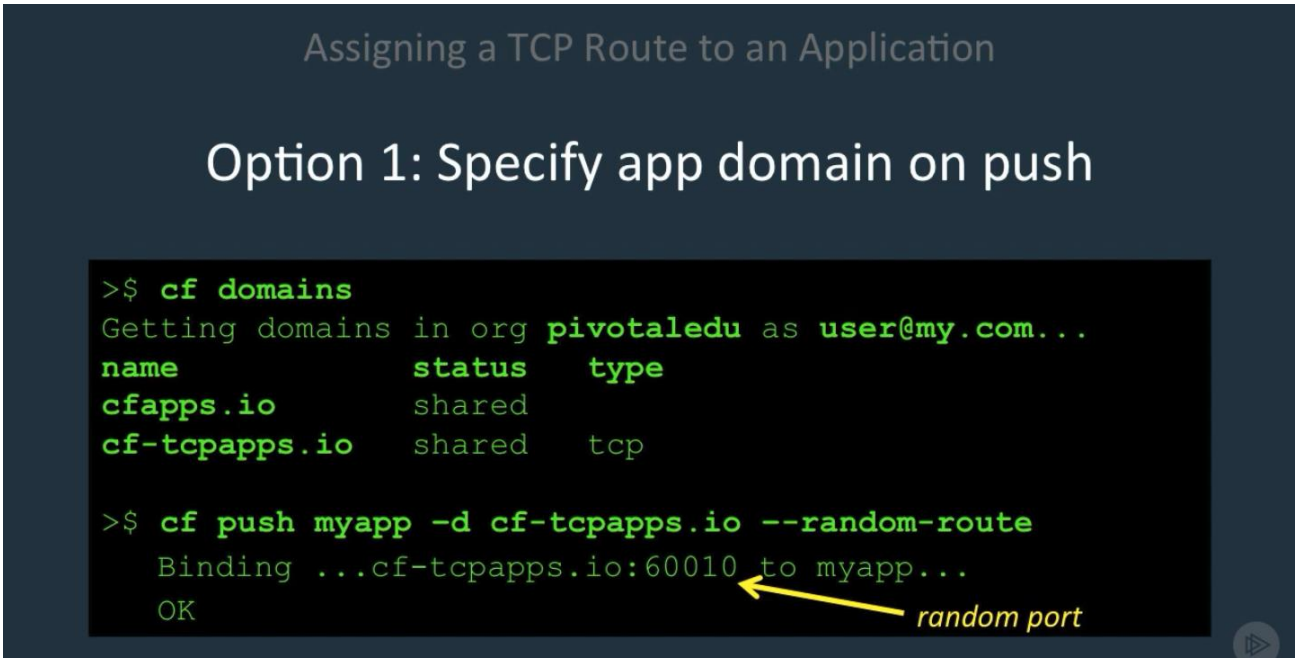
By default, PCF exposes all applications over ports 80 and 443, but there are many other network protocols outside of HTTP that leverage their own custom ports.



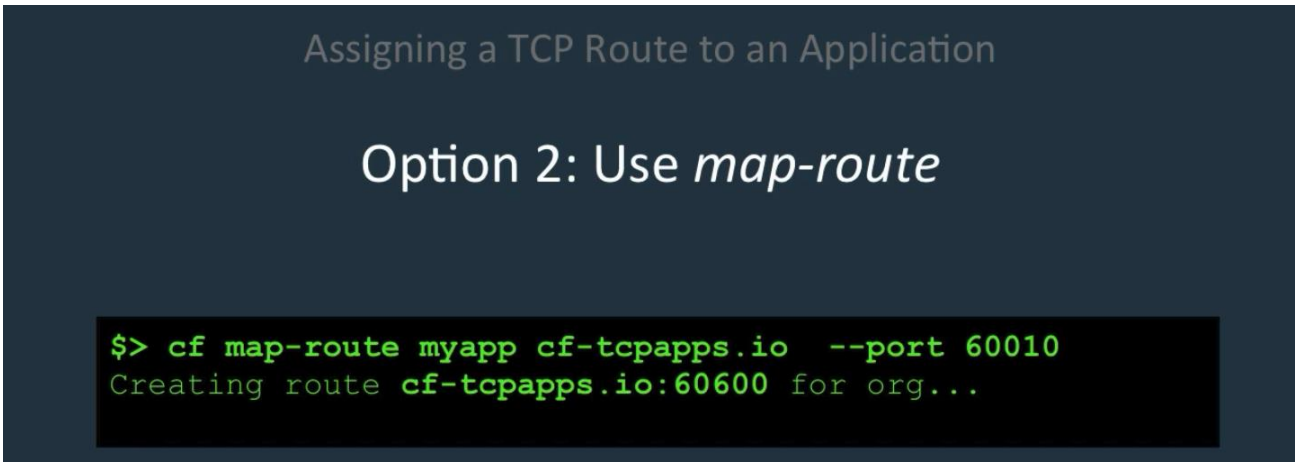
The TCP routing feature can also help with mutual-TLS between applications on the platform.



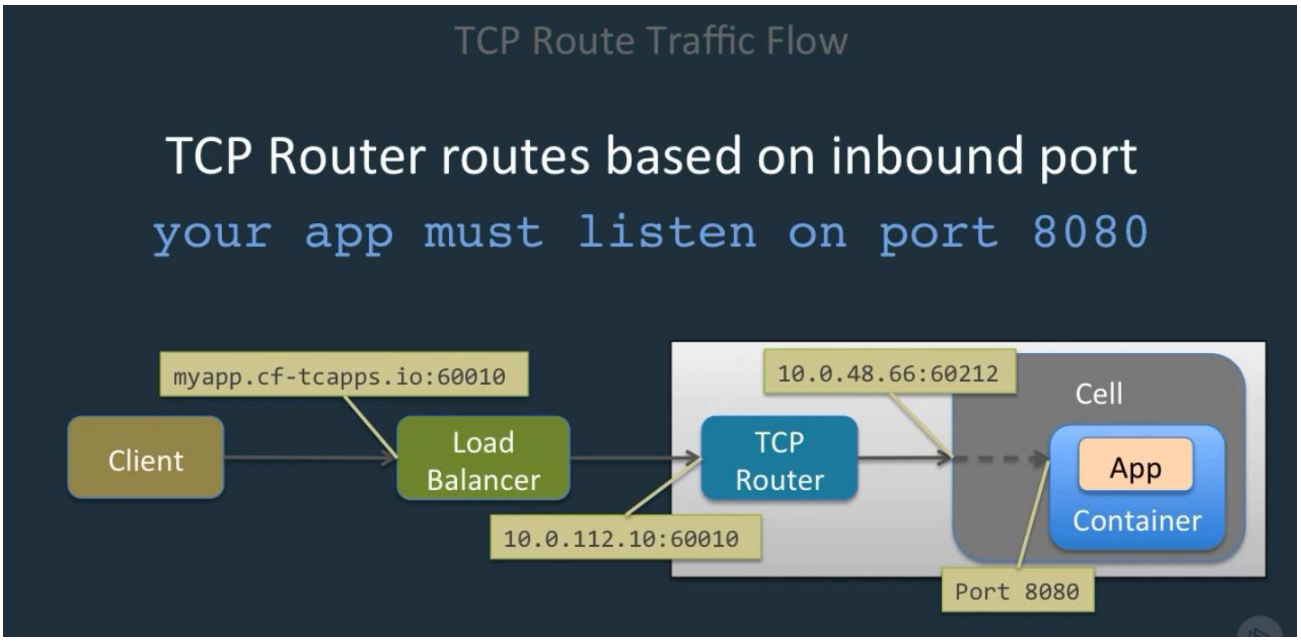
Admins are responsible for enabling mTLS on the platform, developers can check if mTLS is enabled by examining the ***$ cf domains*** command to see the domains available to them and their type.



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.



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.



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.