

# Cloud Native Applications & Cloud Foundry

CLOUD NATIVE, CLOUD NATIVE PLATFORM, CLOUD NATIVE RUNTIME, CLOUD FOUNDRY, TWELVE FACTOR, APPLICATION FRAMEWORKS

# Cloud Native Applications

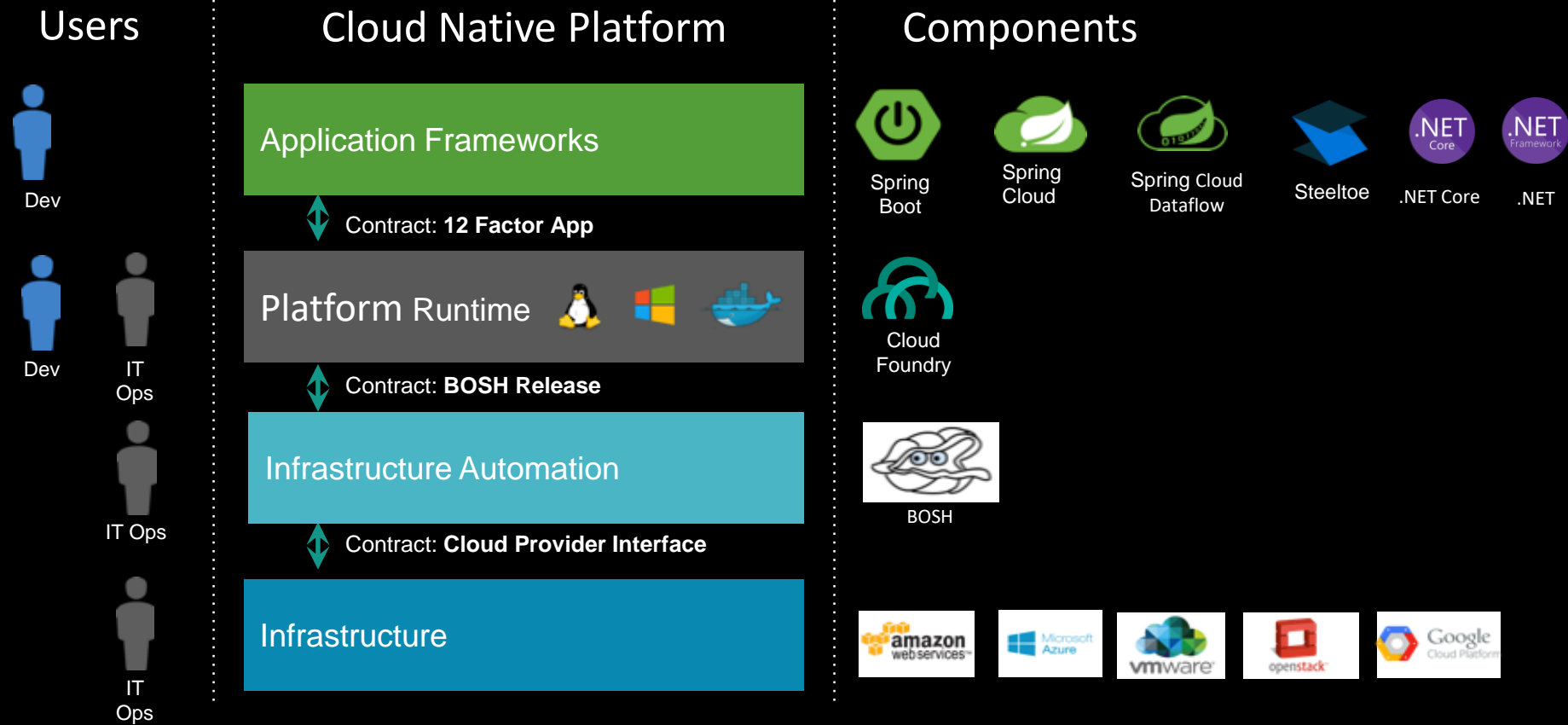
Cloud Native is not about where, but how you build and run your app!

- Microservices Architecture
- Twelve-Factor Methodology
- Containers
- Continuous Delivery
- Shift from Silo IT to DevOps

# Cloud Native Promise

- Automated provisioning & configuration
- Automated scaling
- Infrastructure independence
- Continuous delivery
- Loose coupling
- Rapid recovery
- DevOps
- Security

# Cloud Native Platform





Dev



IT Ops

Platform Runtime



Cloud Foundry

# Everything Needed to Deploy and Operate Cloud Native Applications



1. Roles and Policy



5. Security and Isolation



2. Metrics



6. Blue-Green deployment



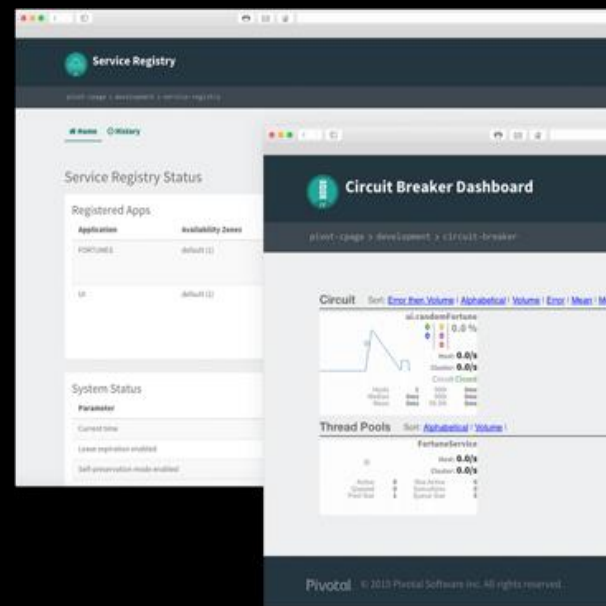
3. Log Aggregation



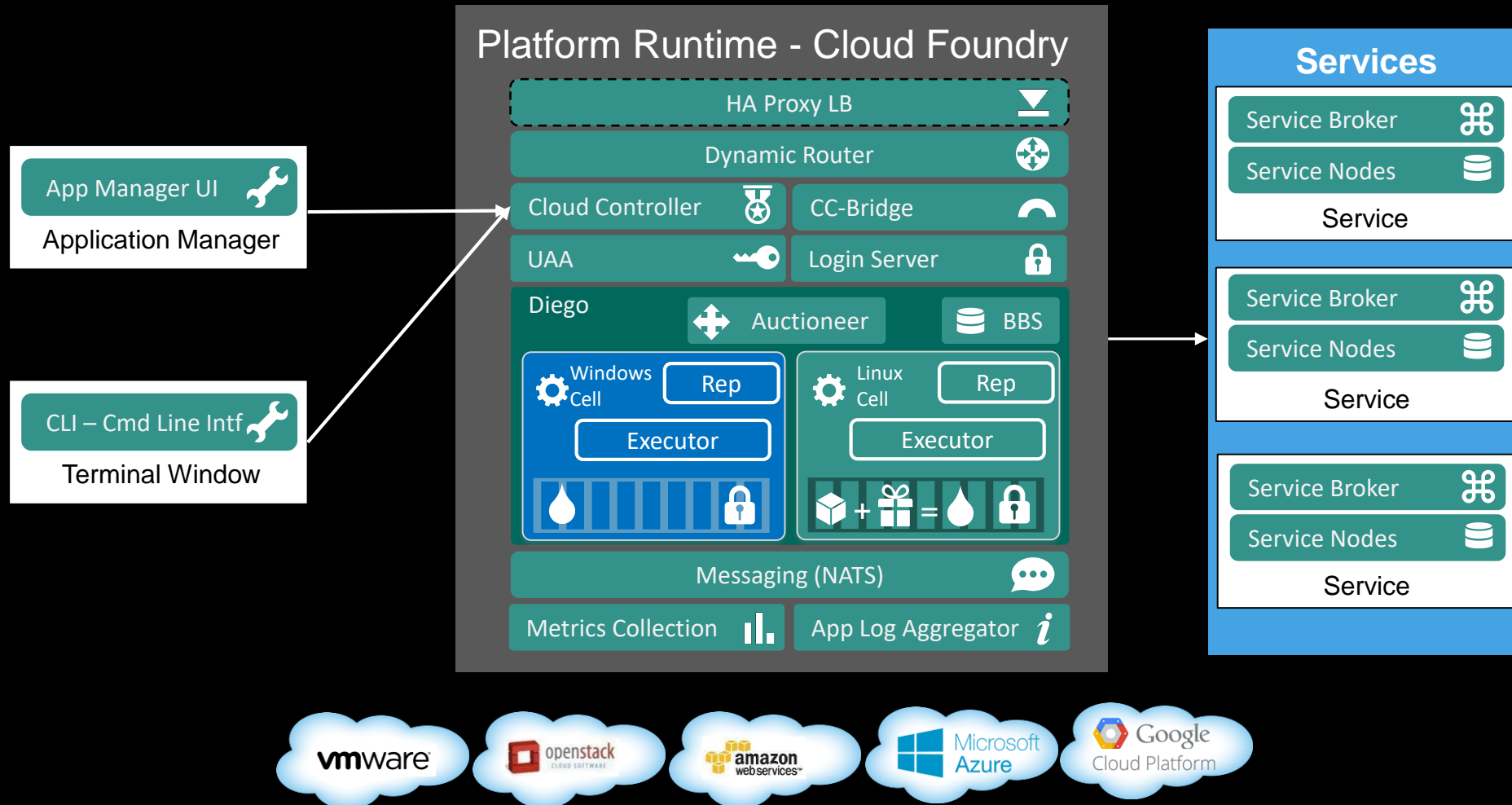
7. Scaling



4. Health management



# Cloud Foundry Architecture



# Twelve Factor Applications – Platform Contract

Architectural and development best practices – <http://12factor.net>

<b>I. Codebase</b> One codebase tracked in SCM, many deploys	<b>II. Dependencies</b> Explicitly declare and isolate dependencies	<b>III. Configuration</b> Store config in the environment
<b>IV. Backing Services</b> Treat backing services as attached resources	<b>V. Build, Release, Run</b> Strictly separate build and run stages	<b>VI. Processes</b> Execute app as stateless processes
<b>VII. Port binding</b> Export services via port binding	<b>VIII. Concurrency</b> Scale out via the process model	<b>IX. Disposability</b> Maximize robustness with fast startup and graceful shutdown
<b>X. Dev/prod parity</b> Keep dev, staging, prod as similar as possible	<b>XI. Logs</b> Treat logs as event streams	<b>XII. Admin processes</b> Run admin / mgmt tasks as one-off processes



## Application Frameworks



Spring Boot



Spring Cloud  
Dataflow



Spring Cloud  
Services



Steeltoe



.NET Core



.NET

## Facilitates Twelve-Factor Contract

### Spring Cloud Services

Powered by Netflix OSS



- Spring Cloud Services
  - Which is built on Spring Boot simplifies distributed, microservice-style architecture by implementing proven patterns to bring resilience, reliability, and coordination to your microservices.
  - When used with PCF customers have a turnkey, secure solution for production operations of this coordination infrastructure—service registry, config server, and circuit breaker dashboard.
  - Steeltoe enables Spring Cloud Services on .NET

## Enabling Cloud Native Applications

### Service Registry

A dynamic directory that enables client side load balancing and smart routing

### Cloud Bus

Application bus to broadcast state changes, leadership election

### Circuit Breaker

Microservice fault tolerance with a monitoring dashboard

### OAuth2 Patterns

Support for single sign on, token relay and token exchange

### Configuration Server

Dynamic, versioned propagation of configuration across lifecycle states without the need to restart your application

### Lightweight API Gateway

Single entry point for API consumers (browsers, devices, other APIs)

### Spring Cloud Services

Turnkey microservice operations and security on Pivotal Cloud Foundry





## Application Frameworks



Spring Boot



Spring Cloud  
Dataflow



Spring Cloud  
Services



Steeltoe



.NET Core



.NET

## Facilitates Twelve-Factor Contract on .NET



## Enabling Cloud Native Applications on .NET

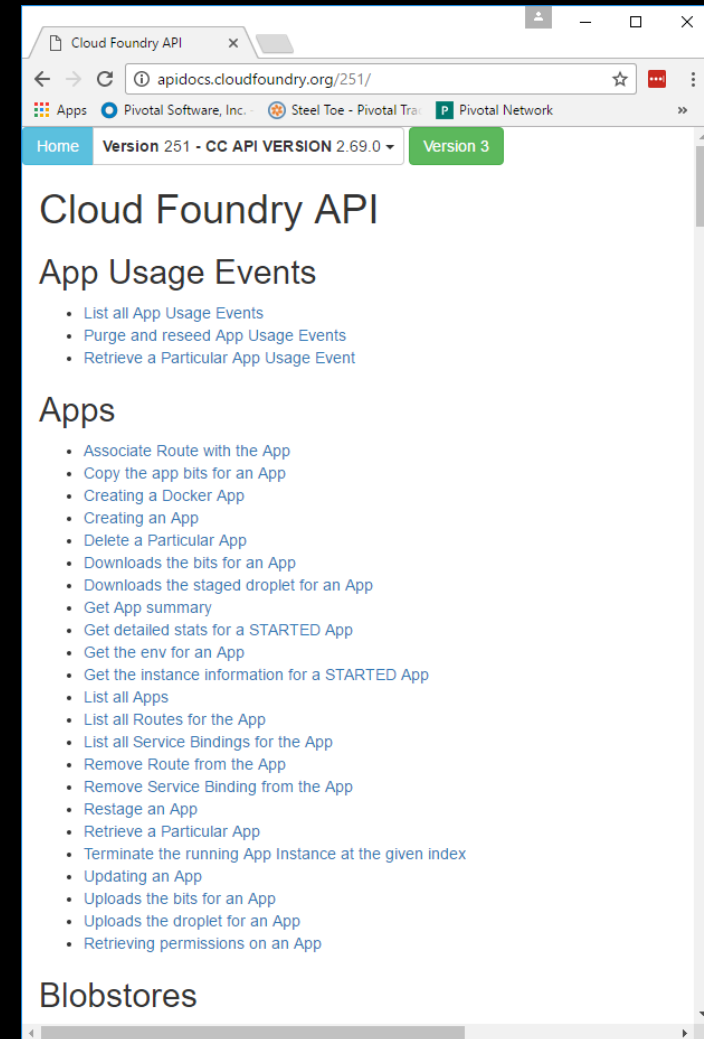
- Simplifies using .NET & ASP.NET on Cloud Foundry
  - Connectors (e.g. MySql, Redis, Postgres, RabbitMQ, OAuth, etc.)
  - Security providers (e.g. OAuth SSO, JWT, Redis KeyRing Storage, etc.)
  - Configuration providers (e.g. Cloud Foundry)
- Simplifies using Spring Cloud Services
  - Configuration server provider (e.g. Config Server, etc.)
  - Service Discovery (e.g. Eureka, etc.)
  - Circuit Breaker (e.g. Hystrix coming)
  - Distributed Tracing (e.g. Slueth coming)

# Cloud Foundry Fundamentals

ORGS, SPACES, USERS, ROLES, CLI, API, APPS MANAGER

# Cloud Controller API

- Cloud Controller (CC) component of Elastic Runtime manages all Cloud Foundry APIs
- CF CLI and other clients like Apps Manager directly call this API
- Before accessing the CC API, you must get an access token from the User Account and Authentication (UAA) server
- <http://apidocs.cloudfoundry.org>



# CLI – Command Line Interface

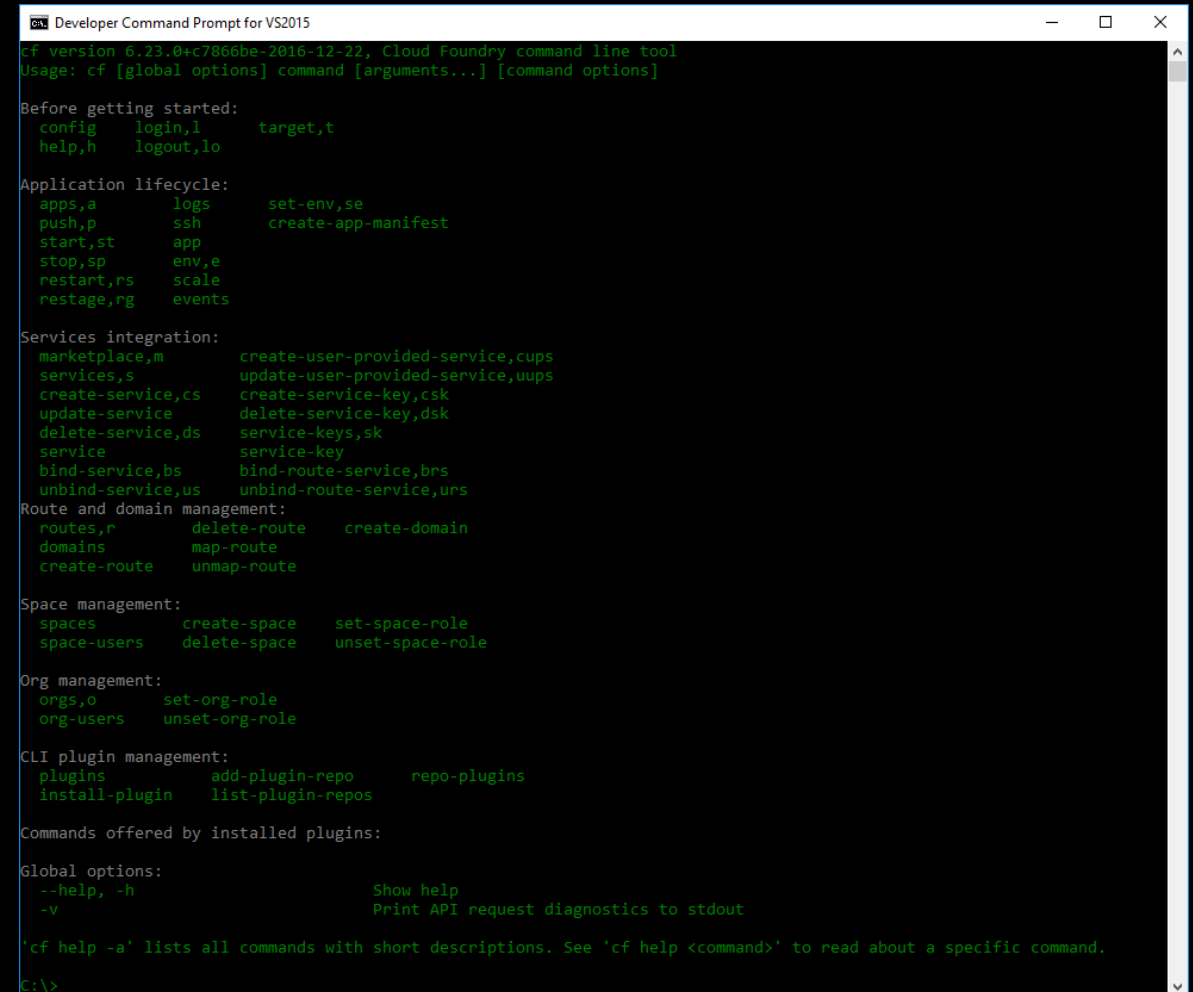
- Command line utility providing easy access to the Cloud Controller API.
- Scriptable
- Fully documented

```
cf help -a
```

```
cf help <command>
```

```
cf api http://foo.bar.com/
```

```
cf login <username>
```



```
Developer Command Prompt for VS2015
cf version 6.23.0+c7866be-2016-12-22; Cloud Foundry command line tool
Usage: cf [global options] command [arguments...] [command options]

Before getting started:
  config login,l      target,t
  help,h  logout,lo

Application lifecycle:
  apps,a      logs      set-env,se
  push,p      ssh       create-app-manifest
  start,st    app
  stop,sp     env,e
  restart,rs  scale
  restage,rg  events

Services integration:
  marketplace,m  create-user-provided-service,cups
  services,s    update-user-provided-service,uups
  create-service,cs  create-service-key,csk
  update-service    delete-service-key,dsk
  delete-service,ds  service-keys,sk
  service           service-key
  bind-service,bs   bind-route-service,brs
  unbind-service,us unbind-route-service,urs

Route and domain management:
  routes,r      delete-route      create-domain
  domains       map-route
  create-route  unmap-route

Space management:
  spaces      create-space      set-space-role
  space-users delete-space      unset-space-role

Org management:
  orgs,o      set-org-role
  org-users   unset-org-role

CLI plugin management:
  plugins      add-plugin-repo      repo-plugins
  install-plugin list-plugin-repos

Commands offered by installed plugins:

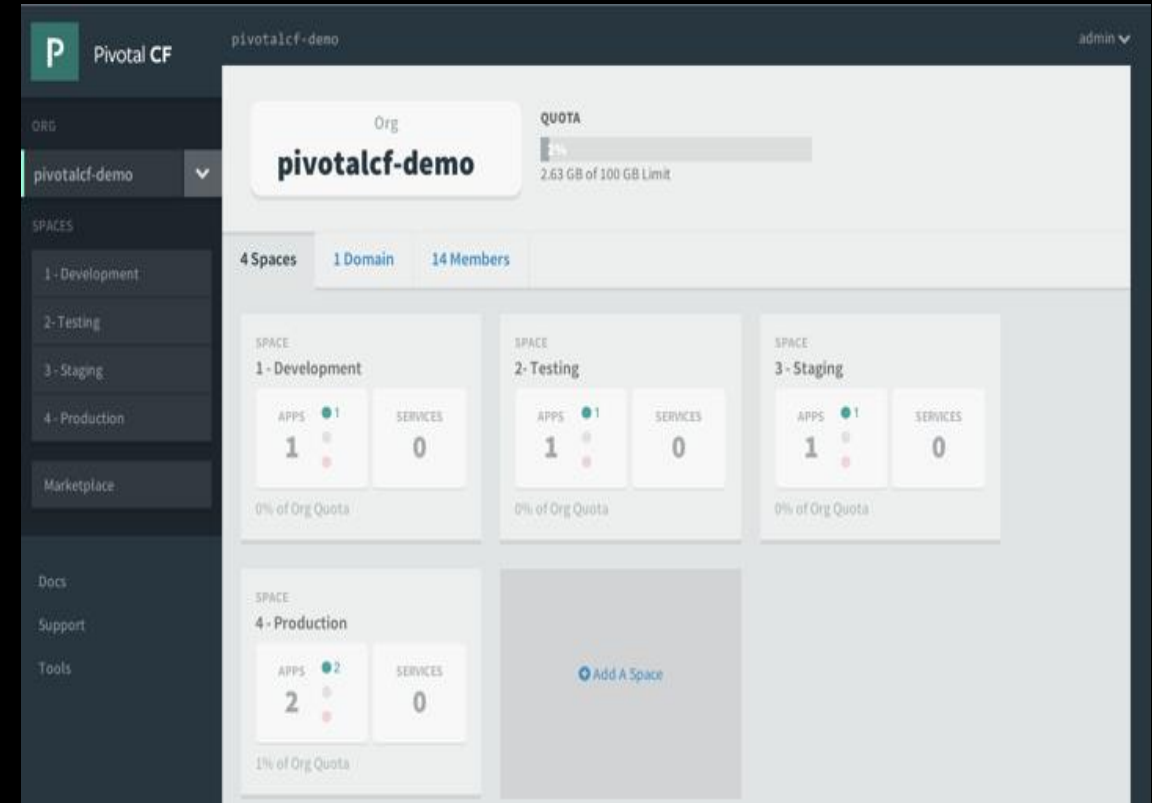
Global options:
  --help, -h      Show help
  -v             Print API request diagnostics to stdout

'cf help -a' lists all commands with short descriptions. See 'cf help <command>' to read about a specific command.

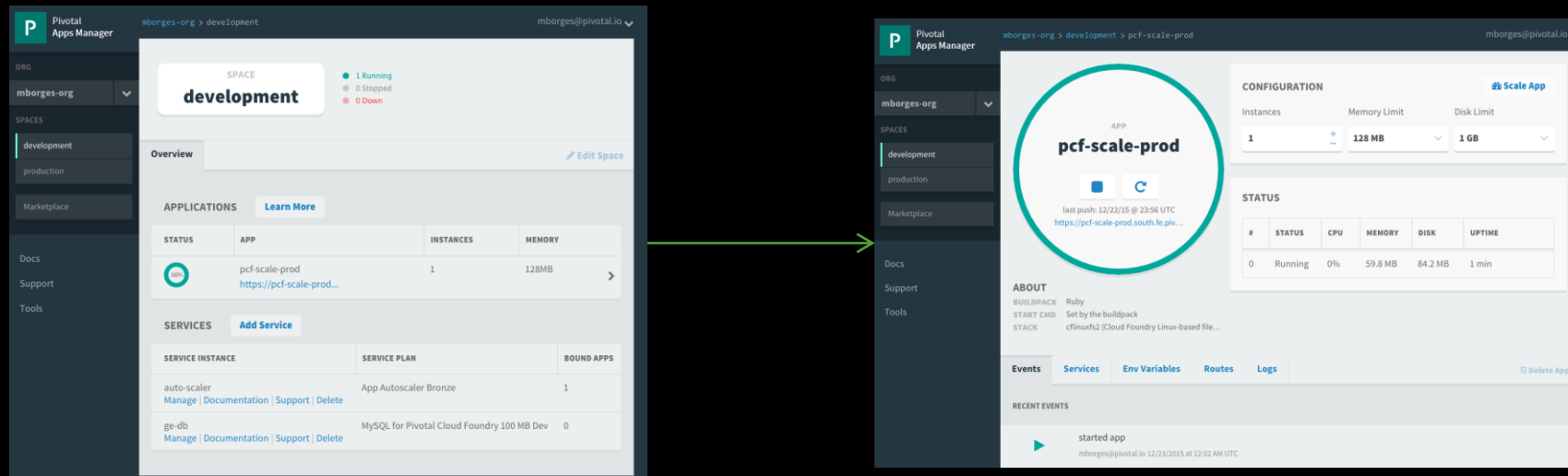
C:\>
```

# Pivotal Apps Manager

- Manage Organizations, users, applications and Spaces
- Monitor applications logs, services and routes
- Access Service Marketplace, create services and bind to applications



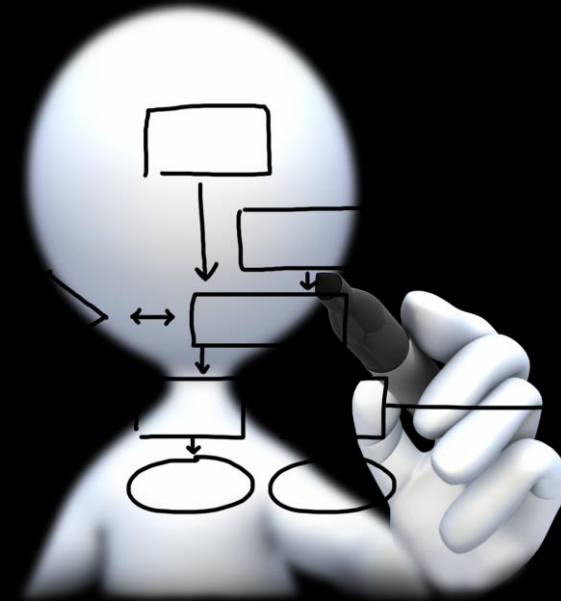
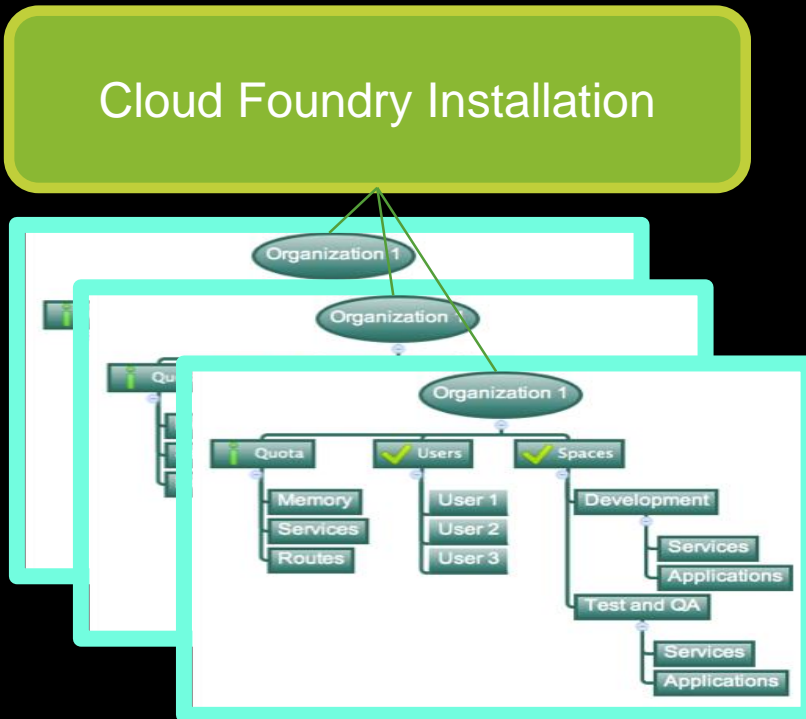
# Pivotal Apps Manager – App View



- Drill into a space to see all application and services instances
- Then drill into an application to see configuration, status, event, logging, routes, environment variables and service instances bound to the application

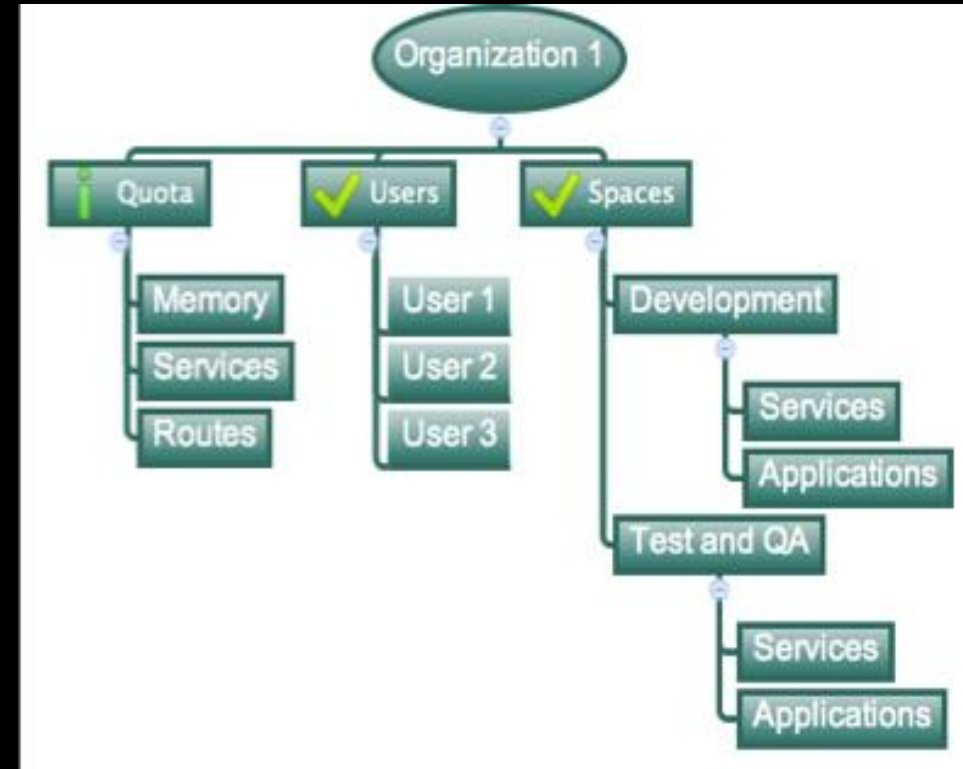
# Orgs, Spaces, Users and Quotas

## Cloud Foundry Installation



# Organizations

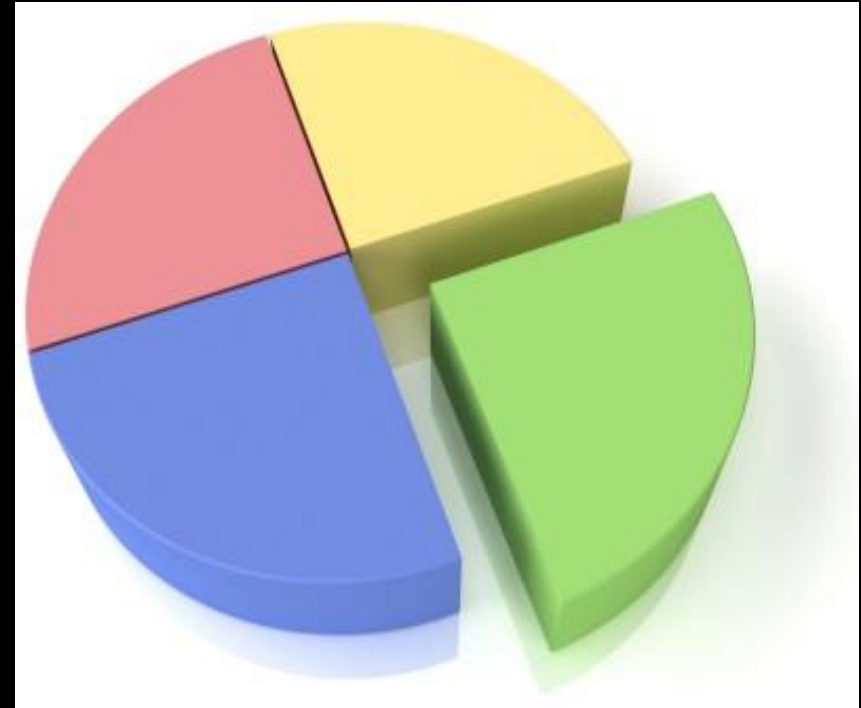
- Top-most administrative unit
- Logical division within a Pivotal Cloud Foundry Install / Foundation
  - Typically a company, department, application suite or large project
- Each organization has its own users and assigned quota
- User permissions / Roles are specified per space within an organization
- Sub-divided into spaces





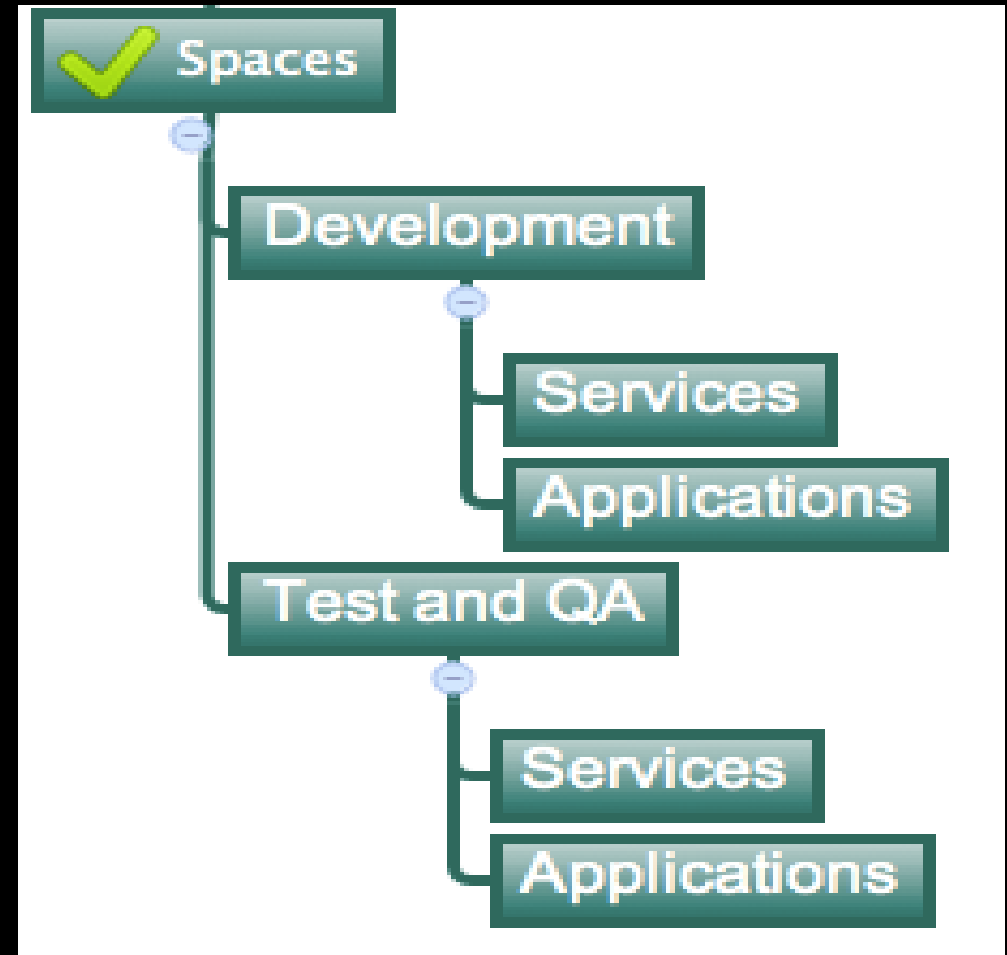
# Quotas

- Different quota limits (e.g. small, enterprise, default, runaway) can be assigned per Organization
- Quotas define
  - Total Memory
  - Total # of services
  - Total # of Routes
- Sub-divided into spaces



# Spaces

- Logical sub-division within an organization
- Users authorized at an organization level can have different roles per space
- Services and Applications are created / target per Space
- Same service name can have different meaning per space



# Users and Roles

- Users are members of an organization
  - Usually they are operators or developers (not application end users)
  - Users are sent an email invite and asked to create an account
- Users have specific organization and space roles
  - Organization roles grant permissions in an organization
  - Space roles grant permissions in a particular space
  - A combination defines the user's overall permissions



# Lab0 – Logging into Cloud Foundry (CF)

- In this lab we are ensuring we all have access to the Workshop environment
  - Ensuring we have the CLI installed
  - Verifying connectivity & credentials
  - Using the CLI to access CF
  - Accessing CF via the Apps Manager

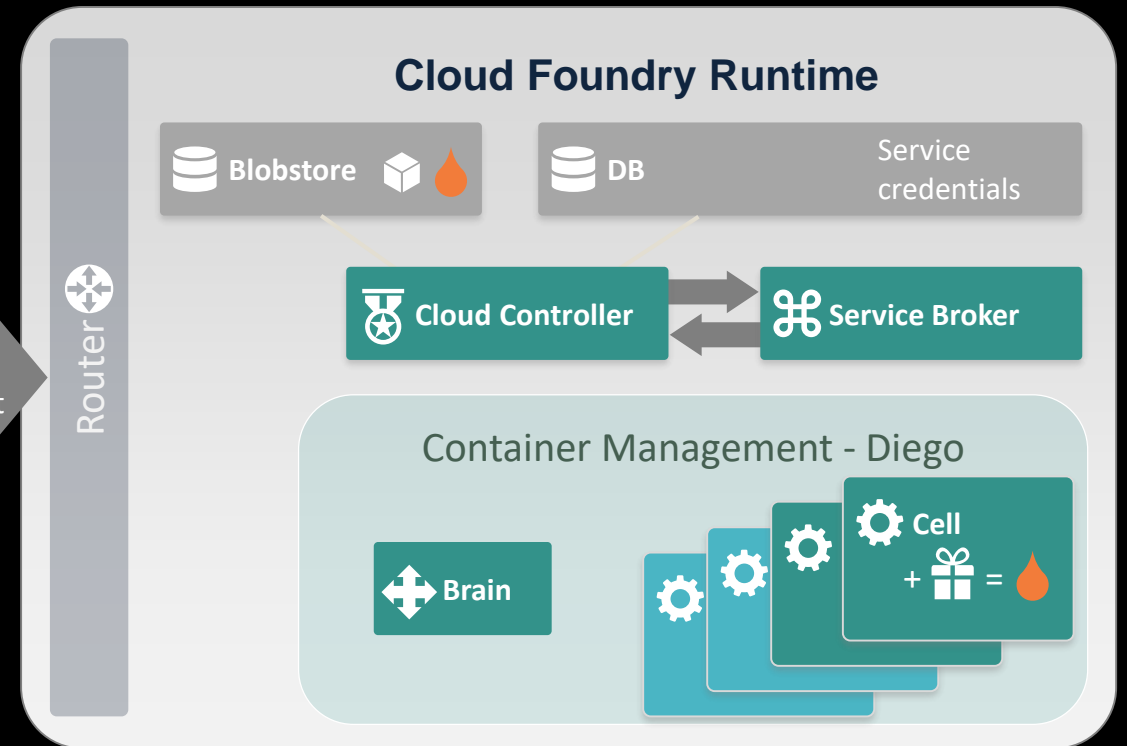
# Running Applications on Cloud Foundry

CF PUSH, MANIFEST, STAGING, BUILD PACKS, CONTAINERS, CELLS, ENVIRONMENT VARIABLES, VCAP\_APPLICATION

# Pushing an Application

1. Upload app bits and metadata
2. Bind services
3. Stage application
4. Save staged application image (i.e. droplet 🍯 )
5. Deploy image to container
6. Manage applications health

```
cf push appname -p <path to bits>  
cf push appname -f <manifest> -p <pathtobits>
```



# Manifest Files

- Application manifests tells `cf push` what to do with applications
- What OS stack to run on: Windows or Linux
- How many instances to create and how much memory to use.
- Helps automate deployment, specially of multiple apps at once
- Can list services to be bound to the application
- YAML format – <http://yaml.org>

```
1 ---|
2 # all applications use these settings and services
3 domain: shared-domain.com
4 memory: 1G
5 instances: 1
6 services:
7 - clockwork-mysql
8 applications:
9 - name: springtock
10   host: tock09876
11   path: ./spring-music/build/libs/spring-music.war
12 - name: springtick
13   host: tick09875
14   path: ./spring-music/build/libs/spring-music.war
```

# Staging an Application – Applying Buildpacks

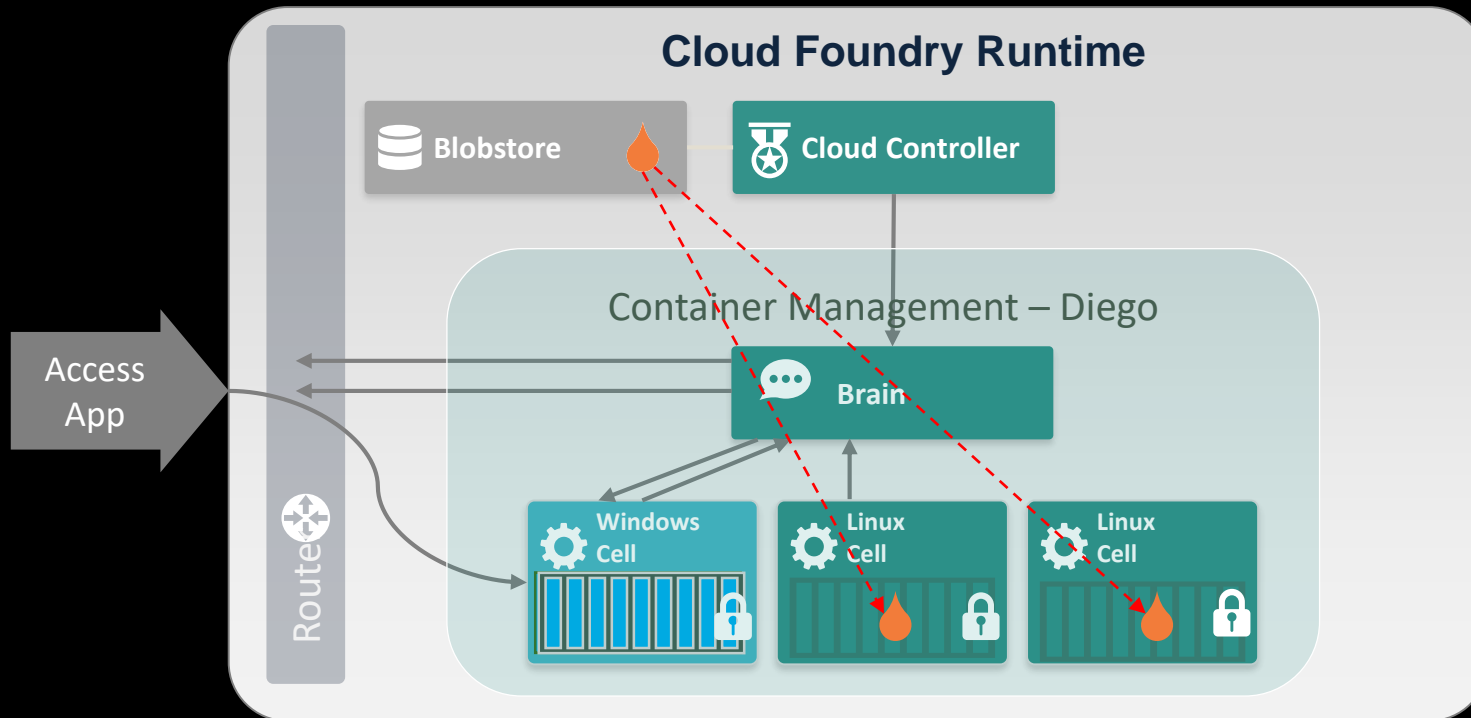
- Buildpacks build container images
- Buildpacks take care of
  - Detecting which type of application is being pushed
  - Installing the appropriate run-time
  - Installing required dependencies or other artifacts
  - Creating the command used to start the application
- Lots of Buildpacks
  - Staticfile
  - Java
  - Ruby
  - Nodejs
  - Go
  - Python
  - PHP
  - .NET Core
  - Binary



# Why Buildpacks

- Control what frameworks/runtimes are used on the platform
- Provides consistent deployments across environments
  - Stops deployments from piling up at operation's doorstep
  - Enables a self-service platform
- Eases ongoing operations burdens:
  - Security vulnerability is identified
  - Subsequently fixed with a new buildpack release
  - Restage applications

# Deploying Image to Container

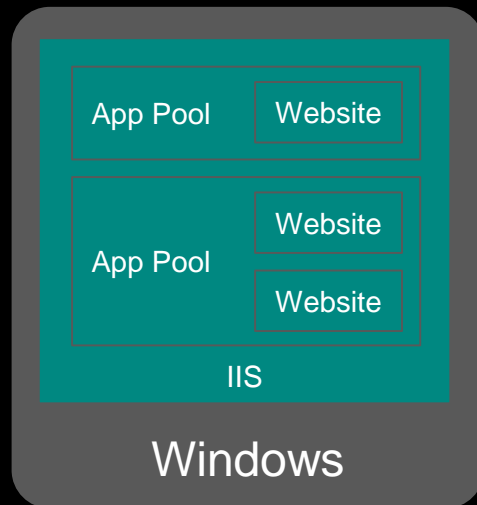


# Why Containers

- Containers are OS level virtualization (i.e. process isolation)
- They are small and allow for much higher packing density
- They are easy to move around and to replicate
- They do not have any redundant or unnecessary operating system elements; they don't need the care and feeding of a large OS stack.
- They are lightweight and have fast startup times,
- Well suited for building hyper-scale, highly resilient infrastructure
- Typical container image is 10s of MB
- Containers start in msec

# Windows Cells on Cloud Foundry

## Traditional Windows Architecture

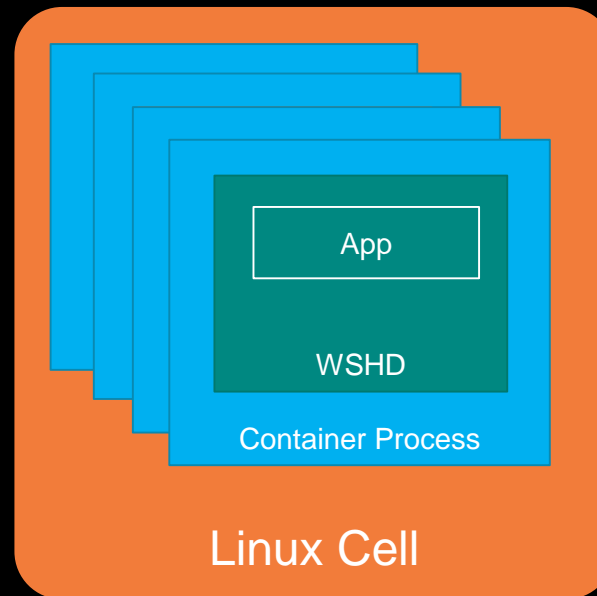


## Cloud Foundry Architecture



# Linux Cells on Cloud Foundry

## Cloud Foundry Architecture



# Container Environment Variables

- Used to communicate apps environment/config to deployed container
  - VCAP\_APPLICATION
    - Application attributes – version, instance index, limits, URLs, etc.
  - VCAP\_SERVICES
    - Bound services – name, label, credentials, etc.
  - CF\_INSTANCE\_\*
    - CF\_INSTANCE\_ADDR, CF\_INSTANCE\_INDEX, etc.

# VCAP\_APPLICATION

```
"VCAP_APPLICATION": {  
  "application_id": "95bb5b8e-3d35-4753-86ee-2d9d505aec7c",  
  "application_name": "fortuneService",  
  "application_uris": [  
    "fortuneservice-glottologic-neigh.apps.testcloud.com"  
  ],  
  "application_version": "40933f4c-75c5-4c61-b369-018febb0a347",  
  "cf_api": "https://api.system.testcloud.com",  
  "limits": {  
    "disk": 1024,  
    "fds": 16384,  
    "mem": 512  
  },  
  "name": "fortuneService",  
  "space_id": "86111584-e059-4eb0-b2e6-c89aa260453c",  
  "space_name": "test",  
  "uris": [  
    "fortuneservice-glottologic-neigh.apps.testcloud.com"  
  ],  
  "users": null,  
  "version": "40933f4c-75c5-4c61-b369-018febb0a347"  
}
```

# Lab1 – Pushing .NET Application to Cloud Foundry

- Push pre-built ASP.NET 4.x MVC application to Cloud Foundry
  - Makes use of Steeltoe Cloud Foundry Configuration provider
  - Steeltoe Configuration provider is used to parse `VCAP_APPLICATION` and add it to applications configuration information
    - Look at this in more detail in upcoming labs
  - Illustrates using Steeltoe components in ASP.NET 4.x applications
    - Several samples like this on github: <https://github.com/SteeltoeOSS/Samples>



# Using Services on Cloud Foundry

MANAGED, USER-PROVIDED, SERVICE BROKERS, INSTANCE CREATION, APPLICATION BINDING, ENVIRONMENT VARIABLES, VCAP\_APPLICATION

# What is a Service?

- Allows resources to be easily provisioned on-demand
- Typically an external “component” necessary for applications
  - Database, cache, message queue, microservice, etc.
- Can be a persistent, stateful layer



# Types of Services

- Managed - Fully integrated, with fully lifecycle management
- User-Provided – Created and managed external to the platform



# Managed Services

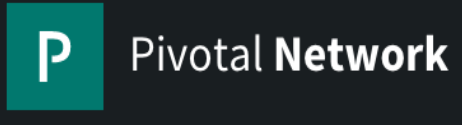
- Integrated with Cloud Foundry
  - Implements a required API for which the cloud controller is the client
- **Service Broker** implements the required API
  - Advertise a catalog of service offerings and service plans
  - Handle calls from the Cloud Controller
  - Fetch catalog
  - Create service instances
  - Bind applications to service instances
  - Unbind applications from service instances
  - Delete service instances

# User Provided Services











- Service instances managed outside of Cloud Foundry
- Behave like other service instances once created
- Familiar CLI commands ('create-service') provide service instance configuration

EXAMPLE: AN ORACLE DATABASE MANAGED OUTSIDE OF, AND UNKNOWN TO CLOUD FOUNDRY

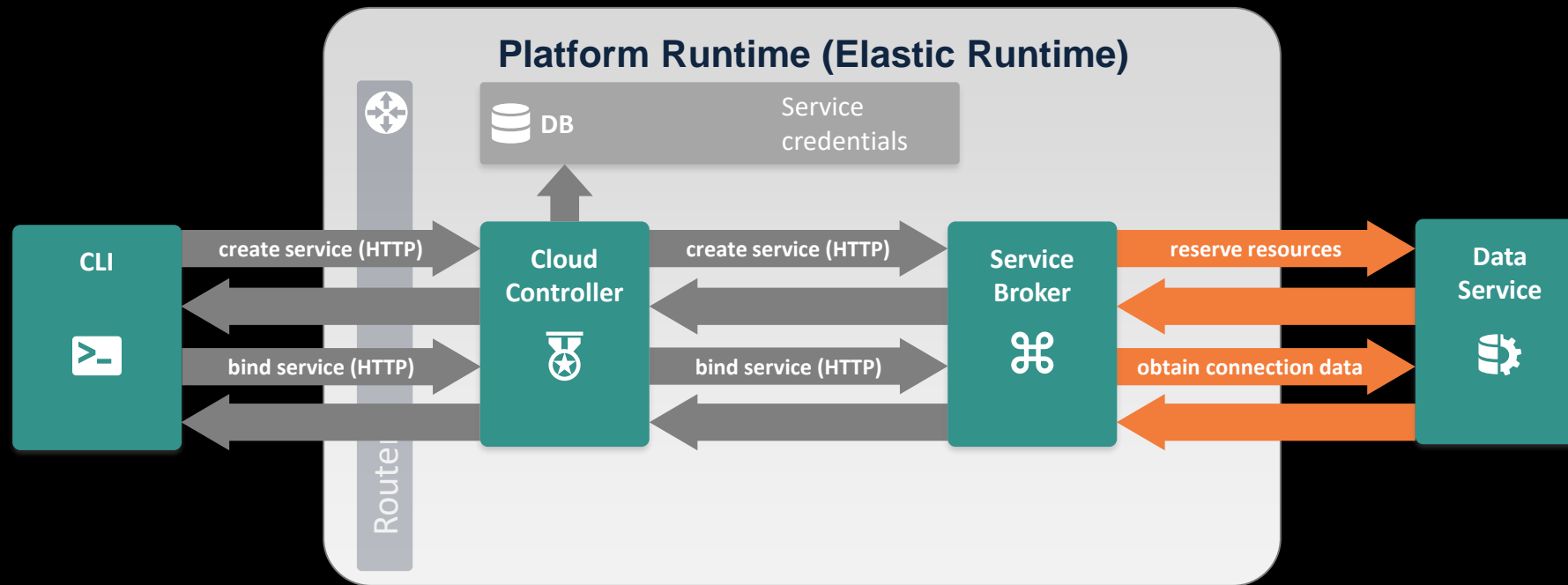
# Examples of Managed Services



<https://network.pivotal.io/>

 <b>Pivotal Cloud Foundry Service Broker for AWS</b>	 <b>GemFire for PCF</b>
 <b>Push Notification for PCF</b>	 <b>MySQL for PCF</b>
 <b>Redis for PCF</b>	 <b>Pivotal Tracker for PCF</b>
 <b>Session State Caching Powered by GemFire for PCF</b>	 <b>RabbitMQ for PCF</b>
 <b>Spring Cloud Services for PCF</b>	 <b>Single Sign-On for PCF</b>

# Creating and Binding Services



# Container Environment Variables

- Used to communicate apps environment/config to deployed container
  - VCAP\_APPLICATION
    - Application attributes – version, instance index, limits, URLs, etc.
  - VCAP\_SERVICES
    - Bound services – name, label, credentials, etc.
  - CF\_INSTANCE\_\*
    - CF\_INSTANCE\_ADDR, CF\_INSTANCE\_INDEX, etc.



# VCAP\_SERVICES

```
"VCAP_SERVICES": {  
  "p-identity": [  
    {  
      "credentials": {  
        "client_id": "e3ca311d-999b-4e4f-b056-b50138cfff9f",  
        "client_secret": "a995365e-d7b7-4727-95b8-463df2842f64",  
        "auth_domain": "https://sso1.login.run.haas-76.pez.pivotal.io"  
      },  
      "syslog_drain_url": null,  
      "label": "p-identity",  
      "provider": null,  
      "plan": "sso1",  
      "name": "sso",  
      "tags": []  
    }  
  ]  
}
```

## Lab2 – Creating and Binding Services

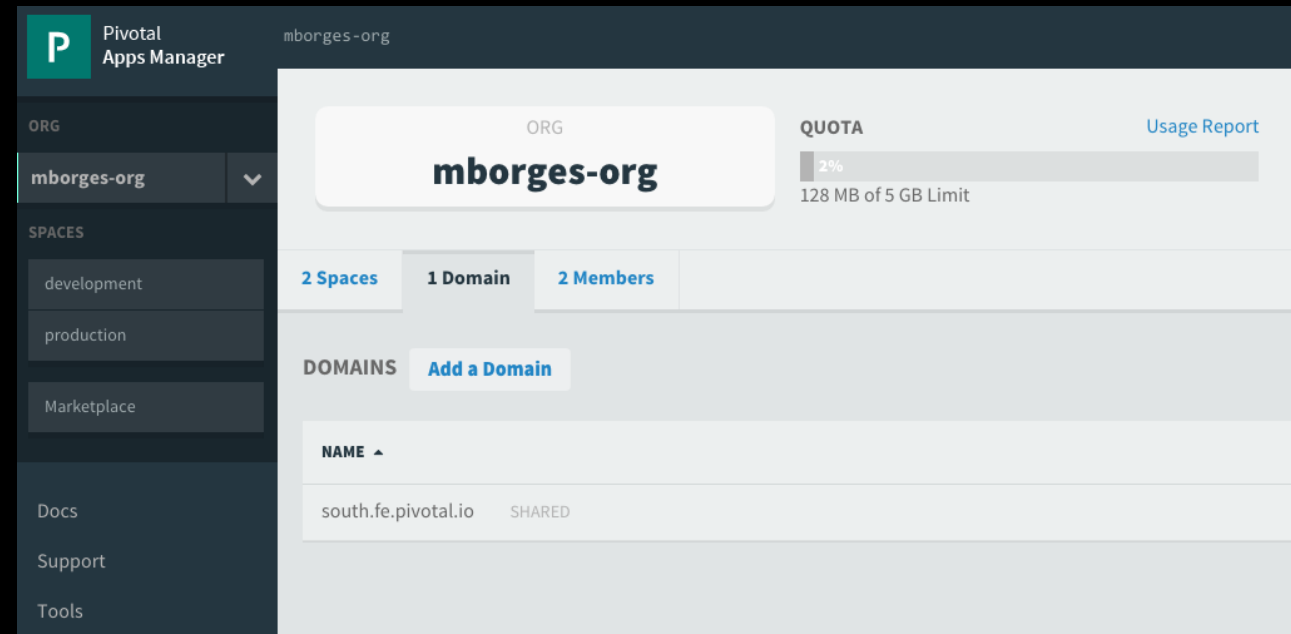
- Use same pre-built ASP.NET 4.x MVC application
  - Illustrates how Steeltoe Configuration provider can be used to parse `VCAP_SERVICES` and add it to applications configuration information
    - Look at how Steeltoe Connectors make use of this provider in future labs

# Configuring Application Routes and Instances on Cloud Foundry

DOMAINS, DNS, ROUTES, SCALING VIA CLI, SCALING VIA APP  
MANAGER

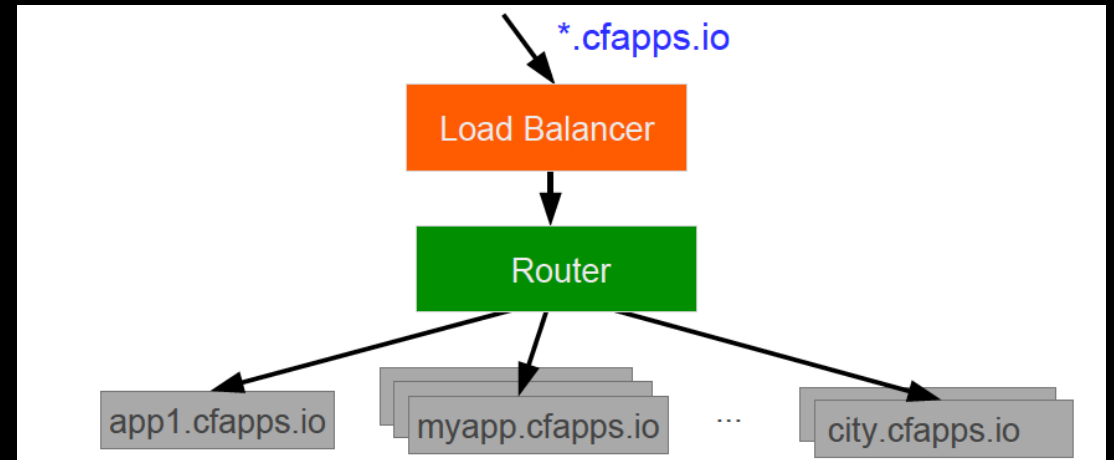
# Domains

- Each Cloud Foundry installation has a default app domain
- Domains provide a namespace from which to create routes
- Requests for any routes created from the domain will be routed to Cloud Foundry.
- Domains can be shared or private in regards to PCF organizations



# Domains – Behind the Scenes

- A wildcard entry (\*) is added to the DNS for the app domain
- That DNS entry points to a load balancer (or Cloud Foundry's HA Proxy), which points to the Cloud Foundry Router
- The Router uses the subdomain to map to application instance(s)



# Routes

- HTTP requests are routed to apps pushed by associating a URL with an application, known as route
- Many app instances can be mapped to a single route resulting in load balanced requests
- Routes belong to a space
- Application can have multiple routes

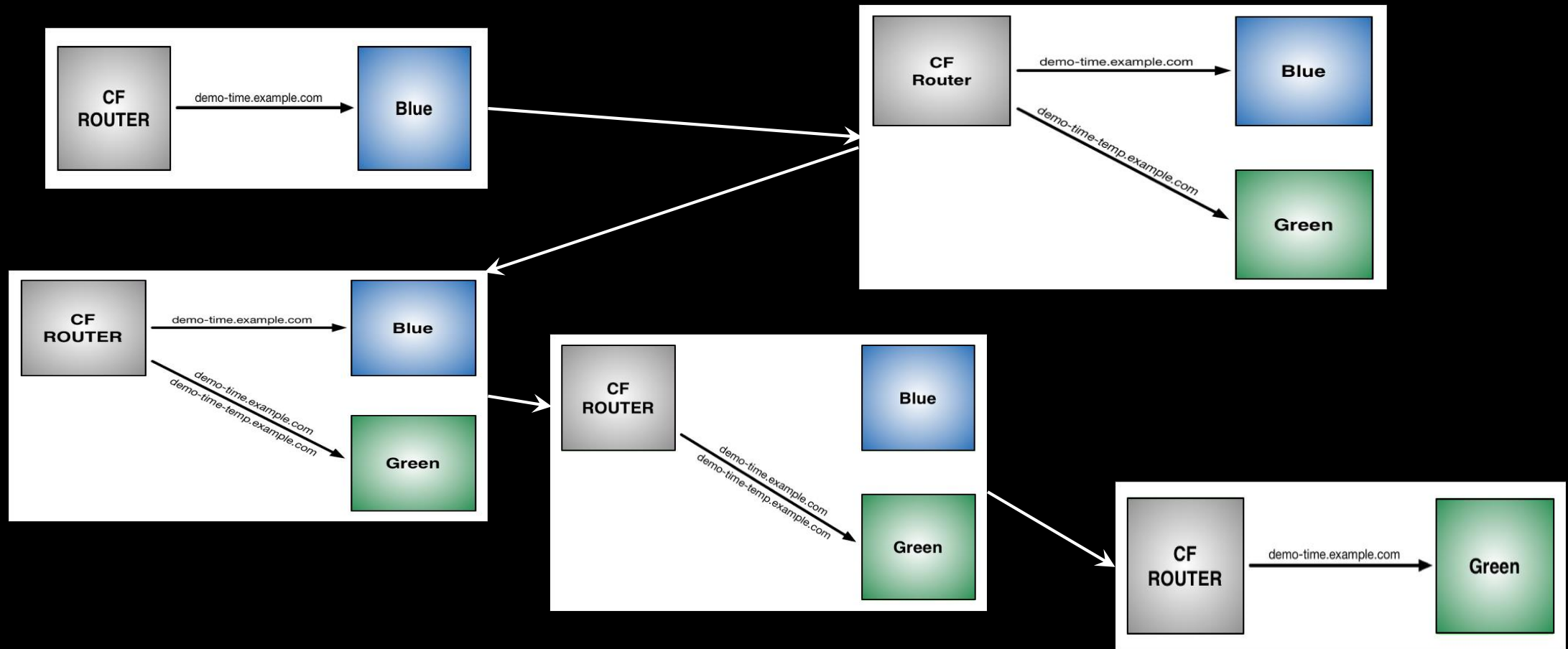
The screenshot displays the Pivotal Apps Manager interface for the application 'pcf-scale-prod' within the 'development' space of the 'mborges-org' organization. The interface includes a sidebar with navigation options like 'ORG', 'SPACES', 'Docs', 'Support', and 'Tools'. The main content area is divided into several sections:

- APP Overview:** Shows the app name 'pcf-scale-prod', a 'last push' timestamp of '12/22/15 @ 23:56 UTC', and a link to the app's URL.
- CONFIGURATION:** Includes a 'Scale App' button and settings for 'Instances' (1), 'Memory Limit' (128 MB), and 'Disk Limit' (1 GB).
- STATUS:** A table showing the app's current status and resource usage.
- ABOUT:** Provides details about the buildpack (Ruby), start command, and stack (cflinuxfs2).
- Routes:** A section with a 'Map a Route' button and a list of existing routes.

#	STATUS	CPU	MEMORY	DISK	UPTIME
0	Running	0%	88.3 MB	84.2 MB	1 d 2 hr 31 min

Route	Action
<a href="https://pcf-scale-prod.south.fe.pivotal.io">https://pcf-scale-prod.south.fe.pivotal.io</a>	Unmap
<a href="https://pcf-scale-v1_2.south.fe.pivotal.io">https://pcf-scale-v1_2.south.fe.pivotal.io</a>	Unmap

# Blue-Green Deployments



<https://docs.pivotal.io/pivotalcf/1-8/devguide/deploy-apps/blue-green.html>

# Scaling

- Can be done via CLI
  - At deployment time (via manifest.yml or as a modifier to cf push)
  - During run time without interrupting operations (via cf scale --instances 10)
- Can also be done via Apps Manager
- Container image started on other available cells

The screenshot displays the Pivotal Apps Manager interface for the application 'pcf-scale-prod'. The left sidebar shows the navigation menu with 'mborges-org' selected under 'ORG' and 'development' selected under 'SPACES'. The main content area is divided into several sections:

- APP:** A circular logo for 'pcf-scale-prod' with a 'last push' timestamp of 12/22/15 @ 23:56 UTC and a link to the application's GitHub repository.
- CONFIGURATION:** A section with three dropdown menus: 'Instances' (set to 1), 'Memory Limit' (set to 128 MB), and 'Disk Limit' (set to 1 GB). A 'Scale App' button is located to the right of these settings.
- STATUS:** A table showing the current status of the application.
- ABOUT:** A section providing details about the application's buildpack, start command, and stack.
- Events:** A section with tabs for 'Events', 'Services', 'Env Variables', 'Routes', and 'Logs'. A 'Delete App' button is located to the right of the 'Logs' tab.
- RECENT EVENTS:** A section showing a recent event: 'updated app' by 'mborges@pivotal.io' on 12/24/2015 at 02:32 AM UTC.

#	STATUS	CPU	MEMORY	DISK	UPTIME
0	Running	0%	100 MB	84.2 MB	13 d 23 hr 58 min



## Lab3 – Creating Routes and Scaling

- Use same pre-built ASP.NET 4.x MVC application

# Monitoring Applications on Cloud Foundry

LOGGING, TAILING LOGS, HEALTH, EVENTS, CLI

# Logs

- Cloud Foundry aggregates an applications logs
  - Application logs should be written to STDOUT /STDERR
- Use the CLI to view an applications logs
  - ``cf logs APP_NAME`` – allows you to tail and applications logs
  - ``cf logs APP_NAME -recent`` – allows you to view recent logs

```
[> cf logs pcf-scale-prod --recent
Connected, dumping recent logs for app pcf-scale-prod in org mborges-org / space development as mborges@pivotal.io...

2015-12-23T20:32:08.97-0600 [API/0]      OUT Updated app with guid b02f5b0c-1e9b-495f-80c8-540f4239795e ({"route"=>"4d5d20e4-e1c3-4cda-ae82-150ed3564d23"})
2015-12-23T20:41:02.81-0600 [RTR/0]      OUT pcf-scale-v1_2.south.fe.pivotal.io - [24/12/2015:02:41:02 +0000] "GET / HTTP/1.1" 200 5355 "https://apps.south.fe.pivotal.io/organizations/691dc08d-91bd-4bf8-80cf-1e28f158c00a/spaces/de43df54-ad24-4dcb-beee-60ba842e7f46/applications/b02f5b0c-1e9b-495f-80c8-540f4239795e" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/47.0.2526.80 Safari/537.36" 192.168.5.10:41871 x_forwarded_for:"192.168.5.1" vcap_request_id:ba548a80-dc76-4cdb-59bc-4b072e9dcecb response_time:0.100964736 app_id:b02f5b0c-1e9b-495f-80c8-540f4239795e
2015-12-23T20:41:02.81-0600 [App/0]      ERR 192.168.5.1, 192.168.5.10 - - [24/Dec/2015 02:41:02] "GET / HTTP/1.1" 200 5355 0.0465
2015-12-23T20:41:02.91-0600 [App/0]      ERR 192.168.5.1, 192.168.5.10 - - [24/Dec/2015 02:41:02] "GET /css/bootstrap.min.css HTTP/1.1" 200 103314 0.0207
2015-12-23T20:41:02.91-0600 [RTR/0]      OUT pcf-scale-v1_2.south.fe.pivotal.io - [24/12/2015:02:41:02 +0000] "GET /css/bootstrap.min.css HTTP/1.1" 200 103314 "https://pcf-scale-v1_2.south.fe.pivotal.io/" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/47.0.2526.80 Safari/537.36" 192.168.5.10:41872 x_forwarded_for:"192.168.5.1" vcap_request_id:65a07c84-2fd7-4bbc-4628-928a7b1839f3 response_time:0.031567972 app_id:b02f5b0c-1e9b-495f-80c8-540f4239795e
2015-12-23T20:41:02.96-0600 [App/0]      ERR 192.168.5.1, 192.168.5.10 - - [24/Dec/2015 02:41:02] "GET /js/bootstrap.min.js HTTP/1.1" 200 31596 0.0139
2015-12-23T20:41:02.96-0600 [RTR/0]      OUT pcf-scale-v1_2.south.fe.pivotal.io - [24/12/2015:02:41:02 +0000] "GET /js/bootstrap.min.js HTTP/1.1" 200 31596 "https://pcf-scale-v1_2.south.fe.pivotal.io/" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/47.0.2526.80 Safari/537.36" 192.168.5.10:41874 x_forwarded_for:"192.168.5.1" vcap_request_id:129e0f87-4b70-45b1-54cb-300c197485f5 response_time:0.027457693 app_id:b02f5b0c-1e9b-495f-80c8-540f4239795e
```

# Health

- Cloud Foundry proactively monitors health of application containers
  - Restarts them if they fail
- Use the CLI to view an applications health & status
  - ``cf app APP_NAME`` - allows you view health status of an application

```
-> cf app pcf-scale-prod
Showing health and status for app pcf-scale-prod in org mborges-org / space development as mborges@pivotal.io...
OK

requested state: started
instances: 1/1
usage: 128M x 1 instances
urls: pcf-scale-prod.south.fe.pivotal.io, pcf-scale-v1_2.south.fe.pivotal.io
last uploaded: Tue Dec 22 23:56:50 UTC 2015
stack: cflinuxfs2
buildpack: Ruby
```

	state	since	cpu	memory	disk	details
#0	running	2015-12-22 06:02:59 PM	0.1%	88.3M of 128M	84.2M of 1G	

# Application Events

- Cloud Foundry records all changes to an application as events
  - Container state changes
  - Configuration changes
- Use the CLI to view an applications events
  - ``cf events APP_NAME`` – allows you view recent events of an application

```
[-> cf events pcf-scale-prod
Getting events for app pcf-scale-prod in org mborges-org / space development as mborges@pivotal.io...
```

time	event	actor	description
2015-12-23T20:32:08.00-0600	audit.app.update	mborges@pivotal.io	
2015-12-23T20:32:08.00-0600	audit.app.map-route	mborges@pivotal.io	
2015-12-22T18:02:56.00-0600	audit.app.update	mborges@pivotal.io	state: STARTED
2015-12-22T18:02:56.00-0600	audit.app.update	mborges@pivotal.io	state: STOPPED
2015-12-22T17:56:55.00-0600	audit.app.update	mborges@pivotal.io	state: STARTED
2015-12-22T17:56:45.00-0600	audit.app.update	mborges@pivotal.io	
2015-12-22T17:56:45.00-0600	audit.app.map-route	mborges@pivotal.io	
2015-12-22T17:56:44.00-0600	audit.app.create	mborges@pivotal.io	instances: 1, memory: 128, state: STOPPED, environment_json: PRIVATE DATA HIDDEN

## Lab4 – Monitoring Applications

- Use same pre-built ASP.NET 4.x MVC application

# Cloud Native Application Development

CLOUD NATIVE, TWELVE FACTOR, MICROSERVICES

# Cloud Native Applications

Cloud Native is not about where, but how you build and run your app!

- Microservices Architecture
- Twelve-Factor Methodology
- Containers
- Continuous Delivery
- Shift from Silo IT to DevOps



# Twelve Factor Applications – Platform Contract

Architectural and development best practices – <http://12factor.net>

<b>I. Codebase</b> One codebase tracked in SCM, many deploys	<b>II. Dependencies</b> Explicitly declare and isolate dependencies	<b>III. Configuration</b> Store config in the environment
<b>IV. Backing Services</b> Treat backing services as attached resources	<b>V. Build, Release, Run</b> Strictly separate build and run stages	<b>VI. Processes</b> Execute app as stateless processes
<b>VII. Port binding</b> Export services via port binding	<b>VIII. Concurrency</b> Scale out via the process model	<b>IX. Disposability</b> Maximize robustness with fast startup and graceful shutdown
<b>X. Dev/prod parity</b> Keep dev, staging, prod as similar as possible	<b>XI. Logs</b> Treat logs as event streams	<b>XII. Admin processes</b> Run admin / mgmt tasks as one-off processes

# Microservice Definition

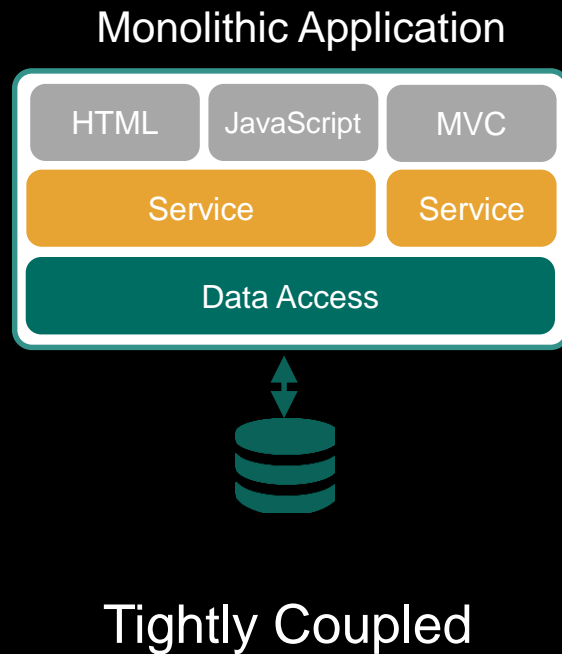
If every service has to be updated in concert,  
it's not loosely coupled!

**“Loosely coupled service oriented  
architecture with bounded contexts”**

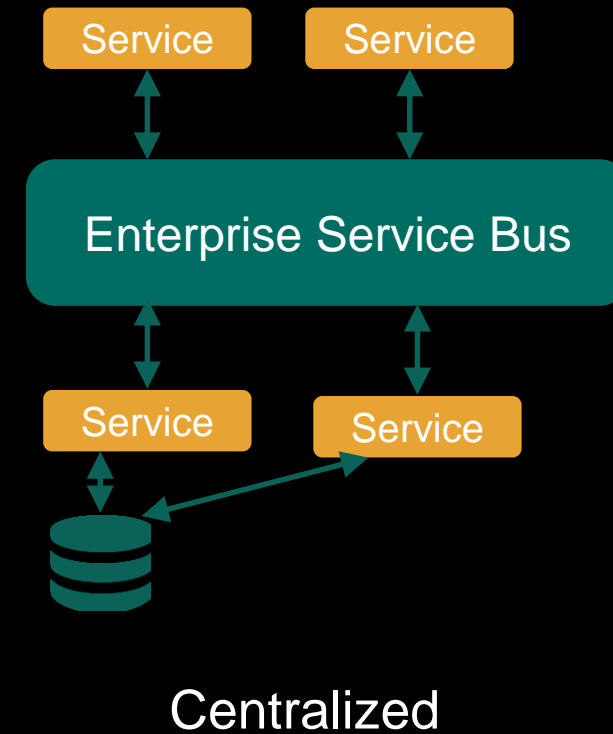
If you have to know about surrounding  
services you don't have a bounded context.

- Adrian Cockcroft

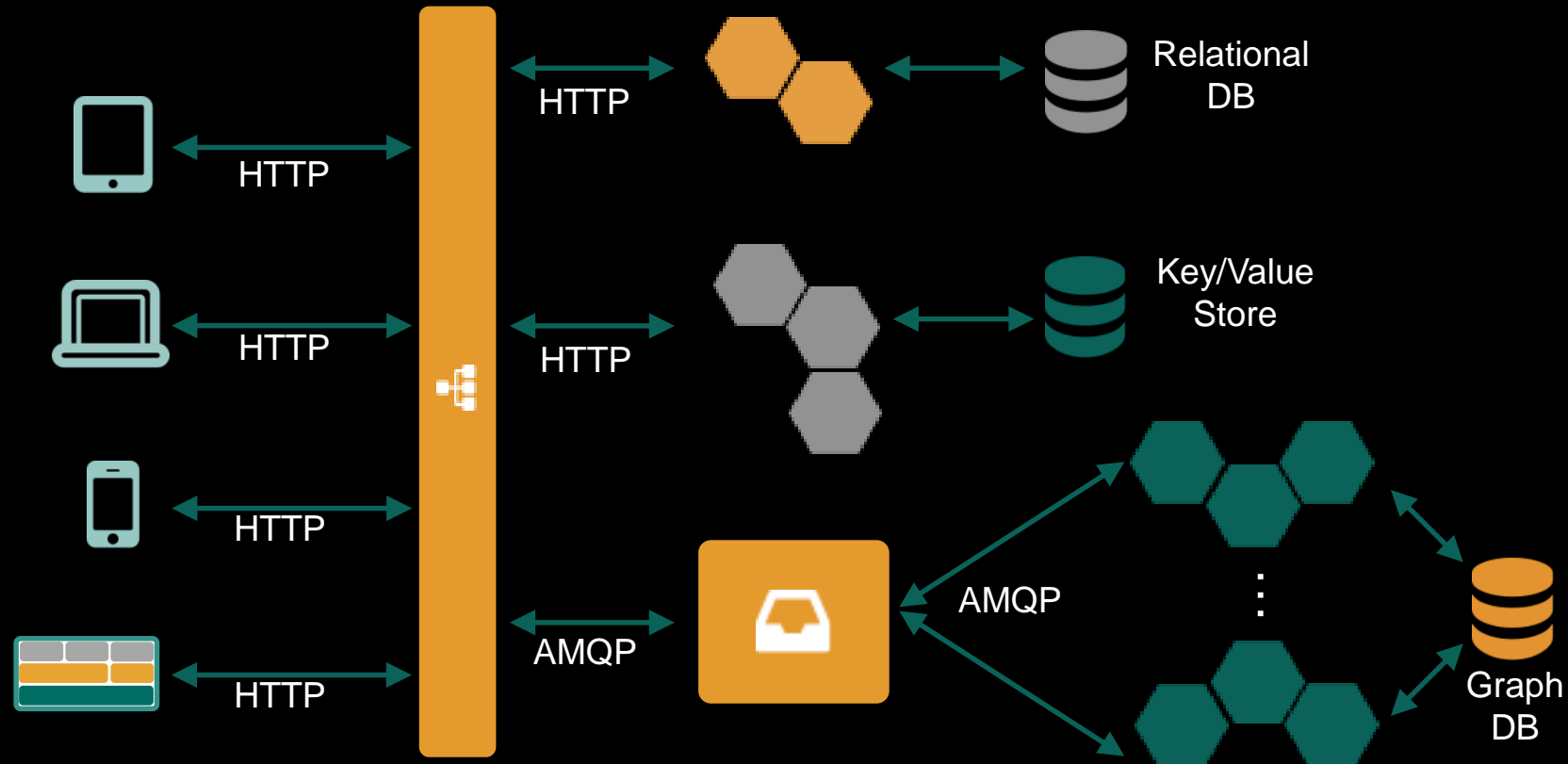
# Microservices are NOT



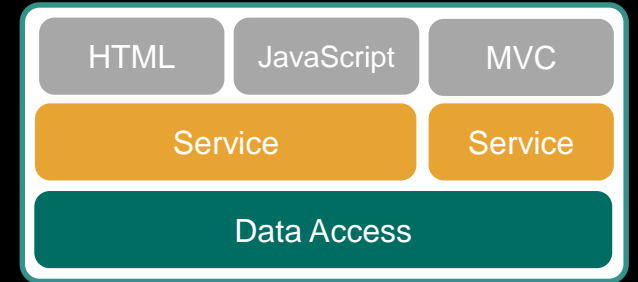
OR



# Microservices Architecture



# Monolith Challenges



- Traditional monolithic design patterns are not appropriate for the cloud.
- Monoliths couple change cycles together.
- Monoliths services can't be scaled independently.
- Difficult coordination: too many developers in one code base.
- Developers struggle to understand a large codebase.
- Long term commitment to the tech stack.



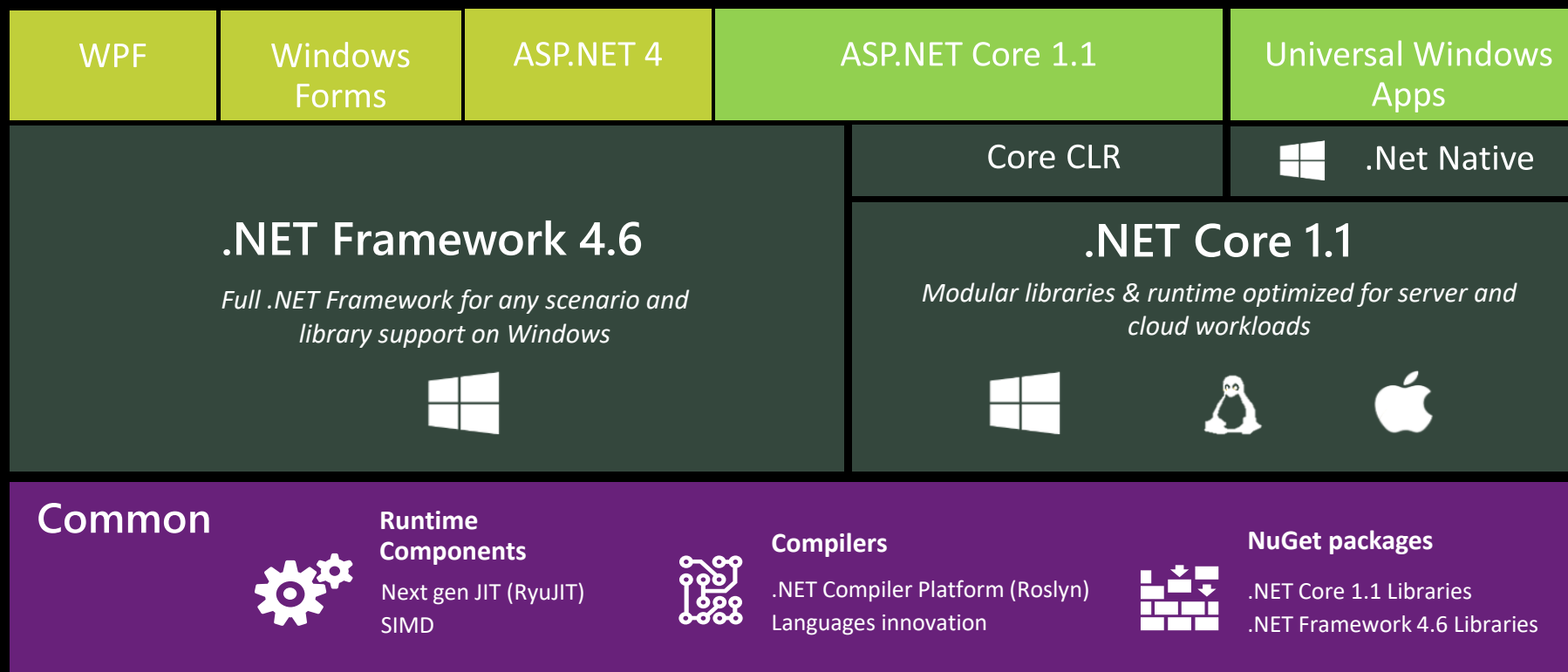
## Microservice Benefits

- Change cycles are decoupled: Enabling frequent deploys
- Allow for efficient and independent scaling
- Developers learn a smaller codebase faster
- Better coordination and scaling of development: Fewer developers in each code base
- Eliminate long-term commitment to technical stack

# Cloud Native .NET Development on Cloud Foundry

.NET, .NET CORE, ASP.NET, ASP.NET CORE, MICROSERVICES,  
CONTAINERS, CLOUD FOUNDRY

# .NET Today





# .NET Core

- Runs cross platform
  - Runtimes, Libraries and Compilers for Windows, Linux, OSX
  - .NET Core tooling for Windows, Linux, OSX
    - Command Line Interface (CLI) – ‘dotnet’
    - ‘project.json’ file is the new ‘.csproj’ (soon to be old)
    - Cross platform Code editor – Visual Studio Code
    - Visual Studio 2015/2017 - IDE (Windows and Mac only)
- Fully open source
  - Runtime (i.e. CoreCLR) - <https://github.com/dotnet/coreclr>
  - Framework Libraries (i.e. CoreFx) - <https://github.com/dotnet/corefx>
  - Compilers (i.e. Roslyn) - <https://github.com/dotnet/roslyn>
- Installers for Windows, Linux, OSX
  - Binaries = Runtime & Libraries only
  - SDK = Development Tools + Runtime & Libraries

# .NET Core

- Modular – built on NuGet packaging system
  - Packaged and distributed as lots and lots of NuGets
    - CoreCLR, Libraries & Compilers
  - Packages include everything needed to run
    - Even native code dependencies
  - Enables 'a la carte' .NET development
- Application types
  - .NET Console applications
- Missing various .NET Framework classes
  - .NET Core Framework libraries are an API Subset/Superset
  - NET Standard Library – effort to standardize BCL for cross platform
    - Current -> .NET Standard 1.6 -> .NET Core 1.0 & .NET 4.6.?
    - Future -> .NET Standard 2.0 -> .NET Core vNext & .NET 4.6.1

# .NET Core

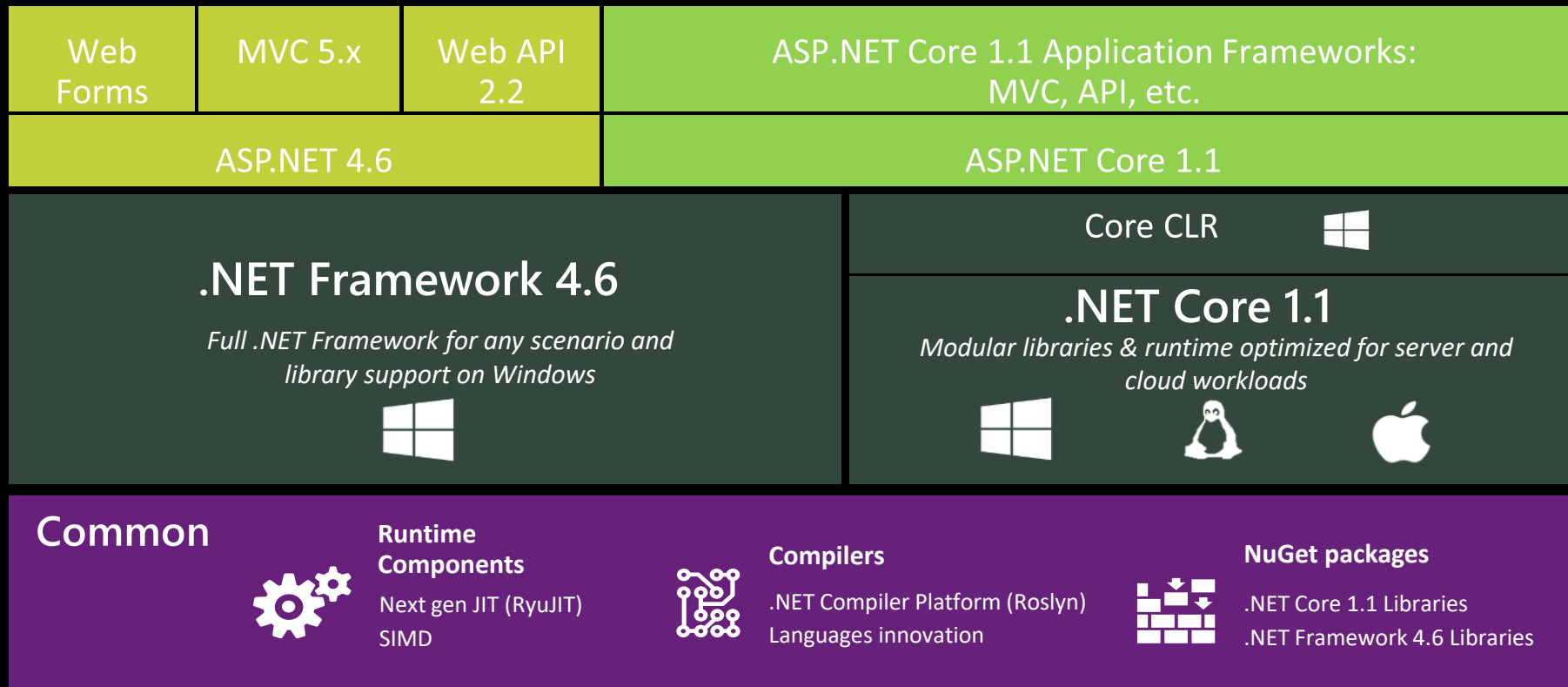
- .NET Core tooling

- `'dotnet <command> <args>'` – Command Line Interface (CLI)
  - ``new`` - create a new project
  - ``restore`` – restore dependencies declared in ``project.json``
  - ``build`` – compile code to assembly – debug or release
  - ``run`` – compile (if necessary) and run the assembly (i.e. launcher)
  - ``publish`` – package app and all dependencies for deployment
- ``project.json`` - it is the project ``recipe`` (i.e. defines target frameworks & runtimes, compile info and NuGet dependencies, etc.)
  - ``frameworks`` - section specifies the target frameworks the app/project supports (e.g. .NET Framework 4.5.2, .NET Standard Library 1.6, UWP, etc.)
    - Specified via a TFM (Target Framework Moniker) – e.g. (net452, netcoreapp1.0, netstandard1.6)
  - ``runtimes`` - section specifies the runtimes supported – used when publishing self-contained deployments
    - Specified via RIDs (Runtime Identifiers) e.g. (win7-x64, ubuntu.14.04-x64, osx.10.11-x64)
  - ``dependencies`` - section specifies package dependencies, can be specified per framework and globally
- ``global.json`` - optional, can specify the .NET Core SDK to use

# .NET Core

- Application deployment options (i.e. `dotnet publish`)
  - Self-contained: Application = .NET Core runtime + App dependencies + App code
  - Portable: Application = App dependencies + App code
- .NET Core != ASP.NET Core
- Lets look at some code!

# ASP.NET Core



# ASP.NET Core

- Runs cross framework
  - .NET Core – Windows, Linux & OSX
  - .NET Framework 4.x - Windows
- Fully open source - <https://github.com/aspnet>
- Modular
  - Packaged and distributed as lots of NuGets
    - Host, Server, Configuration, Dependency Injection, Session, Static Files, etc.
  - Separate from .NET Core or .NET Framework
  - Enables 'pay for play' ASP.NET development
- Use .NET Core CLI – '`dotnet <command> <args>`' (i.e. build, run, publish, etc)
- Deployment options when targeting .NET Core
  - Application can be fully self-contained
    - Application = .NET Core NuGets + ASP.NET Core NuGets + App dependencies + App code
  - Application can be portable
    - Application = .ASP.NET Core NuGets + App dependencies + App code

# ASP.NET Core

- Hosting – self hosting
  - It's just a console application
  - Kestrel – a cross platform web server
  - IIS – used as reverse-proxy with Kestrel
    - ACM – native IIS module used
  - WebListener – windows only web server (built on http.sys)
- Dependency Injection (DI) baked in
- Middleware handles request processing
  - Routing, Static Files, Session, Authentication, Authorization, etc.
- Configuration – separate application config from `web.config`
  - Multiple configuration sources (i.e. JSON, INI, XML files, Environment Variables, etc.)
- Startup class
  - Builds Configuration
  - Configures the Service Container (i.e. DI)
  - Configures the Middleware
- Logging

# ASP.NET Core

- Application frameworks
  - MVC – Integrated UI and Web API framework
    - Layered on top of ASP.NET Core
      - Uses DI, Logging, Self-hosting, etc.
    - Controllers, Actions, Views, Filters, Model binding
    - Razor Views & new Tag Helpers
  - WebSockets
  - SignalR – release coming
- Lets look at some code!



# .NET 4.x Support on Cloud Foundry

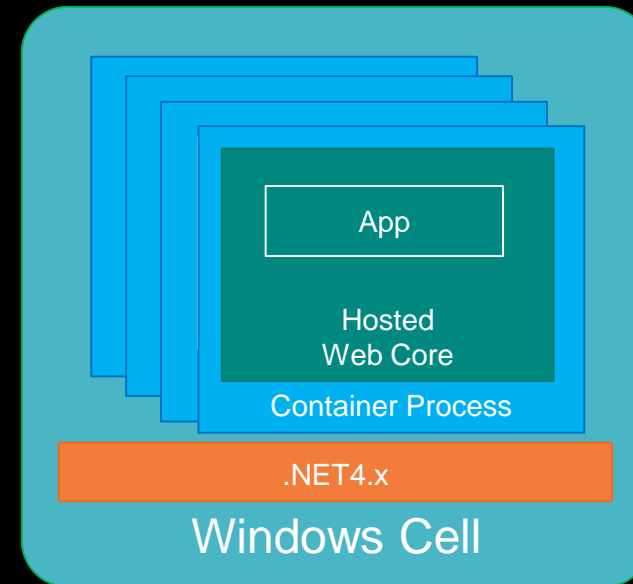
- Always push to Windows cell - `stack: windows2012R2`
  - Uses `binary` build pack (default)
- Application types
  - ASP.NET - MVC, WebForm, WebAPI, WCF
  - ASP.NET OWIN based apps
  - ASP.NET Core web apps
    - Targeting .NET Framework
  - .NET “Background processes”
    - Command line/Console apps

```
---
applications:
- name: env
  random-route: true
  health-check-type: none
  memory: 1G
  stack: windows2012R2
  env:
    MY_ENVIRONMENT: production
```

# .NET 4.x on Windows Cell

- Container image created using
  - Binary buildpack
- .NET 4.x shared by all container processes
- Resource isolation
  - Kernel job object
  - Disk quotas
- Namespace isolation
  - User accounts
  - Hosted Web Core

Windows Cell Architecture



# .NET Core Support on Cloud Foundry

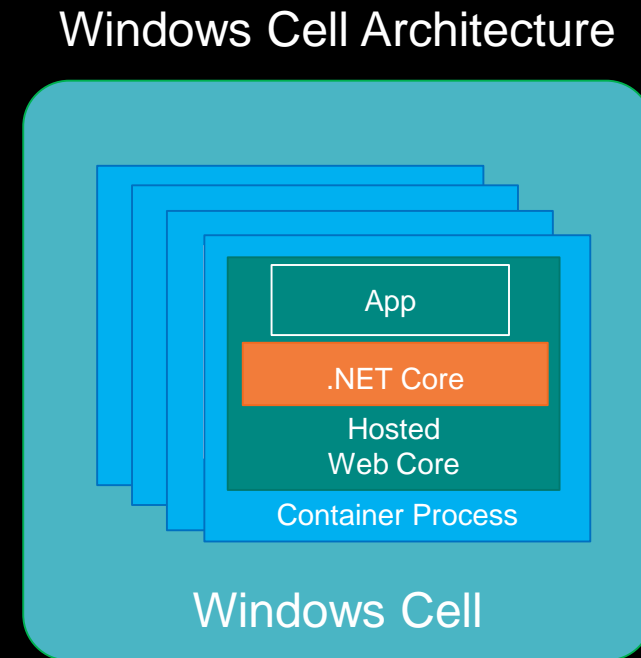
- Push to windows cell - `stack: windows2012R2`
  - Use `binary` build pack (default)
  - Publish & push self-contained application
- Push to Linux cell - `stack: cflinuxfs2` (manifest default)
  - Use .NET Core build pack
  - Publish & push either self-contained, portable, or source
- Application types
  - ASP.NET Core web apps
  - .NET Core “Background processes”
    - Command line apps/Console apps

```
applications:  
- name: env  
  random-route: true  
  health-check-type: none  
  memory: 512M  
  stack: windows2012R2  
  command: cmd /c .\env  
  env:  
    MY_ENVIRONMENT: production
```

```
applications:  
- name: env  
  random-route: true  
  memory: 512M  
  buildpack: dotnet_core_buildpack  
  command: ./env  
  env:  
    MY_ENVIRONMENT: production
```

# .NET Core on Windows Cell

- Isolation same as .NET 4.x on Windows cell
- Container image created using
  - .NET Core buildpack
- .NET Core not shared by all container processes
  - Different versions possible per container



# .NET Core Buildpack – Linux Cells

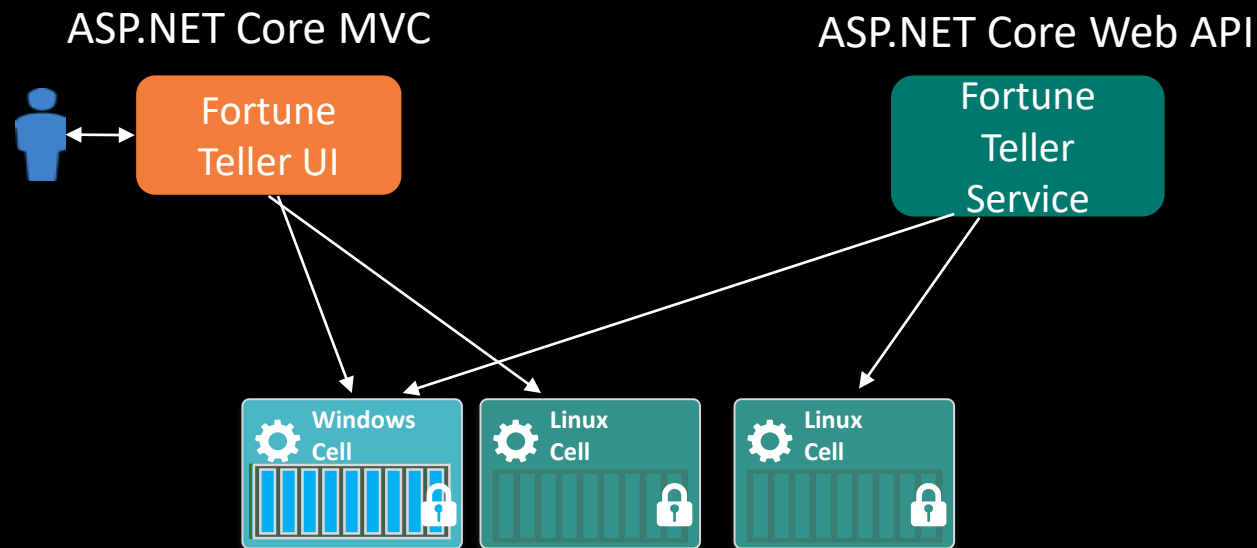
- Creates container images ready to run .NET Core applications on Linux cells
  - Supports running app from pushed source
    - Pushed source directory must contain `project.json`
  - Supports running app from pushed binaries
    - Pushed directory contains NO `project.json` and is a self-contained or portable app
- Pushing source
  - Installs .NET Core runtime – version specify via `global.json`, else build pack chooses
  - Restores application dependencies
  - Generates the command to run the application
- Pushing binaries
  - Portable applications
    - Installs .NET Core runtime – version specify via `global.json`, otherwise build pack chooses
    - Generates the command to run the application
  - Self-contained applications
    - Installs `libunwind.so`
    - You specify command to run

# Writing Twelve Factor ASP.NET Applications

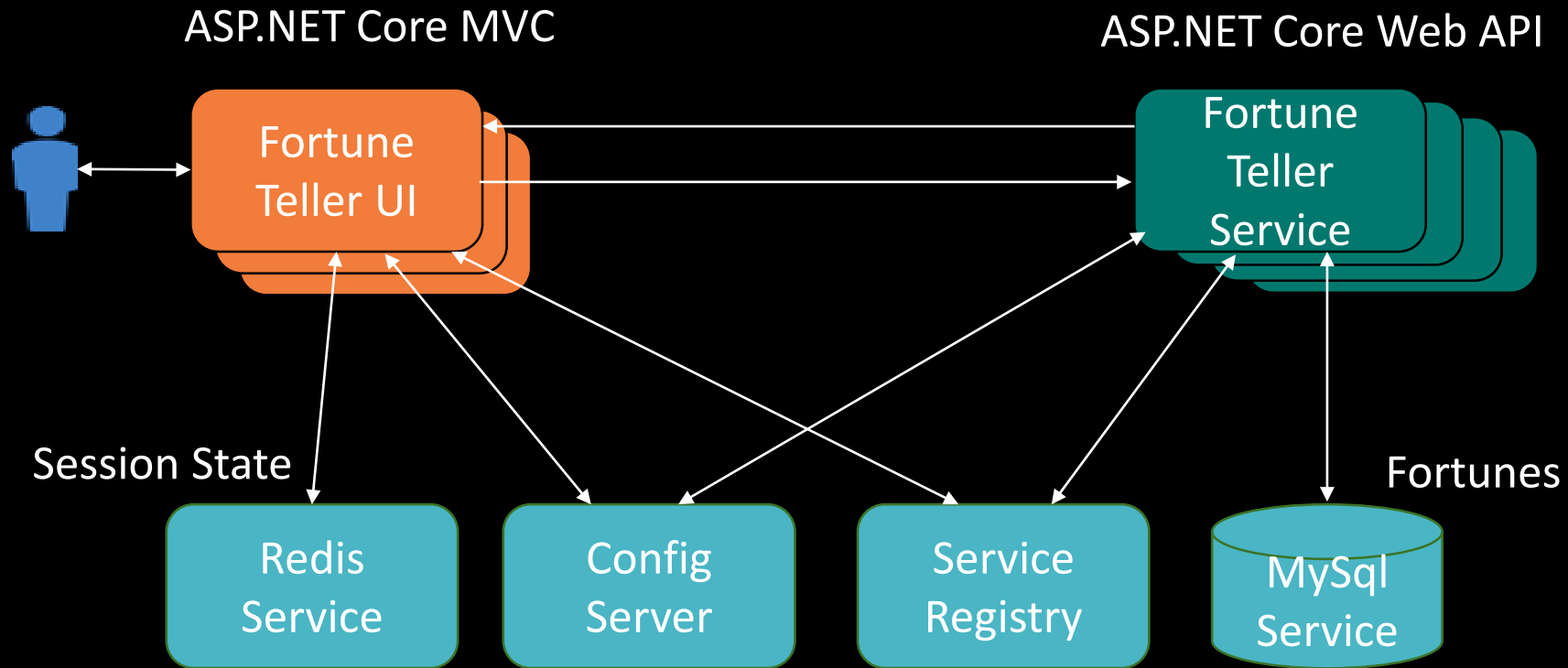
- Avoid in-process session state
- For ASP.NET override `MachineKey` in `web.config` and on ASP.NET Core avoid persisting keyring to filesystem
- On ASP.NET avoid environment specific configuration in `web.config`
- Avoid Integrated Windows Authentication
- Avoid the GAC
- Avoid custom IIS handlers
- Avoid anything that uses the Windows registry
- Avoid using local disk for storing application state
- Avoid using any Windows specific or disk based logging
- Avoid any 32-bit specific libraries or libraries that can't be bin deployable

# Lab5 – Running .NET Core Locally & On Cloud Foundry

- Become comfortable with restoring, running and publishing .NET Core applications
  - From command line
  - From within Visual Studio

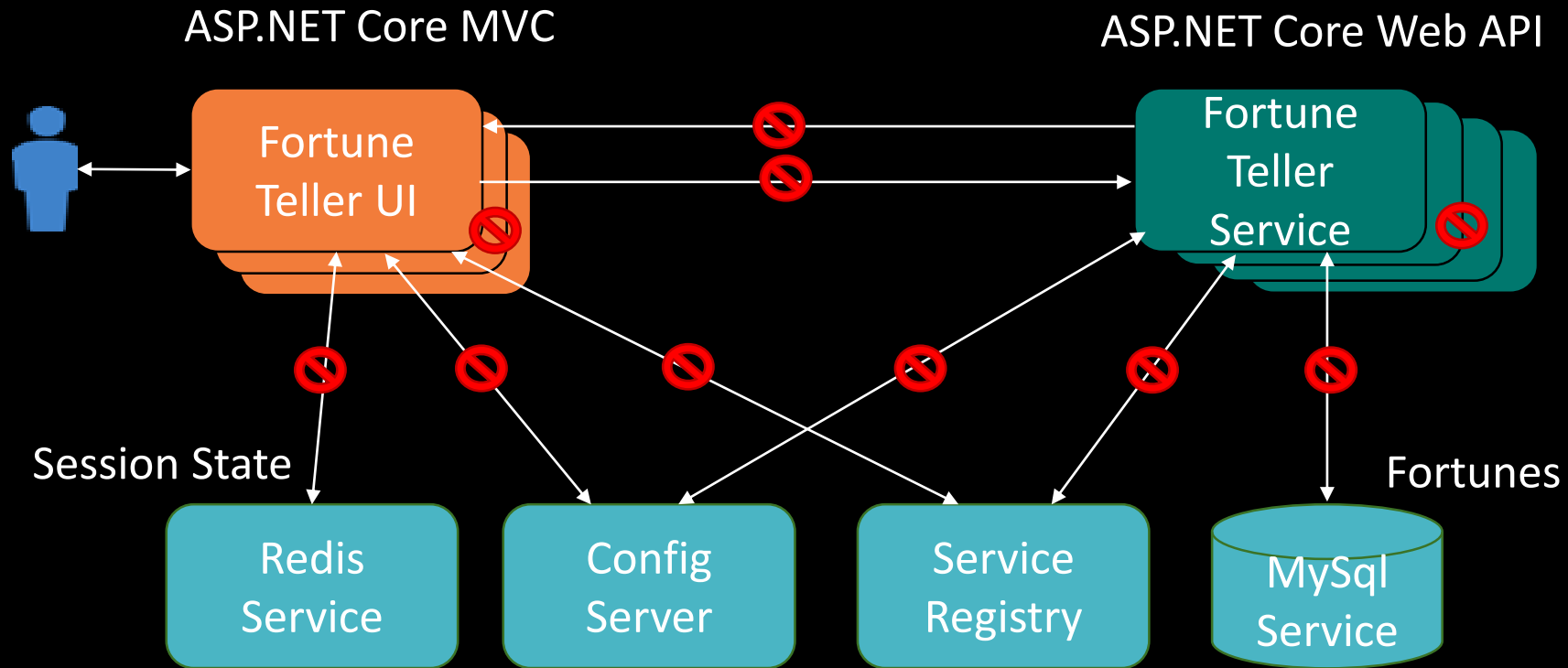


# Lab5 – Fortune Teller App – When done!





# Lab5 – Fortune Teller App - Current State



# ASP.NET Core Programming Fundamentals

HOST, SERVER STARTUP, DEPENDENCY INJECTION, SERVICES,  
MIDDLEWARE

# ASP.NET Core Startup

- Everything starts in `Program.cs`
  - Build a host using `WebApplicationBuilder`
    - Configure web server, listen address, etc.
    - Specify startup class
- Startup class
  - `Startup()` - constructor builds applications configuration
  - `ConfigureServices()` - configures the service collection (Dependency Injection)
  - `Configure()` - configures the middleware pipeline

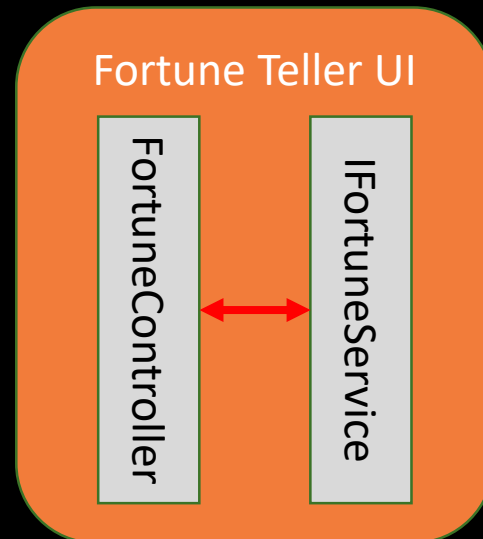
# ASP.NET Core Dependency Injection

- Basic, minimal feature set, but fundamental to operation of ASP.NET Core
  - Can be replaced with other DI systems (e.g. Autofac, etc.)
  - Supports constructor based injection (i.e. specify dependencies via arguments)
- `IServiceProvider` – is the container
  - Manages `services`
- Use `IServiceCollection` to add services at startup (i.e. inside `ConfigureServices()`)
  - Add framework services via `AddServiceName()` extension methods
    - e.g. `services.AddMvc()`, `services.AddSession()`, etc.
  - Add application services via `AddTransient()`, `AddSingleton()` & `AddScoped()`
    - e.g. `services.AddSingleton<IFortuneRepository, FortuneRepository>()`
- Lets look at the code!

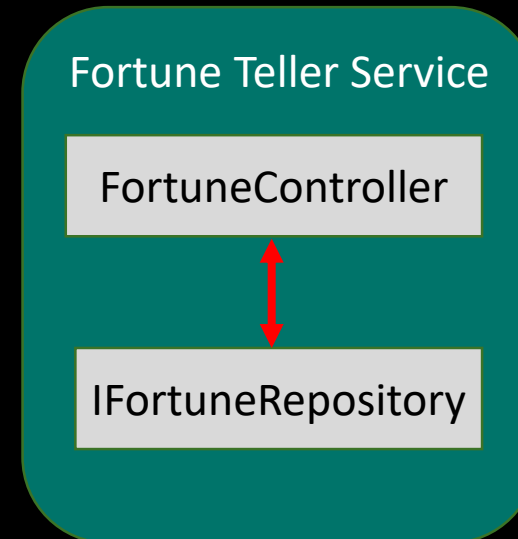
# Lab6 – Programing ASP.NET Core Dependency Injection

- Become comfortable with ASP.NET Core Dependency Injection
  - Use it to hook up application components

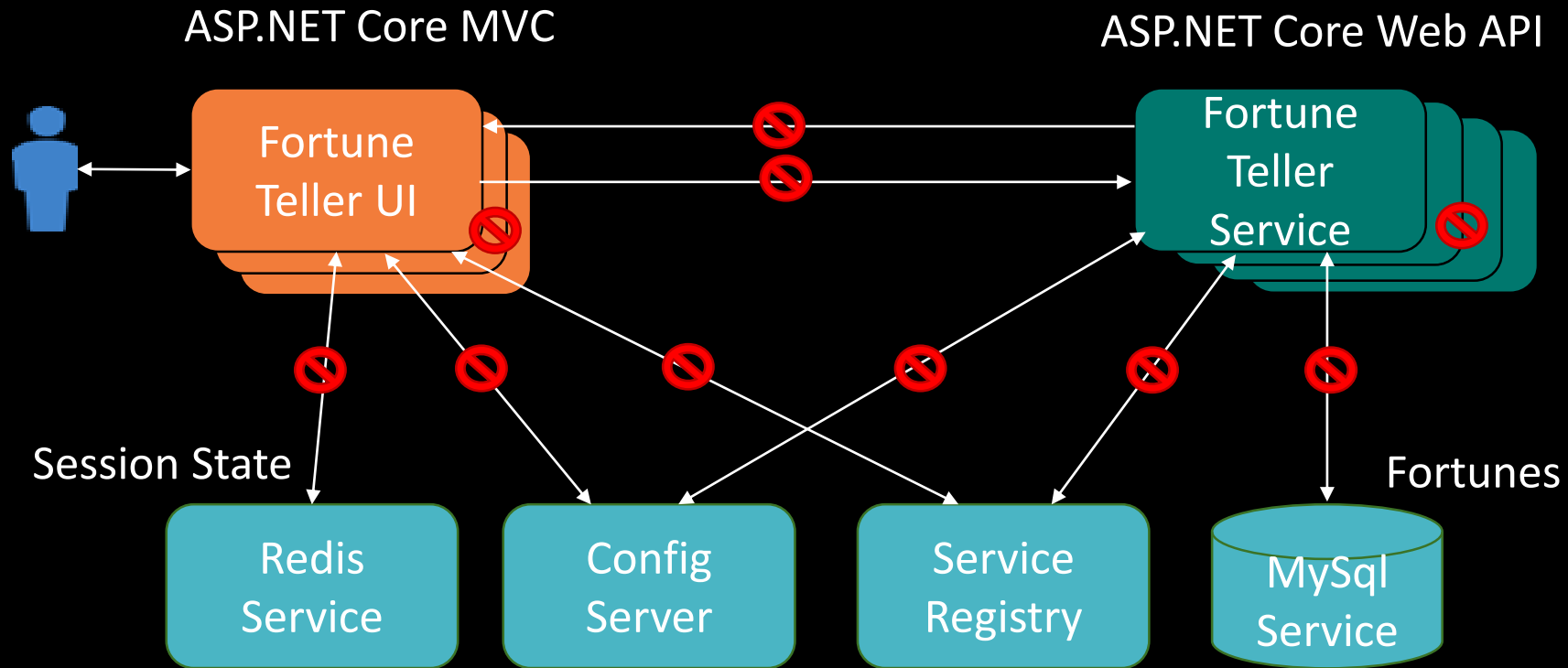
ASP.NET Core MVC



ASP.NET Core Web API



# Lab6 – Fortune Teller App – After Lab



# Cloud Native Applications and Spring Cloud Services

# Cloud Native Architectures

- Light Side of the Cloud
  - Scalability
  - High Availability
  - Velocity: Continuous Delivery
  - On-Demand Provisioning

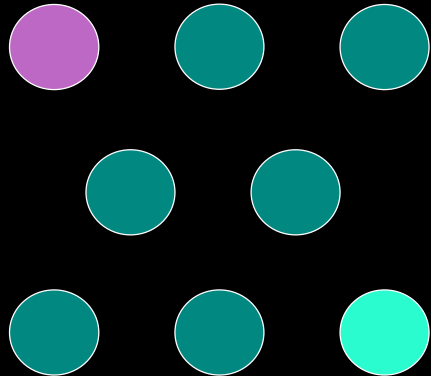




# Cloud Native Architectures

- Dark Side of the Cloud
  - Services: Finding them

Where are you?



Over here!



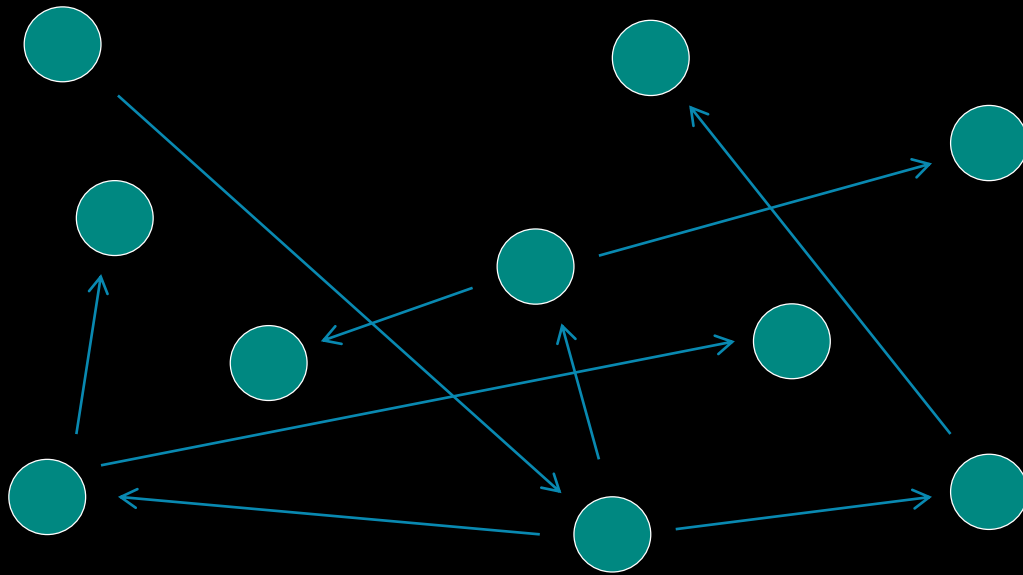
# Cloud Native Architectures

- Dark Side of the Cloud
  - Configuration: Managing Differences



# Cloud Native Architectures

- Dark Side of the Cloud
  - Failures: Following call graphs

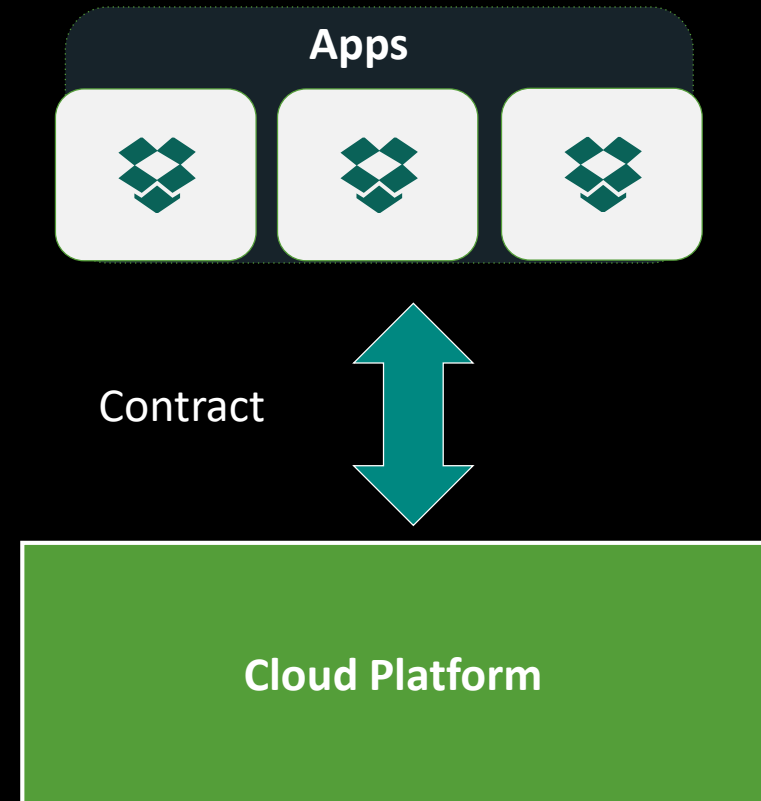


WHAT'S GOING ON?!!?



# Cloud Native Principles

- Twelve-Factor App (<http://12factor.net>) principles popular and important
- Twelve Factors include Dependencies, Config, Processes, and Disposability
- Cloud native apps recognize they are ephemeral, and minimize dependencies on the underlying platform
- Principles establish a contract between cloud native apps and the underlying platform



# Netflix Cloud Libraries

- Netflix needed to be faster to win/disrupt
- Pioneer and vocal proponent of microservices
  - Key to speed and success
- Netflix OSS supplies parts, but its not a full solution



# Open Source Cloud Libraries

- Twitter, Facebook and Hashicorp have open-sourced other cloud infrastructure libraries
- Complementary and competing solutions
  - Form a bazaar of ideas and solutions



# Spring Cloud

- Easy developer access to curated selection of open source cloud infrastructure
- Spring Cloud API encapsulates access to underlying libraries
  - Pluggable implementations
  - Allows best of breed



# Spring Cloud

- Additional capabilities include
  - Cloud Connectors
  - Config Server
- Spring philosophy
  - Convention over configuration
  - Opinionated defaults
  - Developer simplicity





# Spring Cloud Services



## Services Marketplace



### Circuit Breaker

Circuit Breaker Dashboard for Spring Cloud Applications



### Config Server

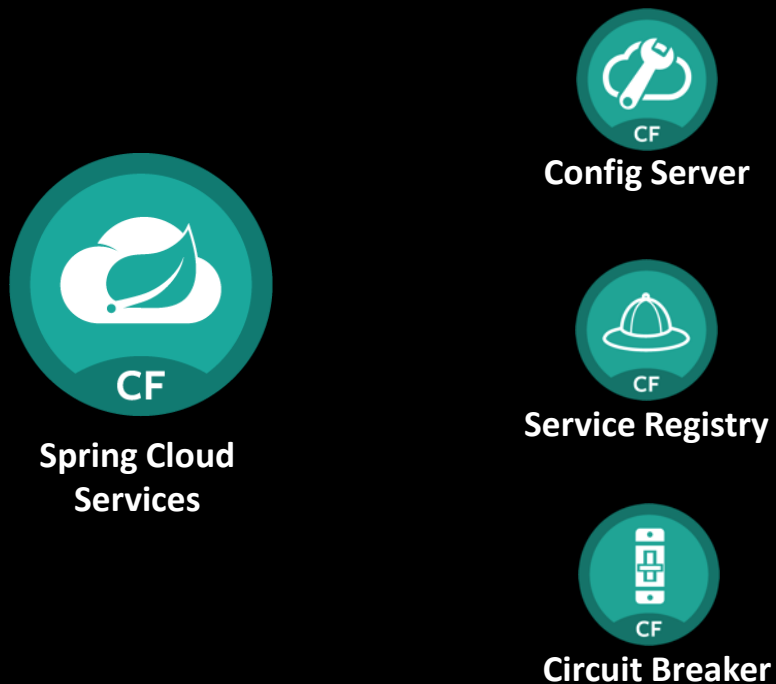
Config Server for Spring Cloud Applications



### Service Registry

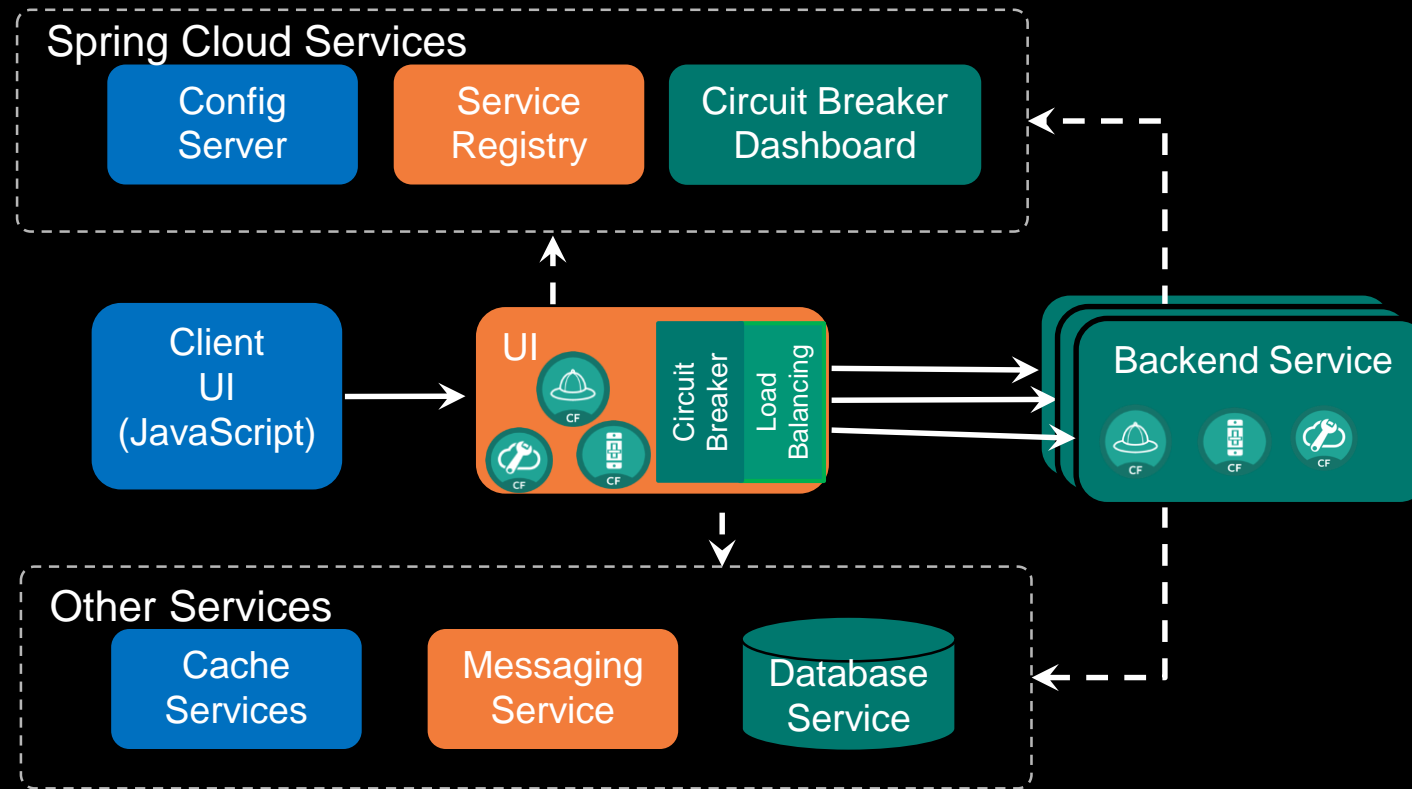
Service Registry for Spring Cloud Applications

# Spring Cloud Services



- Config Server
  - Centrally manage app configuration
  - Single tenant, scope to CF Space
- Service Registry
  - Registration/Discovery via Netflix Eureka
  - Registration via CF Route
- Circuit Breaker
  - Via Netflix Turbine and Hystrix Dashboard
  - Aggregation via AMQP (RabbitMQ)

# Spring Cloud Services Application Architecture



# Spring Cloud Services

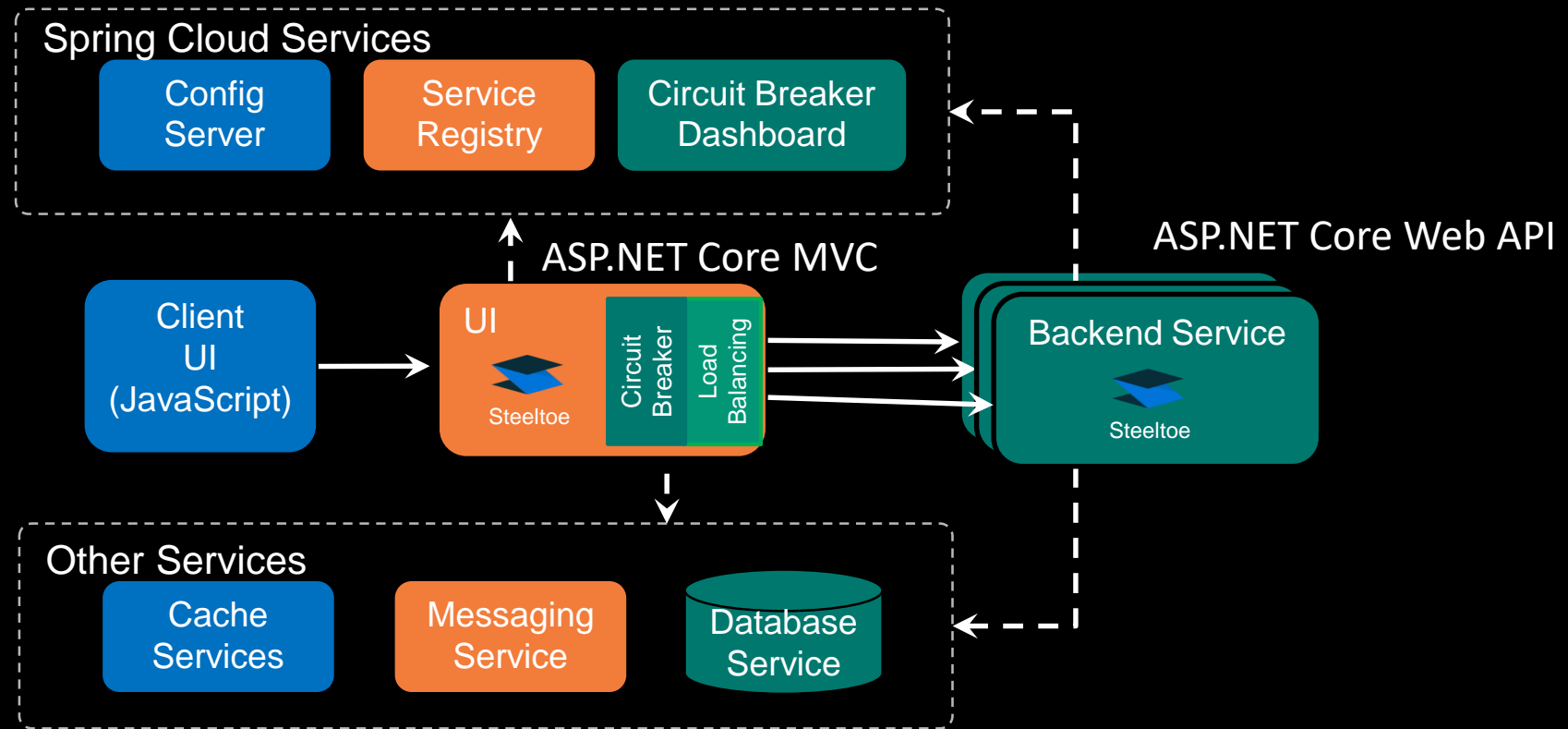
- Battle tested infrastructure, out of the box
- High availability, Enterprise class implementations
- Integrated with logging, monitoring and administration of Pivotal Cloud Foundry



# Steeltoe Overview

CONNECTORS, CONFIGURATION, DISCOVERY, SECURITY

# Spring Cloud Services Application Architecture on .NET





## Application Frameworks



Spring Boot



Spring Cloud  
Dataflow



Spring Cloud  
Services



Steeltoe



.NET Core



.NET

## Facilitates Twelve-Factor Contract on .NET



## Enabling Cloud Native Applications on .NET

- Simplifies using .NET & ASP.NET on Cloud Foundry
  - Connectors (e.g. MySQL, Redis, Postgres, RabbitMQ, OAuth, etc.)
  - Security providers (e.g. OAuth SSO, JWT, Redis KeyRing Storage, etc.)
  - Configuration providers (e.g. Cloud Foundry)
- Simplifies using Spring Cloud Services
  - Configuration server provider (e.g. Config Server, etc.)
  - Service Discovery (e.g. Eureka, etc.)
  - Circuit Breaker (e.g. Hystrix coming)
  - Distributed Tracing (e.g. Slueth coming)

# Steeltoe Overview

- Open Source
  - <https://github.com/SteeltoeOSS>
  - Lots of documentation in Readmes
  - Functionally organized (Configuration, Discovery, Connectors, etc.)
- .NET support
  - .NET Core (Windows, Linux & OSX)
  - .NET Framework
- Application type support
  - ASP.NET - MVC, WebForm, WebAPI, WCF
  - ASP.NET Core
  - Console apps (.NET Framework and .NET Core)



# Steeltoe Overview

- Connectors – simplifies configuring and injecting the following as services
  - MySql, MySql EF6, MySql EFCore – Connections and `DbContexts`
  - Postgres, Postgres EFCore – Connections and `DbContexts`
  - RabbitMQ – Connections
  - Redis – Microsoft `IDistributedCache` & `StackExchange ConnectionMultiplexor`
  - OAuth – access connection details from CF UAA & Pivotal SSO Service
- Configuration – additional `ConfigurationBuilder` providers
  - CloudFoundry – parse `VCAP_*`, `CF_*` and add to apps configuration
  - Config Server Client – access to Config Server

# Steeltoe Overview

- Discovery – service registry clients
  - Netflix Eureka Client – registration and discovery via Eureka Server
- Security – providers for Cloud Foundry and ASP.NET Core security integration
  - OAuth2 provider – Cloud Foundry integration with UAA/Pivotal SSO
  - JWT provider - Cloud Foundry integration with UAA/Pivotal SSO
  - Redis DataProtection KeyStorage connector – use Cloud Foundry Redis service for key ring storage
- Samples – functional area & full featured samples available
  - All samples: <https://github.com/SteeltoeOSS/Samples>
  - MusicStore – micro-services app built from the ASP.NET Core reference app
  - Freddy's BBQ a polyglot (i.e. Java and .NET) micro-services based sample app

# Steeltoe Overview

- NuGet feeds:
  - Development: <https://www.myget.org/gallery/steeltoedev>
  - Stable: <https://www.myget.org/gallery/steeltoemaster>
  - Release & Release Candidates: <https://www.nuget.org/>

# Steeltoe Coming Soon

- Circuit Breaker (CB) – enables implementing CB pattern in .NET
  - Netflix Hystrix – full integration with Spring Cloud Circuit Breaker dashboard
- Distributed Tracing – tracing support for micro-services based apps
  - Slueth – full integration with Spring Cloud Tracing dashboard

# Steeltoe Configuration Providers

ASP.NET CORE CONFIGURATION, OPTIONS, STEELTOE PROVIDERS

# Steeltoe Configuration Providers

- Built on ASP.NET Core Configuration & Options extensions
  - <https://github.com/aspnet/Configuration>
  - <https://github.com/aspnet/Options>
  - <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration>
- Two providers
  - Steeltoe.Extensions.Configuration.CloudFoundry
  - Steeltoe.Extensions.Configuration.ConfigServer
- Application type support
  - ASP.NET - MVC, WebForm, WebAPI, WCF
  - ASP.NET Core
  - Console apps (.NET Framework and .NET Core)

# Understanding ASP.NET Core Configuration

- Configuration built using `ConfigurationBuilder`; usually in `Startup` constructor
  - Add configuration sources to the builder via `Addxxx()` methods
  - Call `Build()` to actually read in and construct configuration
  - Results in a `IConfigurationRoot`
- Several configuration sources available
  - File based (e.g. INI, JSON and XML)
  - Command line arguments
  - Environment variables
  - Custom (e.g. Steeltoe source providers)
- Sources read in order added
  - Later sources override any settings from previous
- Keys are name-value pairs grouped into multilevel hierarchy – hierarchy separated by “:”
  - e.g. ``section:subsect:key``
  - Access values via `IConfigurationRoot` indexer (e.g. `var v=root["section:subsect:key"]`)

# Understanding ASP.NET Core Environments

- ASP.NET Core has support for multiple `environments`
  - Built in understanding of `production`, `staging` & `development` - case sensitive on Linux only
  - Set via environment variable `ASPNETCORE\_ENVIRONMENT`
- Use `IHostingEnvironment` to access environment
  - Injectable
  - Use `env.IsEnvironment("Cloud")` to test – case insensitive way
  - Use `env.IsDevelopment()`, `IsProduction()`, or `IsStaging()` also
- Lets look at some code!



# Understanding ASP.NET Core Options

- Enables using custom classes to hold related configuration settings
  - Class must have public parameter-less constructor and attributes to hold values
  - Built in binder for binding configuration to class; attribute names -> config keys
- To bind configuration data to a class and add it to DI container:
  - Use `services.Configure<SomeClass>(config)` method on `IServiceCollection` to bind configuration to `SomeClass`
  - This also adds an `IOptions<SomeClass>` and `IOptionsSnapshot<SomeClass>` to the container for injection
  - Can also bind subsections of a configuration
    - e.g. `services.Configure<SomeClass>(root.GetSection("subsection"));`
- Lets look at some code!

# Steeltoe CloudFoundry Configuration Provider

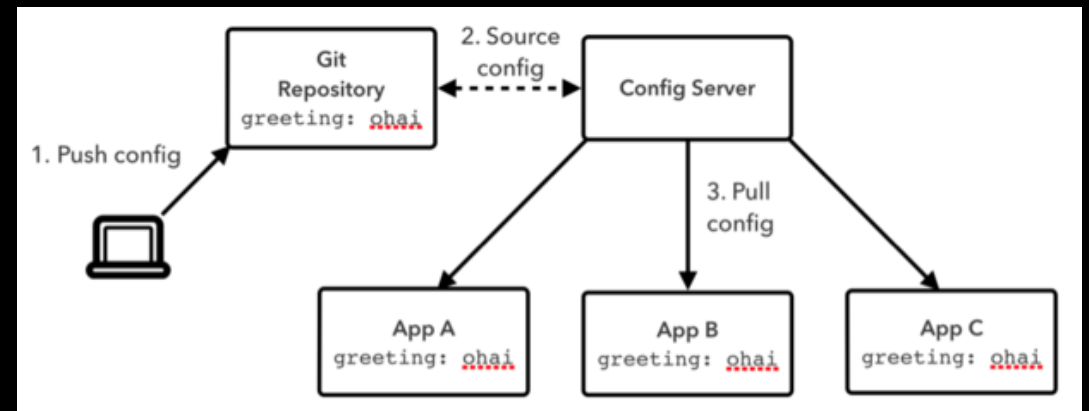
- Parses Cloud Foundry environment variables; adds them to Configuration
  - **NuGet:** `Steeltoe.Extensions.Configuration.CloudFoundry`
  - **Parses** `VCAP_APPLICATION`, `VCAP_SERVICES`, `CF_INSTANCE_*`
- **Use** `AddCloudFoundry()` **extension method with** `ConfigurationBuilder`
  - `VCAP_APPLICATION` & `CF_INSTANCE_*` -> **section** ``vcap:application``
  - `VCAP_SERVICES` -> **section** ``vcap:services``
  - **Access values using the indexer**
- **Optionally, use Options feature with configuration data**
  - `services.Configure<CloudFoundryApplicationOptions>(config);`
  - `services.Configure<CloudFoundryServicesOptions>(config);`
- **Lets look at some code!**

# Steeltoe ConfigServer Client Provider

- Enables Spring Cloud Config Server to be used as a configuration source
  - OSS Config Server - `Steeltoe.Extensions.Configuration.ConfigServer``
  - SCS Config Server - `Pivotal.Extensions.Configuration.ConfigServer``
- Application type support
  - ASP.NET - MVC, WebForm, WebAPI, WCF
  - ASP.NET Core
  - Console apps (.NET Framework and .NET Core)

# Spring Cloud Config Server Basics

- Supports different back ends
  - Local or remote git repos
  - File System
- Exposes resource HTTP API
  - Default starts on port=8080
- Configuration pulled by
  - AppName
  - Profile
  - Label – optional



# Spring Cloud Config Server Basics

- Serves resources:
  - /application.yml
  - /application.properties
  - /{appname}-{profile}.yml
  - /{appname}-{profile}.properties
  - /{appname}/{profile}/{label}
  - /{label}/{appname}-{profile}.yml
  - /{label}/{appname}-{profile}.properties
- For a git backend
  - Configure URL of git repo
  - {appname} & {profile} maps to file & directory in repo
  - {label} -> git branch
- Git backend example:
  - Repo contains files:
    - application.yml
    - application-development.yml
    - foo.yml
    - foo-development.yml
    - bar.yml
  - Client request contains:
    - {appname}=foo
    - {profile}=development
  - Config server returns in precedence order:
    - foo-development.yml
    - foo.yml
    - application-development.yml
    - application.yml

# Steeltoe ConfigServer Client Provider

- Use `AddConfigServer(environment)` extension method on `ConfigurationBuilder` to add Config Server client provider
  - At `Build()`, client calls Config Server and retrieves configuration data
- Must configure the Config Server client settings
  - Easiest to put settings in ``appsettings.json`` or other file based config source
  - Must add the settings provider before ``AddConfigServer(environment)`` so client can find settings
  - Two settings are required at minimum
    - ``spring:application:name`` defines the ``{appName}`` portion of the Config Server request
    - ``spring:cloud:config:uri`` defines the REST endpoint of the Config Server
  - `IHostEnvironment.EnvironmentName` is used for ``{profile}`` portion of Config Server request

# Steeltoe ConfigServer Client Settings

- **Config Server Client settings:**
  - ``spring:cloud:config:enabled`` - enable/disable Config Server client, default(true)
  - ``spring:cloud:config:uri`` - endpoint of Config Server, default("http://localhost:8888")
  - ``spring:cloud:config:validate_certificates`` - enable/disable cert validation, default(true)
  - ``spring:cloud:config:label`` - comma separated list of labels to request, default(empty)
  - ``spring:cloud:config:failFast`` - enable/disable failure at startup, default(false)
  - ``spring:cloud:config:retry:enabled`` - enable/disable retry logic, default(false), failFast enabled
  - ``spring:cloud:config:retry:maxAttempts`` - max number retries if retry enabled, default(6)
  - ``spring:cloud:config:retry:initialInterval`` - starting interval, default(1000)
  - ``spring:cloud:config:retry:multiplier`` - retry interval multiplier, default(1.1)
  - ``spring:cloud:config:retry:maxInterval`` - maximum interval, default(2000)
  - ``spring:cloud:config:username`` - username for Basic auth, default(empty)
  - ``spring:cloud:config:password`` - password for Basic auth, default(empty)

# Using Config Server on Cloud Foundry

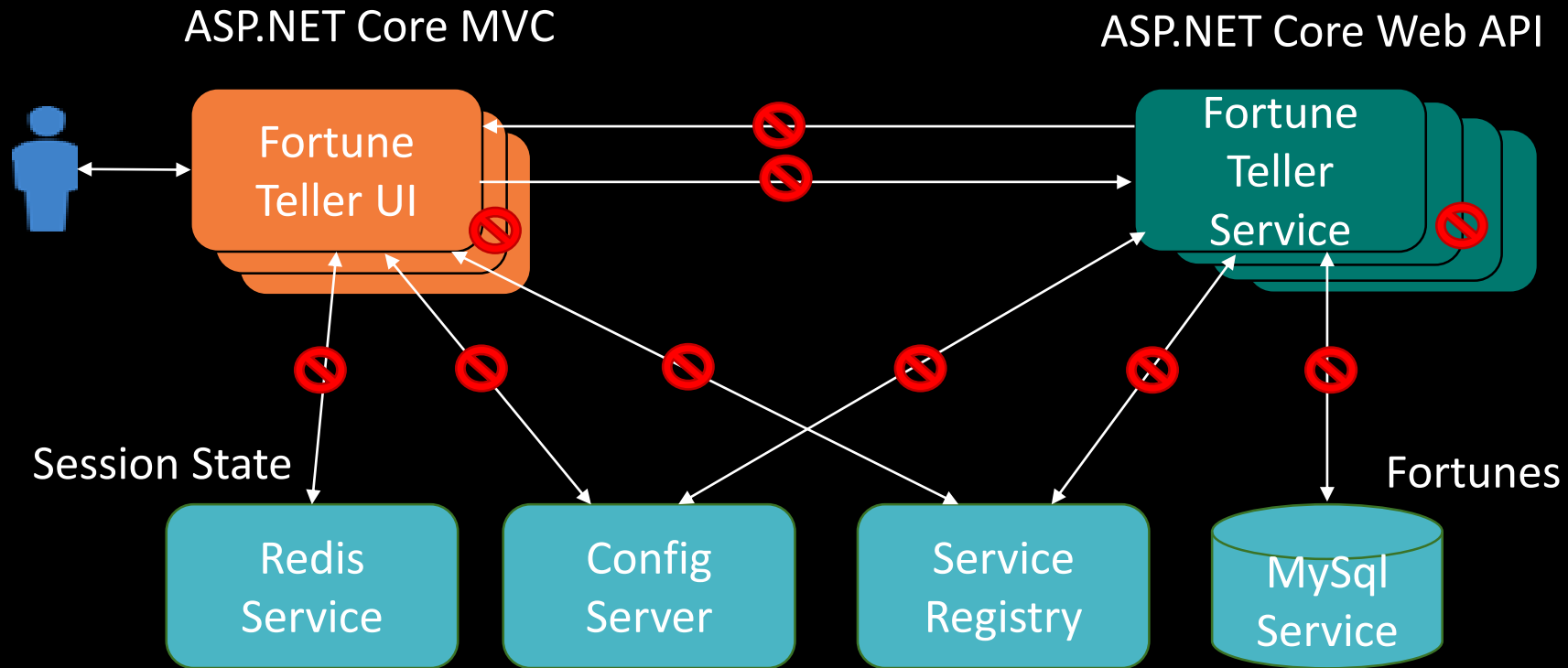
- Create instance of Config Server using CF CLI
  - ``cf create-service p-config-server standard cserver -c config.json``
  - Spins up config server in org: p-spring-cloud-services, space: instances
  - ``config.json`` specifies URL of git repo it uses for configuration data
  - Use ``cf service`` to check status of service
- Bind instance to applications
  - ``cf bind-service appName cserver``
  - Also specify binding in `manifest.yml`
    - ``services:`` section-> add ``cserver``
- Steeltoe Config Server Client detects p-config-server binding
  - Overrides ``appsettings.json`` client settings with binding information
  - Enables easier development and testing locally; and then push to CF with no changes
- Lets look at some code!



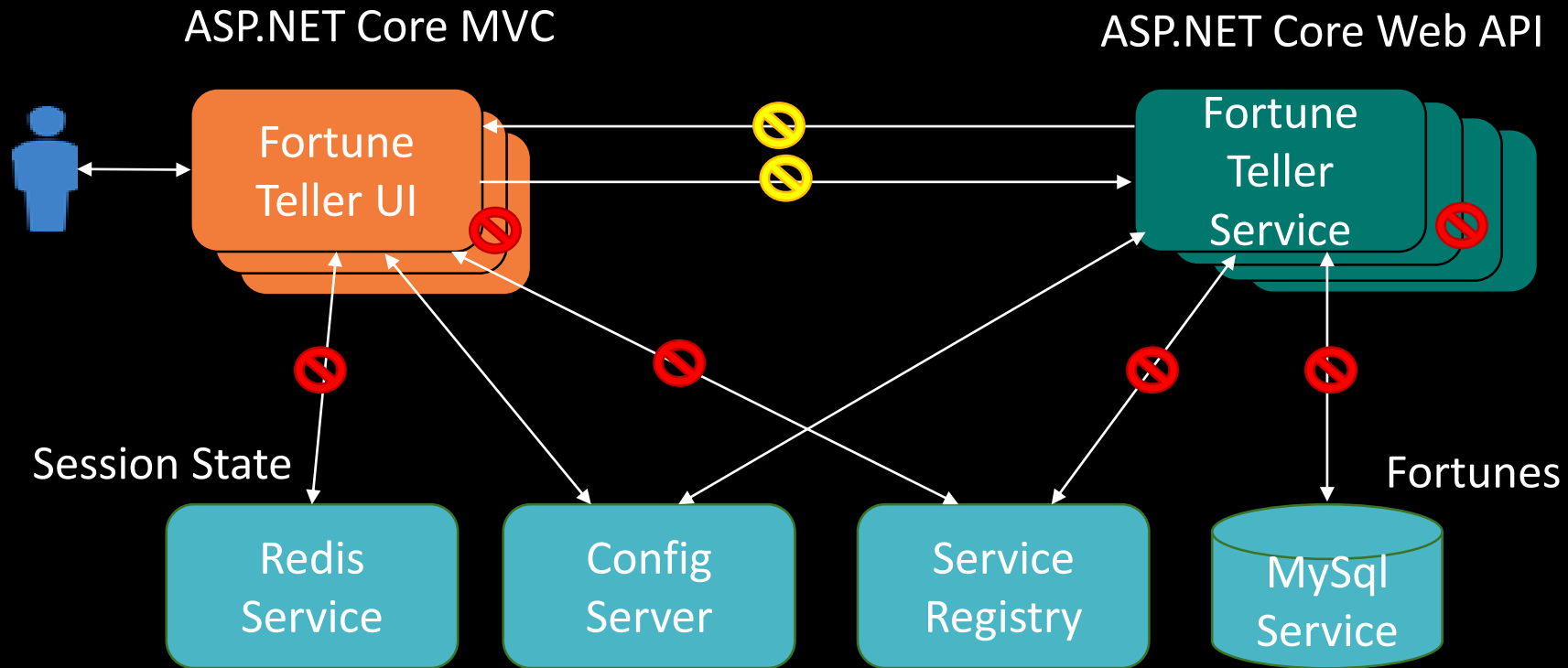
# Lab7 – Fortune Teller App

- Multiple Exercises:
  - Setup and use different configurations for development and production
    - Want to use full debug logging when in development mode
  - Use Configuration & Options to configure `FortuneServiceClient`
    - Use it to configure the REST endpoint of the Fortune Teller Service
    - `FortuneServiceClient` can then communicate with it
  - Use Steeltoe and Spring Cloud Config Server
    - Move as much configuration as possible to Config Server
    - Run Config Server locally for test/development
  - Deploy to Cloud Foundry
    - Move local Config Server data to git repo
    - Does app still work?

# Lab7 – Fortune Teller App – Before Lab



# Lab7 – Fortune Teller App – After Lab



# Steeltoe Service Discovery

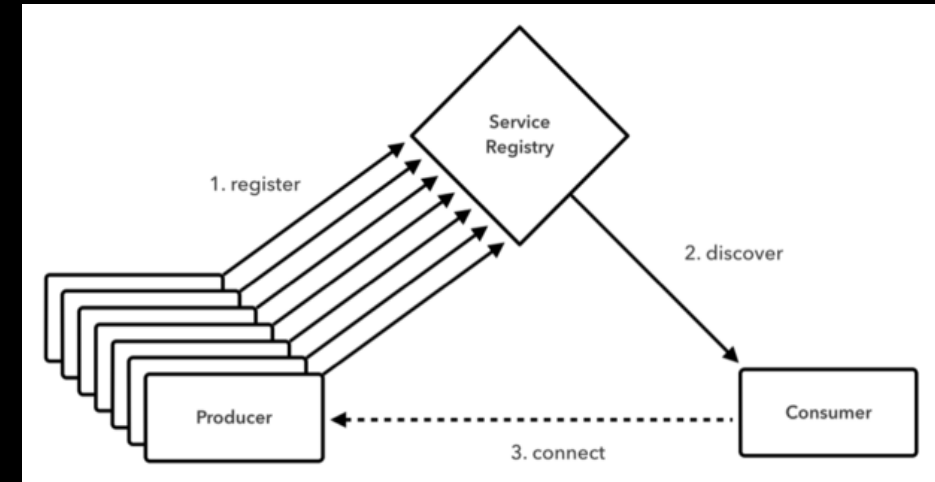
ASP.NET CORE CONFIGURATION, OPTIONS, STEELTOE PROVIDERS

# Steeltoe Service Discovery Client Overview

- Provides configurable generalized interface for Service Registry interaction
  - `Steeltoe.Discovery.Client`
- Single configurable provider today:
  - `Steeltoe.Discovery.Eureka.Client` – client for Netflix Eureka Service registry
- Application type support
  - ASP.NET - MVC, WebForm, WebAPI, WCF
  - ASP.NET Core
  - Console apps (.NET Framework and .NET Core)

# Spring Cloud Service Registry Basics

- Use Service IDs, not URLs, to locate services
- Client-side or server-side load balancing



# Steeltoe Eureka Client

- Enables interaction with Netflix Eureka Service Registry
  - OSS Netflix Eureka Server – use `Steeltoe.Discovery.Client``
  - PCF Netflix Eureka Server - use `Pivotal.Discovery.Client``
- Three step process to enable the Steeltoe Eureka client
  - Configure Discovery client settings – use values from the built Configuration
  - Add Discovery client to service container - `AddDiscoveryClient(Configuration)``
  - Use Discovery client – `UseDiscoveryClient()` – ``starts up client background thread``
    - For Eureka, registered services are pulled from Eureka Server
- Service Discovery client settings
  - Normally just add settings to Configuration
    - Put in ``appsettings.json``, or Config Server repo, etc.
  - Settings for discovering services: ``eureka:client:....``
  - Settings for registering as a service: both ``eureka:instance:....`` & ``eureka:client:....``

# Steeltoe Eureka Client Settings

- Some of the many Eureka settings:
  - ``eureka:client:serviceUrl`` - Eureka server URL, default(`http://localhost:8761/eureka/`)
  - ``eureka:client:shouldRegisterWithEureka`` - enable/disable registering as a service, default(`true`)
  - ``eureka:client:shouldFetchRegistry`` - enable/disable fetching registry periodically, default(`true`)
  - ``eureka:client:validate_certificates`` - enable/disable cert validation, default(`true`)
  - ``eureka:client:registryFetchIntervalSeconds`` - fetch interval, default(`30`)
  - ``eureka:client:shouldFilterOnlyUpInstances`` - only UP instances, default(`true`)
  - ``eureka:instance:name`` - name to register under, default(``spring:application:name``)
  - ``eureka:instance:instanceId`` - unique ID scoped to ``name``, default(`hostname`)
  - ``eureka:instance:port`` - port number to register on, default(`80`)
  - ``eureka:instance:hostName`` - host name to register on, default(`hostname`)
  - ``eureka:instance:leaseRenewalIntervalInSeconds`` - how often heartbeats are sent, default(`30`)
  - ``eureka:instance:leaseExpirationDurationInSeconds`` - heartbeat lost delay, default(`90`)
- Lets look at some code!



# Using Steeltoe Eureka Client

- When `AddDiscoveryClient (config)` is done in Startup class:
  - Adds Steeltoe `IDiscoveryClient` as a Singleton to service container
  - Can be injected into Controllers, Views, etc.
  - Can use the interface to access registered services by name – `GetInstances(name)`
- When `UseDiscoveryClient ()` is done in Startup class:
  - Starts background thread fetching and registering services

# Using Steeltoe Eureka Client to Find Services

- Inject `IDiscoveryClient` into your Controller, View, etc.
  - Can use the interface to access registered services by name – `GetInstances(name)`
- Alternatively, use `DiscoveryHttpClientHandler` – an `HttpClientHandler` for use with `HttpClient`
  - Integrates service lookup with issuing `HttpClient` requests
  - Handler intercepts requests and attempts to resolve `host` portion of the request URL as a service name
    - Replaces it with resolved address if successful
    - Leaves alone if not

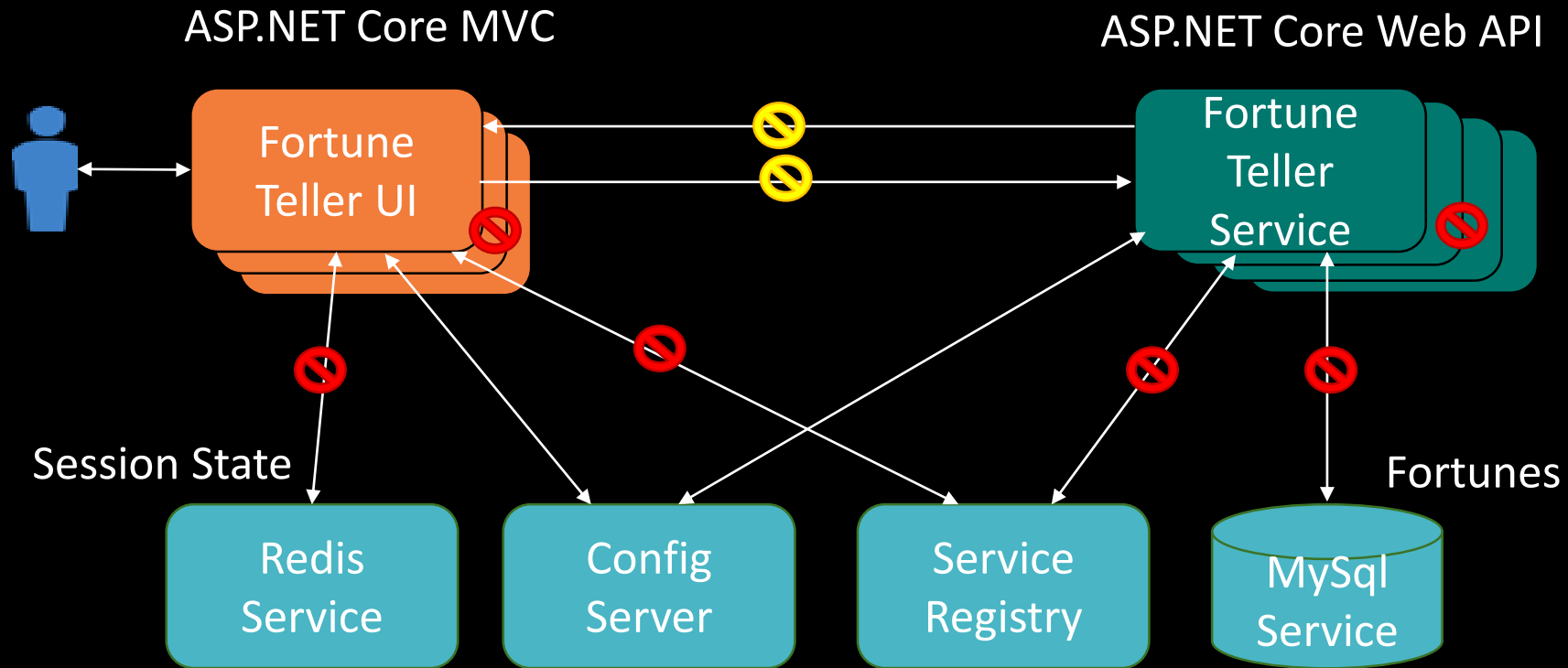
# Using Eureka Server on Cloud Foundry

- Create instance of Eureka Server using CF CLI
  - ``cf create-service p-service-registry standard myDiscoveryService``
  - Spins up eureka server in org: p-spring-cloud-services, space: instances
  - Use ``cf service`` to check status of service
- Bind instance to applications
  - ``cf bind-service appName myDiscoveryService``
  - Also specify binding in `manifest.yml`
    - ``services:`` section-> add ``myDiscoveryService``
- Steeltoe Discovery Client detects p-service-registry binding
  - Overrides ``appsettings.json`` client settings with binding information
- Use Eureka Server dashboard to examine registered services - <http://localhost:8761/>
- Lets look at some code!

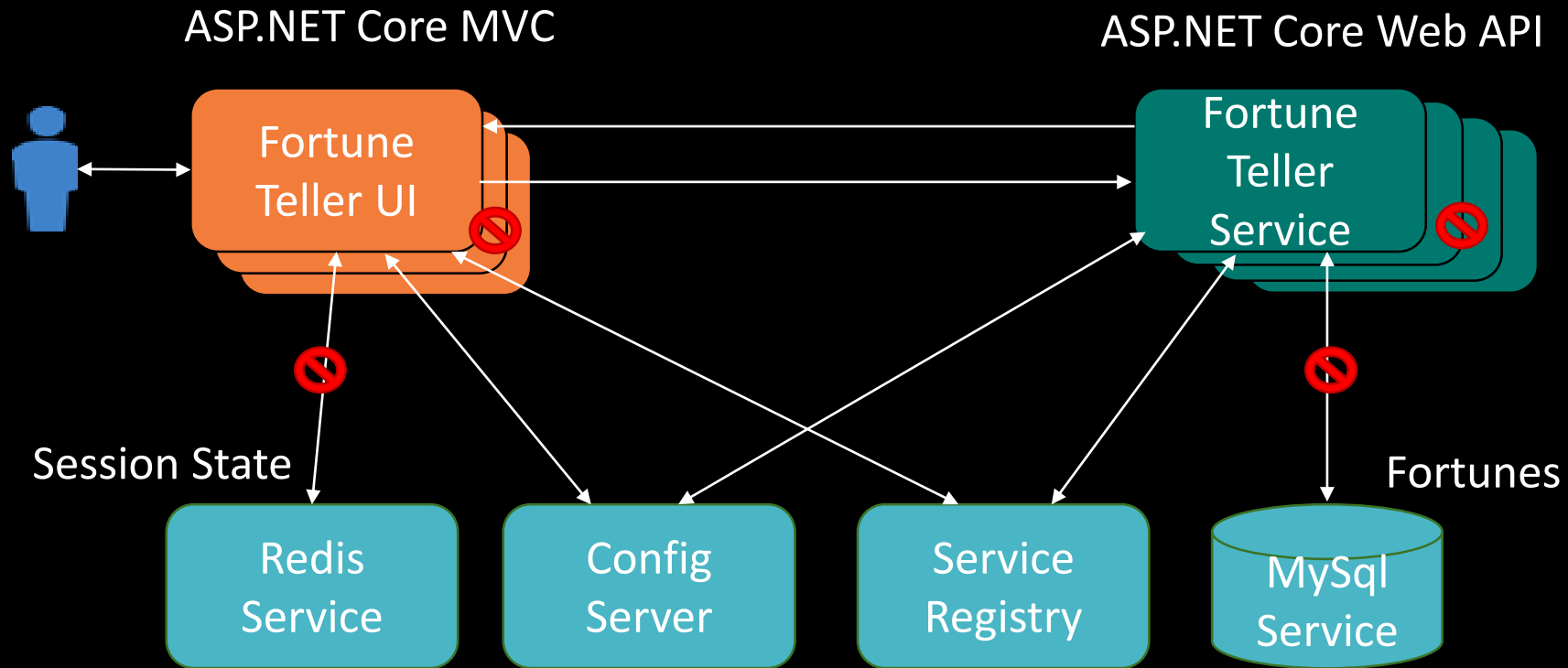
# Lab8 – Fortune Teller App

- Use Steeltoe and Spring Cloud Eureka Server
  - Update configuration for Eureka
  - Update code to use Steeltoe Discovery client
  - Run Eureka Server locally for test/development
  - Deploy to Cloud Foundry

# Lab8 – Fortune Teller App – Before Lab



# Lab8 – Fortune Teller App – After Lab



# Steeltoe Connectors

ASP.NET CORE SESSION, KEY RING, CONNECTORS

# Steeltoe Connectors Overview

- Simplify using Cloud Foundry services
  - Can configure settings for local usage (e.g. `appsettings.json`, Config Server, etc.)
  - When app pushed to Cloud Foundry bindings auto detected and override settings
  - Adds `Connection` or `DbContext` objects to service container
- Several connector NuGets
  - `Steeltoe.CloudFoundry.Connector.MySql`
  - `Steeltoe.CloudFoundry.Connector.Postgres`
  - `Steeltoe.CloudFoundry.Connector.Rabbit`
  - `Steeltoe.CloudFoundry.Connector.Redis`
  - `Steeltoe.CloudFoundry.Connector.OAuth`
- Application type support
  - ASP.NET - MVC, WebForm, WebAPI, WCF
  - ASP.NET Core
  - Console apps (.NET Framework and .NET Core)



# Steeltoe MySql Connector Usage

- Configures and adds a `MySqlConnection` or `DbContext` to the container
  - Built on latest version of Oracle Connector/NET
  - Supports both `EntityFramework` and `EntityFrameworkCore` `DbContexts`
- Usage
  - Add `Steeltoe.CloudFoundry.Connector.MySql` NuGet
  - Add `CloudFoundry` config provider to `Configuration Builder` (i.e. `AddCloudFoundry()`)
    - Not needed if already using `Steeltoe Config Server` client (i.e. `AddConfigServer()` )
  - Configure `MySql` client settings
  - Add `MySqlConnection` or `DbContext` to service container
- `MySql` client settings
  - Add settings to `Configuration` via ``appsettings.json``, `Config Server`, etc.
  - Settings for configuring client-> ``mysql:client:...``

# Steeltoe MySql Connector Usage

- **MySQL client settings:**
  - ``mysql:client:server`` - hostname/address of server, default(localhost)
  - ``mysql:client:port`` - port number for server, default(3306)
  - ``mysql:client:username`` - username for authentication, default(empty)
  - ``mysql:client:password`` - password for authentication, default(empty)
  - ``mysql:client:database`` - schema to connect to, default(empty)
  - ``mysql:client:connectionString`` - full connection string, default(empty), use instead of the above
- **Add to the service container**
  - `AddMySQLConnection(Configuration)` – to inject `MySQLConnection`
  - `AddDbContext<YourContext>(Configuration)` – for EF 6
  - `AddDbContext<YourContext>(o => o.UseMySQL(Configuration))` – for EF Core

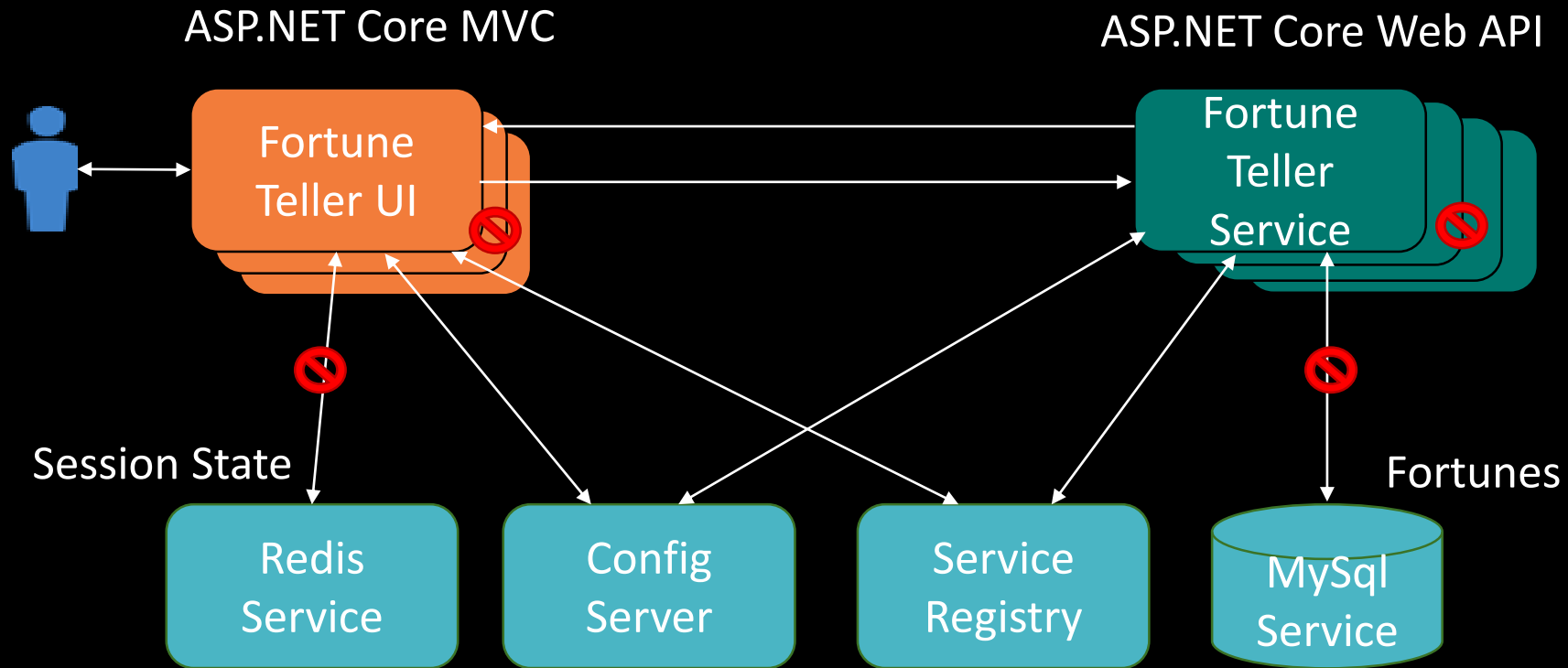
# Using MySql Service on Cloud Foundry

- Create instance of MySql service using CF CLI
  - ``cf create-service p-mysql 100mb myMySQLService``
  - Creates a database tenant in server
- Bind instance to applications
  - ``cf bind-service appName myMySQLService``
  - Also specify binding in `manifest.yml`
    - ``services:` section-> add `myMySQLService``
- Steeltoe MySql connector detects `p-mysql` binding
  - Overrides any “`appsettings.json`” client settings with binding information
- Lets look at some code!

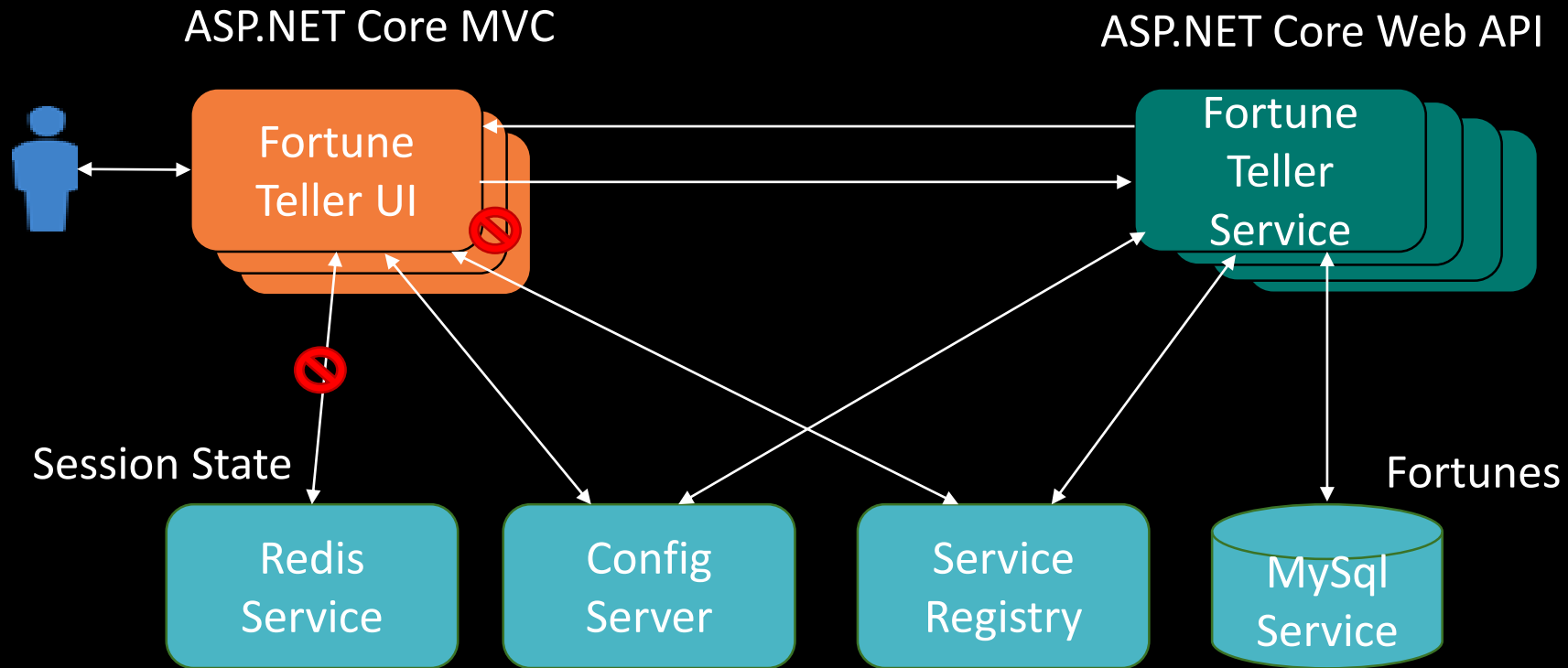
## Lab9a – Fortune Teller App

- Use MySQL Connector to connect to MySQL service
  - When running in development mode use a `FortuneContext` configured using in-memory database
  - When running in any other mode use a `FortuneContext` configured using Steeltoe MySQL connector
- Deploy to Cloud Foundry

# Lab9a – Fortune Teller App – Before Lab



# Lab9a – Fortune Teller App – After Lab



# Understanding ASP.NET Core Session

- ASP.NET Core has middleware (i.e. Session) for managing session state
  - Session added as a service – `AddSession()`
  - Session added as middleware to pipeline – `UseSession()`
  - Session service expects to find an `IDistributedCache` in container for storage
- Session state
  - Stored in dictionary
  - Dictionary saved in `IDistributedCache`, defaults to In-Memory cache
  - Session ID used to save and fetch state
  - Session IDs stored in cookie & sent to browser
  - Session IDs are encrypted using `DataProtection` services before adding to cookie
- Access to session is via `HttpContext.Session`
  - `GetXXX/SetXXX` methods

# Understanding ASP.NET Core DataProtection

- Crypto services for protecting data and for key management
  - Used both internally and optionally by application code
  - Added as a service using `AddDataProtection()`
  - Extension methods used to configure its behavior:
    - `PersistKeysToFileSystem(...)`
    - `PersistKeysToRedis(...)`
- Keys generated and held in key-ring and then stored in a repository
  - Default is to store key ring in local file system repo



# Steeltoe Redis Connector Usage

- Configures and adds a ASP.NET Core `RedisCache` and/or `StackExchange ConnectionMultiplexor` to the container
  - `RedisCache` built on top of `StackExchange NuGets`
- Usage
  - Add `Steeltoe.CloudFoundry.Connector.Redis NuGet`
  - Add `CloudFoundry` config provider to `Configuration Builder` (i.e. `AddCloudFoundry()`)
    - Not needed if already using `Steeltoe Config Server` client (i.e. `AddConfigServer()` )
  - Configure `Redis` client settings
  - Call `AddDistributedRedisCache()` or `AddRedisConnectionMultiplexor()` to add to service container
- Redis client settings
  - Add settings to `Configuration` via ``appsettings.json``, `Config Server`, etc.
  - Settings for configuring client-> ``redis:client:...``

# Steeltoe Redis Connector Usage

- Some of the Redis client settings:
  - ``redis:client:host`` - hostname/address of server, default(localhost)
  - ``redis:client:port`` - port number for server, default(6379)
  - ``redis:client:password`` - password for server, default(empty)
  - ``redis:client:clientName`` - id for connection within server, default(stackexchange)
  - ``redis:client:connectRetry`` - number times to retry connect, default(stackexchange)
  - ``redis:client:connectionString`` - stackexchange connection string - default(empty), use instead of
  - ``redis:client:instanceId`` - RedisCache instanceid for partitioning- default(empty), only RedisCache
- Add to the service container
  - `AddDistributedRedisCache(config)` – to inject a RedisCache as an IDistributedCache
  - `AddRedisConnectionMultiplexer(Configuration)` – to inject a StackExchange ConnectionMultiplexer

# Using Redis Service on Cloud Foundry

- Create instance of Redis service using CF CLI
  - ``cf create-service p-redis shared-vm myRedisService``
  - Creates Redis server instance in a shared VM
- Bind instance to applications
  - ``cf bind-service appName myRedisService``
  - Also specify binding in `manifest.yml`
    - ``services:` section-> add `myRedisService``
- Steeltoe Redis connector detects `p-redis` binding
  - Overrides any “`appsettings.json`” client settings with binding information
- Lets look at some code!

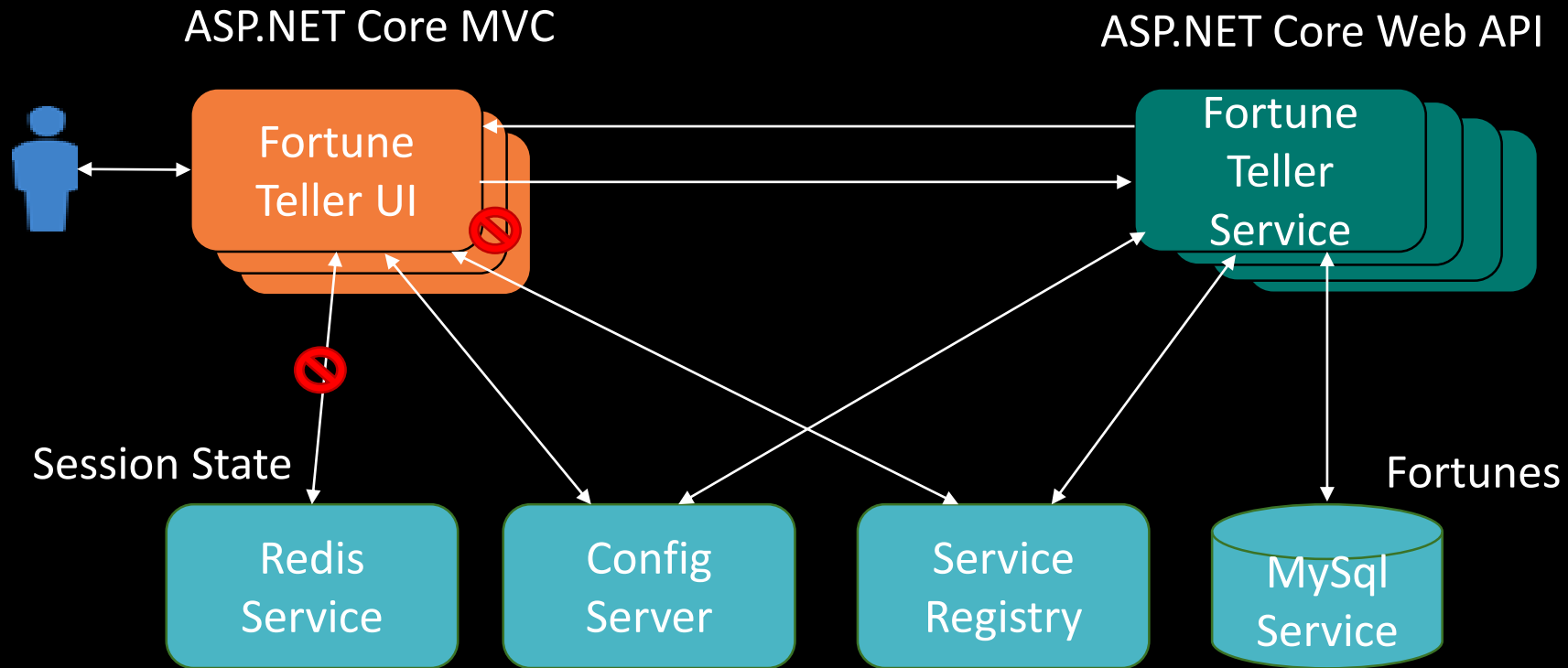
# Steeltoe Redis KeyStorage Connector Usage

- Configures `DataProtection` to use a Cloud Foundry Redis service for Key Ring storage
  - Takes a `StackExchange ConnectionMultiplexor` as constructor argument
- Usage
  - Add `Steeltoe.Security.Dataprotection.Redis` NuGet
  - Add CloudFoundry config provider to Configuration Builder (i.e. `AddCloudFoundry()`)
    - Not needed if already using Steeltoe Config Server client (i.e. `AddConfigServer()`)
  - Configure Redis client settings
  - Call `AddRedisConnectionMultiplexor()` to add to service container
  - Call `PersistKeysToRedis()` to use Redis for Key Storage
- Redis client settings
  - Same as discussed earlier
- Lets look at some code!

## Lab9b – Fortune Teller App

- Use Redis Connector to connect to Redis service and setup Session cache storage as follows:
  - When running in development mode use default in-memory cache for session storage
  - When running in any other mode use the Redis based cache for session storage
- Use Redis KeyStorage Connector to configure the DataProtection service to use Redis for key ring storage as follows:
  - When running in development mode use default file system key ring storage
  - When running in any other mode use the Redis based cache for key ring storage
- Deploy to Cloud Foundry

# Lab9b – Fortune Teller App – Before Lab



# Lab9b – Fortune Teller App – After Lab

