# Steeltoe .NET Developer Workshop

# Workshop Goals

- Provide attendees with a solid understanding of the tools and techniques used to build enterprise-class ASP.NET applications on Cloud Foundry
  - Pivotal Cloud Foundry & Services
  - Micro-services using ASP.NET Core
  - Centralized application configuration
  - Service discovery
  - Horizontal scaling
  - Fault tolerance using Circuit Breakers
  - Security
  - Production Management & Monitoring
- Build a sample application using the above tools & techniques
  - Incrementally build a Fortune Teller micro-services application

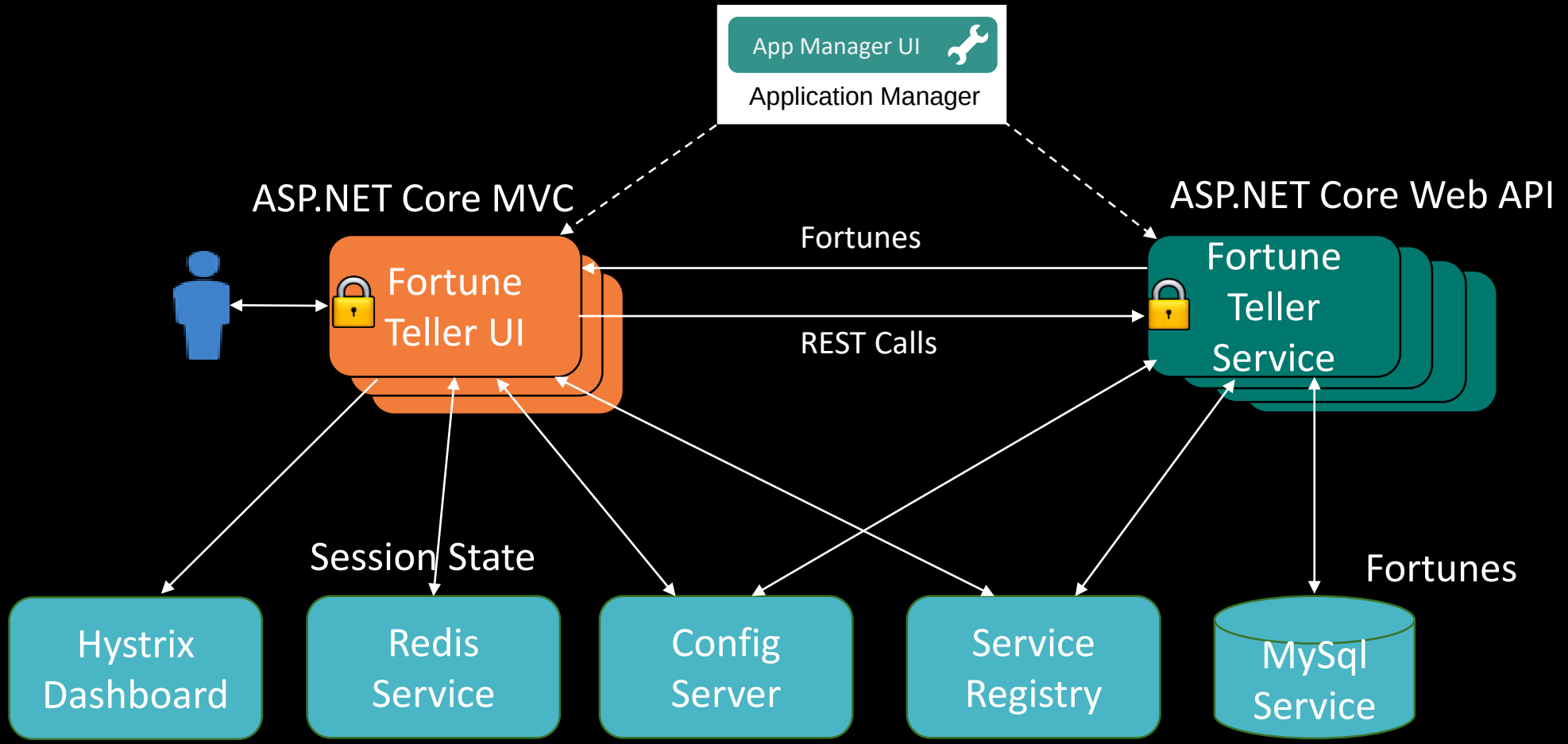
# Workshop Participant Experience

- .NET Application development experience
  - Previous .NET, ASP.NET experience
- .NET Core, ASP.NET Core experience
  - Nice to have, will be very helpful
- Some Pivotal Cloud Foundry experience
  - Push, Logs, Scaling, Services, etc.

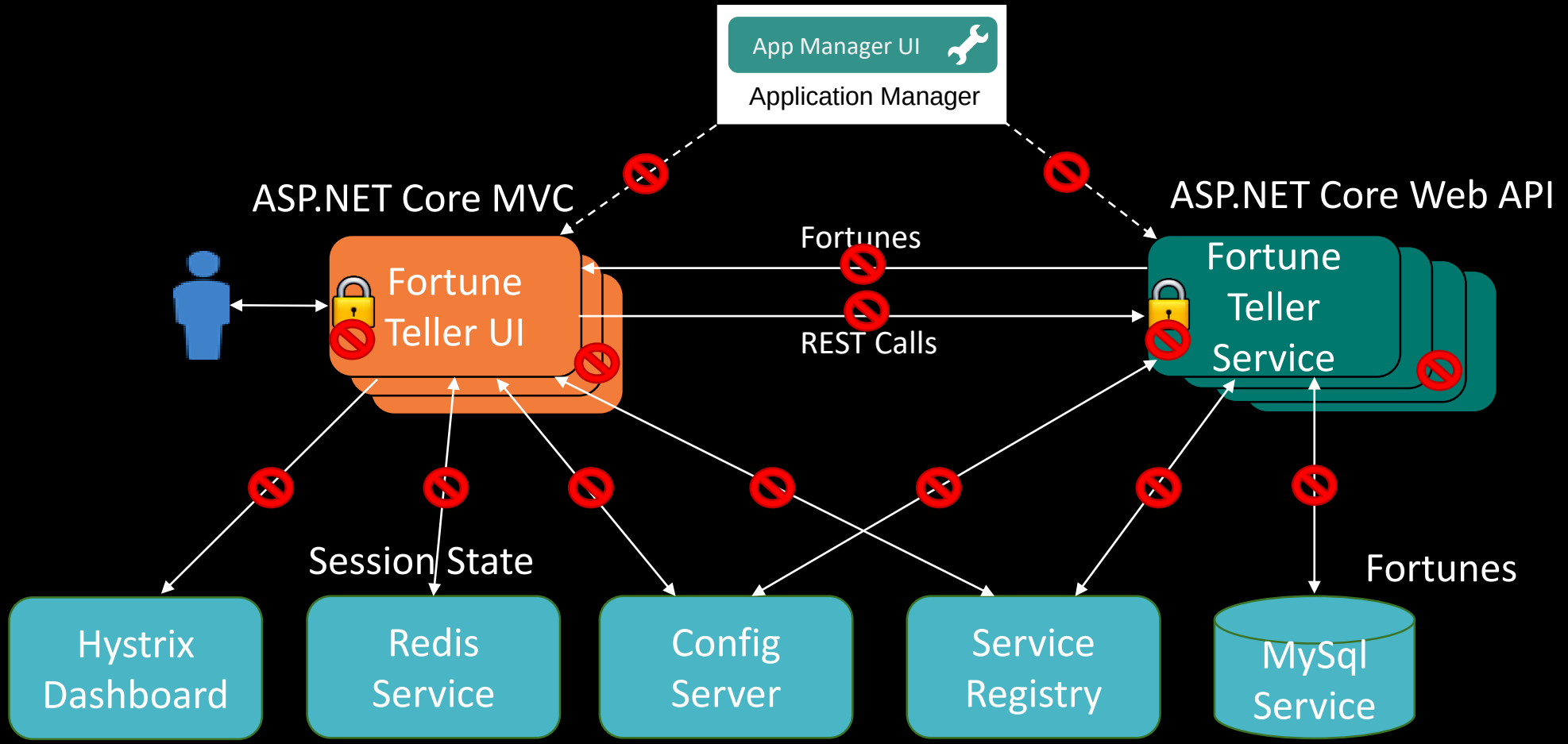
# Workshop Format

- Combination of Presentation, Hands-on-Labs
- Start with introductory Pivotal Cloud Foundry(PCF) overview
  - 4 hands-on labs focused on fundamentals of using  PCF
- Finish with building and deploying ASP.NET Core micro-services app on Cloud Foundry using Steeltoe
  - 7 hands-on labs focused on incrementally building Fortune Teller uServices app

# Fortune Teller App – When done!



# Fortune Teller App – Starting with!



# Lab Agenda

- Pivotal Cloud Foundry Overview
  - Lab00 – Install Prerequisites & Log into Cloud Foundry
  - Lab01 – Running .NET Application on Cloud Foundry
  - Lab02 – Creating and Binding to Cloud Foundry Services
  - Lab03 – Scaling and Operating Applications
  - Lab04 – Monitoring Applications
- Building Fortune Teller Application
  - Lab05 – ASP.NET Core Programming Fundamentals
  - Lab06 – Centralizing Application Configuration – Using Config Server
  - Lab07 – Service Discovery – Using Eureka Server
  - Lab08 – Scaling Horizontally – Using Redis and MySql Services
  - Lab09 – Fault Tolerance & Monitoring – Using Hystrix
  - Lab10 – Securing Application Endpoints – Using OAuth and JWT Bearer tokens
  - Lab11 – Production Monitoring & Management – Using Pivotal Apps Manager

# Workshop Technical Prerequisites

- .NET Core SDK 2.0 SDK
  - https://www.microsoft.com/net/download
- Visual Studio Development Environment – pick either
  - Visual Studio 2017 – Windows only
    - https://www.visualstudio.com/downloads/
  - Visual Studio Code – on Mac, Linux or Windows
    - http://code.visualstudio.com/
    - Add C# extension - http://code.visualstudio.com/docs/languages/csharp
- Java JDK 8 – needed to run Spring Cloud Servers locally
  - Could use Docker instead
- Cloud Foundry CLI
- Git command line tools

# Workshop Materials

- Workshop is Open Source on GitHub
  - Open - https://github.com/SteeltoeOSS/Workshop
- Workshop Slides
  - Link to slides in README.md
- Workshop Labs
  - /Workshop/LabXY – completed code for that specific lab
    - Lab descriptions in README.md
    - Hint: Get stuck in a lab, look in these directories!
  - /Workshop/Final – completed code for Workshop
    - Comment blocks in code outlining what changes were made for each lab
  - /Workshop/Lab01 – code for Lab1-Lab4
  - /Workshop/Start – starting code for Lab5 -> Lab11

# The Big Picture

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

# Cloud Native Promise – The Hype

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

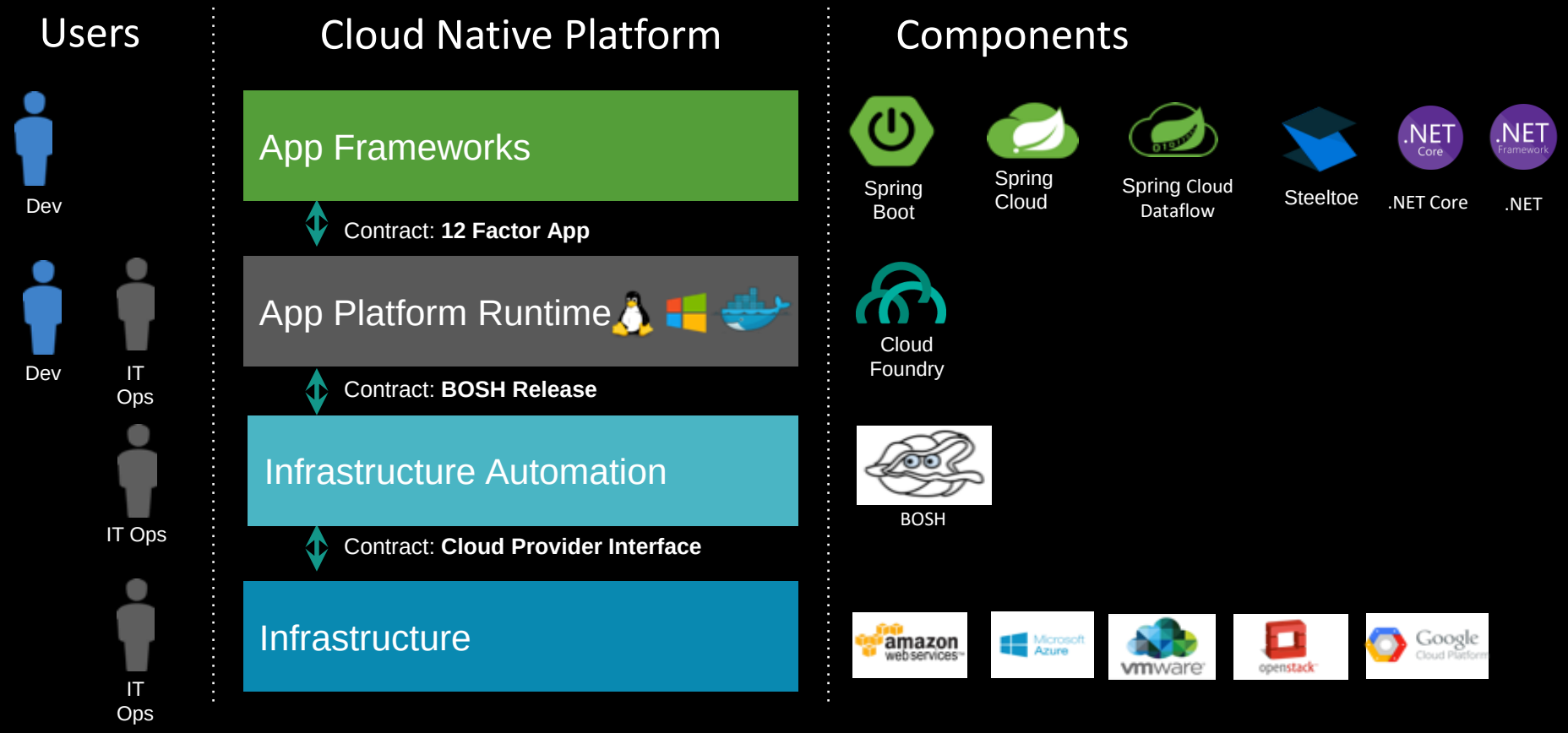
# Cloud Native Applications

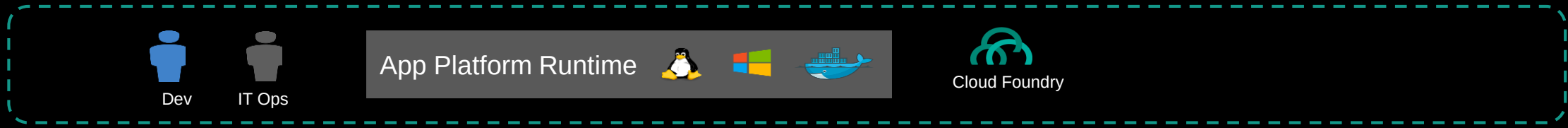
Cloud Native is not just about **where** you run your app, it's about **how** you write, build and run your app!

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

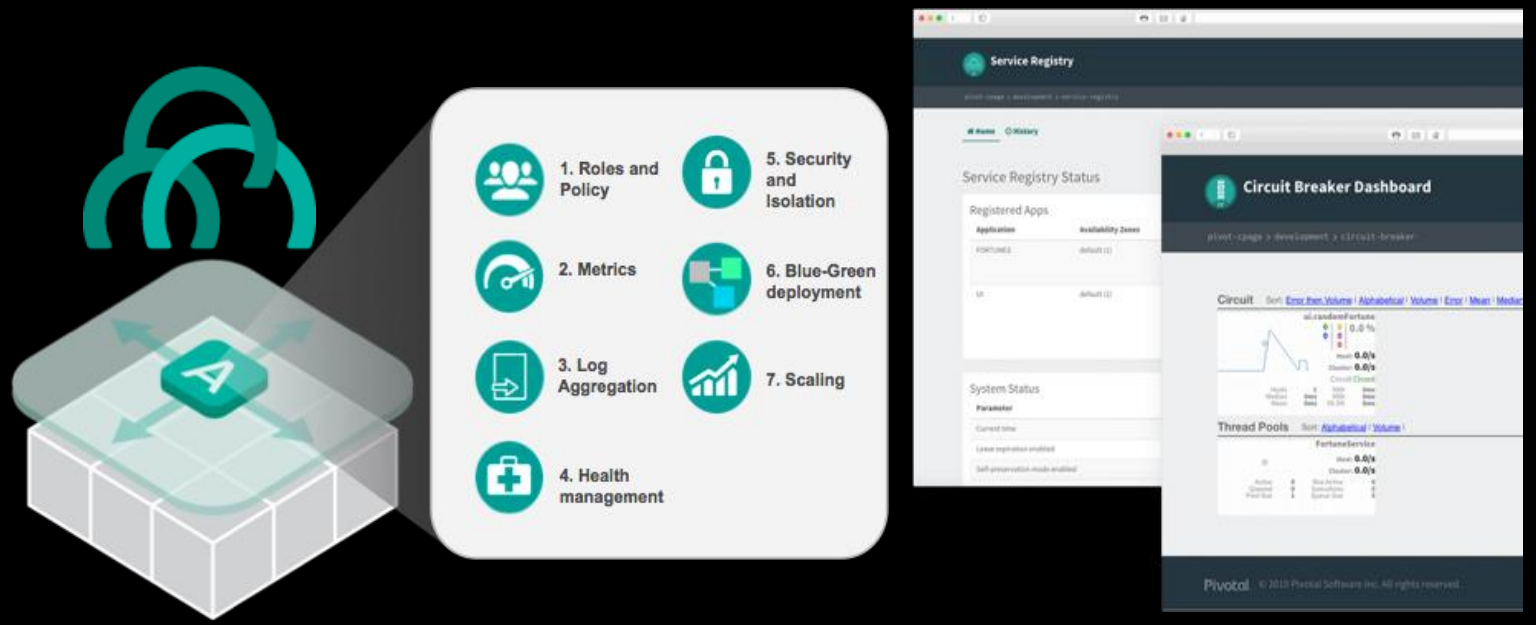
Cloud Native Platform

# Cloud Native Platform





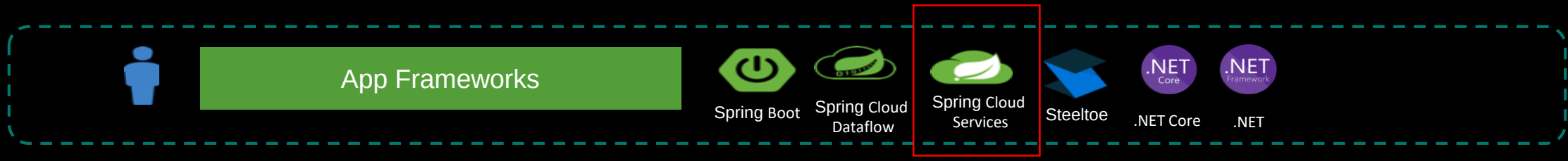

# Cloud Foundry
# Everything Needed to Deploy and Operate Cloud Native Applications



# Twelve Factor Applications – Platform Contract

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

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




# 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 using Spring Cloud Services and more in .NET based applications

# 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**

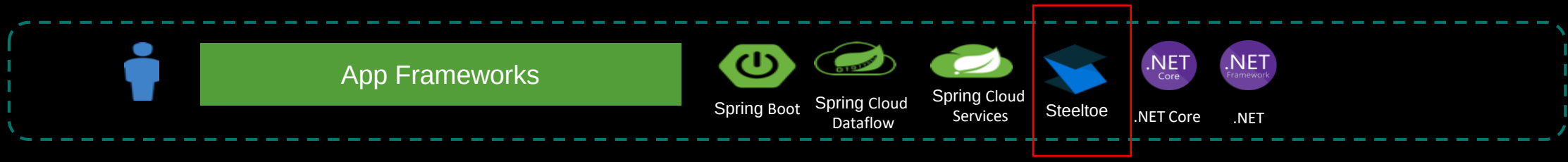
Dynamic, versioned propagation of configuration across lifecycle states without the need to restart your application

**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




# Facilitates Twelve-Factor Contract on .NET

Steeltoe
http://steeltoe.io

NETFLIX OSS

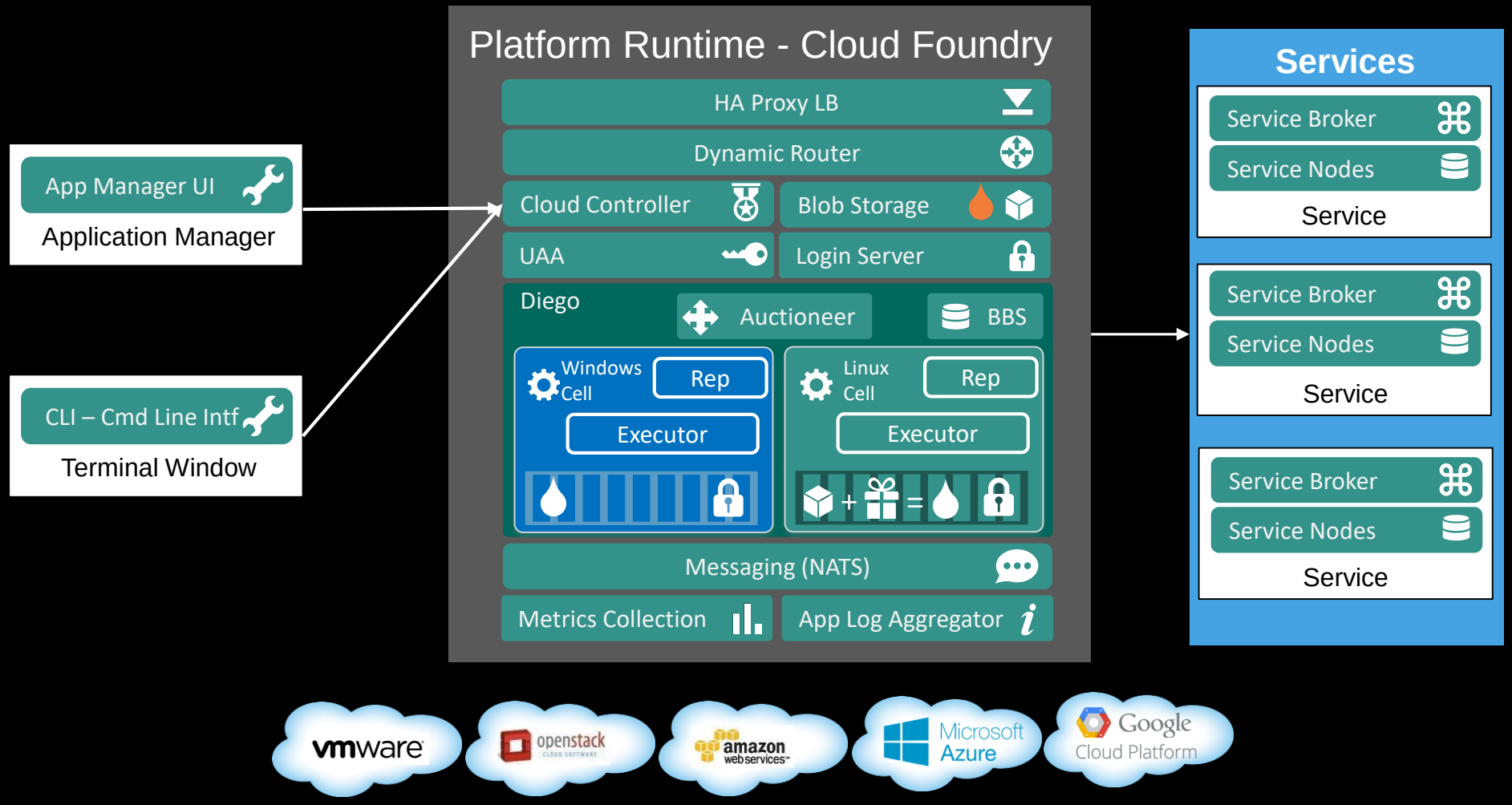
# Enabling Cloud Native Applications on .NET

- 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)
  - Management & Monitoring
- Simplifies using Spring Cloud Services
  - Configuration (e.g. Config Server, etc.)
  - Service Discovery (e.g. Netflix Eureka, etc.)
  - Circuit Breaker (e.g. Netflix Hystrix)
  - Distributed Tracing (e.g. Slueth coming)

# Pivotal Cloud Foundry Fundamentals

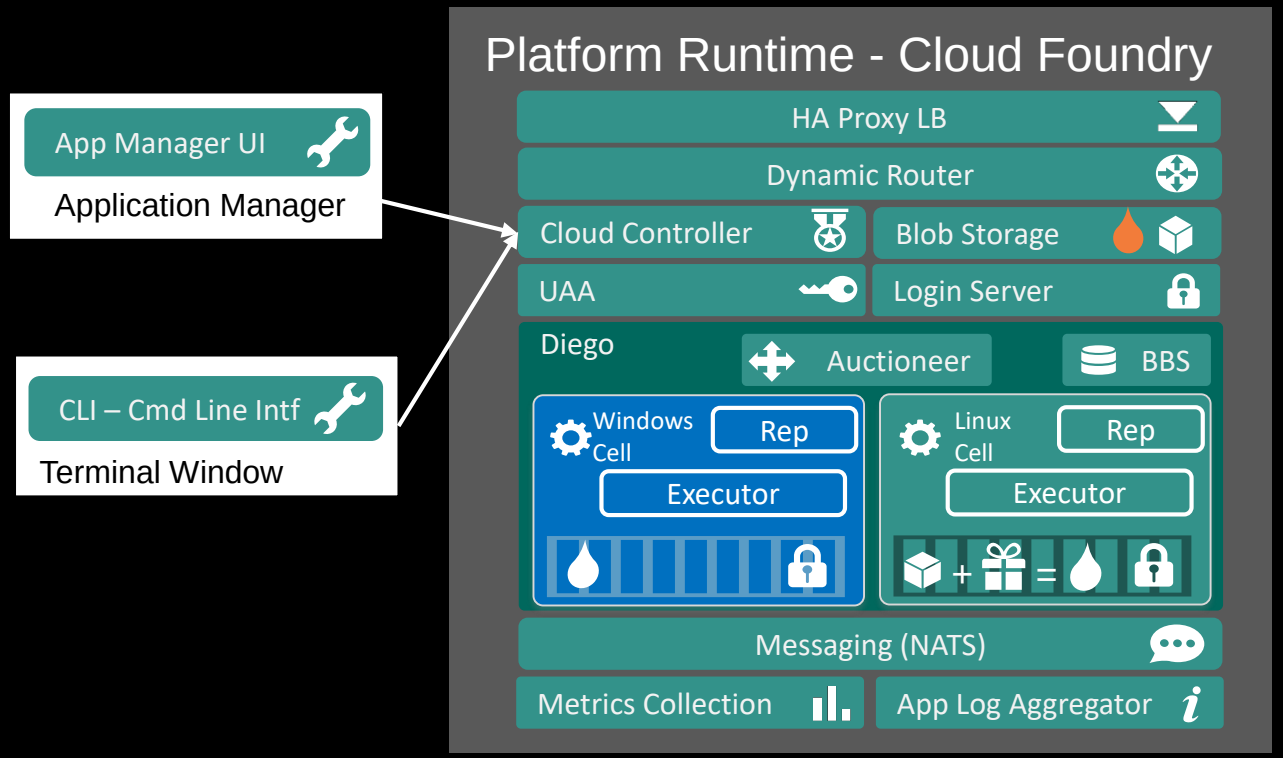
ORGS, SPACES, USERS, ROLES, CLI, API, APPS MANAGER

# Pivotal Cloud Foundry (PCF) Architecture



# Cloud Controller API

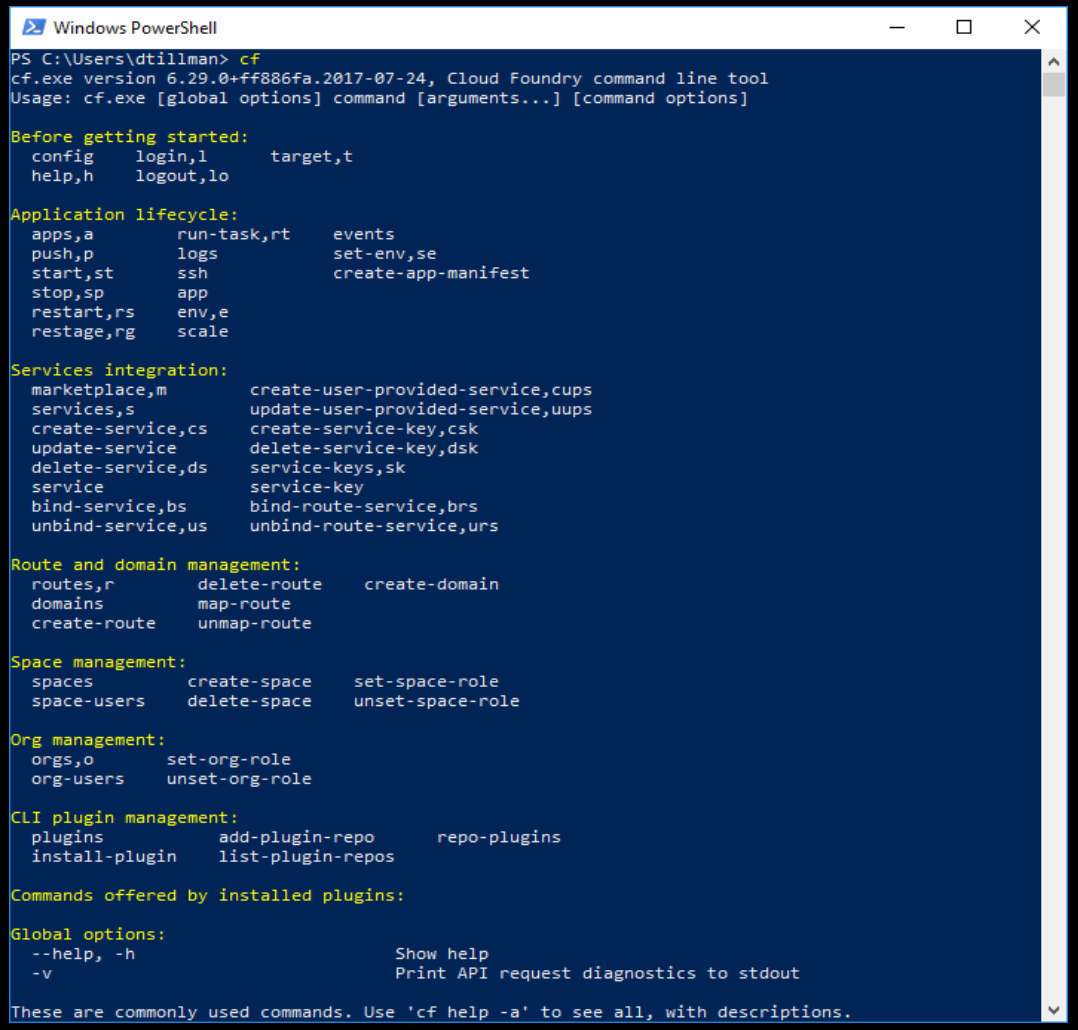
- Cloud Controller (CC) component of Cloud Foundry manages all APIs
- CF CLI and other clients like Apps Manager directly call this API
- Before accessing the CC API, you must get an access token from the User Account and Authentication (UAA) server
- http://apidocs.cloudfoundry.org



# CLI – Command Line Interface

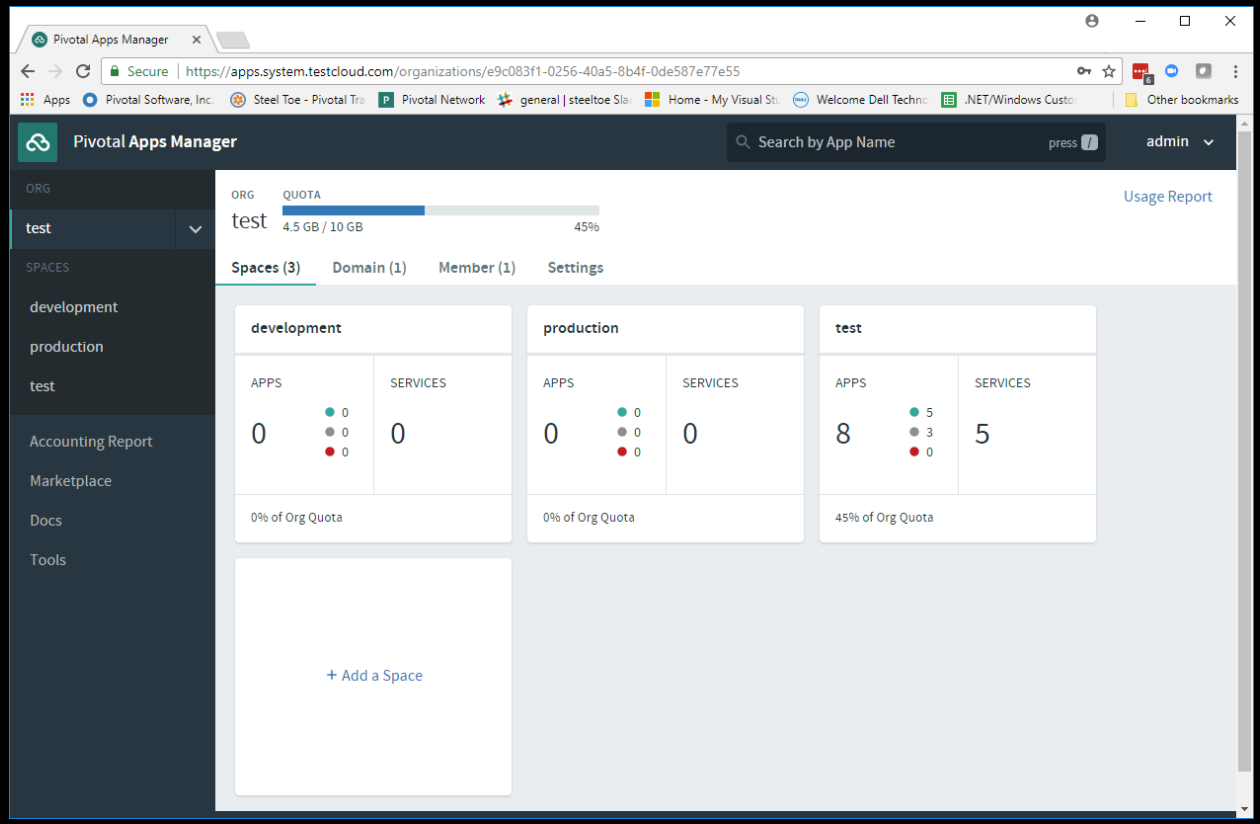
- Command line utility providing easy access to the Cloud Controller (CC) API.
- Scriptable
- Fully documented

```
cf help –a
cf help <command>
cf api http://foo.bar.com
cf login <username>
```

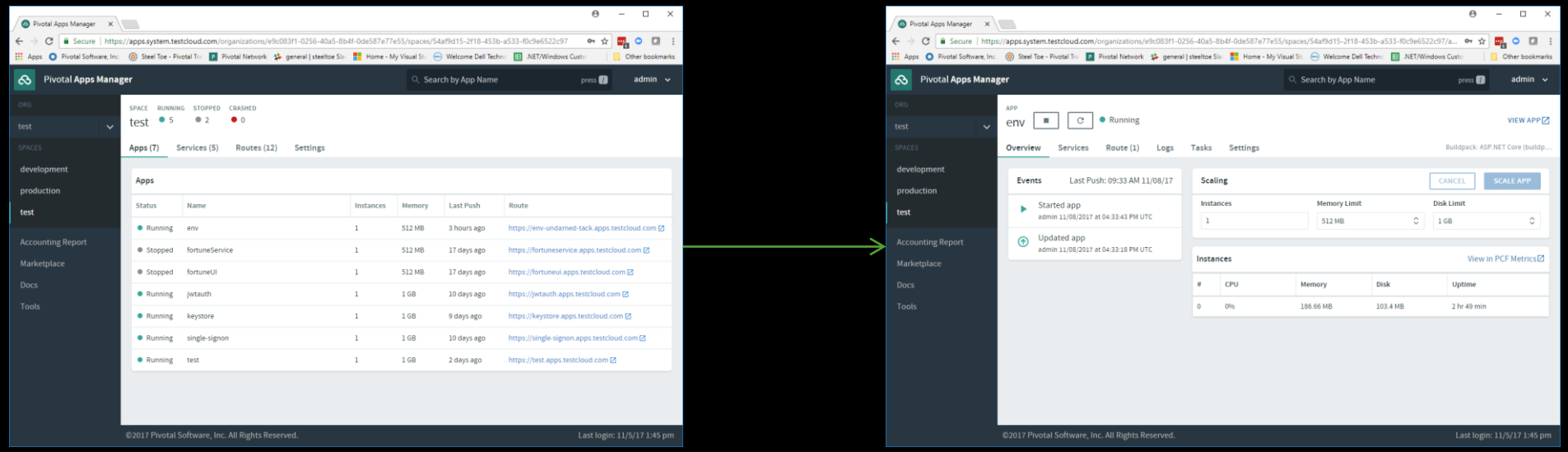


# Pivotal Apps Manager

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

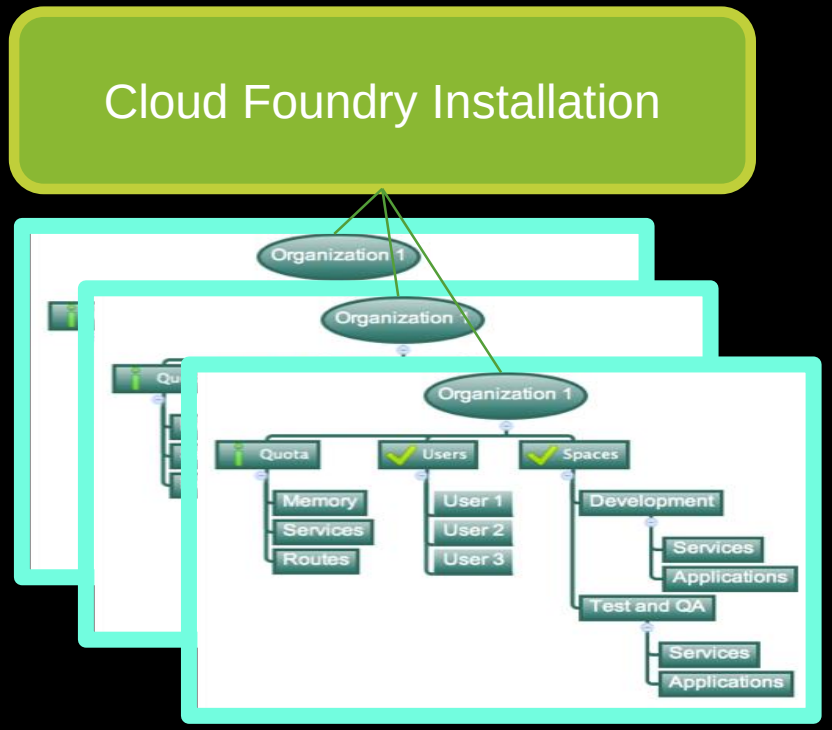


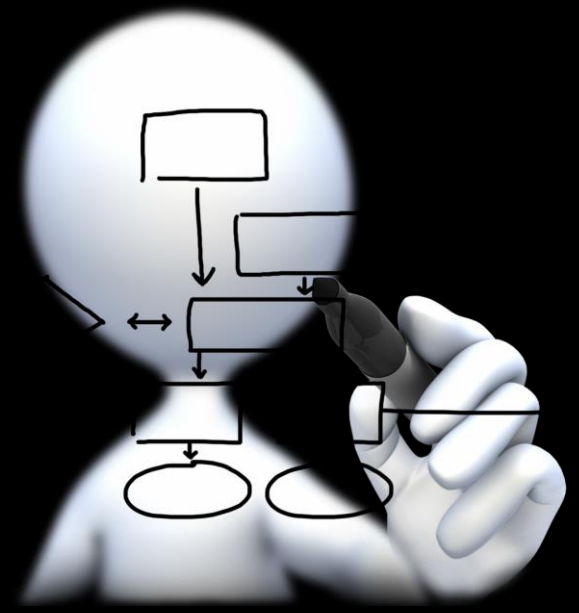
# Pivotal Apps Manager – App View




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

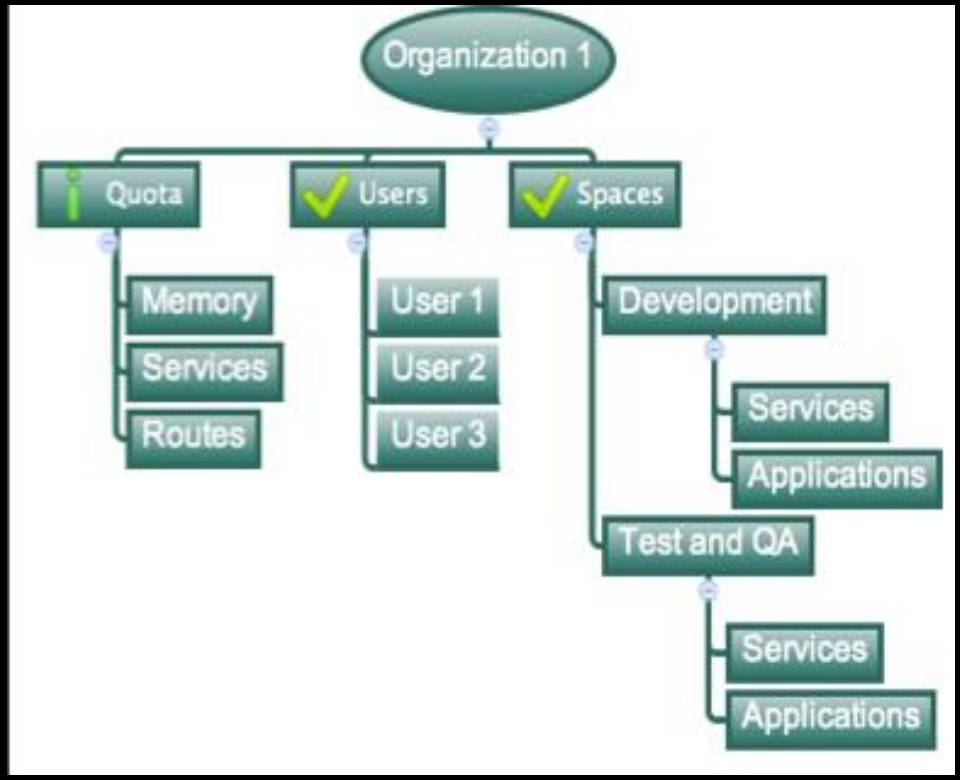
# Orgs, Spaces, Users and Quotas



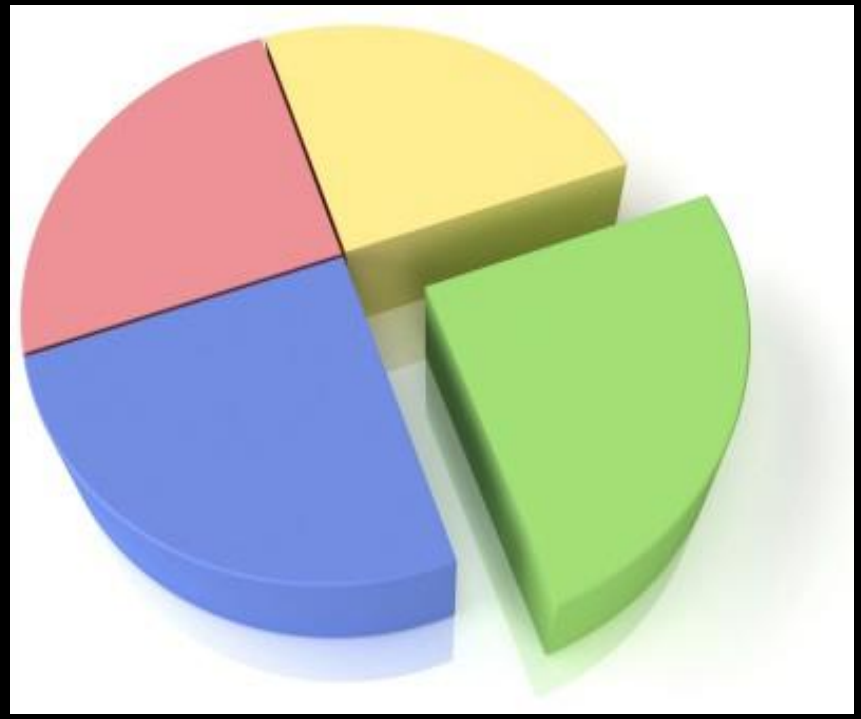
# Organizations

- 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



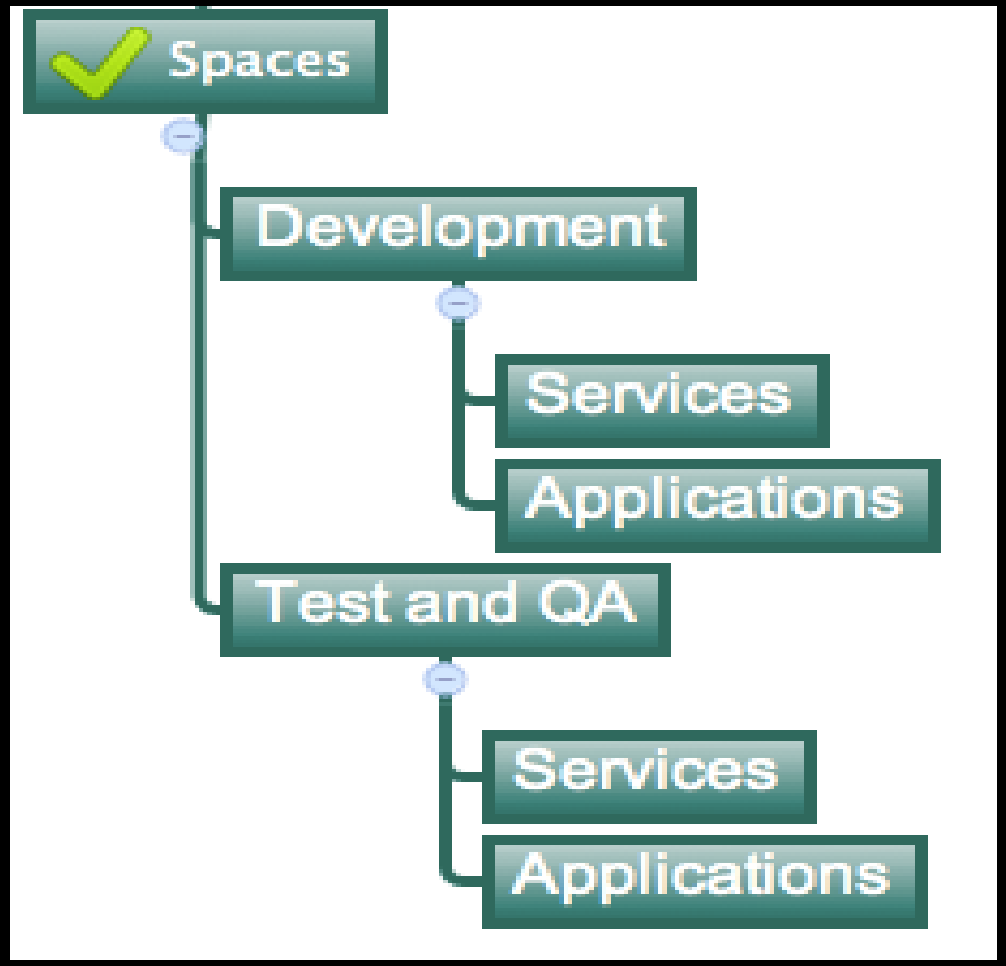
# Quotas

- 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
- Roles
  - Admin, Admin-R/O, Auditor, OrgMgr, OrgAuditor, OrgBillingMgr, OrgUser, SpaceMgr, SpaceDev, SpaceAuditor

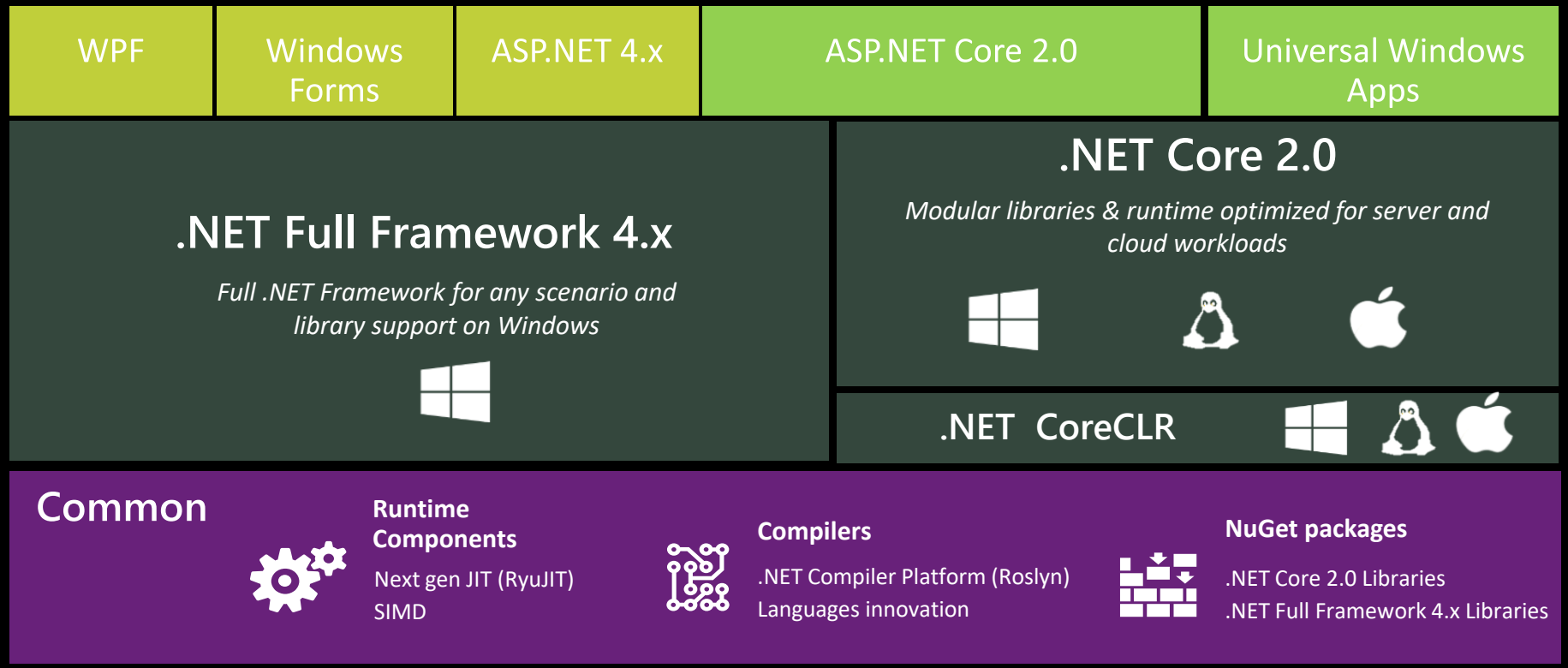
# Lab0

- Install and verify Workshop pre-requisites
- Verify access to the Workshop Cloud Foundry environment
  - Ensure we have the CLI installed and working
  - Verifying connectivity & credentials
    - Verify access to Cloud Foundry org and space
  - Use the CLI to access Cloud Foundry
    - cf target, cf login
  - Verify access to Cloud Foundry using the Apps Manager
- Follow Lab Description
  - https://github.com/SteeltoeOSS/Workshop/tree/master/Lab00

# .NET Core Fundamentals

.NET, .NET CORE, ASP.NET, ASP.NET CORE

# .NET Today


WPF
Windows Forms
ASP.NET 4.x
ASP.NET Core 2.0
Universal Windows Apps
.NET Full Framework 4.x
Full .NET Framework for any scenario and library support on Windows
.NET Core 2.0
Modular libraries & runtime optimized for server and cloud workloads
.NET CoreCLR
Common
Runtime Components
Next gen JIT (RyuJIT)
SIMD
Compilers
.NET Compiler Platform (Roslyn)
Languages innovation
NuGet packages
.NET Core 2.0 Libraries
.NET Full Framework 4.x Libraries

# .NET Core

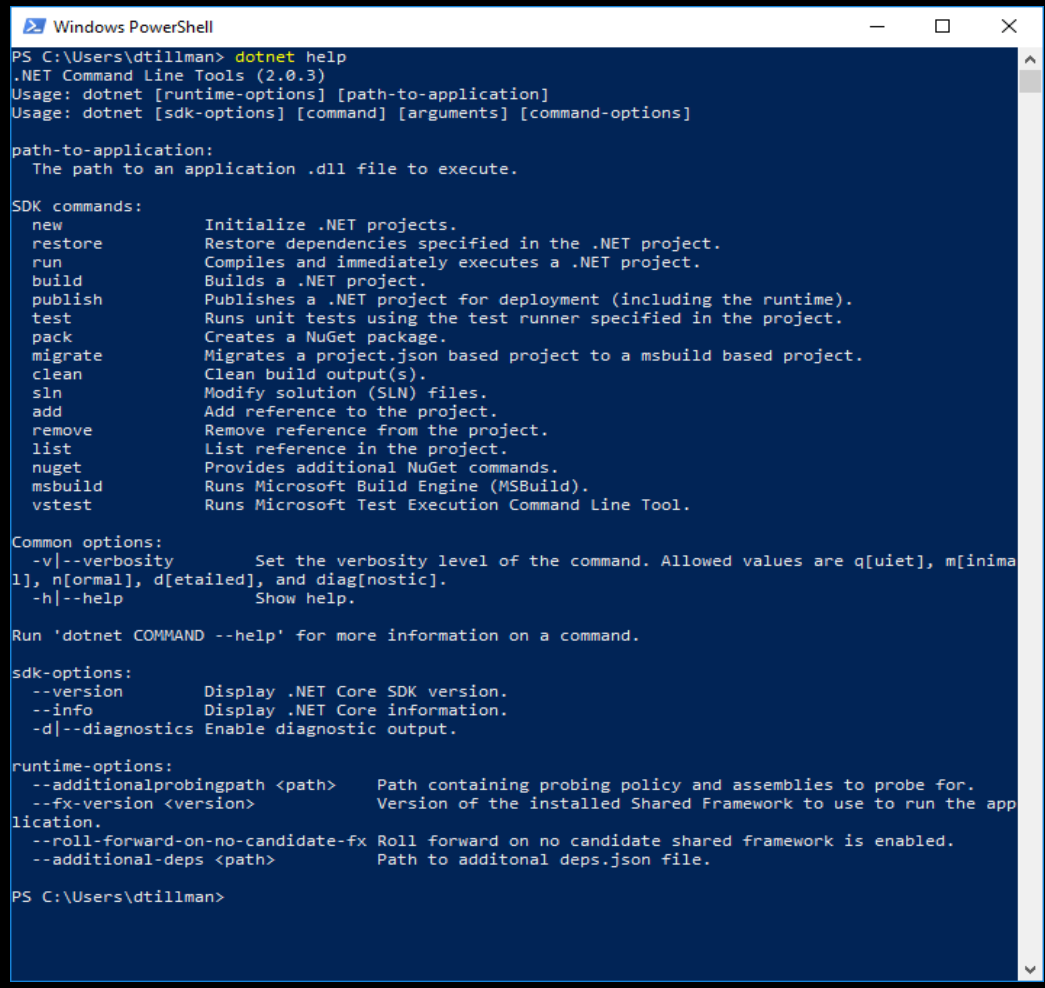
- Runs cross platform
  - Runtimes, Libraries and Compilers for Windows, Linux, OSX
  - .NET Core tooling for Windows, Linux, OSX
    - Command Line Interface (CLI) – ‘dotnet’
    - New `.csproj` files (MSBuild)
    - Cross platform Code editor – Visual Studio Code
    - Visual Studio 2017 - IDE (Windows 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
  - Binary = Runtime & Libraries only
  - SDK = Development Tools + Runtime & Libraries

# .NET Core cont'd

- Modular – built on NuGet packaging system
  - Packaged and distributed as several NuGets
    - CoreCLR, Libraries & Compilers, etc.
  - Packages include everything needed to run
    - Even native code dependencies
  - Enables 'a la carte' .NET development
- Application types
  - .NET Console applications
- .NET Core 2.0 implements .NET Standard 2.0
  - .NET 4.6.1 implements .NET Standard 2.0
    - .NET Core 2.0 API not necessarily supported xPlatform
- .NET Core != ASP.NET Core

# .NET Core SDK – Command Line

- Command Line Interface - dotnet
  - `new` - create a new project
  - `restore` – restore dependencies
  - `build` – compile code to assembly
  - `run` – compile (and run the assembly
  - `publish` – package app and dependencies



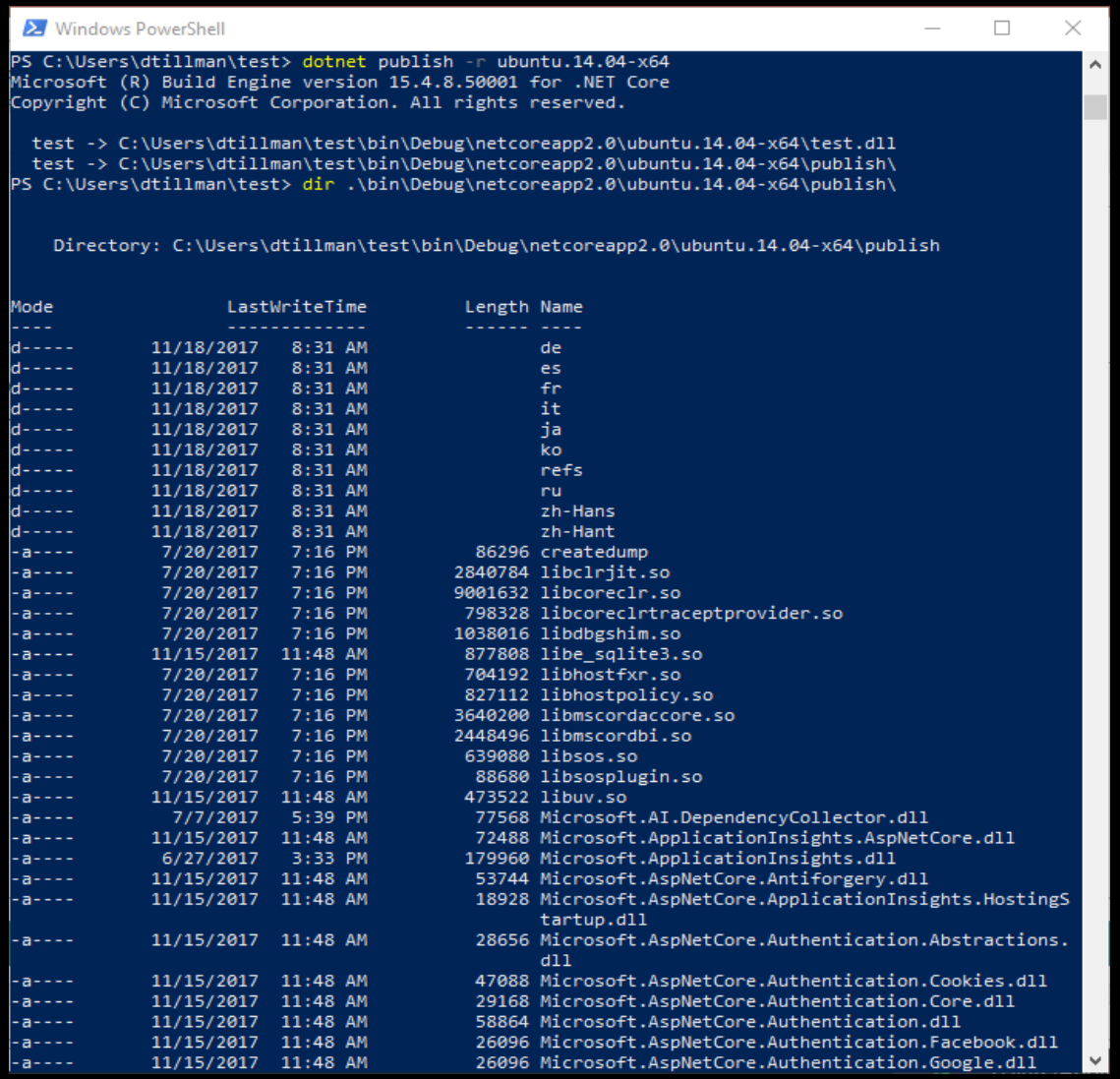
# .NET Core SDK – Project file

- Project file – new simpler `.csproj`
  - `<TargetFramework>` - specifies the target frameworks for application or library
    - Specified via a TFM (Target Framework Moniker)
      - net461, netcoreapp2.0, netstandard2.0)
  - `<PackageReference>` - specifies package dependencies
  - `<RuntimeIdentifiers>` - specifies the runtimes the application or library supports
    - Specified via RIDs (Runtime Identifiers)
      - win7-x64, ubuntu.14.04-x64, osx.10.11-x64

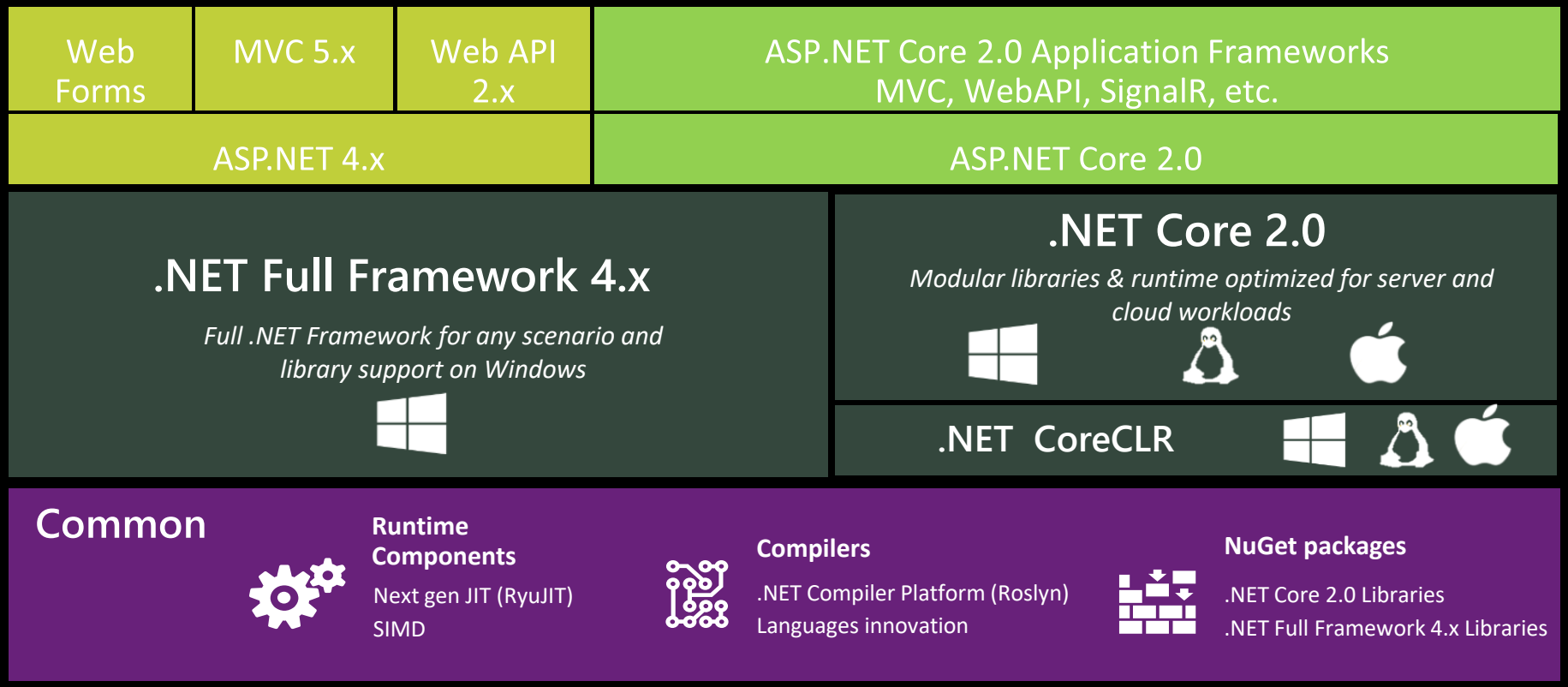
```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <Folder Include="wwwroot\" />
  </ItemGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.3" />
  </ItemGroup>
</Project>
```

# .NET Core SDK - Publishing

- Application deployment options (i.e. `dotnet publish`)
  - Framework dependent deployment (FDD)
    - Application = App dependencies + App code
      - .NET Full Framework
      - .NET Core
  - Self-contained deployment (SCD)
    - Application = .NET runtime + App dependencies + App code
      - .NET Core



# ASP.NET Today



# ASP.NET Core

- Runs cross framework - (e.g. target netcoreapp2.0 or net461)
  - .NET Core – Windows, Linux & OSX
  - .NET Framework 4.x - Windows
- Fully open source - https://github.com/aspnet
- Modular
  - Packaged and distributed as lots of multi-target NuGets (.NET Core & .NET Full Framework)
    - Host, Server, Configuration, Dependency Injection, Session, Static Files, etc.
    - Enables ‘pay for play’ ASP.NET configuration
  - Aggregator package pulls in everything
    - <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.3" />

# ASP.NET Core cont'd

- Use .NET Core CLI – 'dotnet <command> <args>'
  - build, run, publish, etc.
- Deployment options when targeting .NET Core
  - FDD - Application can be portable
    - Application = .ASP.NET Core NuGets + App dependencies + App code
  - SCD - Application can be fully self-contained
    - Application = .NET Core NuGets + ASP.NET Core NuGets + App dependencies + App code

# ASP.NET Core cont'd

- Hosting – self hosting
  - It's just a console application
  - Kestrel – a cross platform web server & is default one used
  - HttpSysServer – Windows only web server (built on http.sys)
  - IIS – used as reverse-proxy with Kestrel
    - ACM – IIS module used
- Dependency Injection (DI) baked in
- Middleware handles request processing
  - Routing, Static Files, Session, Authentication, Authorization, etc.

# ASP.NET Core

- Configuration – separate application config processing
  - Multiple configuration sources (i.e. JSON, INI, XML files, Environment Variables, etc.)
- Startup class
  - Configures the Service Container (i.e. DI)
  - Configures the Middleware
- Program
  - Builds & configures the Web Host ( i.e. Web Server, Logging, etc.)
  - Builds applications configuration

# ASP.NET Core

- ASP.NET Core Application frameworks layered on top
  - MVC – Integrated UI and Web API framework
    - Uses DI, Logging, Self-hosting, etc.
    - Adds Controllers, Actions, Views, Filters, Model binding
    - Razor Views & new Tag Helpers
  - WebSockets
  - SignalR

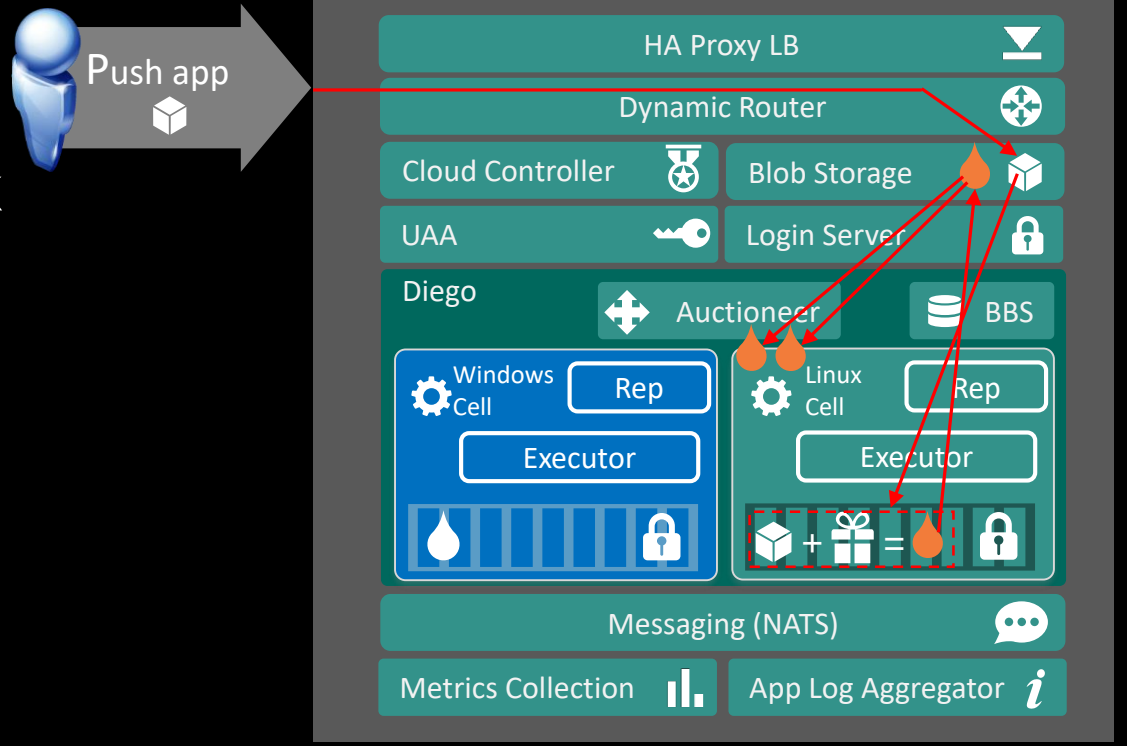
# Running .NET Applications on Pivotal Cloud Foundry

CF PUSH, MANIFEST, STAGING, BUILD PACKS, CONTAINERS, CELLS, ENVIRONMENT VARIABLES, VCAP_APPLICATION

# Running an App on Pivotal Cloud Foundry (PCF)

1. Upload app bits using metadata from manifest to blob store
2. Bind services – *covered in next section*
3. Stage application & apply buildpack
4. Save staged application image (i.e. droplet )
5. Deploy & run image in container
6. Manage applications health

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



# Manifest Files – Application Metadata

- Application manifests tell cf push what to do with applications
- What OS stack
  - Linux default – cflinuxfs2
  - Windows – windows2012R2
- How many instances to create and how much memory to use
- Helps automate deployment, including multiple apps at once
- Can list services to be bound to the application
- Command line to use to start the application

```
---
applications:
- name: fortuneui
  random-route: true
  buildpack: binary_buildpack
  memory: 512M
  stack: windows2012R2
  command: cmd /c .\Fortune-Teller-UI --server.urls http://*:%PORT%
  env:
    ASPNETCORE_ENVIRONMENT: Production
  services:
   - myConfigServer
   - myDiscoveryService
   - myRedisService
   - myHystrixService
   - myOAuthService
```

```
---
applications:
- name: fortuneService
  random-route: true
  buildpack: dotnet_core_buildpack
  env:
    ASPNETCORE_ENVIRONMENT: Production
  services:
   - myConfigServer
   - myDiscoveryService
   - myMySqlService
   - myOAuthService
```

# Staging an Application – Applying Buildpack

- Build container images (i.e. droplets)
  - Saved in Blobstore
- Takes care of
  - Detecting which type of application is being pushed
  - Installing the appropriate run-time if needed
  - Installing required dependencies or other artifacts if needed
  - Creating the command used to start the application
- Lots of Build packs
  - Staticfile, Java, Ruby, Nodejs, Go, Python, PHP, .NET Core, Binary, HWC
- Configurable on Cloud Foundry
  - cf buildpacks, cf create-buildpack, cf update-buildpack, cf delete-buildpack, etc.

# 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
- Three buildpacks used for staging .NET applications
  - .NET Core – `dotnet_core_buildpack`
  - Binary - `binary_buildpack`
  - .NET HWC - `hwc_buildpack`

# .NET Core Buildpack

- Used to create container images ready to run .NET Core applications on Linux cells
  - Target Stack: cflinuxfs2
  - Supports pushing two types of directories
    - Source – project source code
    - Binaries – `dotnet publish` output
- Pushing source
  - Must contain `.csproj`
  - Installs .NET Core runtime – version specify via global.json, else build pack chooses
  - Restores application dependencies
- Pushing binaries – two types of binary directories supported
  - Must NOT contain `.csproj`
  - FDD directory - Portable .NET Core application
    - Installs .NET Core runtime – version specified via global.json, otherwise build pack chooses
  - SCD directory - Self-contained .NET Core application
    - Installs Linux dependencies (i.e. libunwind.so)

```
---
applications:
- name: env
  random-route: true
  memory: 1G
  stack: cflinuxfs2
  buildpack: dotnet_core_buildpack
```

# Binary Buildpack

- Used to create container images ready to run .NET Core applications on Windows cells
  - Target Stack: Windows2012R2
  - Supports pushing
    - Binaries – `dotnet publish` output
  - Copies image, as is, no additional dependencies added
  - Provide shell command to be used to start the application
- Pushing binary – two types of binary directories supported
  - FDD directory - Portable .NET Core application
    - Would require the .NET Core runtime to have been installed on the Windows machine
  - SCD directory - Self-contained .NET Core application

```
---
applications:
- name: fortuneService
  random-route: true
  buildpack: binary_buildpack
  memory: 512M
  stack: windows2012R2
  command: cmd /c .\Fortune-Teller-Service --server.urls http://*:%PORT%
```

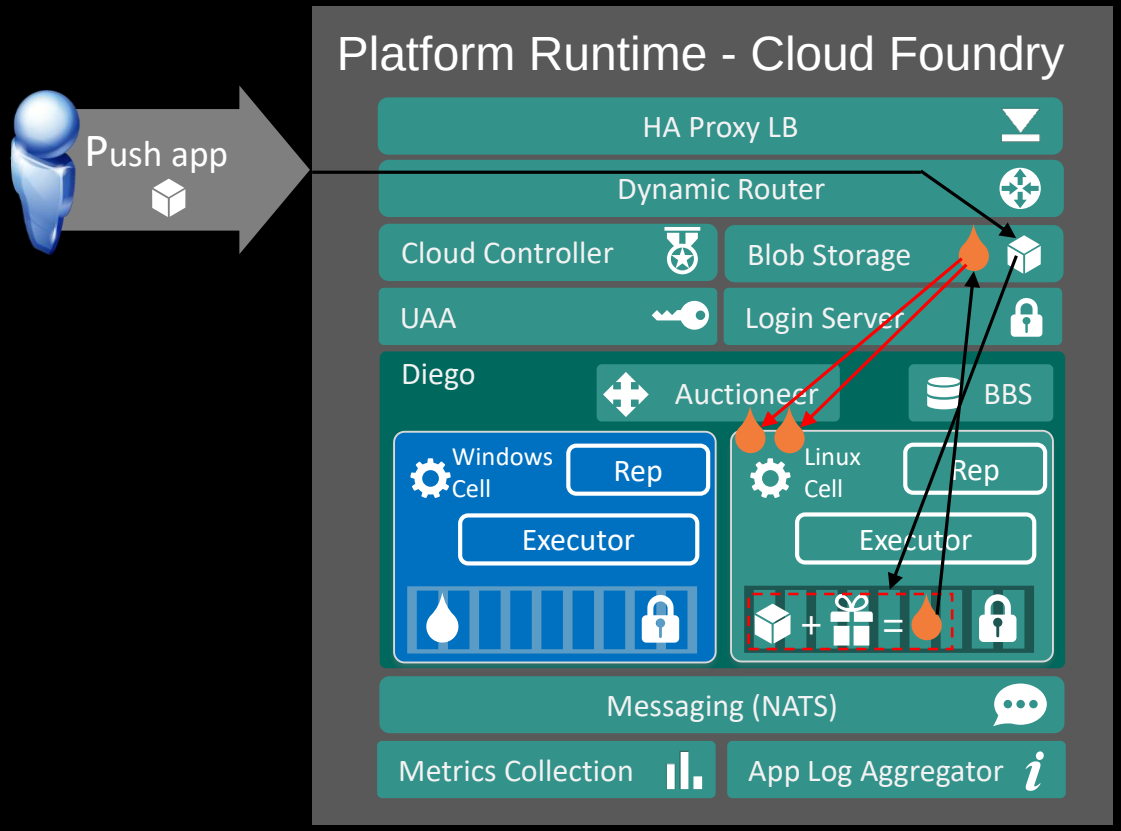
# .NET Hosted Web Core (HWC) Buildpack

- Used to create container images ready to run ASP.NET/IIS applications on Windows cells
  - Target Stack: Windows
  - Buildpack ensures web.config is present
  - Installs `hwc.exe` and configures application to launched under it
    - Runs the application as a “Windows Hosted Web Core” application

```
---
applications:
- name: env
  random-route: true
  memory: 1G
  stack: windows2012R2
  buildpack: hwc_buildpack
```

# Deploying Image to Containers in Cells

- Diego Container management handles deployment
  - Auction process determines what cells are selected
    - Stack requirements (i.e. Windows/Linux)
    - Cells load and resources
    - Availability requirements
  - Containers are created in each cell
    - Logs streamed to log system
  - Droplet image downloaded to cell and started in container
  - When instance health is good, Router notified of instance

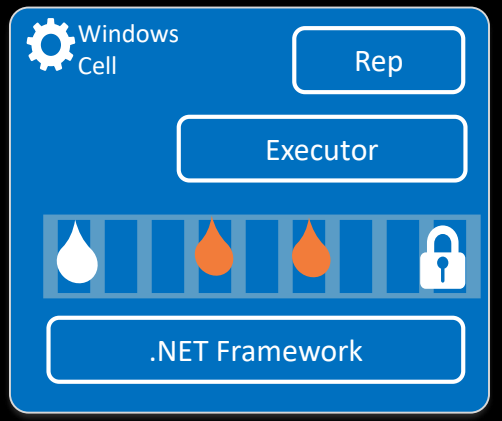


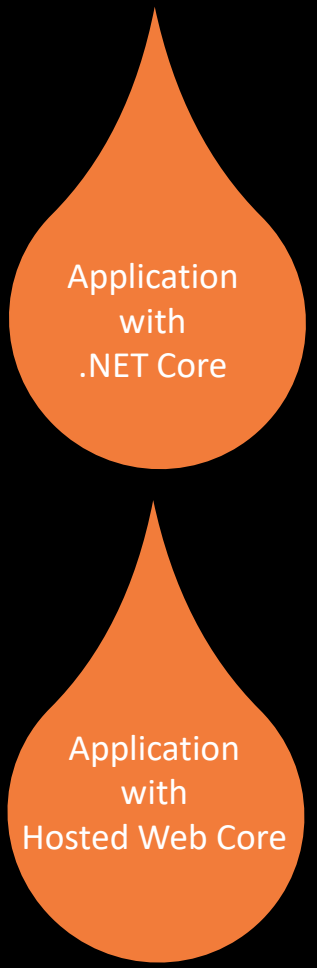
# Why Containers

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

# .NET Application Images on Windows Cells

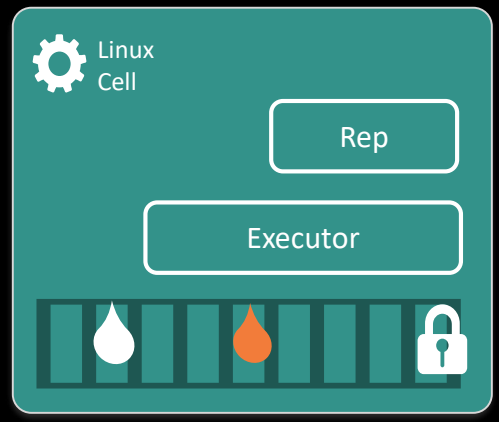
- Run container images prepared by either
  - Binary buildpack
  - HWC buildpack
- Application types supported
  - .NET Full “Background processes”
    - Command line/Console apps
  - ASP.NET 4 applications
    - MVC, WebForm, WebAPI, WCF
  - .NET Core “Background processes”
    - Command line apps/Console apps
  - ASP.NET Core web apps
    - .NET Full Framework
    - .NET Core

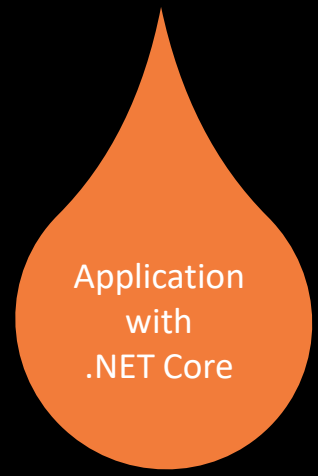


# .NET Application Images on Linux Cells

- Run container images prepared by
  - .NET Core buildpack
- Application types supported
  - .NET Core “Background processes”
    - Command line apps/Console apps
  - ASP.NET Core web apps
    - .NET Core only



# Cloud Foundry Container Environment Variables

- Used to communicate application environment & configuration to deployed containers
  - 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 Example


```
"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"
}
```

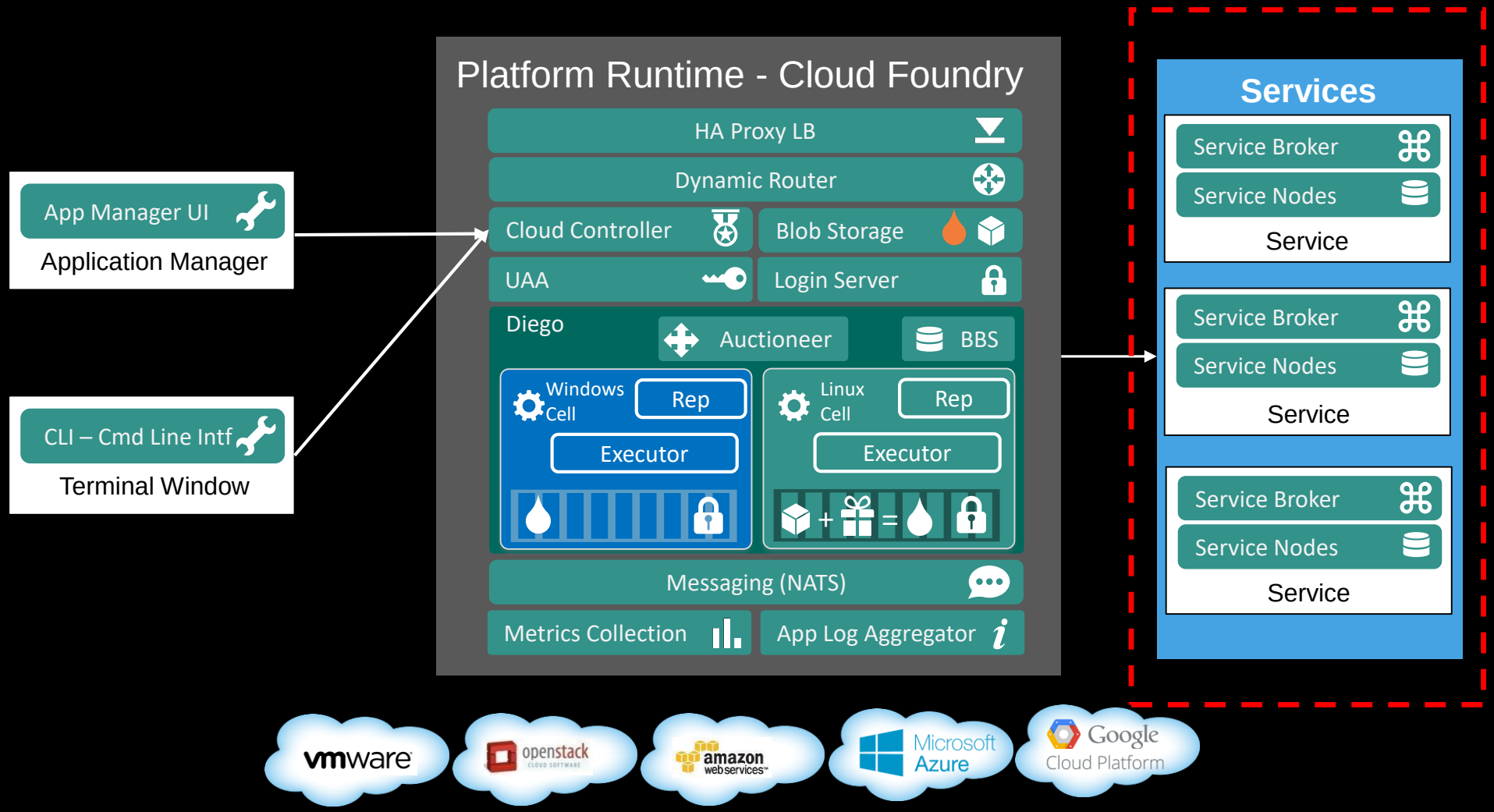
# Lab1 – Push ASP.NET Core Application to Cloud Foundry

- Push ASP.NET Core MVC application to help understand
  - Cloud Foundry's support for .NET and .NET Core applications
  - Cloud Foundry processing during a push:
    - Staging, Droplets, Manifest files, Buildpacks, Cells, Containers, Container Environment variables (e.g. VCAP_APPLICATION )
  - Usage of dotnet CLI to build and publish application for Cloud Foundry
  - Usage CF CLI: push, stop, delete, restart, a, apps
  - Usage of Steeltoe CloudFoundry Configuration Provider
    - Used to parse VCAP_APPLICATION
- Follow Lab Description
  - Description: https://github.com/SteeltoeOSS/Workshop/tree/master/Lab01
  - Code: /Workshop/Lab01/Lab01.sln

# Using Services on Pivotal Cloud Foundry

MANAGED, USER-PROVIDED, SERVICE BROKERS, INSTANCE CREATION, APPLICATION BINDING, ENVIRONMENT VARIABLES, VCAP_APPLICATION

# Pivotal Cloud Foundry (PCF) Architecture



# 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

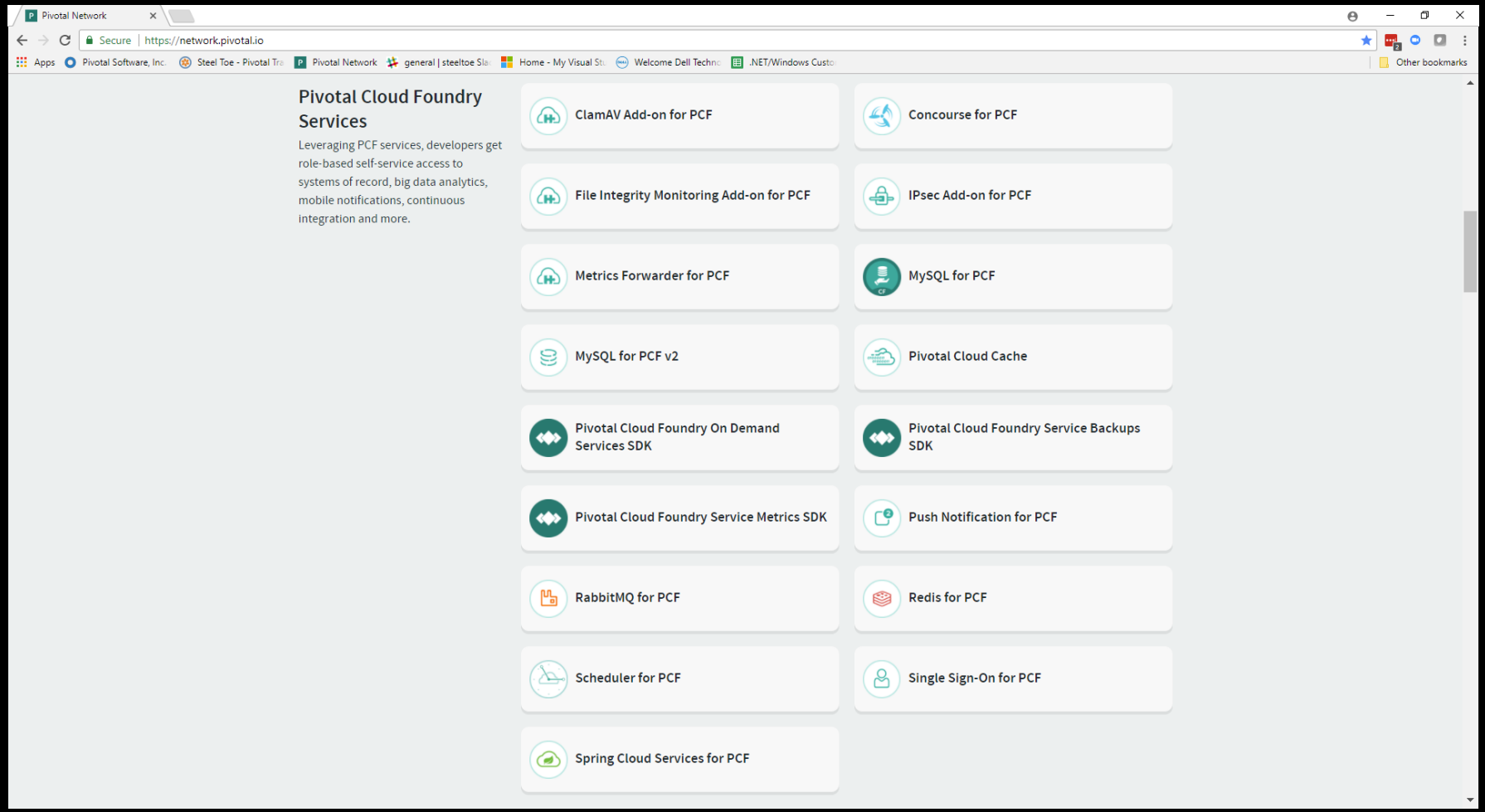
# Types of Services

- Managed - Fully integrated, with fully lifecycle management
  - Part of the market place of services
- User-Provided – Created and managed external to the platform

# Pivotal Managed Services

Pivotal Network

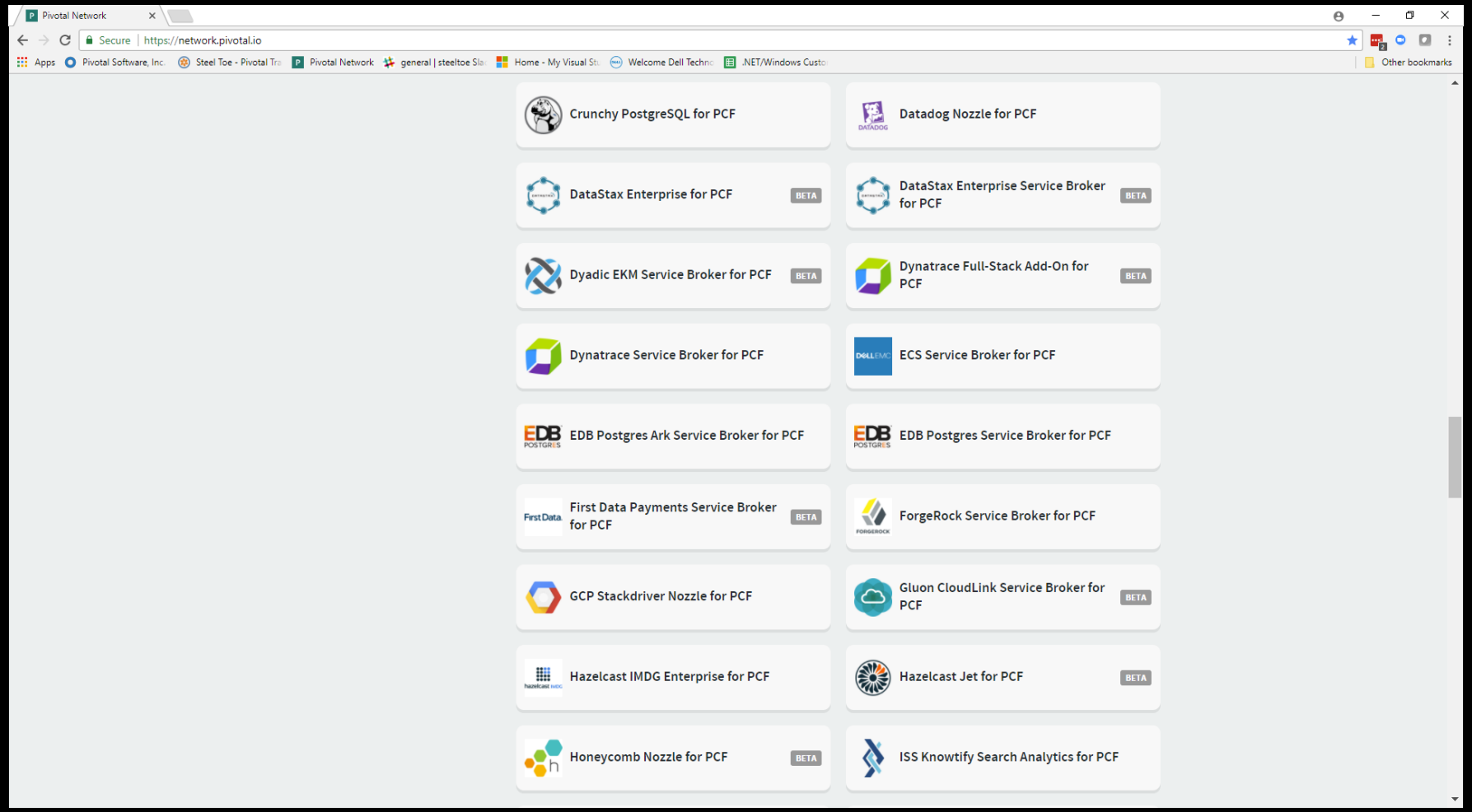
https://network.pivotal.io/



# Third Party Managed Services

Pivotal Network

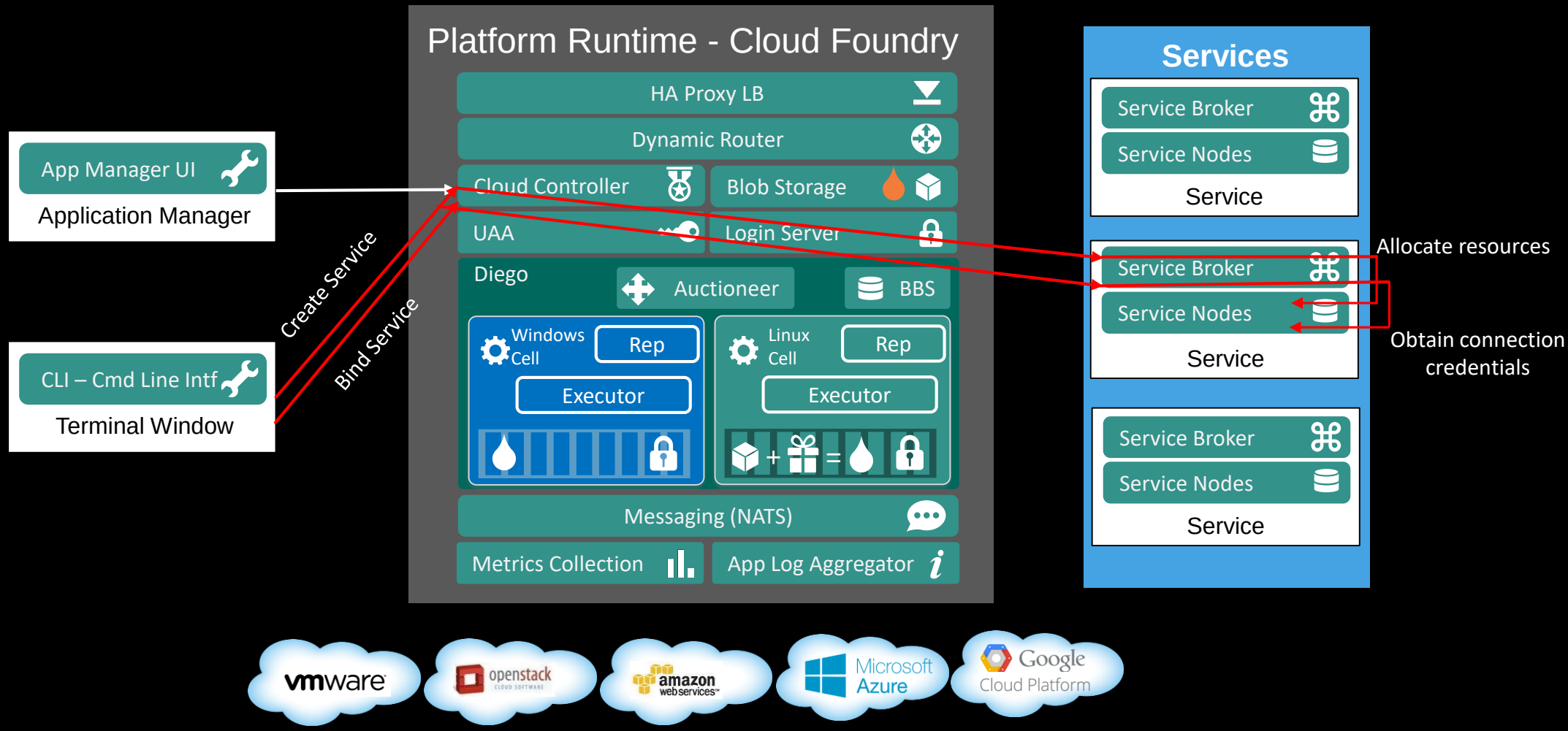
https://network.pivotal.io/



# Managed Services

- Integrated with Cloud Foundry
  - Services integrated with Cloud Foundry must implement the Service Broker API
  - Cloud Controller (CC) manages services using the API
- Service Brokers implement the API
  - Advertise a catalog of service offerings and service plans
  - Create service instances
  - Bind applications to service instances
  - Unbind applications from service instances
  - Delete service instances

# Creating and Binding Services



# User Provided Services

- Service instances managed outside of Cloud Foundry
- Behave like other service instances once created
- Familiar CLI commands provide service instance configuration
  - cf create-user-provided-service ….

EXAMPLE: AN ORACLE DATABASE MANAGED OUTSIDE OF CLOUD FOUNDRY

# Cloud Foundry Container Environment Variables

- Used to communicate application environment & configuration to deployed containers
  - 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 Example


```
"VCAP_SERVICES": {
   "p-config-server": [
    {
     "credentials": {
      "uri": "https://config-bd112dd4-9870-4819-b9a6-62eb3311e27b.apps.testcloud.com",
      "client_secret": "X3e2gKs5Oqhp",
      "client_id": "p-config-server-5f0d1211-75f1-4105-9f94-7ec010de2d3a",
      "access_token_uri": "https://p-spring-cloud-services.uaa.system.testcloud.com/oauth/token"
     },
     "syslog_drain_url": null,
     "volume_mounts": [],
     "label": "p-config-server",
     "provider": null,
     "plan": "standard",
     "name": "myConfigServer",
     "tags": [
      "configuration",
      "spring-cloud"
     ]
    }
   ],
   "p-service-registry": [
.....
```

# Lab2 – Binding Services to ASP.NET Core Application

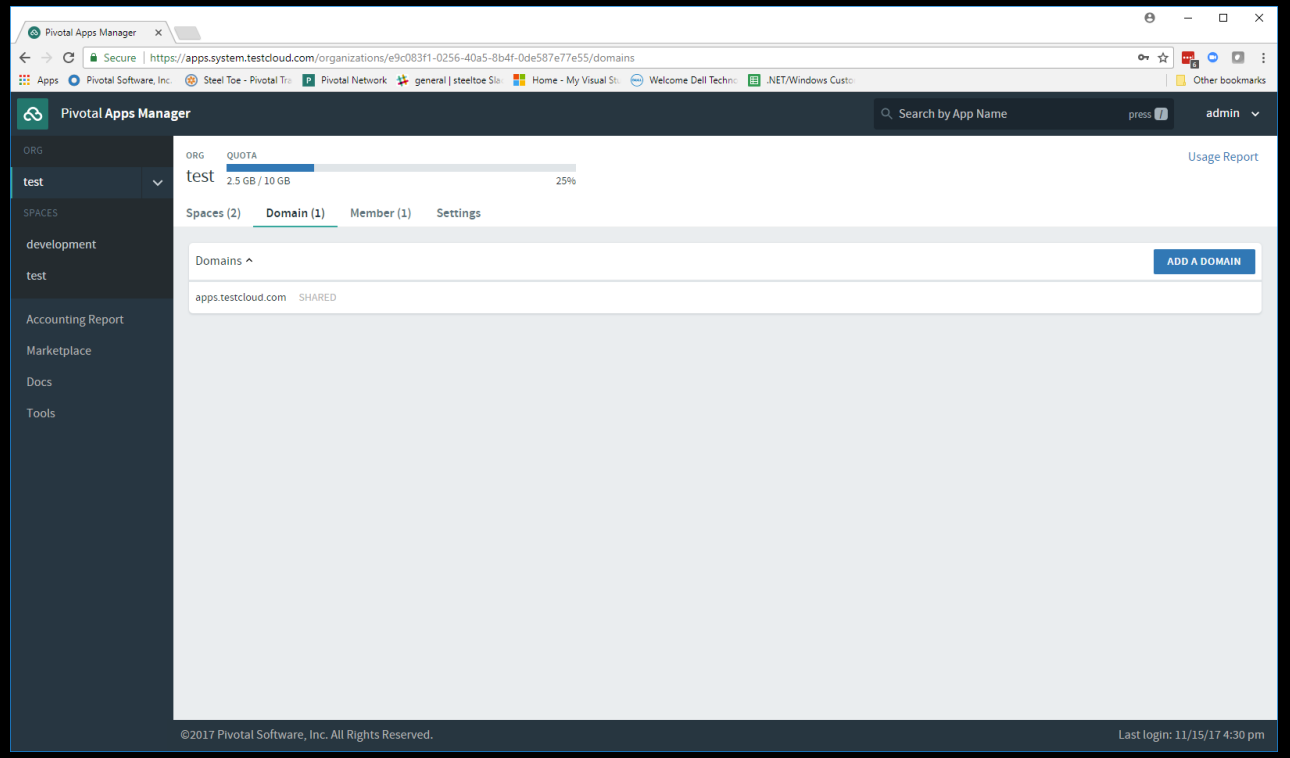
- Use same ASP.NET Core MVC application to understand
  - Cloud Foundry services, both Managed and User-Provided
  - Cloud Foundry processing during a service creation and binding
    - Service Marketplace, Service Brokers, Service instance, Service binding, Container Environment variables (e.g. VCAP_SERVICES )
  - Usage of CF CLI: create-service, delete-service, bind-service, unbind-service, create-user-provided-service, restage,etc.
  - Usage of Steeltoe CloudFoundry Configuration Provider
    - Used to parse VCAP_SERVICES
- Follow Lab Description
  - Description: https://github.com/SteeltoeOSS/Workshop/tree/master/Lab02
  - Code: /Workshop/Lab01/Lab01.sln

# Scaling and Operating Applications on Cloud Foundry

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

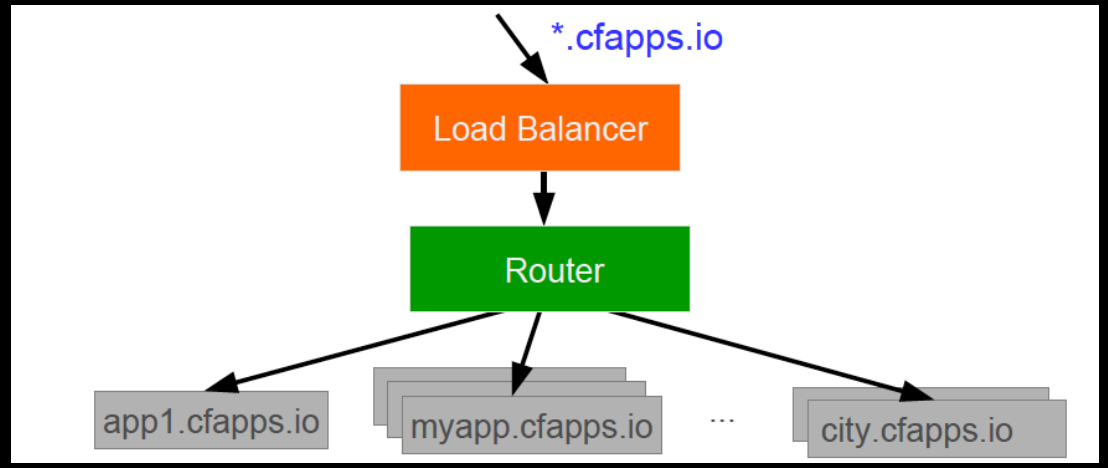
# Cloud Foundry Domains

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



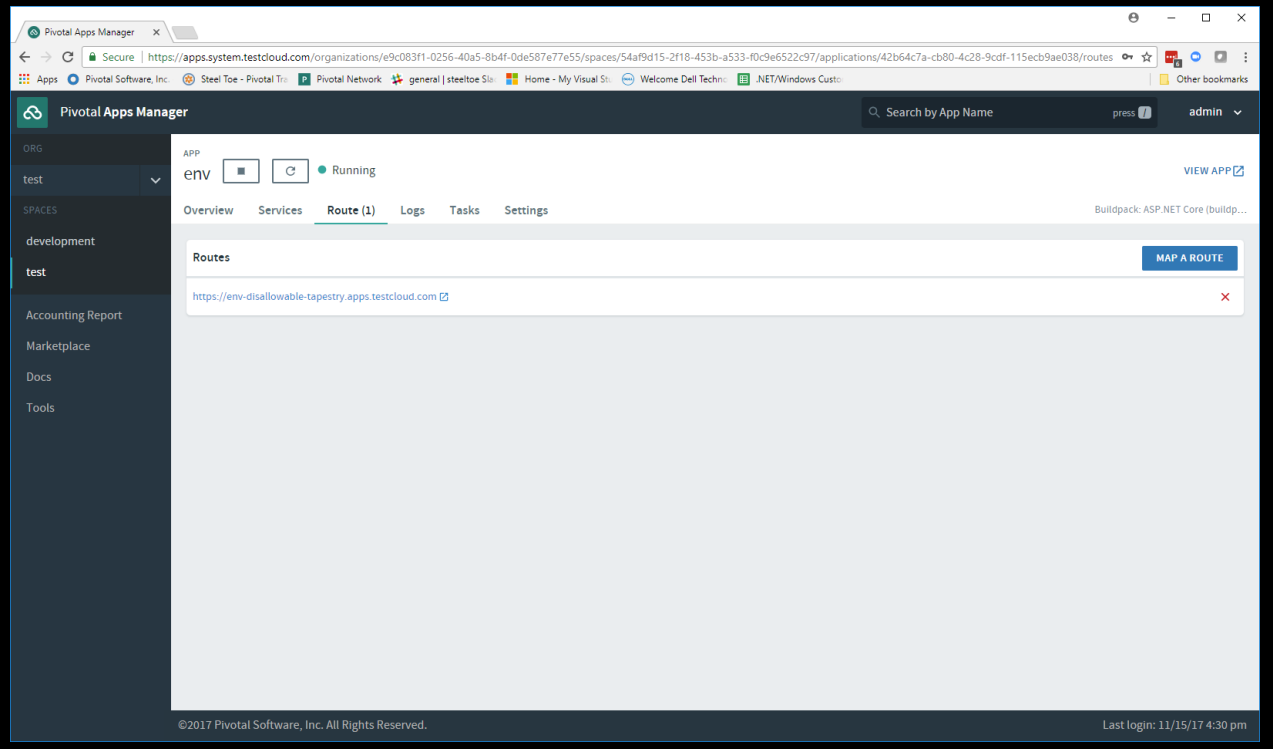
# Domains – Behind the Scenes

- 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)

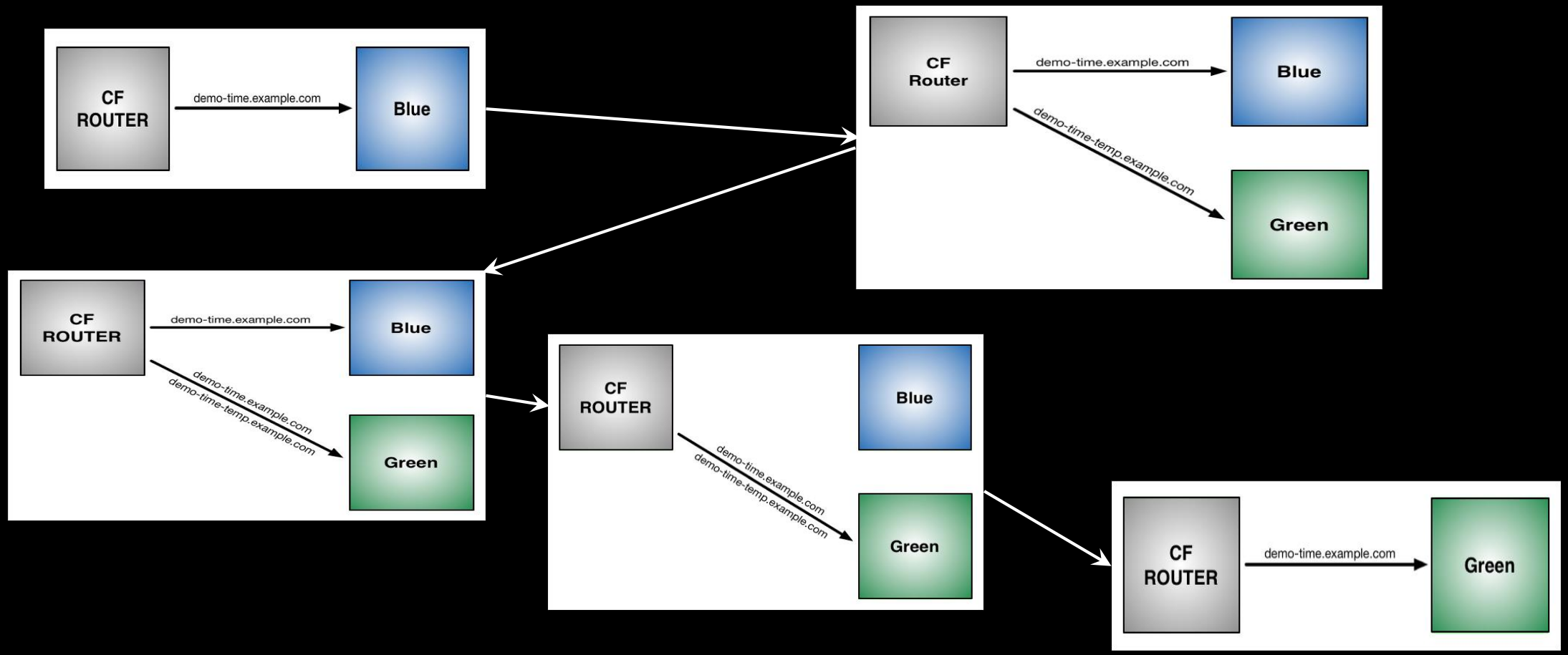


# Cloud Foundry Routes

- HTTP requests are routed to apps by associating a Route with the application
  - Route = Hostname + Domain
- 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



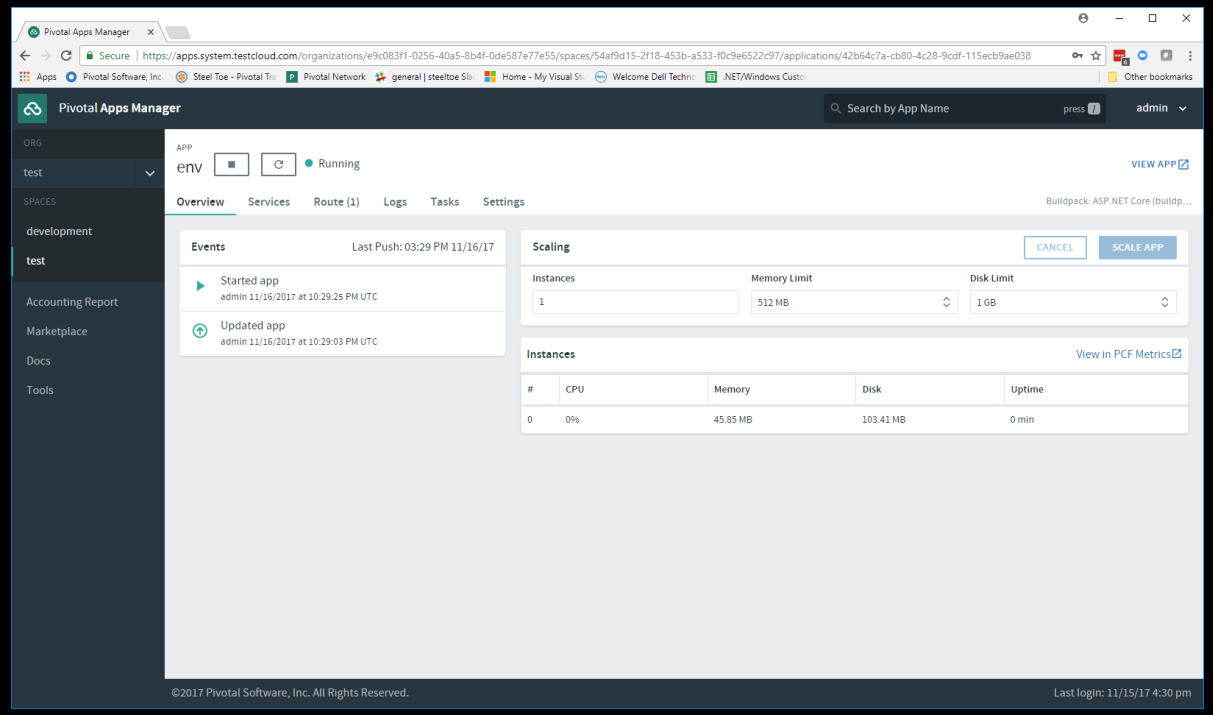
# Blue-Green Deployments




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

# Scaling

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



# Lab3 – Scaling Applications on Cloud Foundry

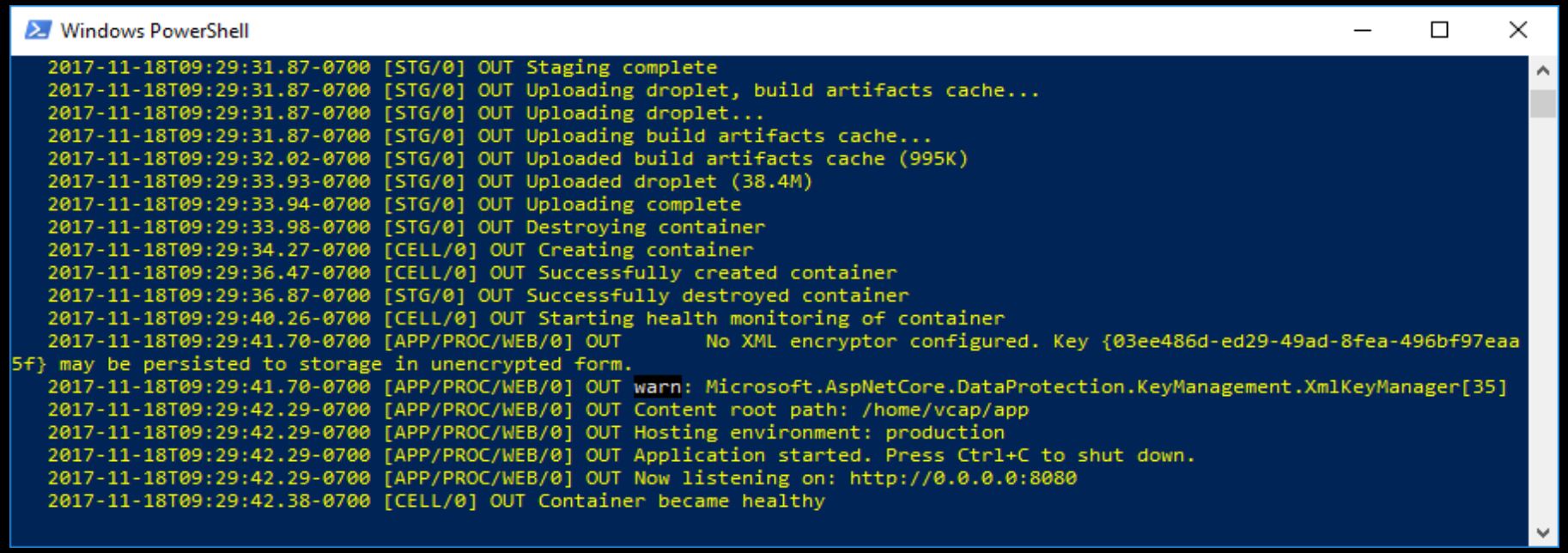
- Use same ASP.NET Core MVC application to understand:
  - Cloud Foundry Domains, Routes & Hosts
  - Usage of CF CLI: scale, routes, create-route, map-route, etc.
  - Usage of Steeltoe CloudFoundry Configuration Provider
    - Used to see what instance your interacting with
- Follow Lab Description
  - Description: https://github.com/SteeltoeOSS/Workshop/tree/master/Lab03
  - Code: /Workshop/Lab01/Lab01.sln

# Monitoring Applications on Pivotal Cloud Foundry

LOGGING, TAILING LOGS, HEALTH, EVENTS, CLI

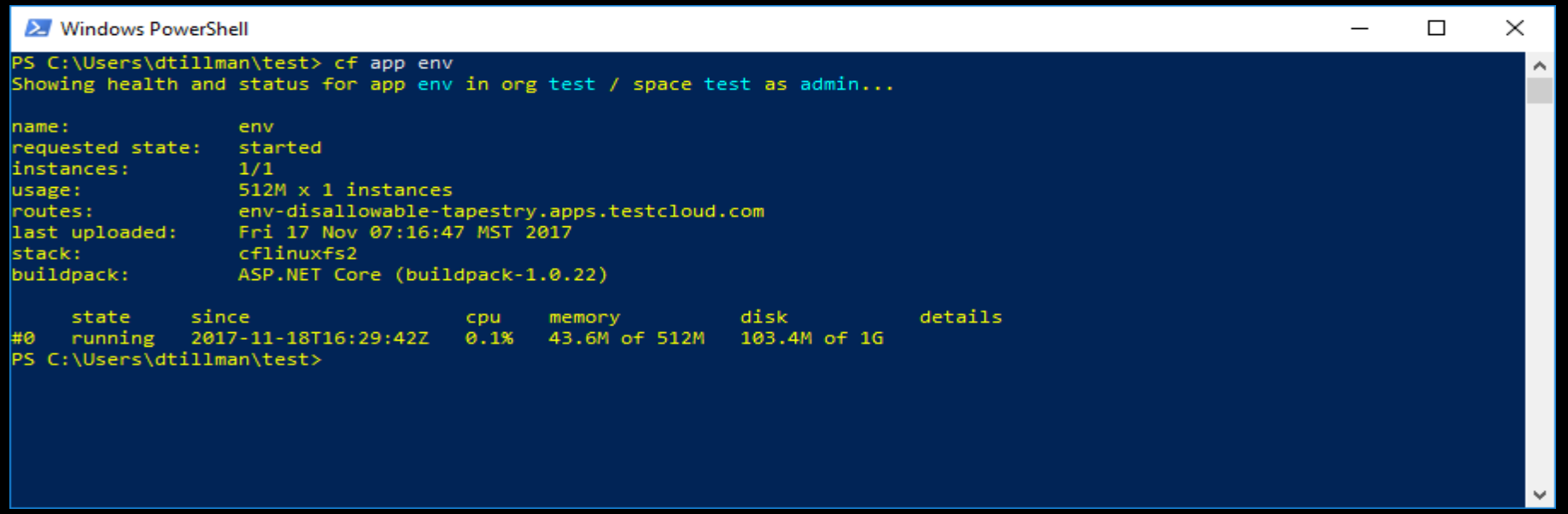
# Logging

- Cloud Foundry aggregates 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 applications logs
  - `cf logs APP_NAME –recent` - allows you to view recent logs



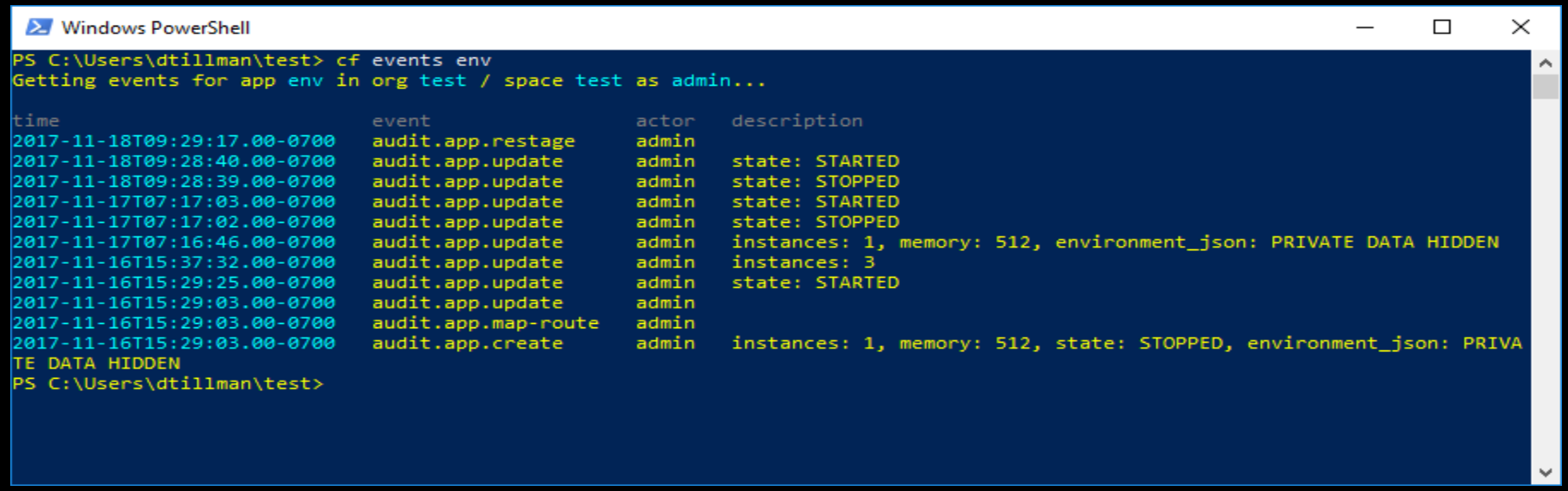
# Health

- 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 to view health of an application



# Application Events

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



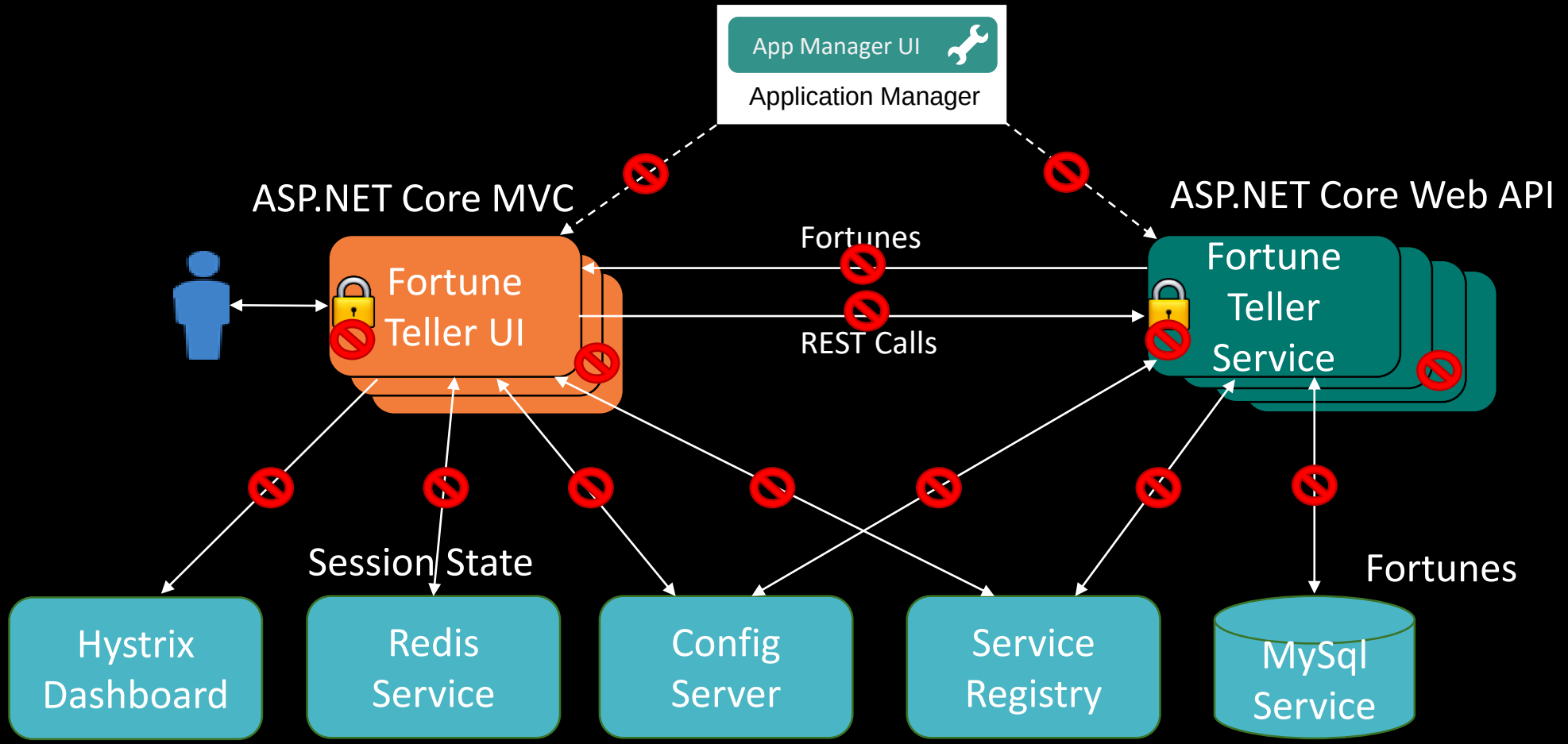
# Lab4 – Monitoring Applications on Cloud Foundry

- Use same ASP.NET Core MVC application to understand
  - Cloud Foundry Logging – STG, CELL,  APP,  RTR
  - Usage of CF CLI: cf logs, events, app, etc.
  - Usage of PCF Metrics to view application performance data
- Follow Lab Description
  - Description: https://github.com/SteeltoeOSS/Workshop/tree/master/Lab04
  - Code: /Workshop/Lab01/Lab01.sln

# ASP.NET Core Programming Fundamentals

## HOST, SERVER STARTUP, DENDENCY INJECTION, SERVICES, MIDDLEWARE

# Fortune Teller App – Current State



# ASP.NET Core – Everything starts in Program.Main

- `CreateDefaultBuilder` sets up the default Web Host
  - Web Server - Kestrel
    - IIS Integration
  - Logging - Console and Debug output
  - Builds application Configuration from
    - `appsettings.json`
    - `appsettings-{env}.json`
    - Environment variables
    - Command line arguments
  - Specifies `Startup` class
    - Configure Dependency Injection
    - Configure Middleware Pipeline

```
public class Program
{
  public static void Main(string[] args)
  {
     BuildWebHost(args).Run();
  }

  public static IWebHost BuildWebHost(string[] args) =>
      WebHost.CreateDefaultBuilder(args)
             .UseStartup<Startup>()
             .Build();
}
```

# ASP.NET Core – Startup Class

- Configures the application
  - Constructor - Save applications `IConfiguration`
  - `ConfigureServices()`
    - Configure Dependency Injection
  - `Configure()`
    - Configure Middleware Pipeline

```
public class Startup
{
  public Startup(IConfiguration configuration)
  {
     Configuration = configuration;
  }
  public IConfiguration Configuration { get; }
  public void ConfigureServices(IServiceCollection services)
  {
     services.AddMvc();
     services.AddScoped<IFortuneRepository, FortuneRepository>();
  }
  public void Configure(IApplicationBuilder app, IHostingEnvironment env)
  {
     if (env.IsDevelopment())
     {
       app.UseDeveloperExceptionPage();
     }
     app.UseStaticFiles();
     app.UseAuthentication();
     app.UseMvc();
  }
}
```

# ASP.NET Core - Dependency Injection

- Fundamental to ASP.NET Core
  - Replaceable with other DI providers like Autofac, etc.
- `IServiceProvider`
  - The container, manages `services`
- `IServiceCollection` to add to it
  - Framework services added via Add<Service>()
  - Application services added via
    - `AddTransient()`
    - `AddSingleton()`
    - `AddScoped()`

```
public class Startup
{
  public Startup(IConfiguration configuration)
  {
     Configuration = configuration;
  }
  public IConfiguration Configuration { get; }
  public void ConfigureServices(IServiceCollection services)
  {
     services.AddMvc();
     services.AddScoped<IFortuneRepository, FortuneRepository>();
  }
  public void Configure(IApplicationBuilder app, IHostingEnvironment env)
  {
     if (env.IsDevelopment())
     {
       app.UseDeveloperExceptionPage();
     }
     app.UseStaticFiles();
     app.UseAuthentication();
     app.UseMvc();
  }
}
```

# ASP.NET Core - Dependency Injection cont'd

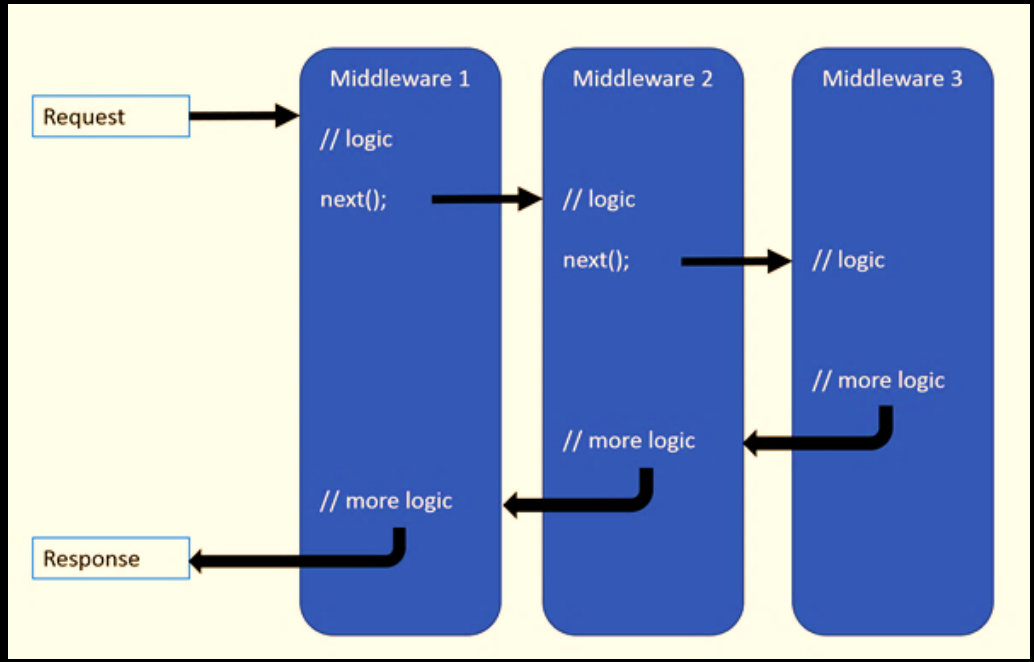
- Constructor based
  - Must be public
  - Multiple injectable arguments
  - One suitable constructor
- Supports chaining dependencies
  - Resolves dependency graph
- Service lifetimes
  - Singleton
  - Transient
  - Scoped
- Disposes what it creates

```
public class FortuneRepository : IFortuneRepository
{
  private FortuneContext _db;
  Random _random = new Random();

  public FortuneRepository(FortuneContext db)
  {
    _db = db;
  }
  ……
}

public class FortunesController : Controller
{
  private IFortuneRepository _fortunes;
  public FortunesController(IFortuneRepository fortunes)
  {
     _fortunes = fortunes;
  }
   ……
}
```

# ASP.NET Core - Middleware

- Components which process each request and response
  - Optionally pass request on to next component
  - Execute logic before & after
- Built-in ASP.NET Core Middleware
  - Authentication
  - Session
  - StaticFiles
  - Routing

Middleware Pipeline



# ASP.NET Core - Middleware

- Use `IApplicationBuilder` to add to pipeline
  - Framework Middleware added via Use<Middleware>()
  - Order is important
- Example
  1. Respond with dev exception page for 500 errors
  2. Static file serving
  3. Authentication processing
  4. MVC processing

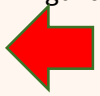
```
public class Startup
{
  public Startup(IConfiguration configuration)
  {
     Configuration = configuration;
  }
  public IConfiguration Configuration { get; }
  public void ConfigureServices(IServiceCollection services)
  {
     services.AddMvc();
     services.AddScoped<IFortuneRepository, FortuneRepository>();
  }
  public void Configure(IApplicationBuilder app, IHostingEnvironment env)
  {
     if (env.IsDevelopment())
     {
       app.UseDeveloperExceptionPage();
     }
     app.UseStaticFiles();
     app.UseAuthentication();
     app.UseMvc();
  }
}
```

# ASP.NET Core - Customized Web Host Example

```
public static IWebHost BuildWebHost(string[] args)
{
   var builder = new WebHostBuilder()
     .UseKestrel()
     .UseIISIntegration()
     .ConfigureAppConfiguration((hostingContext, config) => {
        var env = hostingContext.HostingEnvironment;
        config.AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
              .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true, reloadOnChange: true);
        config.AddEnvironmentVariables();
        if (args != null) {
          config.AddCommandLine(args);
        }
      })
     .ConfigureLogging((hostingContext, logging) => {
         logging.AddConfiguration(hostingContext.Configuration.GetSection("Logging"));
         logging.AddDebug();
         logging.AddConsole();
      })
     .UseStartup<Startup>()
     .UseContentRoot(Directory.GetCurrentDirectory())
     .UseDefaultServiceProvider((context, options) => {
         options.ValidateScopes = context.HostingEnvironment.IsDevelopment();
      });
     if (args != null) {
         builder.UseConfiguration(new ConfigurationBuilder().AddCommandLine(args).Build());
     }
     return builder.Build();
}
```

WebServer

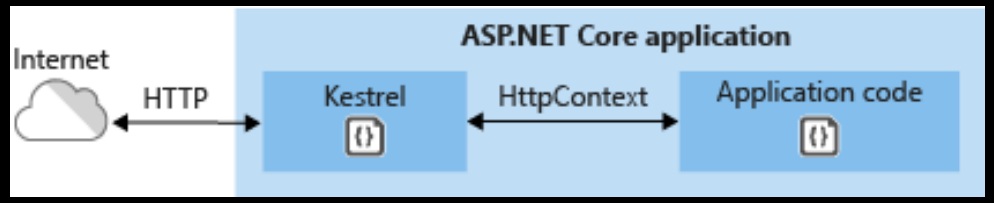
App Config

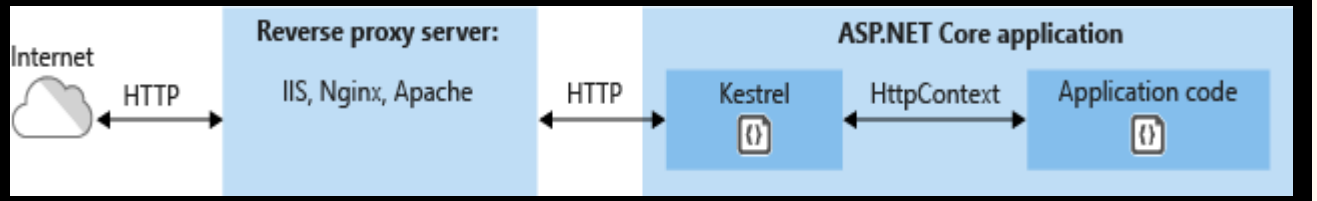


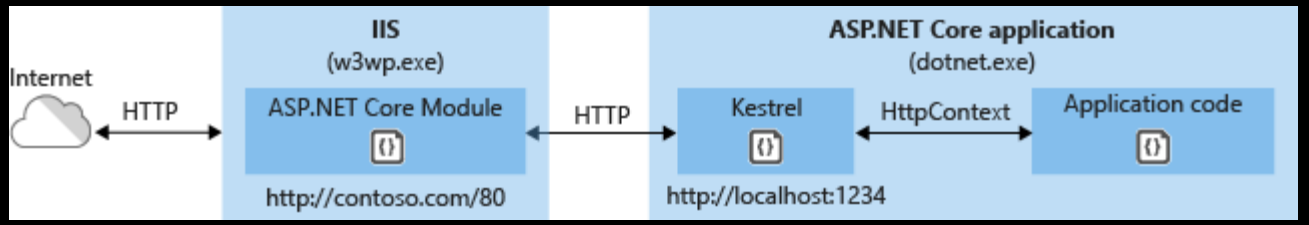
Logging

Startup

# ASP.NET Core – Kestrel Web Server




```
public static IWebHost BuildWebHost(string[] args) =>
 ……….
    .UseKestrel(options => {
       options.Limits.MaxConcurrentConnections = 100;
       options.Limits.MaxConcurrentUpgradedConnections = 100;
       options.Limits.MaxRequestBodySize = 10 * 1024;
       options.Limits.MinRequestBodyDataRate =
         new MinDataRate(100,TimeSpan.FromSeconds(10));
       options.Limits.MinResponseDataRate =
          new MinDataRate(100 TimeSpan.FromSeconds(10));
       options.Listen(IPAddress.Loopback, 5000);
       options.Listen(IPAddress.Loopback, 5001, lo => {
          lo.UseHttps("testCert.pfx", "testPassword");
        })
     .UseIISIntegration()

…….
}).Build();
```

# ASP.NET Core - Environments

- Support for multiple `environments`
  - Built in understanding of `Production`, `Staging` & `Development`
  - Set via environment variable `ASPNETCORE_ENVIRONMENT`
- Use `IHostingEnvironment` to access
  - Check with `IsDevelopment()`, `IsProduction()`, or `IsStaging()`
  - Use `IsEnvironment("Cloud")` to test non-standard settings

```
public static IWebHost BuildWebHost(string[] args)
{
  var builder = new WebHostBuilder()
    .UseKestrel()
    .UseIISIntegration()
    .ConfigureAppConfiguration((hostingContext, config) => {
       var env = hostingContext.HostingEnvironment;
       config.AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
             .AddJsonFile($"appsettings.{env.EnvironmentName}.json", ….
       config.AddEnvironmentVariables();
       if (args != null) {
         config.AddCommandLine(args);
       }
     })
    .ConfigureLogging((hostingContext, logging) => {
       logging.AddConfiguration(hostingContext.Configuration.GetSection("Logging"));
       logging.AddDebug();
       logging.AddConsole();
     })
    .UseStartup<Startup>()
    .UseContentRoot(Directory.GetCurrentDirectory())
    .UseDefaultServiceProvider((context, options) => {
       options.ValidateScopes = context.HostingEnvironment.IsDevelopment();
     });
    if (args != null) {
        builder.UseConfiguration(new ConfigurationBuilder()……
    }
    return builder.Build();
}
```

# ASP.NET Core – Configuration

- Built using `ConfigurationBuilder`
  - Add configuration sources to the builder
    - Add<Source>() methods
  - Call `Build()` to actually construct configuration
  - Produces a `IConfiguration`
- Several configuration sources available
  - File based (e.g. INI, JSON and XML)
  - Command line arguments
  - Environment variables
  - Custom (e.g. Steeltoe providers)
- Sources are read in order added
  - Later sources override any settings from previous

```
public static IWebHost BuildWebHost(string[] args)
{
  var builder = new WebHostBuilder()
    .UseKestrel()
    .UseIISIntegration()
    .ConfigureAppConfiguration((hostingContext, config) => {
       var env = hostingContext.HostingEnvironment;
       config.AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
             .AddJsonFile($"appsettings.{env.EnvironmentName}.json", ….
       config.AddEnvironmentVariables();
       if (args != null) {
         config.AddCommandLine(args);
       }
     })
    .ConfigureLogging((hostingContext, logging) => {
       logging.AddConfiguration(hostingContext.Configuration.GetSection("Logging"));
       logging.AddDebug();
       logging.AddConsole();
     })
    .UseStartup<Startup>()
    .UseContentRoot(Directory.GetCurrentDirectory())
    .UseDefaultServiceProvider((context, options) => {
       options.ValidateScopes = context.HostingEnvironment.IsDevelopment();
     });
    if (args != null) {
        builder.UseConfiguration(new ConfigurationBuilder()……
    }
    return builder.Build();
}
```

# ASP.NET Core – Configuration cont'd

- Configuration is list of name-value pairs grouped into multilevel hierarchy  separated by ":"
  - e.g. `spring:cloud:config:uri`
- Access values via indexer
  - v=config["Logging:IncludeScopes"]
- Access sections
  - config =GetSection("Logging")
    - v = config["IncludeScopes"]

```
{
  "Logging": {
    "IncludeScopes": false,
    "Debug": {
      "LogLevel": {
        "Default": "Warning"
      }
    },
    "Console": {
      "LogLevel": {
        "Default": "Warning"
      }
    }
  },
 "spring": {
    "application": {
      "name": "fortuneservice"
    },
    "cloud": {
      "config": {
        "uri": "http://localhost:8888",
        "validate_certificates": false
      }
    }
  }
}
```

# ASP.NET Core - Options

- Supports de-serializing config settings into a custom object (i.e. Options class)
  - Built in binder for binding configuration to Options class
- Options class must have
  - Public parameter-less constructor
  - Attributes to hold values
  - Attribute names map to config keys

```
public class FortuneServiceOptions
{
  public string Scheme { get; set; } = "http";
  public string Address { get; set; }
  public string RandomFortunePath { get; set; }
  public string AllFortunesPath { get; set; }}
}
```

```
{
  "Logging": {
    "IncludeScopes": false,
    "Debug": {
      "LogLevel": {
        "Default": "Information"
      }
    },
    "Console": {
      "LogLevel": {
        "Default": "Information"
      }
    }
  },
  "fortuneService": {
    "scheme": "http",
    "address": "localhost:5000",
    "randomFortunePath": "api/fortunes/random",
    "allFortunesPath": "api/fortunes/all"
  }
}
```

# ASP.NET Core – Options cont'd

- To bind configuration to Option class and inject into container
  - `AddOptions() service`
  - `Configure<OptionsClass>(.)` for each option class
- Added to container
  - `IOptions<OptionsClass>`
  - `IOptionsSnapshot<OptionsClass>`
  - `IOptionsMonitor<OptionsClass>`

```
public class Startup
{
  public Startup(IConfiguration configuration)
  {
     Configuration = configuration;
  }
  public IConfiguration Configuration { get; }
  public void ConfigureServices(IServiceCollection services)
  {
     services.AddOptions();
     services.Configure<FortuneServiceOptions>(
                    Configuration.GetSection("fortuneService"));
     services.AddScoped<IFortuneService, FortuneServiceClient>();
     services.AddMvc();
  }
……
}
public class FortuneServiceClient : IFortuneService
{
  public FortuneServiceClient(
               IOptionsSnapshot<FortuneServiceOptions> config
  {
     _config = config;
  }
…….
}
```

# ASP.NET Core - Logging

- Logging API with multiple providers
  - Built using LoggingBuilder
  - Add providers to the builder via Add<Provider>() methods
- Providers
  - Console
  - Debug
  - Windows Event Log
  - Custom (e.g. Steeltoe Dynamic Console)
- AddConfiguration(..) sets up Log Filtering Levels using Configuration

```
public static IWebHost BuildWebHost(string[] args)
{
  var builder = new WebHostBuilder()
    .UseKestrel()
    .UseIISIntegration()
    .ConfigureAppConfiguration((hostingContext, config) => {
       var env = hostingContext.HostingEnvironment;
       config.AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
             .AddJsonFile($"appsettings.{env.EnvironmentName}.json", ….
       config.AddEnvironmentVariables();
       if (args != null) {
         config.AddCommandLine(args);
       }
     })
    .ConfigureLogging((hostingContext, logging) => {
       logging.AddConfiguration(hostingContext.Configuration.GetSection("Logging"));
       logging.AddDebug();
       logging.AddConsole();
     })
    .UseStartup<Startup>()
    .UseContentRoot(Directory.GetCurrentDirectory())
    .UseDefaultServiceProvider((context, options) => {
       options.ValidateScopes = context.HostingEnvironment.IsDevelopment();
     });
    if (args != null) {
        builder.UseConfiguration(new ConfigurationBuilder()……
    }
    return builder.Build();
}
```

# ASP.NET Core - Logging cont'd

- Using logging
  - Get `ILogger<>` from DI container
  - Call log method
    - `LogInformation()`, `LogError()`, …
- Log Category & Levels
  - Cat -> Fully qualified name of class
  - Level -> Trace, Debug, … etc.
- Log Filtering Levels
  - Setup by `AddConfiguration()` during Web Host building using Configuration

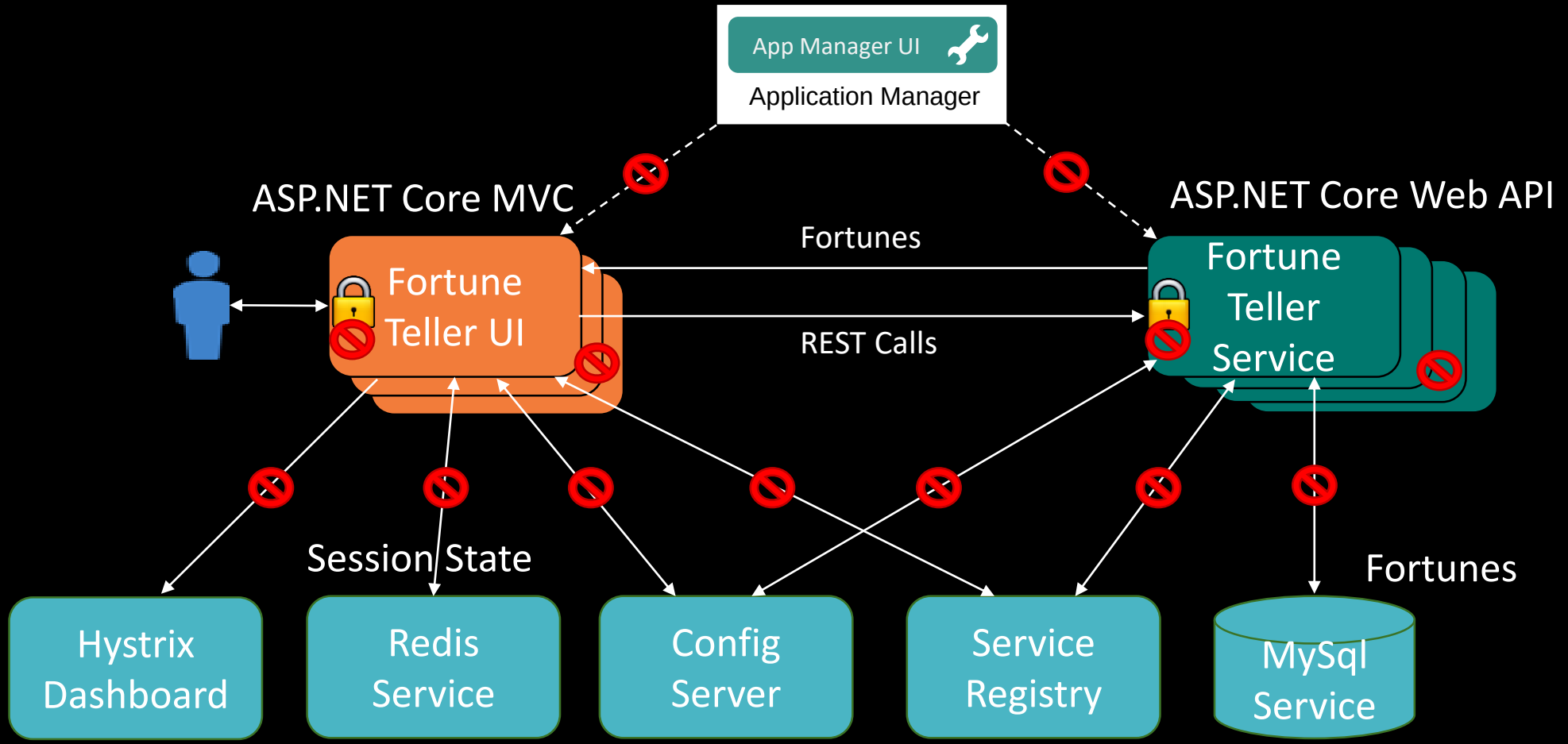
```
public class FortunesController : Controller
{
  ILogger<FortunesController> _logger;
  public FortunesController(ILogger<FortunesController> logger)
  {
     _logger = logger;
  }
  public async Task<Fortune> RandomFortuneAsync()
  {
     _logger.LogDebug("RandomFortuneAsync");
   ……..
  }
}
```

```
{
  "Logging": {
     "Debug": {
      "LogLevel": {
        "Default": "Information"
      }
    },
    "Console": {
      "LogLevel": {
        "Default": "Information",
        "Steeltoe": "Debug"
      }
    }
  }
}
```

# Lab5 – ASP.NET Core Programming Fundamentals

- Familiarize yourself with Fortune-Teller lab code
  - Run application locally & publish & push to Cloud Foundry
- Fortune Service – use an in-memory database to hold Fortunes
  - Add In-Memory EFCore database to store Fortunes
  - Initialize database with some Fortunes
  - Inject `FortuneRepository` and `FortuneContext` into REST API (`FortunesController`)
- Fortune UI – calls Fortune Service to get random Fortunes
  - Inject `FortuneServiceClient` into `FortunesController`
  - Use Configuration & Options framework to configure `FortuneServiceClient`
- Follow Lab Description
  - Description: https://github.com/SteeltoeOSS/Workshop/tree/master/Lab05
  - Code: /Workshop/Start/FortuneTeller.sln

# Fortune Teller App – After Lab5



# Steeltoe & Spring Cloud Services

NEFLIX, SPRING CLOUD, SPRING CLOUD SERVICES

# Netflix Cloud Infrastructure Libraries

- Netflix needed to be faster to win/disrupt
- Pioneer and vocal proponent of microservices
  - Key to speed and success
- Developed several OSS Infrastructure components
  - Eureka Server
  - Hystrix
  - Turbine
  - Ribbon
  - etc.

# Others Contributed Cloud Infrastructure Libraries

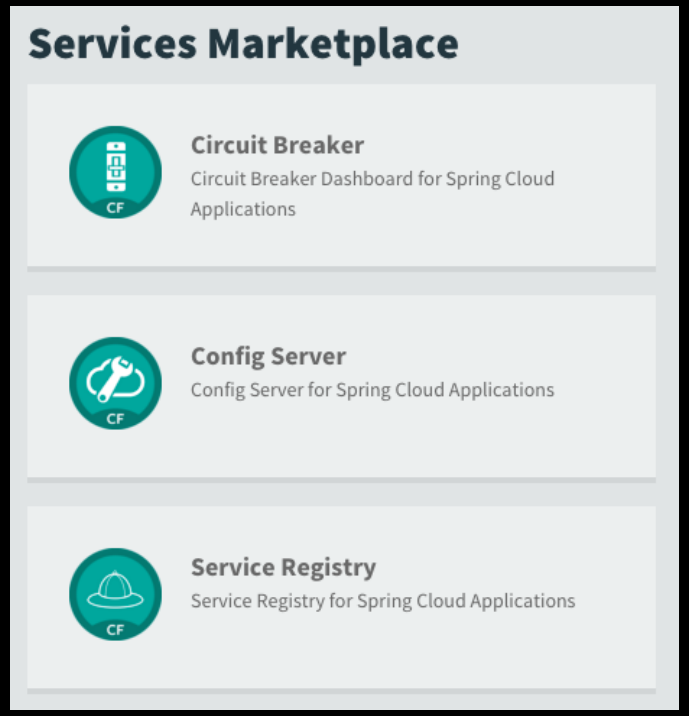
- Twitter, Facebook and Hashicorp all have open-sourced other cloud infrastructure libraries
- Complementary and competing solutions
  - Form a bazaar of ideas and solutions


# Spring Cloud – Open Source Project

- Ease developer access to a selection of open source cloud infrastructure
  - APIs which encapsulates access to underlying libraries
    - Pluggable implementations
    - Allows best of breed
  - Apply Spring philosophy
    - Convention over configuration
    - Opinionated defaults
    - Developer simplicity
- Spring Cloud project added additional cloud infrastructure
  - Connectors
  - Config Server

# Pivotal Spring Cloud Services

## Spring Cloud Services = Spring Cloud Packaged for Pivotal Cloud Foundry

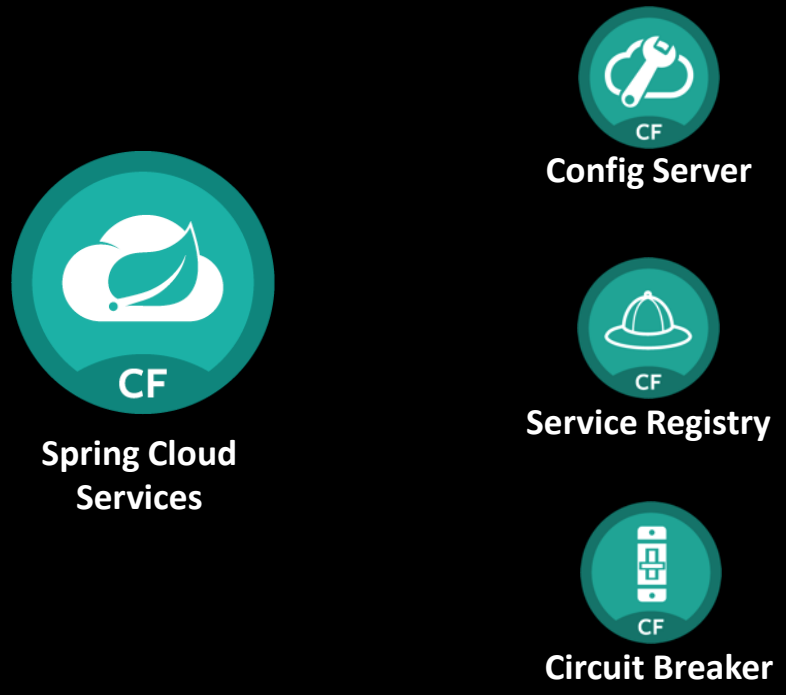




Spring Cloud Services = Spring Cloud + Enhancements

# Spring Cloud Services

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

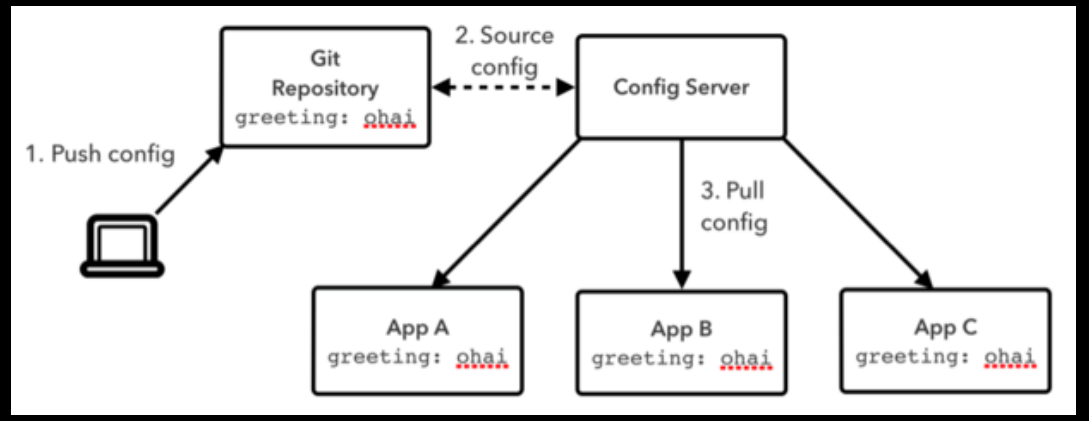
# Spring Cloud Services




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

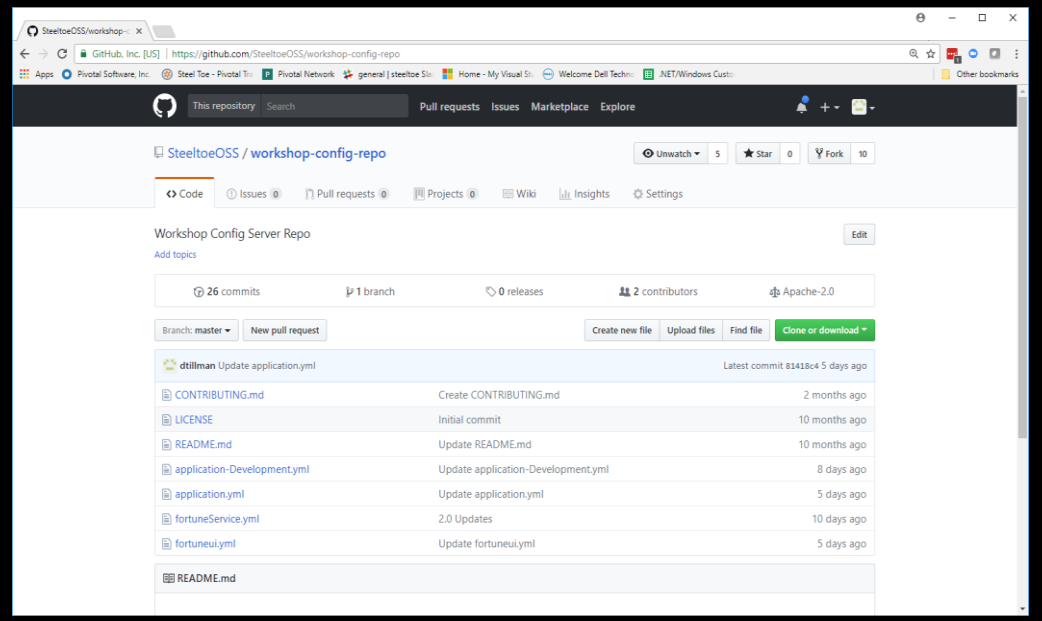
# Spring Cloud Config Server

- Enables centralizing application configurations
- Supports different back end storage options
  - Git repositories
  - File System
  - Hashicorp Vault
  - Composite
- Configuration file format support
  - Java properties (.properties)
  - YAML (.yml)
- Exposes configurations via REST based API, http://localhost:8080/
  - Data returned in JSON
- Client pulls configuration providing
  - {AppName}
  - {Profile} – one or more
  - {Label} – zero or more, meaning varies by back end
- Cloud Foundry version of Config server installed with Spring Cloud Services (SCS)
  - Version is protected by OAuth 2.0 tokens
  - Uses TLS/SSL connections



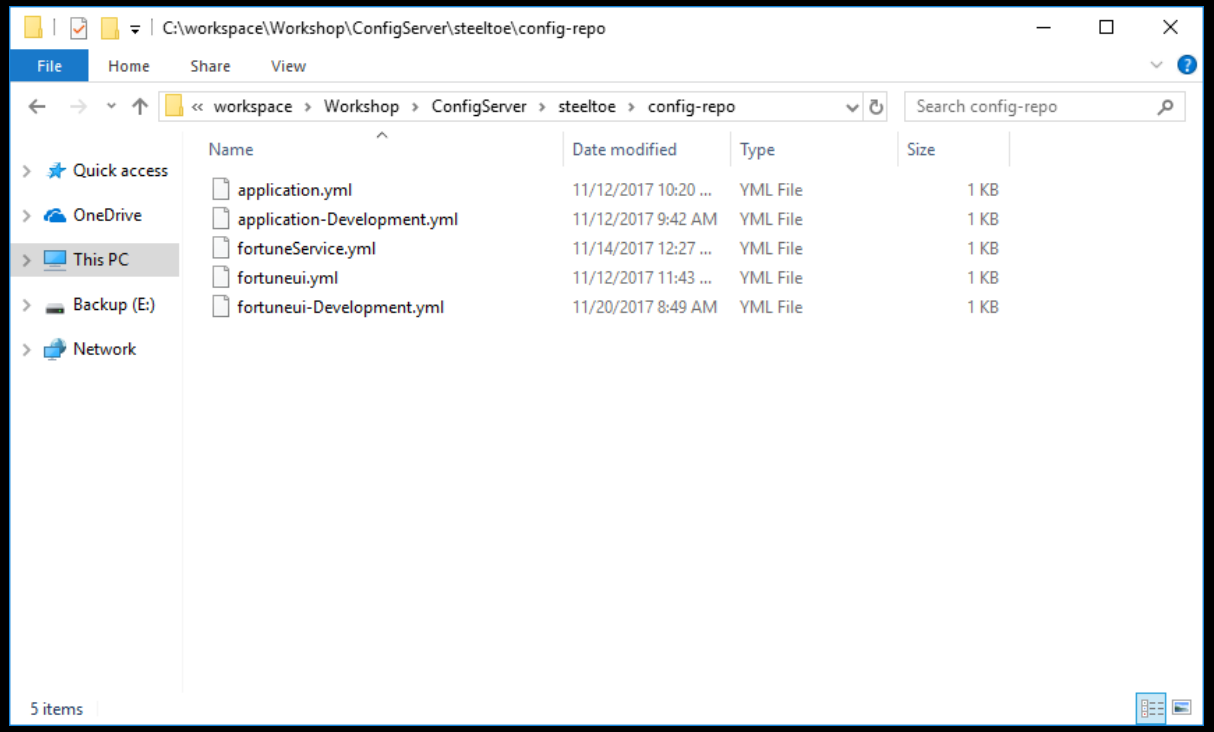
# Spring Cloud Config Server – Git Backend

- Client request parameter mappings
  - {AppName}  -> file name pattern in Git repo
  - {Profile} -> file name pattern in Git repo
  - {Label}  -> Git branch, tag, commit-id, defaults to ‘master’
- Selects configuration files from Git repo with patterns
  - /application.{suffix}
  - /{AppName}.{suffix}
  - /application-{Profile}.{suffix}
  - /{AppName}-{Profile}.{suffix}
- Selected files returned in a list with precedence
  - Above shows list order - least to highest precedence
  - Values in higher precedence files override lower
- Example Config Server requests
  - e.g. `curl localhost:8888/{AppName}/{Profile}`
  - e.g. `curl localhost:8888/{AppName}/{Profile1},{Profile2}`
  - e.g. `curl localhost:8888/{AppName}/{Profile}/{Label}`



# Spring Cloud Config Server – File System Backend

- Client request parameter mappings
  - {AppName}  -> file name pattern in file system
  - {Profile} -> file name pattern in file system
  - {Label}  -> directory name pattern in file system
- Selects configuration files from file system with patterns
  - /application.{suffix}
  - /{AppName}.{suffix}
  - /{Label}/application.{suffix}
  - /{Label}/{AppName}.{suffix}
  - /application-{Profile}.{suffix}
  - /{AppName}-{Profile}.{suffix}
  - /{Label}/application-{Profile}.{suffix}
  - /{Label}/{AppName}-{Profile}.{suffix}
- Selected files returned as a list with precedence
  - Above shows list order - least to highest precedence
  - Values in higher precedence files override lower
- Example Config Server requests
  - e.g. `curl localhost:8888/{AppName}/{Profile}`
  - e.g. `curl localhost:8888/{AppName}/{Profile}/{Label}`



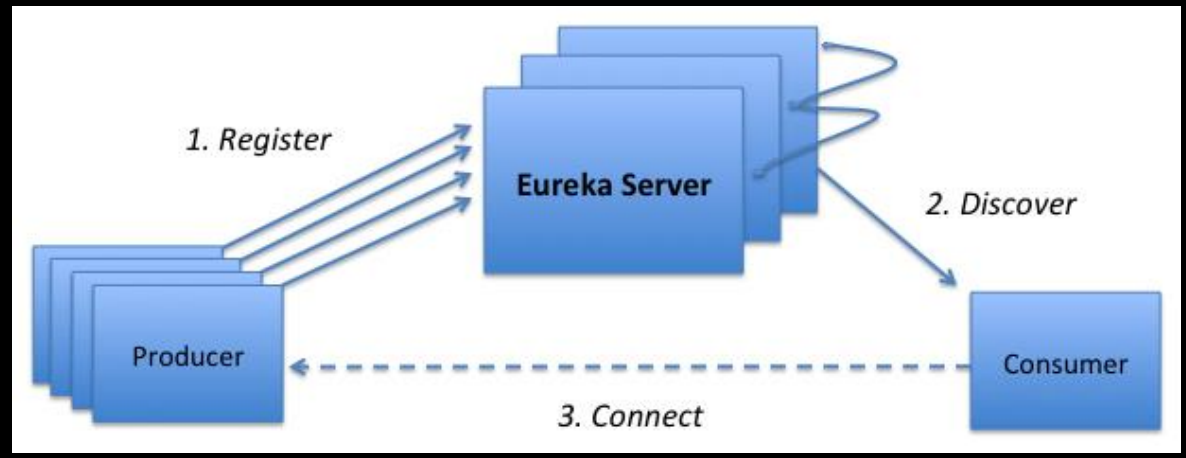
# Workshop Config Server

- Pre-configured Spring Cloud Config Server used for workshop
  - `Workshop/ConfigServer`
- Configured for File System back end
  - `ConfigServer/steeltoe/config-repo`
- Start up in terminal window
  - `cd Workshop/ConfigServer`
  - `mvnw spring-boot:run`
- Configuration
  - `Workshop/ConfigServer/src/main/java/resources/application.yml`
- Config Server documentation
  - Placeholder usage – per app repos, per app/profile repos, etc.
  - Pattern matching – multiple repositories, sub-directories, etc.

```yaml
server:
  port: 8888

management:
  context-path: /admin

logging:
  level:
    org.springframework.cloud: 'DEBUG'

spring:
  profiles:
    active: native
  cloud:
    config:
      server:
        native:
          searchLocations: file:./steeltoe/config-repo
#
# Replace spring:… configuration above with what is below for github repo usage
#
# spring:
#   cloud:
#     config:
#       server:
#         git: https://github.com/SteeltoeOSS/workshop-config-repo.git
```

# Spring Cloud Eureka Server

- Server for registering and discovering services
  - Exposes a REST based API http://localhost:8761/eureka
- Applications register addresses by name with Eureka Server
  - Heartbeats renew leases (30 seconds)
  - Three strikes, you're out (90 seconds)
  - Registrations optionally replicated to peers
- Eureka Client pulls entire service registry
  - Cached, used to lookup services by name
- Resilience
  - Multiple Eureka Servers
  - Clients cache copy of registry



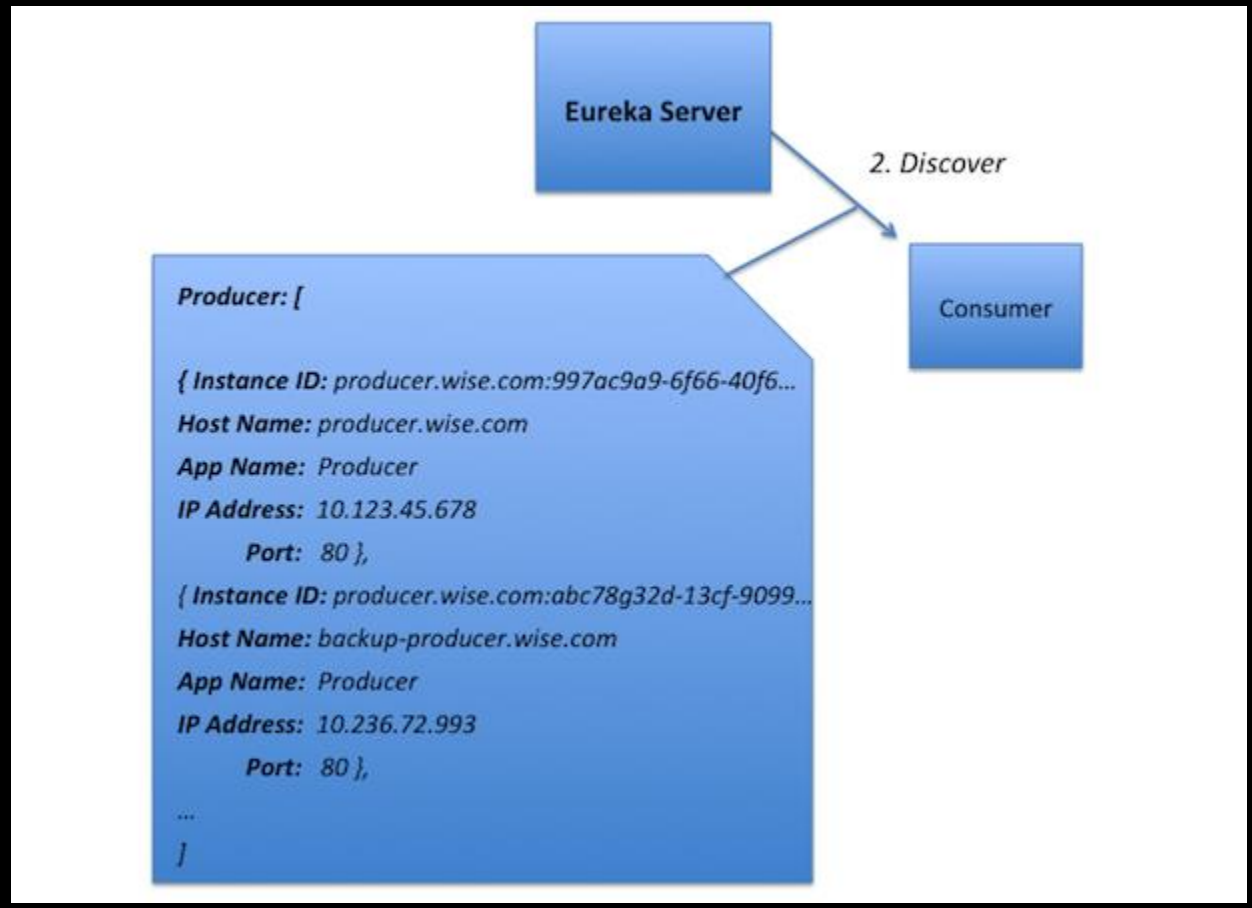

- Cloud Foundry version of Eureka server installed with Spring Cloud Services (SCS)
  - Version is protected by OAuth 2.0 tokens
  - Uses TLS/SSL connections

# Spring Cloud Eureka Server – Service Registration

- Application instances register with Eureka Server
  - Application Name
  - Host Name
  - Port
  - Instance ID
  - IP Address
  - Other metadata
- Default is to register by Host Name
  - Optionally register by IP Address

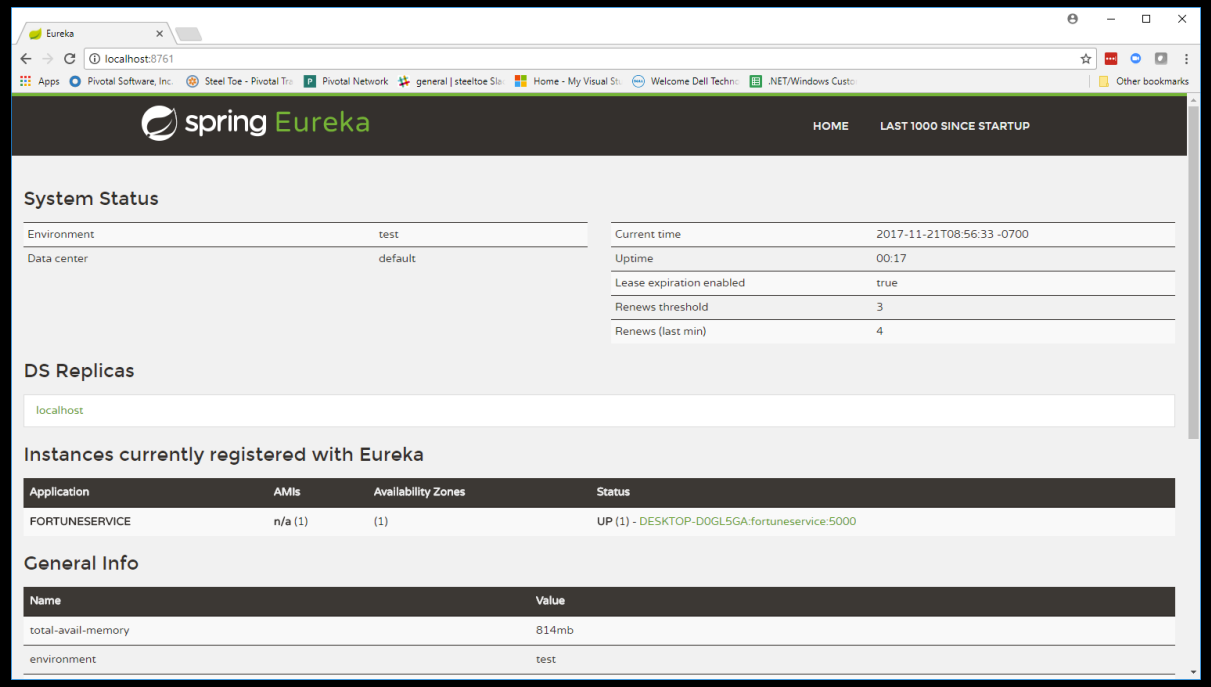
# Spring Cloud Eureka Server – Service Discovery

- Application instances periodically fetch the registry
  - Default every 30 second
  - Cached locally
  - Deltas retrieved after full fetch
- Application uses cache to lookup instances of other applications
  - Client load balancing possible

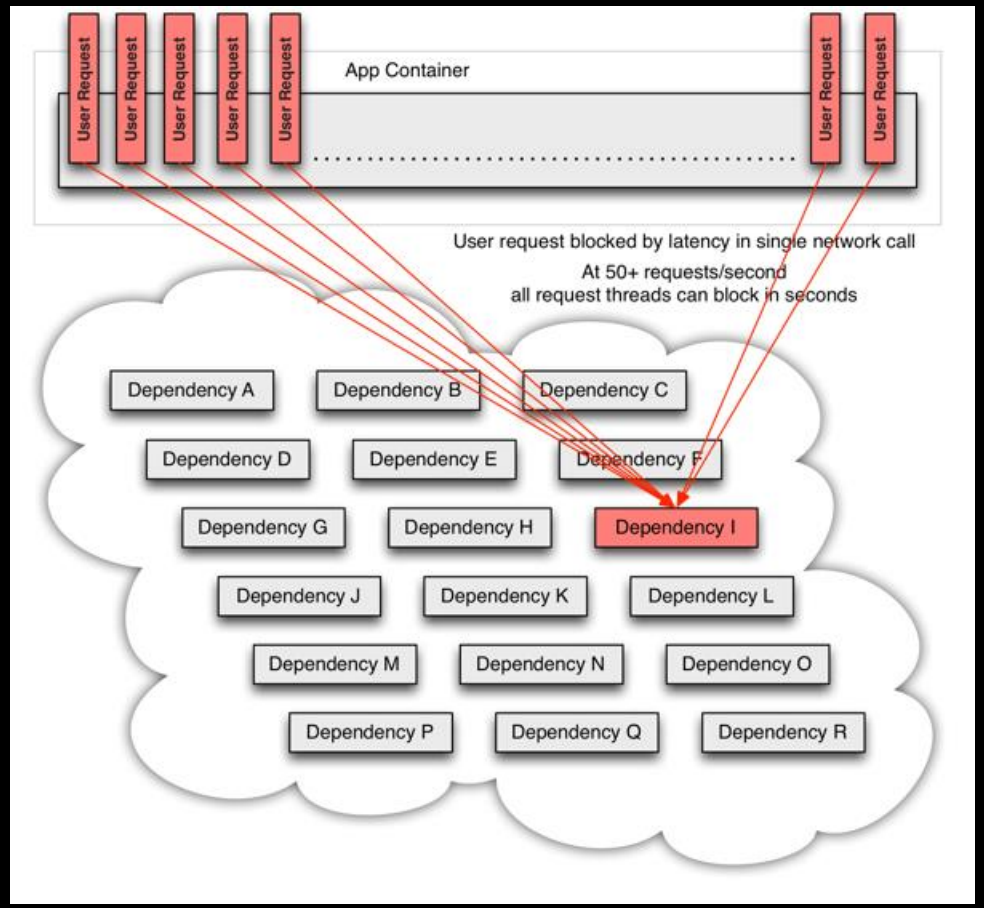


# Workshop Eureka Server

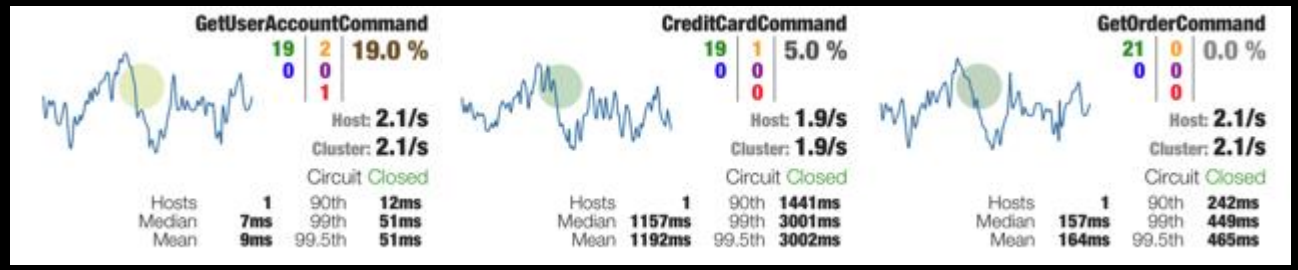
- Pre-configured Spring Cloud Eureka Server used for workshop
  - `Workshop/EurekaServer`
- Configured in standalone mode
  - http://localhost:8761/ - dashboard
  - http://localhost:8761/eureka - REST API
- Start up in terminal window
  - `cd Workshop/EurekaServer`
  - `mvnw spring-boot:run`
- Configuration
  - `EurekaServer/src/main/java/resources/application.yml`
- Eureka Server documentation
  - Peer Awareness – replication



# Spring Cloud Hystrix

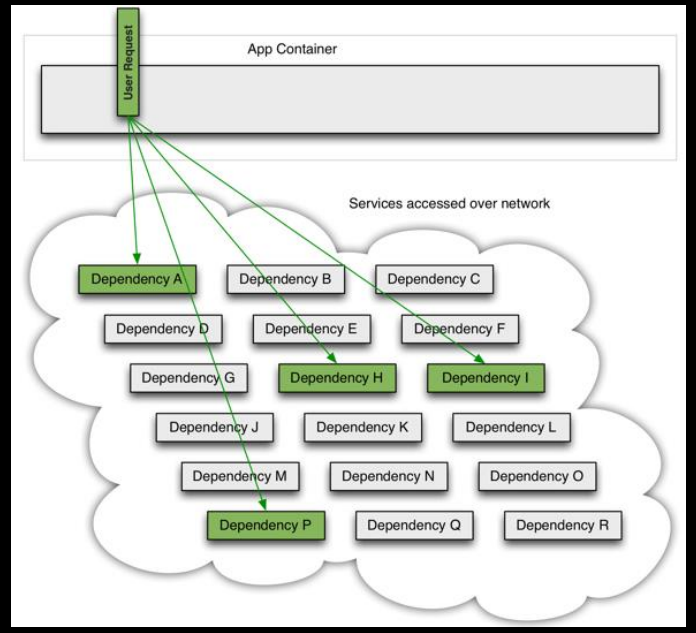



- Hystrix consists of two parts
  - Fault tolerance framework for dealing with failures and latency in distributed systems
    - Isolation patterns to limit impacts
  - Near real-time monitoring & alerting
    - Shorten time-to-discover and time-to-recover from failures

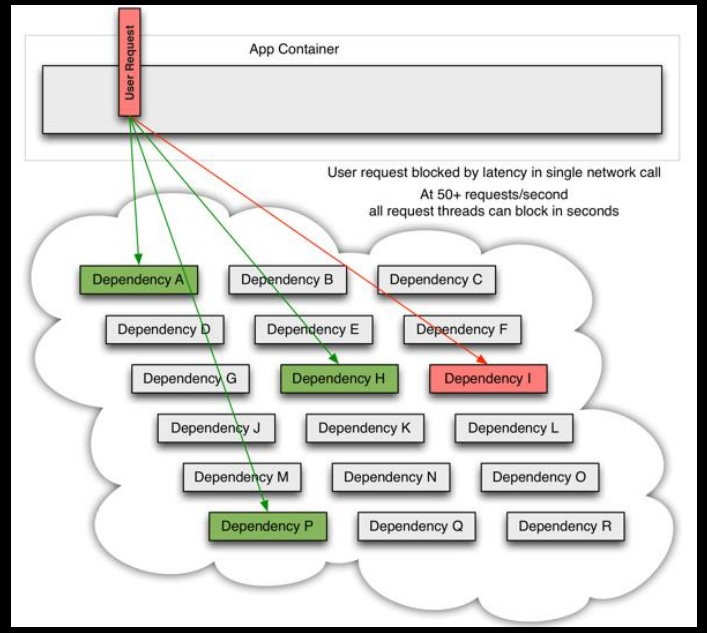


# Spring Cloud Hystrix – Fault Tolerance Framework

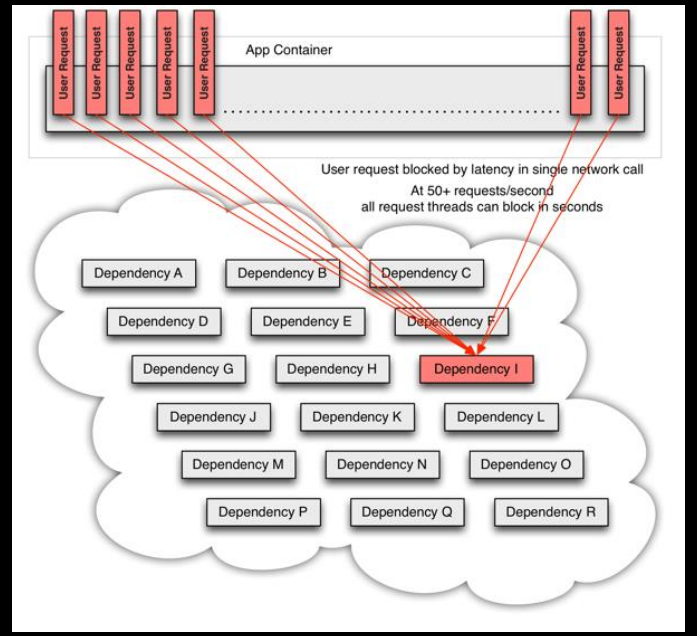
Healthy System




Backend Dependency Becomes Latent

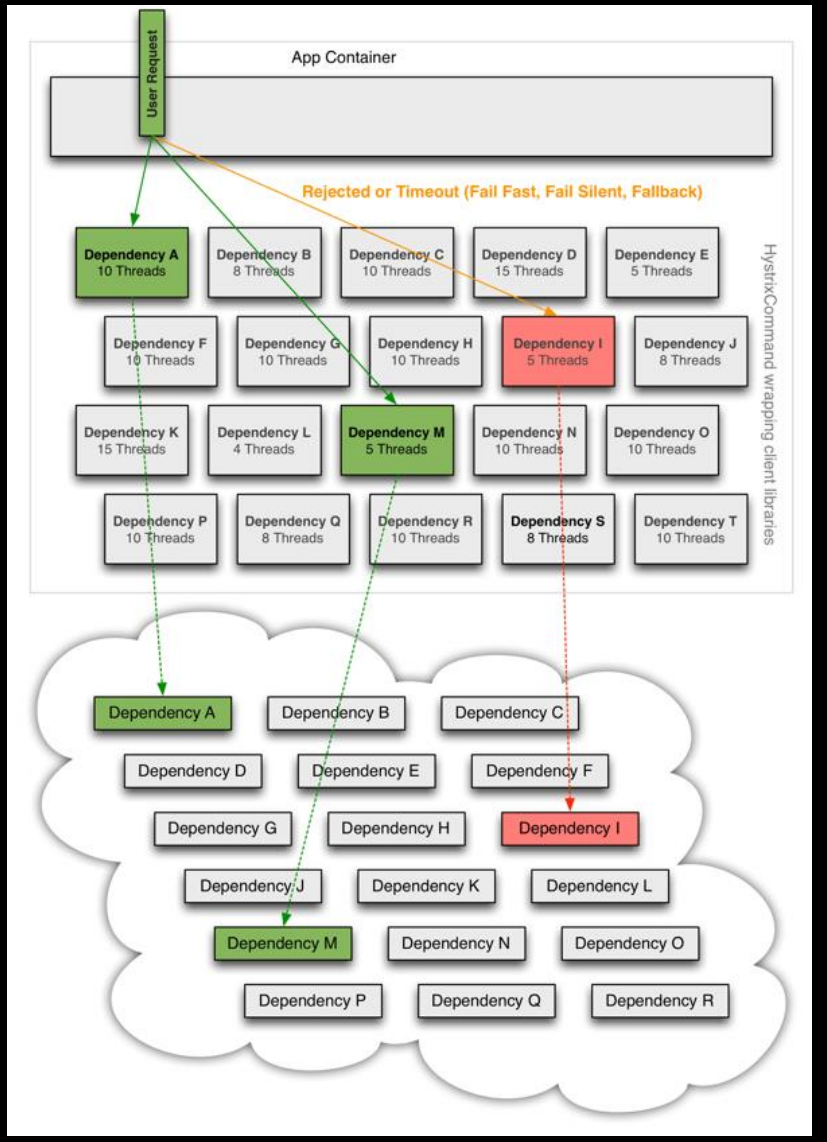



Latency cascades resources become saturated



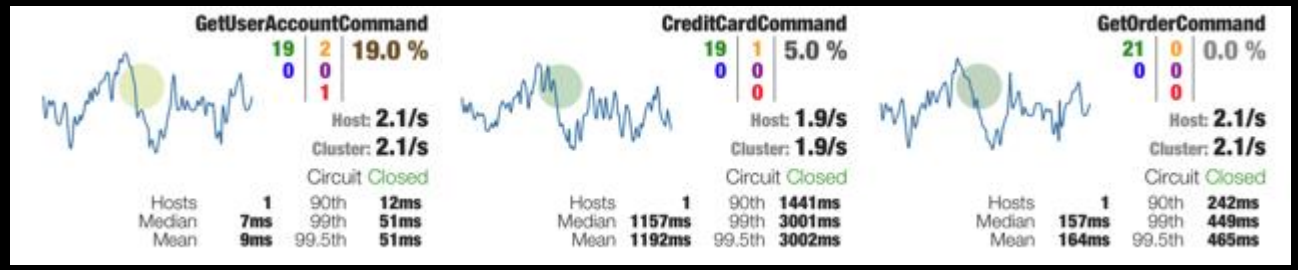
# Spring Cloud Hystrix

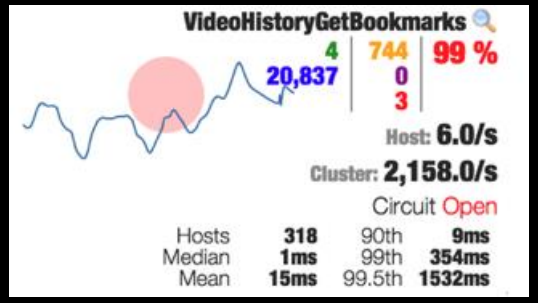
- Fault tolerance framework
  - Wrap all external dependencies in Hystrix commands
    - Executed on separate thread
    - Execute fallback logic on failures
  - Configurable thread pools for each dependency
    - Fail fast if pool consumed
  - Measures success, failures, timeouts, thread rejections, etc.
  - Trip circuit-breaker to stop requests to failed dependency
    - Periodically check
  - Gather metrics and report back to a dashboard for monitoring

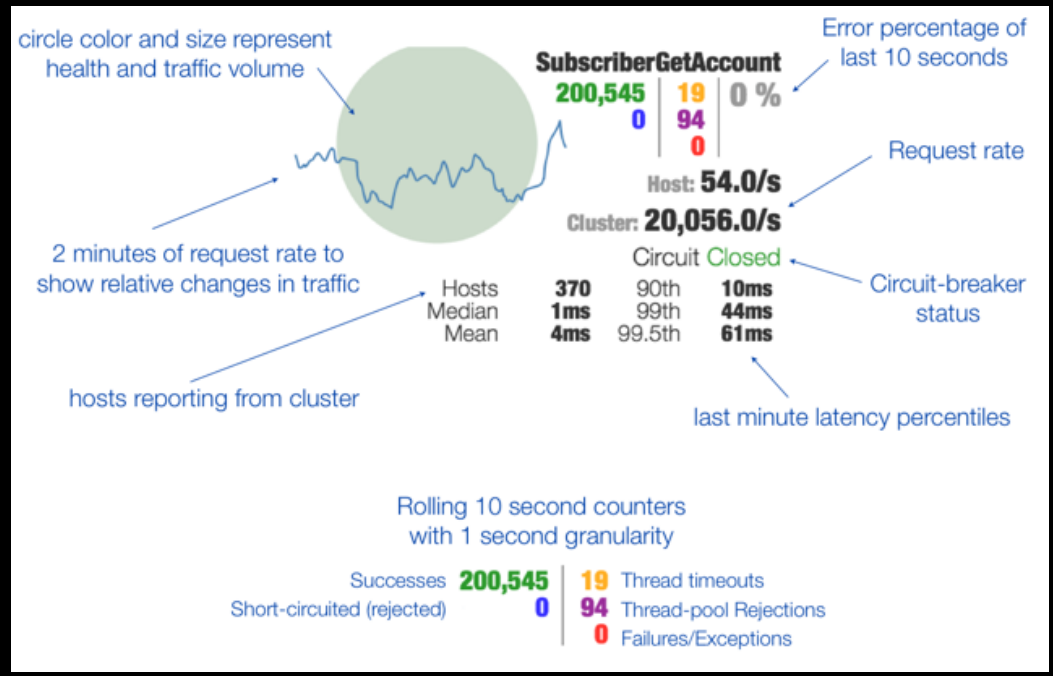


# Spring Cloud Hystrix – Monitoring and Alerting

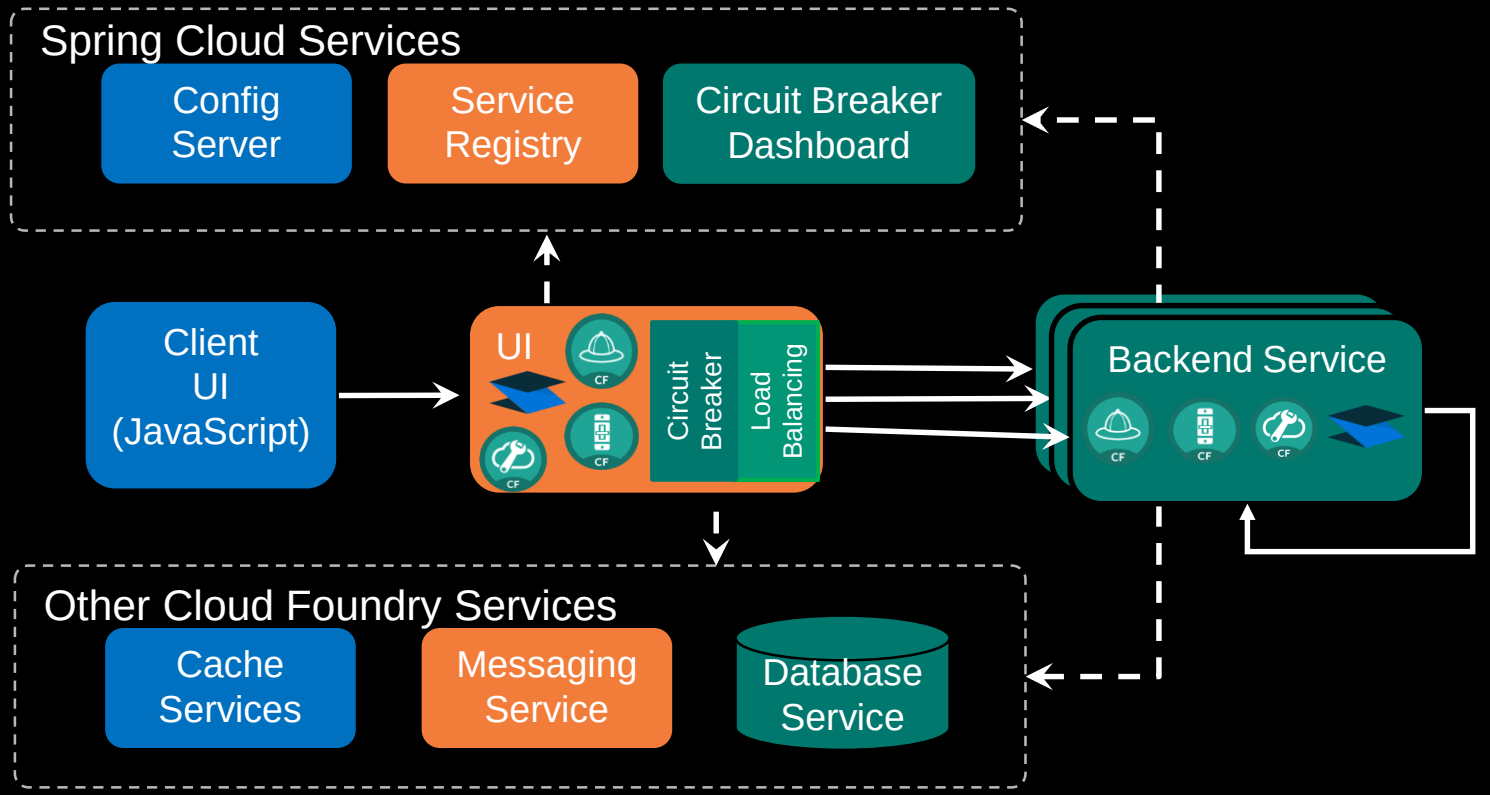
- Near real-time monitoring & alerting
  - Circuit breaker status
  - Thread pool usage
  - Request rates
  - Error percentages



# Putting it all together
# Typical Application Architecture using SCS



# Steeltoe Project

## Facilitates Twelve-Factor Contract on .NET


http://steeltoe.io


## 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)
  - Management & Monitoring
- Simplifies using Spring Cloud Services
  - Configuration server provider (e.g. Config Server, etc.)
  - Service Discovery (e.g. Eureka, etc.)
  - Circuit Breaker (e.g. Netflix Hystrix)
  - Distributed Tracing (e.g. Slueth coming)

# Steeltoe Frameworks

- Frameworks are Open Source
  - https://github.com/SteeltoeOSS
  - Functionally organized (Configuration, Discovery, Connectors, etc.)
- .NET support
  - .NET Core (Windows, Linux & OSX)
  - .NET Framework
- Application type support
  - ASP.NET 4 - MVC, WebForm, WebAPI, WCF
  - ASP.NET Core (.NET Framework and .NET Core)
  - Console apps (.NET Framework and .NET Core)

# Steeltoe Frameworks cont'd

- Configuration – additional .NET Configuration providers
  - CloudFoundry – parse VCAP_*, CF_*  and add to apps configuration
  - Config Server Client – access to Spring Cloud Config Server
- Connectors – simplifies configuring and injecting the following as services
  - MySql, MySql EF6, MySql EFCore – Connections and DbContexts
  - Postgres, Postgres EFCore – Connections and DbContexts
  - SqlServer, SqlServer EF6, SqlServer EFCore – Connections and DbContexts
  - RabbitMQ – Connection Factory
  - Redis – Microsoft IDistributedCache & StackExchange IConnectionMultiplexor
  - OAuth – access connection details from CF UAA & Pivotal SSO Service
- Circuit Breaker
  - Hystrix – Netflix based, latency and fault tolerance library

# Steeltoe Frameworks cont'd

- 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
- Management – production monitoring and management
  - Health
  - Build info – e.g. Git, etc.
  - Dynamically adjustable log levels
  - Request/Response Traces

# Steeltoe NuGets

- NuGet feeds
  - Development: https://www.myget.org/gallery/steeltoedev
  - Stable: https://www.myget.org/gallery/steeltoemaster
  - Release & Release Candidates: https://www.nuget.org/
- NuGet naming conventions
  - Steeltoe.X.Y.Z – core X.Y.Z functionality, application type independent
    - e.g. Steeltoe.Extensions.Configuration.CloudFoundry
  - Steeltoe.X.Y.ZCore – ASP.NET Core DI support
    - e.g. Steeltoe.Extensions.Configuration.CloudFoundryCore
  - Steeltoe.X.Y.ZAutofac – ASP.NET 4 Autofac support
    - e.g. Steeltoe.Extensions.Configuration.CloudFoundryAutofac
  - Others over time

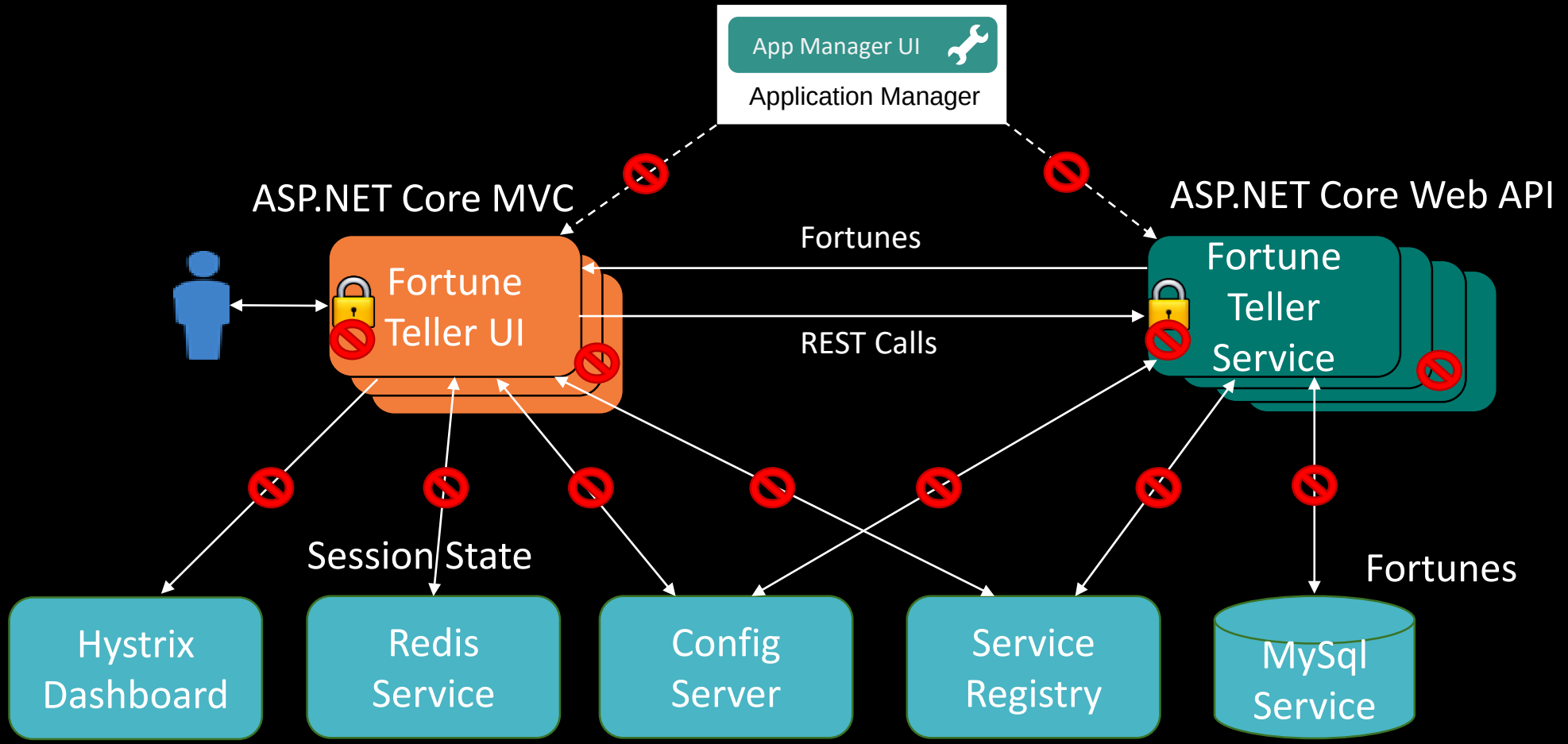
# Steeltoe Help

- Documentation – http://steeltoe.io
- Slack  - https://slack.steeltoe.io/
- Samples – https://github.com/SteeltoeOSS/Samples
  - Functionally - organized by framework area (e.g. Configuration, Discovery, etc.)
    - ASP.NET 4 and ASP.NET Core samples
  - Multi-functional – illustrates using several Steeltoe components
    - MusicStore – micro-services app built from the ASP.NET Core reference app
    - FreddysBBQ - a polyglot (i.e. Java and .NET) micro-services based sample app
    - Workshop – Fortune Teller using all of the Steeltoe components

# Steeltoe Configuration Providers

**ASP.NET 4, ASP.NET CORE, CONFIGURATION, OPTIONS, ENVIRONMENTS, CLOUDFOUNDRY PROVIDER, CONFIG SERVER PROVIDER**

# Fortune Teller App – After Lab5



# Steeltoe Configuration Overview

- Simplifies integration of additional sources of application configuration data into .NET applications
  - Built on .NET Configuration and Options services
- Several Configuration Providers
  - Cloud Foundry
  - Spring Cloud / Spring Cloud Services Config Server Client
- Application type support
  - ASP.NET - MVC, WebForm, WebAPI, WCF
  - ASP.NET Core (.NET Framework and .NET Core)
  - Console apps (.NET Framework and .NET Core)

# Steeltoe Cloud Foundry Provider

- Enables Cloud Foundry configuration to be added to an applications configuration
  - `VCAP_APPLICATION`, `VCAP_SERVICES`, `CF_INSTANCE_*`
- Usage
  - Add appropriate NuGet reference to project
  - Add Cloud Foundry provider to Configuration Builder (i.e. `AddCloudFoundry()`)
  - Optionally configure Cloud Foundry options (i.e. `ConfigureCloudFoundryOptions()`)
  - Optionally, access config values using configuration indexer
  - Optionally, access config values using Options

# Cloud Foundry – Add NuGet References

- Pivotal Cloud Foundry
  - Console, ASP.NET 4 - `Steeltoe.Extensions.Configuration.CloudFoundry`
  - ASP.NET Core - `Steeltoe.Extensions.Configuration.CloudFoundryCore`
  - ASP.NET 4 Autofac -`Steeltoe.Extensions.Configuration.CloudFoundryAutofac`
  - Namespace - `#using Steeltoe.Extensions.Configuration.CloudFoundry;`

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />
    <PackageReference Include="Steeltoe.Extensions.Configuration.CloudFoundryCore" Version="x.y.z" />
  <ItemGroup>

</Project>
```

# Cloud Foundry – Add Provider and Configure Options

- Add while building `WebHost`
  - In `ConfigureAppConfiguration()`
  - Use `AddCloudFoundry()` on provided `ConfigurationBuilder`
    - `vcap:application`
    - `vcap:services`
    - `spring:application:name` set to `vcap:application:name`
  - Add order may be important to you
- Optionally configure Cloud Foundry Options classes
  - `CloudFoundryApplicationOptions`
  - `CloudFoundryServicesOptions`

```
public static IWebHost BuildWebHost(string[] args)
{
  var builder = new WebHostBuilder()
    .UseKestrel()
    .UseIISIntegration()
    .ConfigureAppConfiguration((hostingContext, config) => {
       var env = hostingContext.HostingEnvironment;
       config.AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
             .AddJsonFile($"appsettings.{env.EnvironmentName}.json", ….
       config.AddEnvironmentVariables()
       config.AddCloudFoundry();
       if (args != null) {
         config.AddCommandLine(args);
       }
     })
}

public class Startup
{
  public IConfiguration Configuration { get; }
  public void ConfigureServices(IServiceCollection services)
  {
     services.AddOptions();
     services.ConfigureCloudFoundryOptions(Configuration)
  }
}
```

# Cloud Foundry – Using Indexers

```
"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"
 }
```

```
public class HomeController : Controller
{

  private IConfiguration Config { get; set; }


  public HomeController(IConfiguration config)
  {
    Config = config;
  }

  public IActionResult Info()
  {

     ViewData["appId"] = Config["vcap:application:application_id"];
     ViewData["appName"] = Config["vcap:application:application_name"];
     ViewData["uri0"] = Config["vcap:application:application_uris:0"];
     ViewData["disk"] = Config["vcap:application:limits:disk"];

     return View();
  }
}
```

# Cloud Foundry – Using Options

```
"VCAP_SERVICES": {
 "p-config-server": [
 {
 "credentials": {
   "uri":
      "https://config-bd112dd4-9870-4819-b9a6-62eb3311e27b.apps.testcloud.com",
   "client_secret": "X3e2gKs5Oqhp",
   "client_id": "p-config-server-5f0d1211-75f1-4105-9f94-7ec010de2d3a",
   "access_token_uri":
      "https://p-spring-cloud-services.uaa.system.testcloud.com/oauth/token"
 },
 "syslog_drain_url": null,
 "volume_mounts": [],
 "label": "p-config-server",
 "provider": null,
 "plan": "standard",
 "name": "myConfigServer",
 "tags": [
    "configuration",
    "spring-cloud"
 ]
}
],
"p-service-registry": [
 . . . .
```

```
public class HomeController : Controller
{
  private CloudFoundryServicesOptions Services { get; set; }
  public HomeController(
            IOptions<CloudFoundryServicesOptions> servOptions)
  {
    Services = servOptions.Value;
  }
  public IActionResult About()
  {
     foreach(var service in Services.ServicesList)
     {
       ViewData[service.Name] = service.Name;
       ViewData[service.Plan] = service.Plan;
     }
     return View();
  }
}
```

# Steeltoe Config Server Client Provider

- Enables Spring Cloud and Spring Cloud Services Config Servers to be used as a source of configuration data
- Usage
  - Add appropriate NuGet reference to project
  - Add Config Server provider to Configuration Builder (i.e. `AddConfigServer()`)
  - Configure Config Server Client settings
  - Optionally, configure any Options classes (i.e. `Configure<YourOptionsClass>()`)
  - Access config values using Options or indexer
  - Optionally, create & bind service instance for usage on Cloud Foundry

# Config Server Client – Add NuGet References

- Spring Cloud Config Server
  - Console, ASP.NET 4 - `Steeltoe.Extensions.Configuration.ConfigServer`
  - ASP.NET Core - `Steeltoe.Extensions.Configuration.ConfigServerCore`
  - ASP.NET 4 Autofac -`Steeltoe.Extensions.Configuration.ConfigServerAutofac`
  - Namespace- `#using Steeltoe.Extensions.Configuration.ConfigServer;`
- Spring Cloud  & Spring Cloud Services Config Server
  - Console, ASP.NET 4 - `Pivotal.Extensions.Configuration.ConfigServer`
  - ASP.NET Core - `Pivotal.Extensions.Configuration.ConfigServerCore`
  - ASP.NET 4 Autofac - `Pivotal.Extensions.Configuration.ConfigServerAutofac`
  - Namespace - `#using Pivotal.Extensions.Configuration.ConfigServer;`

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />
    <PackageReference Include="Pivotal.Extensions.Configuration.ConfigServerCore" Version="x.y.z" />
  <ItemGroup>
</Project>
```

# Config Server Client – Add Config Server Provider

- Add while building `WebHost`
  - In `ConfigureAppConfiguration()`
  - Use `AddConfigServer()` on provided `ConfigurationBuilder`
    - `IHostingEnvironment`
  - Order IS important
    - Client looks for its settings from previously added providers
- Pivotal version of client also adds Cloud Foundry provider to builder (i.e. `AddCloudFoundry()` )
  - Service bindings added to configuration
  - `spring:application:name` set equal to `vcap:application:name`
- Client issues REST calls to Config Server when configuration is built
  - i.e. `config.Build();`

```
public static IWebHost BuildWebHost(string[] args)
{
  var builder = new WebHostBuilder()
    .UseKestrel()
    .UseIISIntegration()
    .ConfigureAppConfiguration((hostingContext, config) => {
       var env = hostingContext.HostingEnvironment;
       config.AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
             .AddJsonFile($"appsettings.{env.EnvironmentName}.json", ….
       config.AddConfigServer(env)
       config.AddEnvironmentVariables();
       if (args != null) {
         config.AddCommandLine(args);
       }
     })

    . . . .

    return builder.Build();
}

}
```

# Config Server Client - Configure Providers Settings

- Client looks for settings in previously added providers
- Easiest to define settings in `appsettings.json`
  - Config Server endpoint
    - `spring:cloud:config:uri`
  - Application name: {AppName}
    - `spring:cloud:config:name`
    - `spring:application:name`
    - `Environment.ApplicationName` - default
  - Profiles to pull: {Profile}
    - `spring:cloud:config:env` - comma list
    - `Environment.EnvironmentName`
    - "`Production`" - default
  - Labels to try and pull: {Label}
    - `spring:cloud:config:label` - comma list
    - "`master`" – default

```
{
  "Logging": {
    "IncludeScopes": false,
    "Debug": {
      "LogLevel": {
        "Default": "Warning"
      }
    },
    "Console": {
      "LogLevel": {
        "Default": "Warning"
      }
    }
  },
 "spring": {
    "application": {
      "name": "fortuneservice"
    },
    "cloud": {
      "config": {
        "uri": "http://localhost:8888",
        "validate_certificates": false
      }
    }
  }
}
```

# Config Server Client – Partial List of Settings

- `spring:cloud:config:enabled` - enable/disable Config Server client, default(true)
- `spring:cloud:config:name` – app name to retrieve config for, default(`Environment.ApplicationName`)
- `spring:cloud:config:env` – comma separated list of profiles to request, default(“`Production`”)
- `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)
- `spring:cloud:config:timeout` - timeout in milliseconds, default(6000)

# Config Server Client – Configure Options Classes

- Data returned from Config Server integrates seamlessly with Options framework
  - Define your `Options` class
  - `Configure<Options>(config)`

```
public class Startup
{
  public Startup(IConfiguration configuration)
  {
    Configuration = configuration;
  }
  public void ConfigureServices(IServiceCollection services)
  {
    services.AddOptions();
    services.Configure<FortuneServiceOptions>(
          Configuration.GetSection("fortuneService"));
    services.AddMvc();
  }
}
```

```
public class FortuneServiceOptions
{
  public string Scheme { get; set; } = "http";
  public string Address { get; set; }
  public string RandomFortunePath { get; set; }
  public string AllFortunesPath { get; set; }}
}
```

```
eureka:
  client:
    shouldRegisterWithEureka: false
hystrix:
  stream:
    validate_certificates: false
fortuneService:
  scheme: https
  address: fortuneservice
  randomFortunePath: api/fortunes/random
  allFortunesPath: api/fortunes/all
```

# Config Server Client – Using 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 Config Servers configuration
    - Git URL, instance count, placeholders, patterns, etc.
  - Use `` `cf service` `` to check status of service
- Bind instance to applications
  - `` `cf bind-service appName cserver` ``
  - Also specify binding in `manifest.yml`
- Access is scoped to target org & space
- Remember to use "`Pivotal`" NuGets & Namespace
  - Client detects `p-config-server` binding
  - Overrides `appsettings.json` client settings with binding information

```
---
applications:
- name: fortuneService
  random-route: true
  env:
    ASPNETCORE_ENVIRONMENT: Production
  services:
   - myConfigServer
```

```
>
>cf target –o org –s space
>
>cf create-service p-config-server standard myConfigServer –c config.json
>
>cf bind-service myApp myConfigServer
>
```

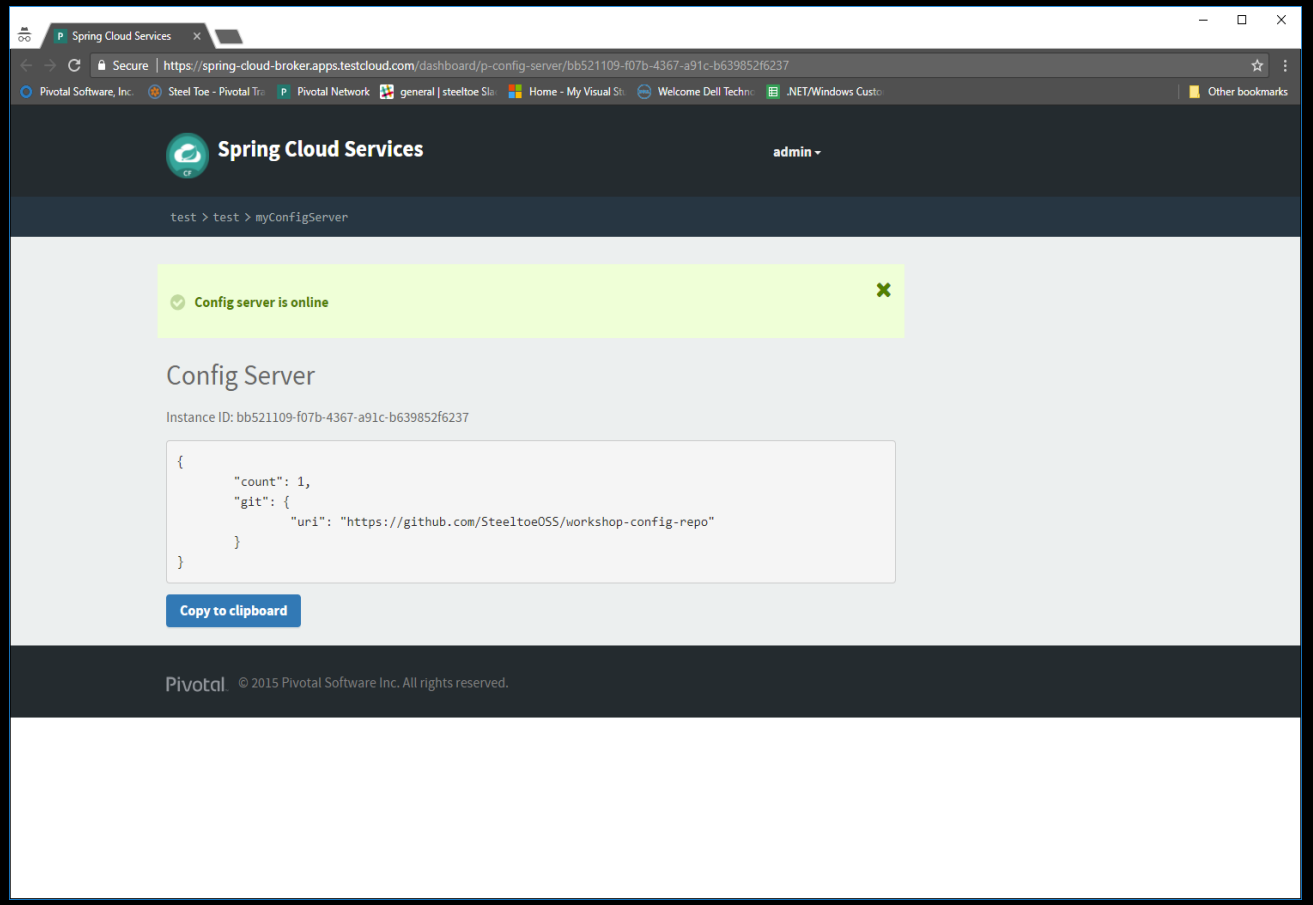
# Config Server Client - Self-signed Certificates

- Communications between Pivotal Config Server client and Spring Cloud Config Server on Cloud Foundry uses TLS/SSL
  - Default behavior for client is to validate server certificate
- Some Cloud Foundry installations will be using self-signed certificates
  - Validation will typically fail unless installation has been configured properly
- Can disable validation using setting
  - `spring:cloud:config:validate_certificates` – set to false to disable

```
{
"spring": {
   "cloud": {
      "config": {
        "uri": "http://localhost:8888",
        "validate_certificates": false
      }
    }
  }
}
```

# Spring Cloud Services Config Server Dashboard

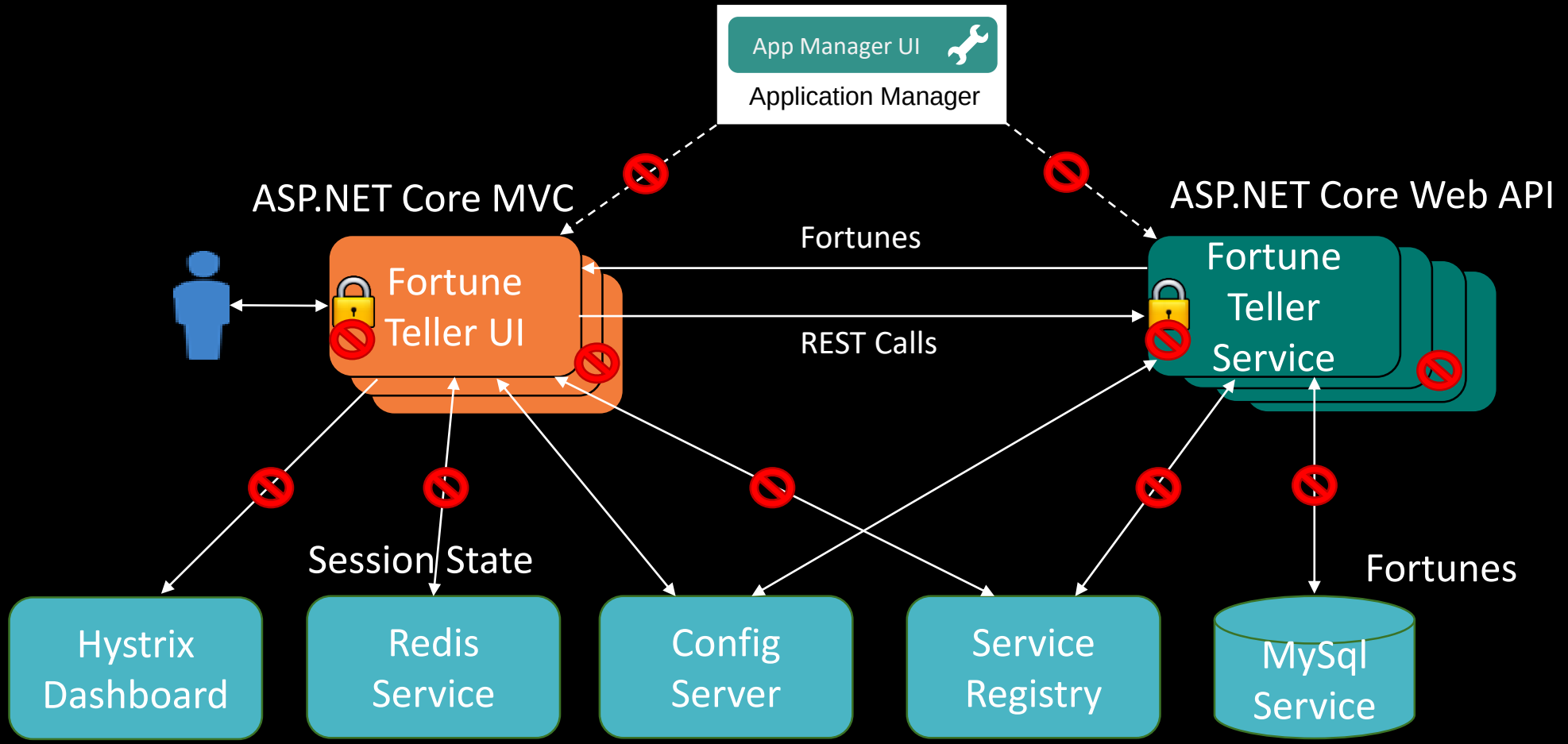
- Determine Config Server Dashboard URL using command line
  - `cf service myConfigServer`



# Lab6 – Centralize Application Configuration

- Make changes to both applications to use Spring Cloud Config Server for centralized configuration
- Follow Lab Description
  - Description: https://github.com/SteeltoeOSS/Workshop/tree/master/Lab06
  - Code: /Workshop/Start/FortuneTeller.sln

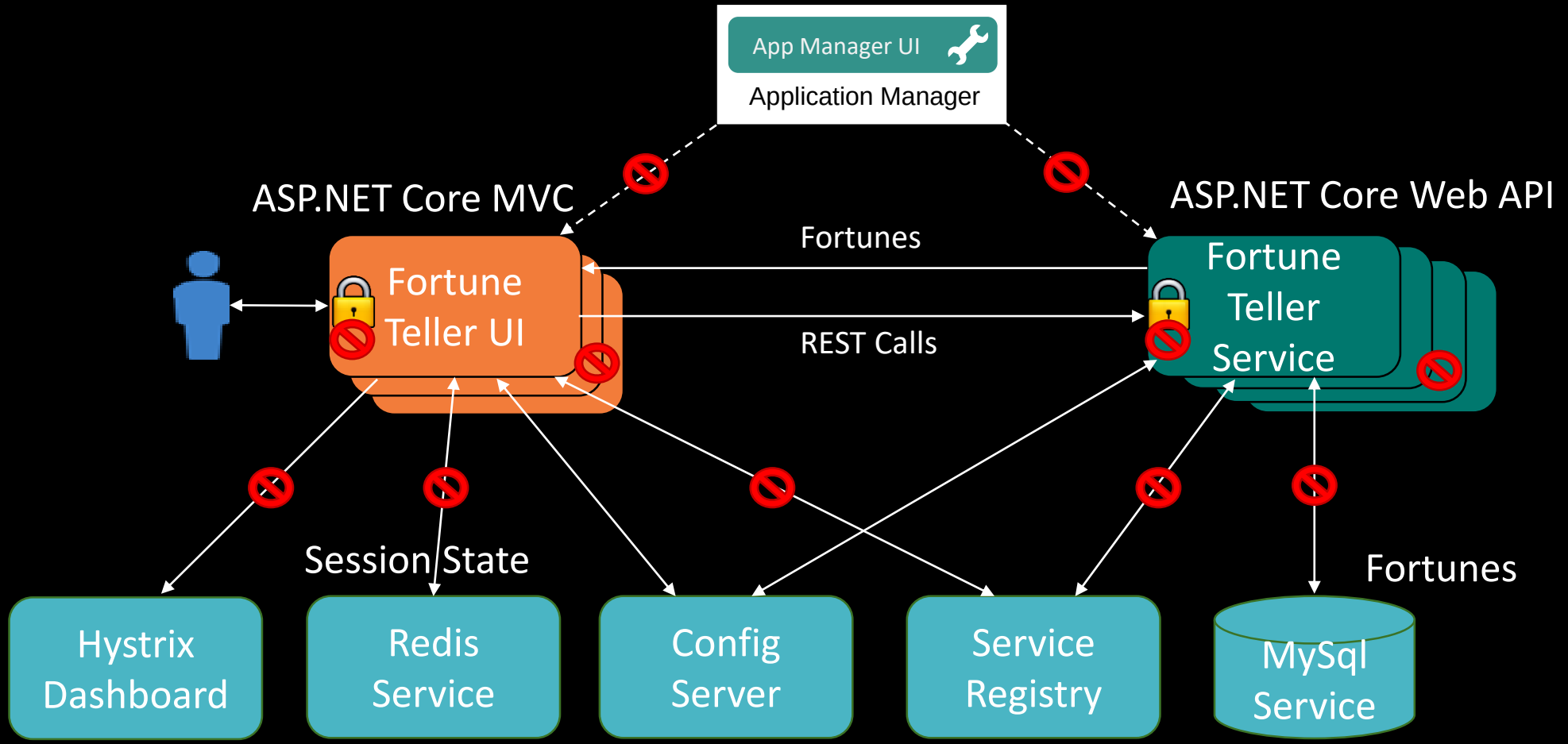
# Fortune Teller App – After Lab6



# Steeltoe Service Discovery

## SPRING CLOUD EUREKA SERVER, SCS EUREKA SERVER, EUREKA CLIENT

# Fortune Teller App – After Lab6



# Steeltoe Discovery Overview

- Provides configurable generalized interface for Service Registry interaction
  - `Steeltoe.Common.Discovery.IDiscoveryClient`
- Single provider
  - Eureka – client for Netflix / Spring Cloud Services Eureka Server
- Application type support
  - ASP.NET - MVC, WebForm, WebAPI, WCF
  - ASP.NET Core (.NET Framework and .NET Core)
  - Console apps (.NET Framework and .NET Core)

# Steeltoe Eureka Server Client

- Enables Spring Cloud and Spring Cloud Services Eureka Servers to be used as a service registry
- Usage
  - Add appropriate NuGet reference to project
  - Add CloudFoundry config provider to Configuration builder (i.e. `AddCloudFoundry()`)
    - Not needed if already using Pivotal Config Server client (i.e. `AddConfigServer()` )
  - Add Discovery Client to service container (i.e. `AddDiscoveryClient()`)
  - Configure Discovery Client settings for Eureka
  - Start the Discovery Client (i.e. `UseDiscoveryClient()`)
  - Inject a `IDiscoveryClient` and use it to lookup services
  - Optionally, create & bind service instance for usage on Cloud Foundry

# Eureka Server Client – Add NuGet References

- Spring Cloud Eureka Server
  - Console, ASP.NET 4 - `Steeltoe.Discovery.Eureka.Client`
  - ASP.NET Core - `Steeltoe.Discovery.ClientCore`
  - ASP.NET 4 Autofac -`Steeltoe.Discovery.ClientAutofac`
  - Namespace- `#using Steeltoe.Discovery.Client;`
- Spring Cloud  & Spring Cloud Services Eureka Server
  - Console, ASP.NET 4 - `Pivotal.Discovery.Eureka.Client`
  - ASP.NET Core - `Pivotal.Discovery.ClientCore`
  - ASP.NET 4 Autofac - `Pivotal.Discovery.ClientAutofac`
  - Namespace- `#using Pivotal.Discovery.Client;`

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />
    <PackageReference Include="Pivotal.Discovery.ClientCore" Version="x.y.z" />
  <ItemGroup>
</Project>
```

# Eureka Server Client – Add and Use Client

- `AddDiscoveryClient(config)`
  - Adds `IDiscoveryClient` as a Singleton to service container
    - Inject into Controllers, Views, etc.
    - Use the interface to lookup services by name
- `UseDiscoveryClient()`
  - Starts background thread fetching and registering/renewing services

```
public class Startup
{
  public IConfiguration Configuration { get; }

  public void ConfigureServices(IServiceCollection services)
  {
    services.AddOptions();

    services.AddDiscoveryClient(Configuration);

    ....
  }

  public void Configure(IApplicationBuilder app, IHostingEnvironment env)
  {
    .....
    app.UseMvc();

    app.UseDiscoveryClient();
  }

}
```

# Eureka Server Client – Configure Settings

- Settings normally contained in Configuration
  - Locally: `appsettings.json`
  - Centralized: Config Server repository
- General settings and settings for discovering service
  - `eureka:client:….`
- Settings for registering as a service
  - `eureka:instance:…`

```
Logging:
  IncludeScopes: false
  Debug:
    LogLevel:
      Default: Information
  Console:
    LogLevel:
      Default: Information
eureka:
  client:
    shouldRegisterWithEureka: false
    serviceUrl: http://localhost:8761/eureka/
    validate_certificates: false
    shouldFetchRegistry: false
  instance:
    port: 5000
```

# Eureka Server Client – Discover Settings

- Settings needed to discover services
  - Eureka Server endpoint
    - `serviceUrl`
  - Fetch registry set true
    - `shouldFetchRegistry`
  - Register as a service set false
    - `shouldRegisterWithEureka`

```
{
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Information"
    }
  },
  "spring": {
    "application": {
      "name": "fortuneUI"
    }
  },
  "eureka": {
    "client": {
      "serviceUrl": "http://localhost:8761/eureka/",
      "shouldRegisterWithEureka": false,
      "shouldFetchRegistry": true
    }
  }
}
```

# Eureka Server Client – Register Settings

- Settings needed to register as a service
  - Eureka Server endpoint
    - `serviceUrl`
  - Fetch registry set false
    - `shouldFetchRegistry`
  - Register as a service set true
    - `shouldRegisterWithEureka`
  - Service name to register
    - `eureka:instance:appName` = name
    - `spring:application:name` = name
    - `unknown` - default
  - Optional, register by IP address
    - `preferIpAddress` - default false
  - Optional, IP port to register
    - `port` - defaults to 80

```
{
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Information"
    }
  },
  "spring": {
    "application": {
      "name": "fortuneUI"
    }
  },
 "eureka": {
    "client": {
      "serviceUrl": "http://localhost:8761/eureka/",
      "shouldFetchRegistry": false,
      "shouldRegisterWithEureka": true
    },
    "instance": {
      "port": 5000,
      “preferIpAddress": false
    }
  }
}
```

# Eureka Server Client – Partial List of 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:appName` - 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)

# Eureka Server Client – Lookup Services

- Inject `IDiscoveryClient` into your Controller, View, Service, etc.
  - Use directly to access registered services: `GetInstances(name)`
  - Alternatively, use with `DiscoveryHttpClientHandler`
    - Integrates service lookup with `HttpClient` requests
    - Assumes URI host component = service name
    - Intercepts request; attempts to resolve service name to an address
    - Replaces service name with resolved address if successful
    - Leaves unchanged if no resolution

```
public class FortuneServiceClient : IFortuneService
{
  DiscoveryHttpClientHandler _handler;

  private const string URL= "http://fortuneService/api/fortunes/random";

  public FortuneServiceClient(IDiscoveryClient client)
  {
    _handler =  new DiscoveryHttpClientHandler(client);
  }

  public async Task<string> RandomFortuneAsync()
  {
     var client = GetClient();
     var result = await client.GetStringAsync(URL);
     return result;
  }

  private HttpClient GetClientAsync()
  {
    return new HttpClient(_handler, false);
  }

}
```

# Eureka Server Client – Using on Cloud Foundry

- Create instance of Config Server using CF CLI
  - `` `cf create-service p-service-registry standard dserver –c config.json` ``
  - Spins up config server in org: `p-spring-cloud-services`, space: `instances`
  - `config.json` specifies Eureka Servers configuration
    - e.g. instance count, peer Eureka servers, etc.
  - Use `` `cf service` `` to check status of service
- Bind instance to applications
  - `` `cf bind-service appName dserver` ``
  - Also specify binding in `manifest.yml`
  - Access is scoped to target org & space; with peer replication across orgs/spaces
- Remember to use "`Pivotal`" NuGets & Namespace
  - Client detects `p-service-registry` binding
  - Overrides any other Eureka settings (i.e. `appsettings.json`, etc.) with binding information
- Ensure you have added CloudFoundry Configuration provider or are using the Config Server provider
  - Service bindings added to configuration
  - `spring:application:name` set equal to `vcap:application:name`

```
---
applications:
- name: fortuneService
  random-route: true
  env:
    ASPNETCORE_ENVIRONMENT: Production
  services:
   - myDiscoveryService
```

```
>
>cf target –o org –s space
>
>cf create-service  p-service-registry standard myDiscoveryService –c config.json
>
>cf bind-service myApp myDiscoveryService
>
```

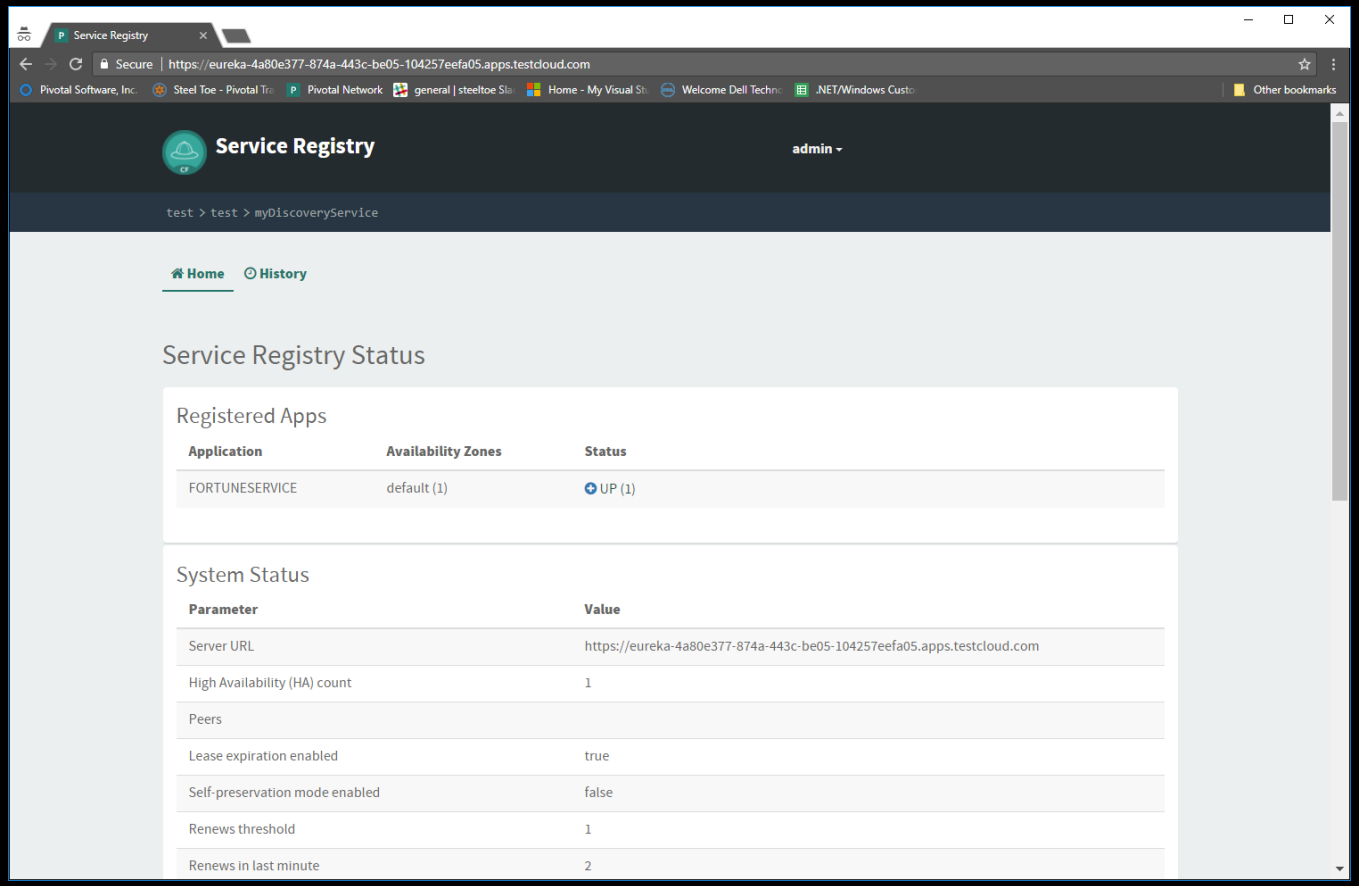
# Eureka Server Client - Self-signed Certificates

- Communications between Pivotal Eureka Server client and Spring Cloud Eureka Server on Cloud Foundry uses TLS/SSL
  - Default behavior for client is to validate server certificate
- Some Cloud Foundry installations will be using self-signed certificates
  - Validation will typically fail unless installation has been configured properly
- Can disable validation using setting
  - `eureka:client:config:validate_certificates` – set to false to disable

```
eureka:
  client:
    serviceUrl: http://localhost:8761/eureka/
    validate_certificates: false
```

# Spring Cloud Services Eureka Dashboard

- Determine Eureka Dashboard URL using command line
  - `cf service myDiscoveryService`



# Enabling Debug Logging

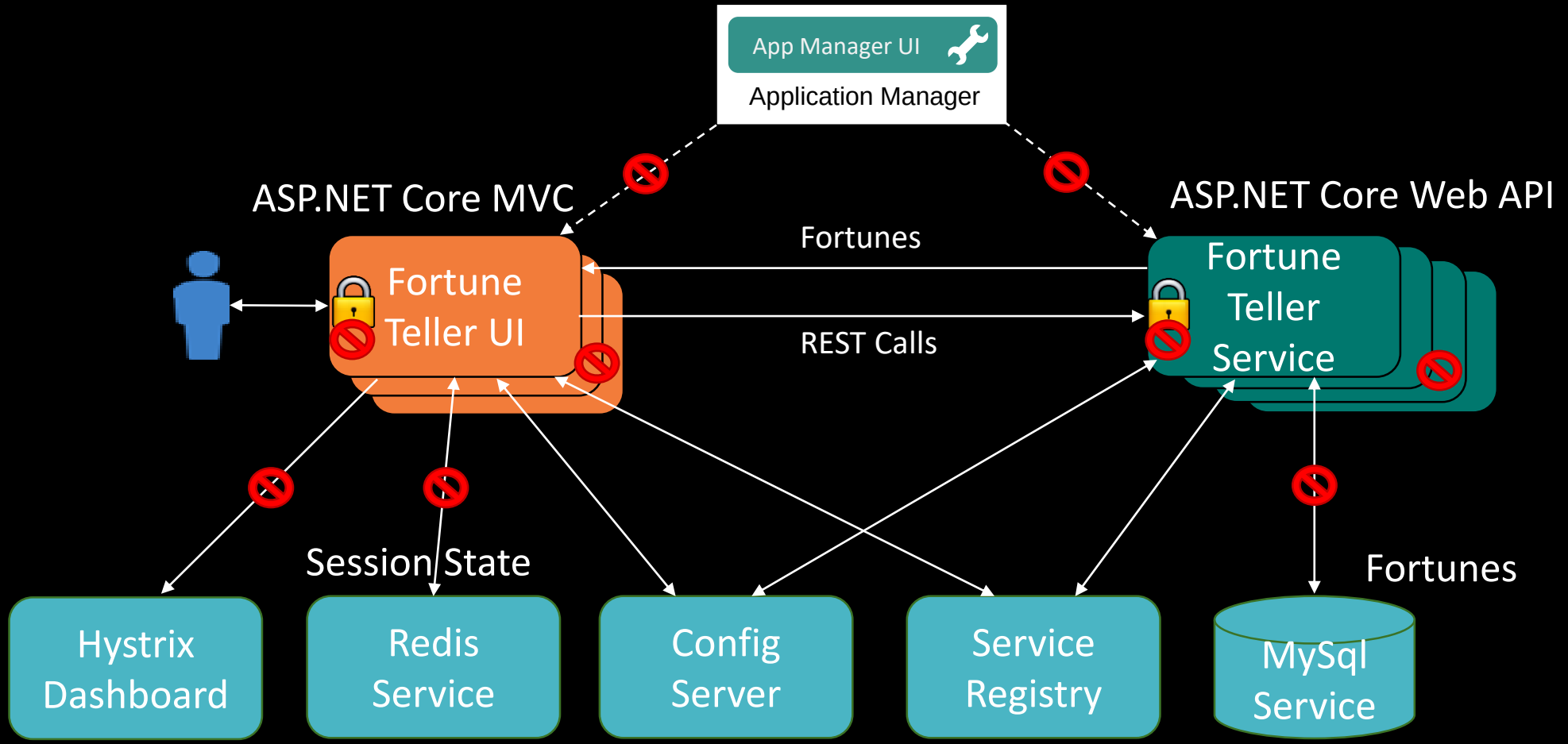
- Turn on logging for Pivotal and Steeltoe classes

```
Logging:
  IncludeScopes: false
  Debug:
    LogLevel:
      Default: Information
      Fortune_Teller_Service: Debug
      Fortune_Teller_UI: Debug
      Pivotal: Debug
      Steeltoe: Debug
  Console:
    LogLevel:
      Default: Information
      Fortune_Teller_Service: Debug
      Fortune_Teller_UI: Debug
      Pivotal: Debug
      Steeltoe: Debug
```

# Lab7 – Use Service Discovery

- Make changes to both applications to use Spring Cloud Eureka Server for service discovery
- Follow Lab Description
  - Description: https://github.com/SteeltoeOSS/Workshop/tree/master/Lab07
  - Code: /Workshop/Start/FortuneTeller.sln

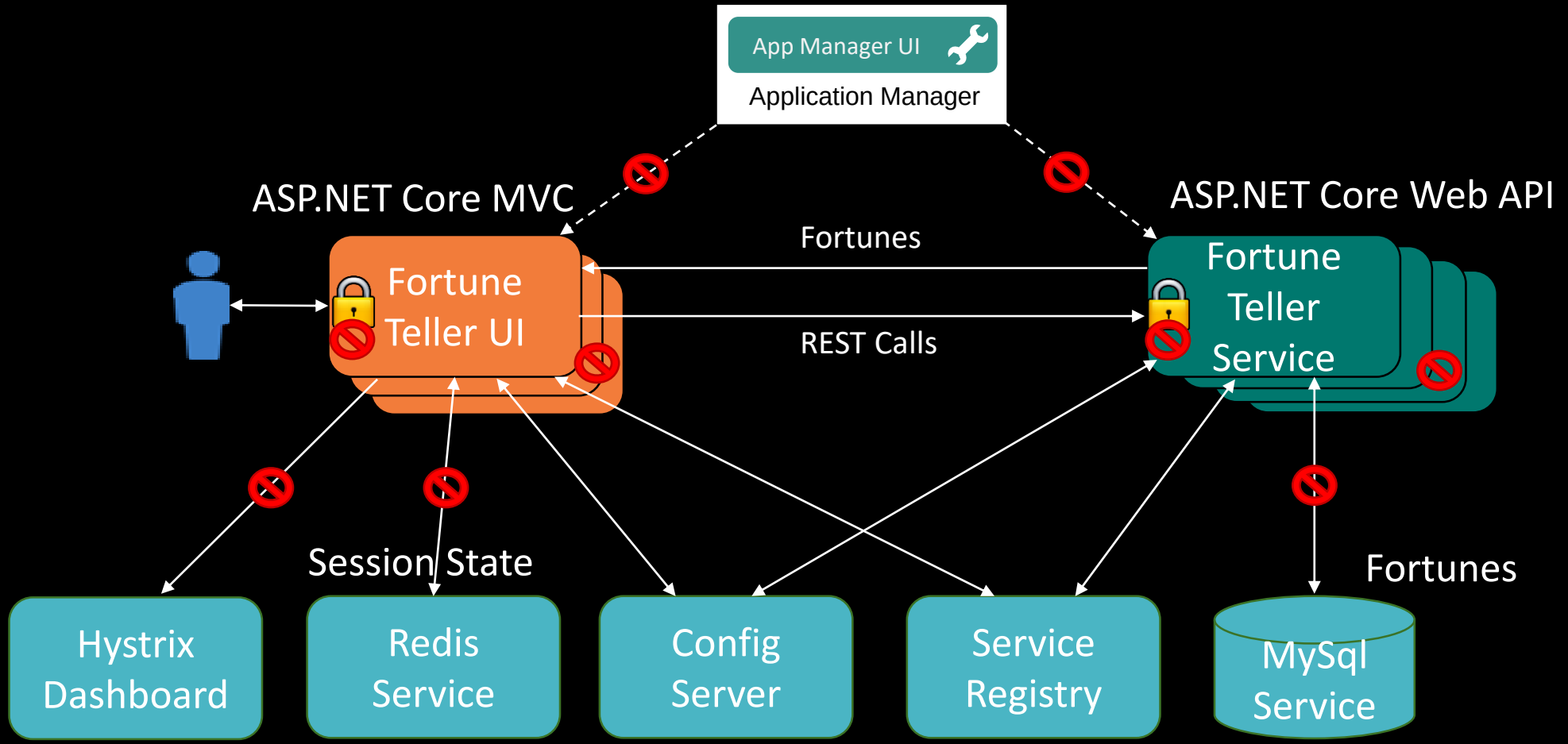
# Fortune Teller App – After Lab7



# Steeltoe Connectors

## ASP.NET CORE SESSION, KEY RING, DATA PROTECTION, CONNECTORS, MYSQL CONNECTOR, REDIS CONNECTOR

# Fortune Teller App – After Lab7



# Understanding ASP.NET Core Data Protection

- 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

# 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`
  - Get<type>()/Set<type>() methods (e.g. `GetString("fortune")`)

# Steeltoe Connectors Overview

- Simplify using Cloud Foundry services
  - Configure settings using `appsettings.json`, Config Server, etc.
  - When application pushed to Cloud Foundry, service bindings auto detected and override other settings
  - Adds Connections, Connection factories, or DbContext objects into service container
- Several Connectors
  - MySQL
  - Redis
  - Postgres
  - RabbitMQ
  - SqlServer
  - OAuth
- Application type support
  - ASP.NET - MVC, WebForm, WebAPI, WCF
  - ASP.NET Core (.NET Framework and .NET Core)
  - Console apps (.NET Framework and .NET Core)

# Steeltoe Redis Connector

- Configures and adds a ASP.NET Core `RedisCache` and/or StackExchange `IConnectionMultiplexor` to the container
  - `RedisCache` built on top of StackExchange NuGets
- Usage
  - Add Connector NuGet reference to project
  - Configure Redis client settings
  - Add `IDistributedCache` or `IConnectionMultiplexer` to service container
  - Inject `IDistributedCache` or `IConnectionMultiplexer` and use
  - Optionally, create & bind Redis service instance for usage on Cloud Foundry
    - Add CloudFoundry config provider to Configuration builder (i.e. `AddCloudFoundry()`)
      - Not needed if already using Steeltoe Config Server client (i.e. `AddConfigServer()` )

# Redis Connector – Add NuGet References

- Redis Connector
  - ASP.NET Core - `Steeltoe.CloudFoundry.ConnectorCore`
  - ASP.NET 4 - `Steeltoe.CloudFoundry.Connector`
  - Namespace - `#using Steeltoe.CloudFoundry.Connector.Redis;`
- Redis Client Providers
  - `StackExchange.Redis.StrongName` - Open Source

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />
    <PackageReference Include="Steeltoe.CloudFoundry.ConnectorCore" Version= "x.y.z" />
    <PackageReference Include="StackExchange.Redis.StrongName" Version= "x.y.z" />
<ItemGroup>

</Project>
```

# Redis Connector – Add to Service Container

- IConnectionMultiplexer
  - Use when you want full featured Redis connection
  - `AddRedisConnectionMultiplexer()` adds configured `IConnectionMultiplexer`
- IDistributedCache
  - Use when you want Session state to be stored in Redis or a simple cache abstraction
  - `AddDistributedRedisCache()` adds configured `IDistributedCache`

```
public class Startup
{
  public IConfiguration Configuration { get; }

  public void ConfigureServices(IServiceCollection services)
  {
    services.AddOptions();

    // This adds a IDistributedCache to the container
    services.AddDistributedRedisCache(Configuration);

    // This adds a IConnectionMultiplexer to the container
    services.AddRedisConnectionMultiplexer(Configuration);
  }

  public void Configure(IApplicationBuilder app, IHostingEnvironment env)
  {
    .....
    app.UseMvc();

    app.UseDiscoveryClient();
  }

}
```

# Redis Connector – Inject and Use Redis

- IDistributedCache
  - Get, Set, Remove, GetAsync, SetAsync, RemoveAsync
- IConnectionMultiplexer
  - Use `GetDatabase()`; returns `IDatabase`
  - Use `GetSubscriber()`; returns `ISubscriber`
  - All Redis database commands and data types available

```
public class HomeController : Controller
{
 private IDistributedCache _cache;
 private IConnectionMultiplexer _conn;
 public HomeController(IDistributedCache cache, IConnectionMultiplexer conn)
 {
   _cache = cache;
   _conn = conn;
 }
 public IActionResult CacheData()
 {
    string key1 = Encoding.UTF8.GetString(_cache.Get("Key1"));
    string key2 = Encoding.UTF8.GetString(_cache.Get("Key2"));
    ViewData["Key1"] = key1;
    ViewData["Key2"] = key2;
    return View();
 }
 public IActionResult ConnData()
 {
    IDatabase db = _conn.GetDatabase();
    string key1 = db.StringGet("ConnectionMultiplexerKey1");
    string key2 = db.StringGet("ConnectionMultiplexerKey2");
    ViewData["ConnectionMultiplexerKey1"] = key1;
    ViewData["ConnectionMultiplexerKey2"] = key2;
    return View();
 }
}
```

# Redis Connector - 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

```
{
 "redis": {
    "client": {
      "connectRetry": 3
    }
  }
}
```

# Redis Connector – Using 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`
- Ensure you have added CloudFoundry Configuration provider or are using the Config Server provider
  - Service bindings added to configuration
- Steeltoe Redis connector detects `p-redis` binding
  - Overrides any other Redis settings with binding information

```
---
applications:
- name: fortuneUI
  random-route: true
  env:
    ASPNETCORE_ENVIRONMENT: Production
  services:
   - myRedisService
```

```
>
>cf target –o org –s space
>
>cf create-service p-redis shared-vm myRedisService
>
>cf bind-service myApp myRedisService
>
```

# Steeltoe Redis KeyStorage Provider

- Configures `DataProtection` to use a Cloud Foundry Redis service for Key Ring storage
  - Requires StackExchange `IConnectionMultiplexor` to be available for injection
- Usage
  - Add appropriate NuGets to project
  - 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
  - Add `IConnectionMultiplexor` to service container
  - Configure Data Protection to use Redis for Key Storage
  - Create & bind Redis service instance for usage on Cloud Foundry

# Redis KeyStorage Provider – Add NuGet References

- Redis KeyStorage Provider
  - ASP.NET Core - `Steeltoe.Security.DataProtection.RedisCore`
  - Namespace - `#using Steeltoe.Security.DataProtection;`
- Redis Client Providers
  - `StackExchange.Redis.StrongName` - Open Source

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />
    <PackageReference Include="Steeltoe.Security.DataProtection.RedisCore" Version= "x.y.z" />
    <PackageReference Include="StackExchange.Redis.StrongName" Version= "x.y.z" />
<ItemGroup>

</Project>
```

# Redis KeyStorage Provider – Configure DataProtection

- Add `IConnectionMultiplexer` to service container
  - Injected into Redis Key Storage provider
- Add DataProtection to service container and configure it
  - `PersistKeysToRedis()` configures DataProtection to save key ring in Redis
  - `SetApplicationName()` enables multiple instances of the app to share protected data

```
public class Startup
{
  public IConfiguration Configuration { get; }

  public void ConfigureServices(IServiceCollection services)
  {
    services.AddOptions();

    services.AddRedisConnectionMultiplexer(Configuration);
    services.AddDataProtection()
                .PersistKeysToRedis()
                .SetApplicationName("fortuneui");
}

  public void Configure(IApplicationBuilder app, IHostingEnvironment env)
  {
    .....
    app.UseMvc();

    app.UseDiscoveryClient();
  }

}
```

# Steeltoe MySql Connector

- Configures and adds a `MySqlConnection` or `DbContext` to the container
  - Supports Oracle and Open Source ADO.NET providers
  - Supports both EntityFramework and EntityFrameworkCore `DbContext`
- Usage
  - Add Connector NuGet reference to project
  - Add CloudFoundry config provider to Configuration builder (i.e. `AddCloudFoundry()`)
    - Not needed if already using Steeltoe Config Server client (i.e. `AddConfigServer()` )
  - Configure MySql connector settings
  - Add `MySqlConnection` or `DbContext` to service container
  - Inject `MySqlConnection` or `DbContext` and use
  - Optionally, create & bind service instance for usage on Cloud Foundry

# MySql Connector – Add NuGet References

- MySql Connector
  - MySqlConnection - `Steeltoe.CloudFoundry.ConnectorCore`
  - Entity Framework Core - `Steeltoe.CloudFoundry.Connector.EFCore`
  - ASP.NET 4 - `Steeltoe.CloudFoundry.Connector`
  - Namespace - `#using Steeltoe.CloudFoundry.Connector.MySql;`
- MySql ADO.NET Providers
  - `MySqlConnector` - Open Source
  - `MySql.Data` - Oracle
- Entity Framework Core Providers
  - `Pomelo.EntityFrameworkCore.MySql` – Open Source
  - `MySql.Data.EntityFrameworkCore` - Oracle

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />
    <PackageReference Include="Steeltoe.CloudFoundry.Connector.EFCore" Version= "x.y.z" />
    <PackageReference Include="Pomelo.EntityFrameworkCore.MySql" Version= "x.y.z" />
  <ItemGroup>

</Project>
```

# MySql Connector - 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:sslMode` – SSL usage option, None, Preferred, Required, default(none)
- `mysql:client:connectionString` - full connection string, default(empty), use instead of the above

```
mysql:
  client:
    database: mydatabase
    username: username
    password: password
```

# MySql Connector – Add MySql to Service Container

- Using Entity Framework Core
  - `AddDbContext<T>(optsAction)`
  - Provide `Action<DbContextBuilder>` that configures the `DbContext`
    - Use Steeltoe Connector method `options.UseMySql(config)`
  - `DbContext` added to container
- Using MySql Connection
  - `AddMySqlConnection(config)` configures connection
  - `MySqlConnection` added to container

```
public class Startup
{
  public IConfiguration Configuration { get; }

  public void ConfigureServices(IServiceCollection services)
  {
    services.AddOptions();
    if (Environment.IsDevelopment()) {
      services.AddEntityFrameworkInMemoryDatabase()
              .AddDbContext<FortuneContext>(
                  options => options.UseInMemoryDatabase("Fortunes"));
    } else {
      services.AddDbContext<FortuneContext>(
                    options => options.UseMySql(Configuration));
    }
  }

  public void Configure(IApplicationBuilder app, IHostingEnvironment env)
  {
    .....
    app.UseMvc();

    app.UseDiscoveryClient();
  }

}
```

# MySql Connector – Inject and Use MySql

- Using Entity Framework Core
  - Inject `DbContext` type and use normally
- Using MySql Connection
  - Inject `MySqlConnection`
  - `Open/Close` connection
  - Use normally

```
public class HomeController : Controller
{
 public IActionResult MySqlData([FromServices] MySqlConnection dbConnection)
 {
   var viewData = new Dictionary<string, string>();
   dbConnection.Open();
   MySqlCommand cmd =
          new MySqlCommand("SELECT * FROM TestData;", dbConnection);
   DbDataReader rdr = cmd.ExecuteReader();

   while (rdr.Read())
   {
      viewData.Add(rdr[0].ToString(), rdr[1].ToString());
   }
      ViewBag.Database = dbConnection.Database;
      ViewBag.DataSource = dbConnection.DataSource;

      dbConnection.Close();

     return View(viewData);
 }
}
```

# MySql Connector - Using 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`
- Ensure you have added CloudFoundry Configuration provider or are using the Config Server provider
  - Service bindings added to configuration
- Steeltoe MySql connector detects `p-mysql` binding
  - Overrides any other MySql settings with binding information

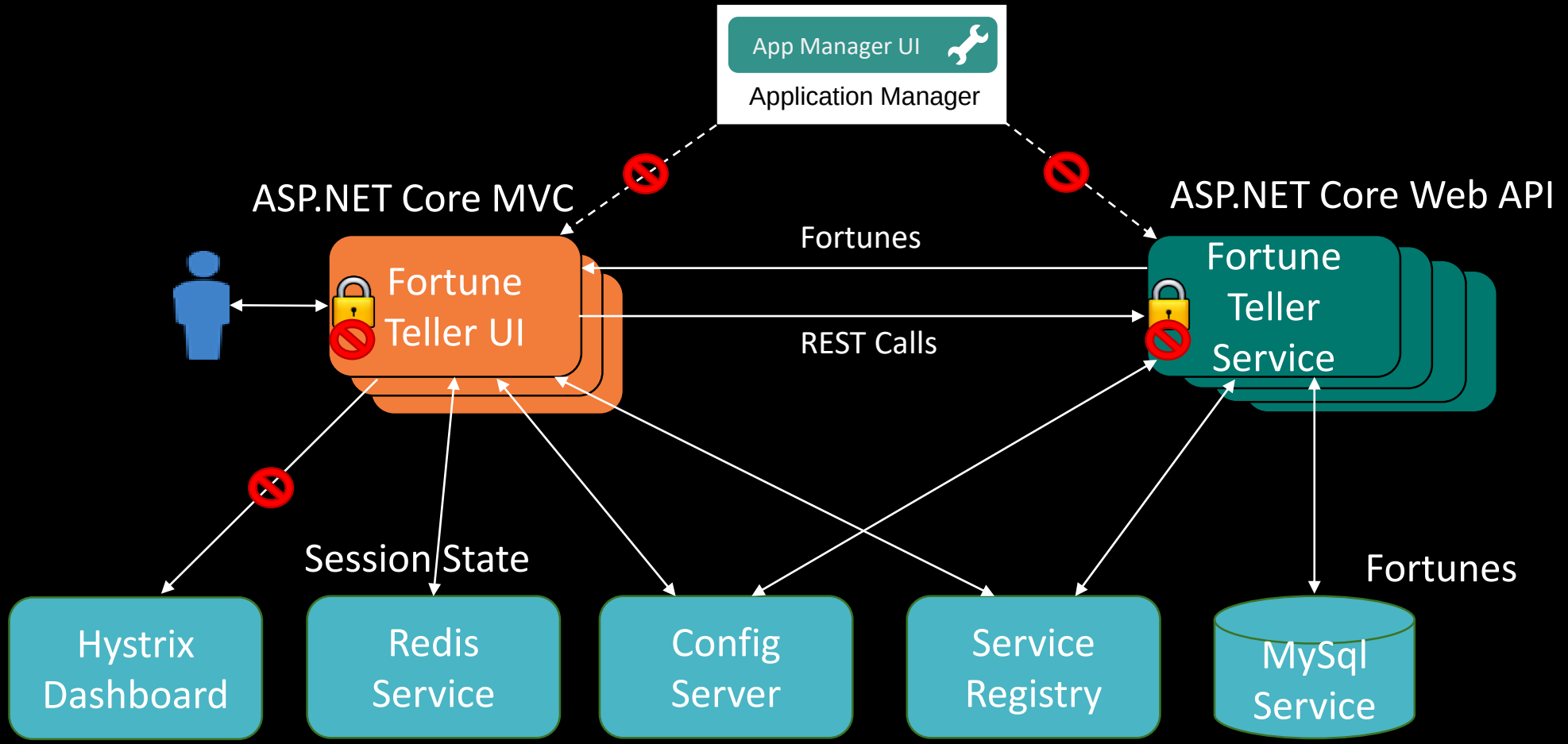
```
---
applications:
- name: fortuneService
  random-route: true
  env:
    ASPNETCORE_ENVIRONMENT: Production
  services:
   - myDiscoveryService
```

```
>
>cf target –o org –s space
>
>cf create-service p-mysql 100mb myMySqlService
>
>cf bind-service myApp myMySqlService
>
```

# Lab8 – Scaling Horizontally

- Use MySql Connector to connect the Fortune Teller Service to MySql service
  - Store Fortunes in MySql
- Use Redis Connector to connect the Fortune Teller UI to Redis service
  - Use Redis for session storage
  - Use Redis for key ring storage
- Scale both components horizontally on Cloud Foundry
- Follow Lab Description
  - Description: https://github.com/SteeltoeOSS/Workshop/tree/master/Lab08
  - Code: /Workshop/Start/FortuneTeller.sln

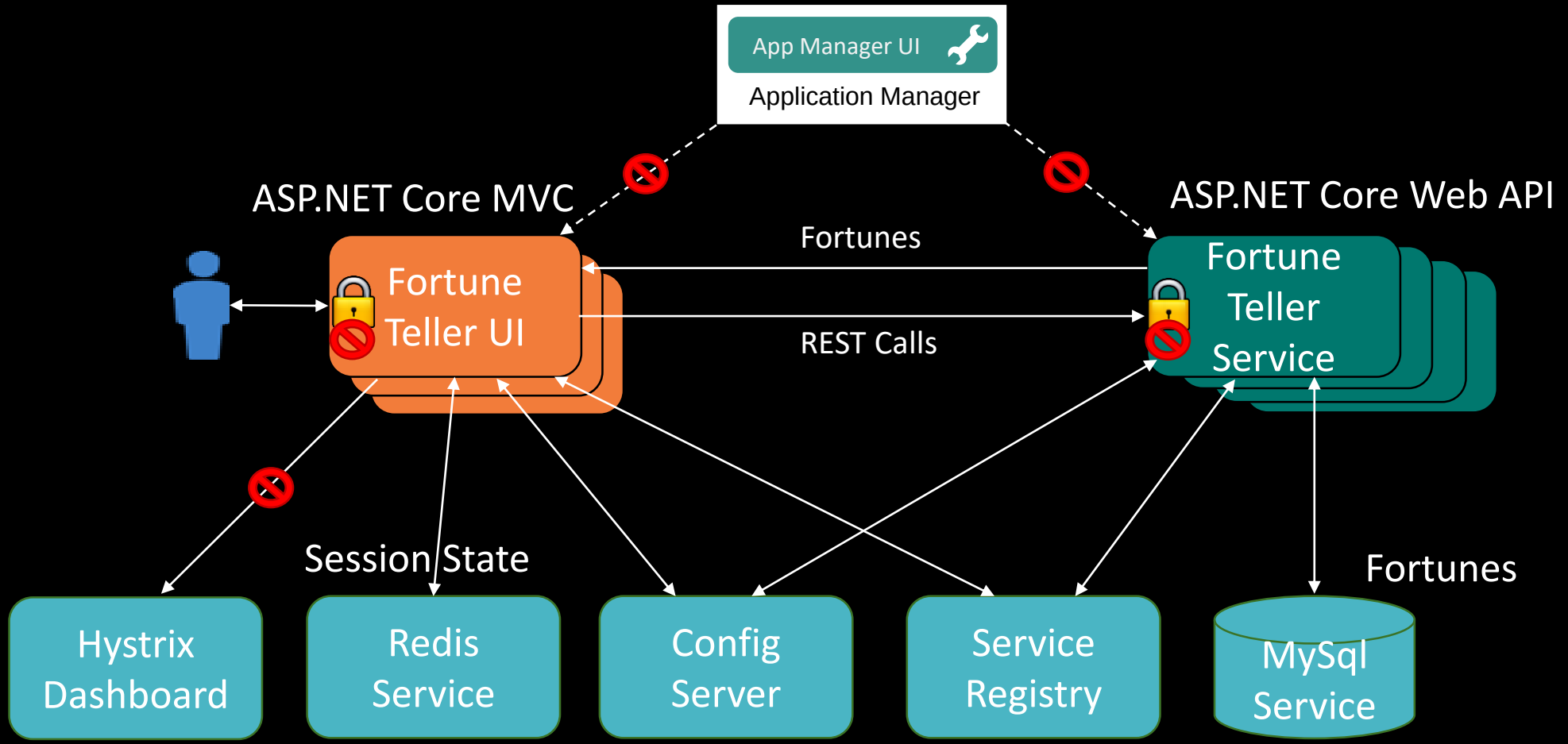
# Fortune Teller App – After Lab8



# Steeltoe Circuit Breakers

## ASP.NET CORE, HYSTRIX, SPRING CLOUD HYSTRIX DASHBOARD

# Fortune Teller App – After Lab8



# Steeltoe Circuit Breaker Overview

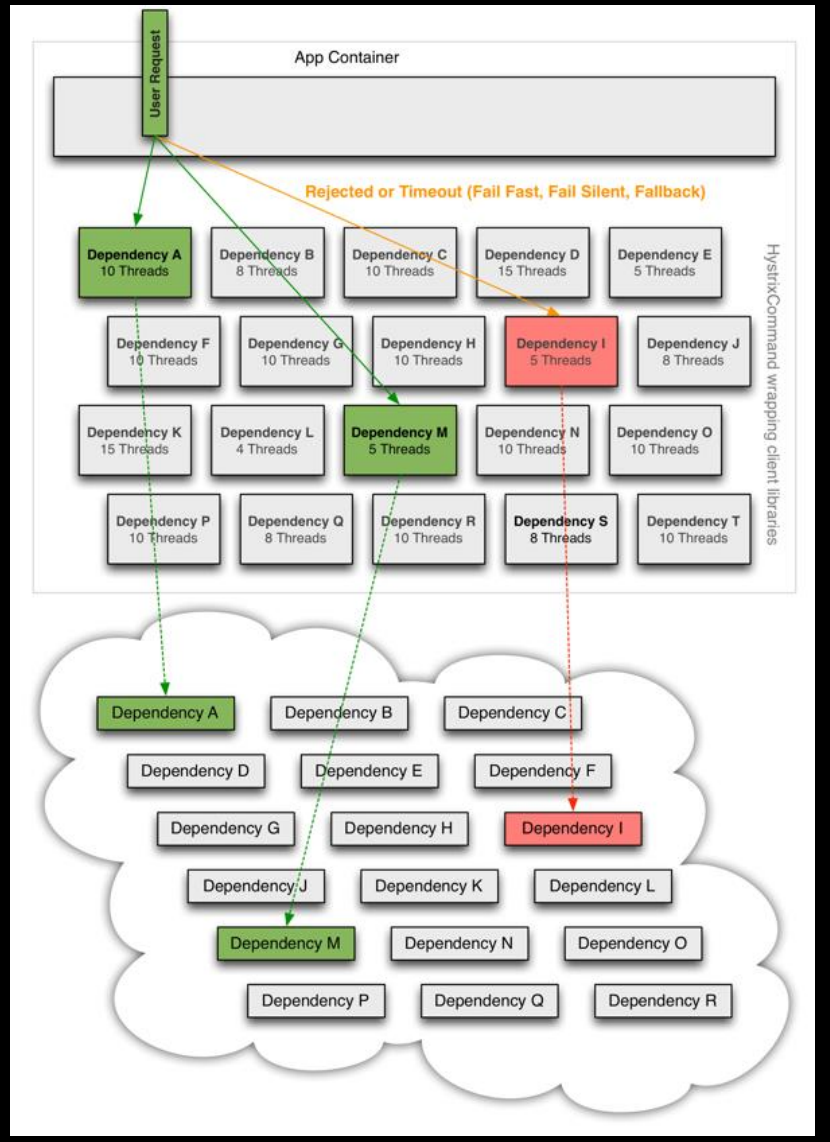
- Provides frameworks which enable .NET developers to leverage common distributed system fault tolerance patterns (e.g. Circuit Breaker, Bulkhead, Swimlane, etc.) in their applications
- Single framework
  - Hystrix – latency and fault tolerance library with real-time monitoring capabilities
- Application type support
  - ASP.NET - MVC, WebForm, WebAPI, WCF
  - ASP.NET Core (.NET Framework and .NET Core)
  - Console apps (.NET Framework and .NET Core)

# Hystrix

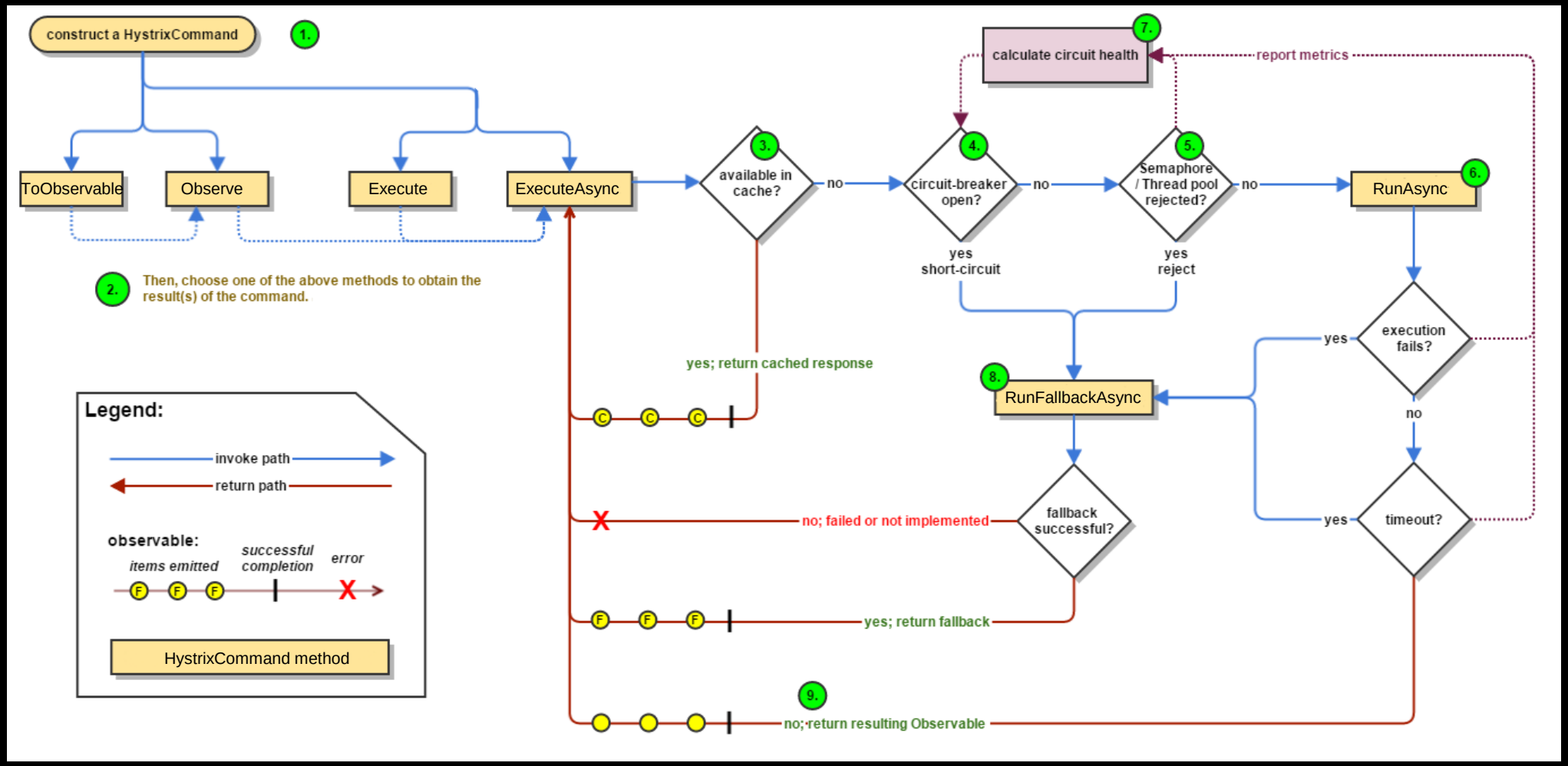
- Hystrix consists of two parts
  - Fault tolerance framework for dealing with failures and latency in distributed systems
    - Isolation patterns to limit impacts
  - Near real-time monitoring & alerting
    - Shorten time-to-discover and time-to-recover from failures

# Hystrix Fault Tolerance Framework

- Wrap all external dependencies in Hystrix commands
  - Executed on separate thread
  - Execute fallback logic on failures
- Configurable thread pools for each dependency
  - Fail fast if pool consumed
- Measures success, failures, timeouts, thread rejections, etc.
- Trip circuit-breaker to stop requests to failed dependency
  - Periodically check
- Gather metrics and report back to a dashboard for monitoring



# Hystrix Command Flow



# Steeltoe Hystrix

- Implementation of the Hystrix command pattern for .NET applications
  - Built using .NET Tasks & underlying .NET thread pool
  - Includes monitoring (i.e. metrics, status, etc.) using the Netflix Dashboard or the Spring Cloud Services Dashboard
- Usage
  - Add NuGet references to project
  - Add CloudFoundry config provider to Configuration builder (i.e. `AddCloudFoundry()`)
    - Not needed if already using Pivotal Config Server client (i.e. `AddConfigServer()` )
  - Create HystrixCommand(s) which wrap external dependency calls
  - Add HystrixCommand(s) to service container (i.e. `AddHystrixCommand()`)
    - Can also simply new <commands> as needed
  - Optionally, add and use Hystrix Metrics stream if using Dashboard
  - Configure any needed settings
  - Use HystrixCommand(s) in application
  - Optionally, create & bind Hystrix Dashboard service instance for usage on Cloud Foundry

# Hystrix – Add NuGet References

- Hystrix
  - Console, ASP.NET 4 - `Steeltoe.CircuitBreaker.Hystrix.Core`
  - ASP.NET Core - `Steeltoe.CircuitBreaker.HystrixCore`
  - ASP.NET 4 Autofac -`Steeltoe.CircuitBreaker.HystrixAutofac`
  - Namespace- `#using Steeltoe.CircuitBreaker.Hystrix;`
- Hystrix Metrics Dashboard
  - Netflix – `Steeltoe.CircuitBreaker.Hystrix.MetricsEventsCore`
  - Spring Cloud Services - `Steeltoe.CircuitBreaker.Hystrix.MetricsStreamCore`
    - RabbitMQ client – `RabbitMQ.Client`

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />
    <PackageReference Include="Steeltoe.CircuitBreaker.Hystrix.HystrixCore" Version= "x.y.z" />
    <PackageReference Include="Steeltoe.CircuitBreaker.Hystrix.MetricsStreamCore" Version= "x.y.z" />
    <PackageReference Include="RabbitMQ.Client" Version="“x.y.z" />
  <ItemGroup>
</Project>
```

# Hystrix – Create HystrixCommand

- Define class which derives from `HystrixCommand<T>`
  - <T> is what the command returns
  - First argument to constructor should be `IHystrixCommandOptions` if using dependency injection
    - Add other arguments as needed
  - Override and implement
    - `RunAsync()` – code executed when command runs
    - `RunFallbackAsync()` – code executed when `RunAsync()` fails
  - Define any other methods or properties you want
- Hystrix commands are stateful objects
  - Can't be reused

```
public class FortuneServiceCommand : HystrixCommand<Fortune>
{
   IFortuneService _fortuneService;
   public FortuneServiceCommand(IHystrixCommandOptions options,
            IFortuneService fortuneService) : base(options)
   {
     _fortuneService = fortuneService;
     IsFallbackUserDefined = true;
   }
   public async Task<Fortune> RandomFortuneAsync()
   {
     return await ExecuteAsync();
   }
   protected override async Task<Fortune> RunAsync()
   {
     return await _fortuneService.RandomFortuneAsync();
   }
   protected override async Task<Fortune> RunFallbackAsync()
   {
     return await Task.FromResult<Fortune>(
         new Fortune() { Id = 9999, Text = "You will have a happy day!" });
   }
}
```

# Hystrix – Add Command and Metrics Stream

- `AddHystrixCommand<>` used to configure and add commands to container
  - Requires first argument to be `IHystrixCommandOptions`
  - Adds with lifetime of Transient
  - Optional, can `new` commands as needed
- `AddHystrixMetricsStream` configures Hystrix to stream metrics to dashboard
  - Requires Hystrix context to be established for each incoming request
    - `UseHystrixRequestContext()`
- `UseHystrixMetricsStream` starts background thread pushing metrics to dashboard
  - Uses RabbitMQ with Spring Cloud Services dashboard
  - Uses HTTP/SSE with Netflix dashboard

```
public class Startup
{
  public IConfiguration Configuration { get; }

  public void ConfigureServices(IServiceCollection services)
  {
    services.AddOptions();

    services.AddHystrixCommand<FortuneServiceCommand>(
                                    "FortuneService", Configuration);
    services.AddHystrixMetricsStream(Configuration);
  }

  public void Configure(IApplicationBuilder app, IHostingEnvironment env)
  {
    . . .
    app.UseStaticFiles();

    app.UseHystrixRequestContext();

    app.UseMvc();

    app.UseHystrixMetricsStream();
  }

}
```

# Hystrix – Using Hystrix Command in Application

- Inject Hystrix command if you used `AddHystrixCommand<>`
  - Also `new` an instance
- Four ways to execute
  - `Execute()` – blocks, returns single value
  - `ExecuteAsync()` – returns Task, which returns single value when complete
  - `Observe()` – returns a “hot” observable which can return multiple values
  - `ToObservable()` – returns a “cold” observable which when you subscribe to it starts execution

```
public class FortunesController : Controller
{
  private FortuneServiceCommand _fortunes;
  public FortunesController(FortuneServiceCommand fortunes)
  {
    _fortunes = fortunes;
  }
  public async Task<IActionResult> RandomFortune()
  {
     var fortune = await _fortunes.ExecuteAsync();
     return View(fortune);

  }
}
```

# Hystrix – Command  & Thread Pool Settings

- Four levels of settings
  - Fixed global – defaults for all Hystrix commands or thread pools; used if nothing else specified
  - Configured global – defaults specified in configuration files which override fixed global settings
  - Configured named settings – settings specified in configuration files targeted at named commands or thread pools; apply to all instances created with that name
  - Code settings – settings applied in command or thread pool constructors; apply to that specific instance
- All Command settings use `hystrix:command` as prefix
  - Configured global use `hystrix:command:default`
  - Configured named use `hystrix:command:<name>`
- All Thread Pool settings use `hystrx:threadpool` as prefix
  - Configured global use `hystrix:threadpool:default`
  - Configured named use `hystrix:threadpool:<name>`

# Hystrix – Command Setting Areas

- Settings that control how commands are executed
  - Execution – controls how RunAsync() is executed
  - Fallback – controls how RunFallbackAsync() is executed
  - CircuitBreaker – controls the behavior of default Hystrix Circuit Breaker
  - Metrics – controls how metrics are captured and reported
  - Request cache – enables or disables request caching
  - Request logging – enables or disables request logs
- Settings that control how Hystrix manages its thread pools
  - Sizing – controls various sizing values for thread pools
  - Metrics – controls how metrics are captured and reported
- See http://steeltoe.io/docs/ for details on all the possible settings

# Hystrix – Execution Settings

- `execution:timeout:enabled` – enable or disable, default(true)
- `execution:isolation:strategy` – THREAD or SEMAPHORE, default(THREAD)
- `execution:isolation:thread:timeoutInMilliseconds` – time allowed for RunAsync(), default(1000)
- `execution:isolation:semaphore:maxConcurrentRequests` – max usage of RunAsync() when SEMAPHORE, default(10)

```
hystrix:
  command:
    foobar:
      execution:
        timeout:
          enabled: false
```

# Hystrix - Using Dashboard on Cloud Foundry

- Create instance of Circuit Breaker Dashboard service using CF CLI
  - `` `cf create-service p-circuit-breaker-dashboard standard myHystrixService` ``
- Bind instance to applications
  - `` `cf bind-service appName myHystrixService` ``
  - Also specify binding in `manifest.yml`
- Ensure you have added CloudFoundry Configuration provider or are using the Config Server provider
  - Service bindings added to configuration
- Steeltoe Hystrix Metrics Stream detects `p-circuit-breaker-dashboard` binding
  - Overrides any other settings with binding information

```
---
applications:
- name: fortuneui
  random-route: true
services:
   - myHystrixService
```

```
>cf target –o org –s space
>
>cf create-service p-circuit-breaker-dashboard standard myHystrixService
>
>cf bind-service myApp myHystrixService
```

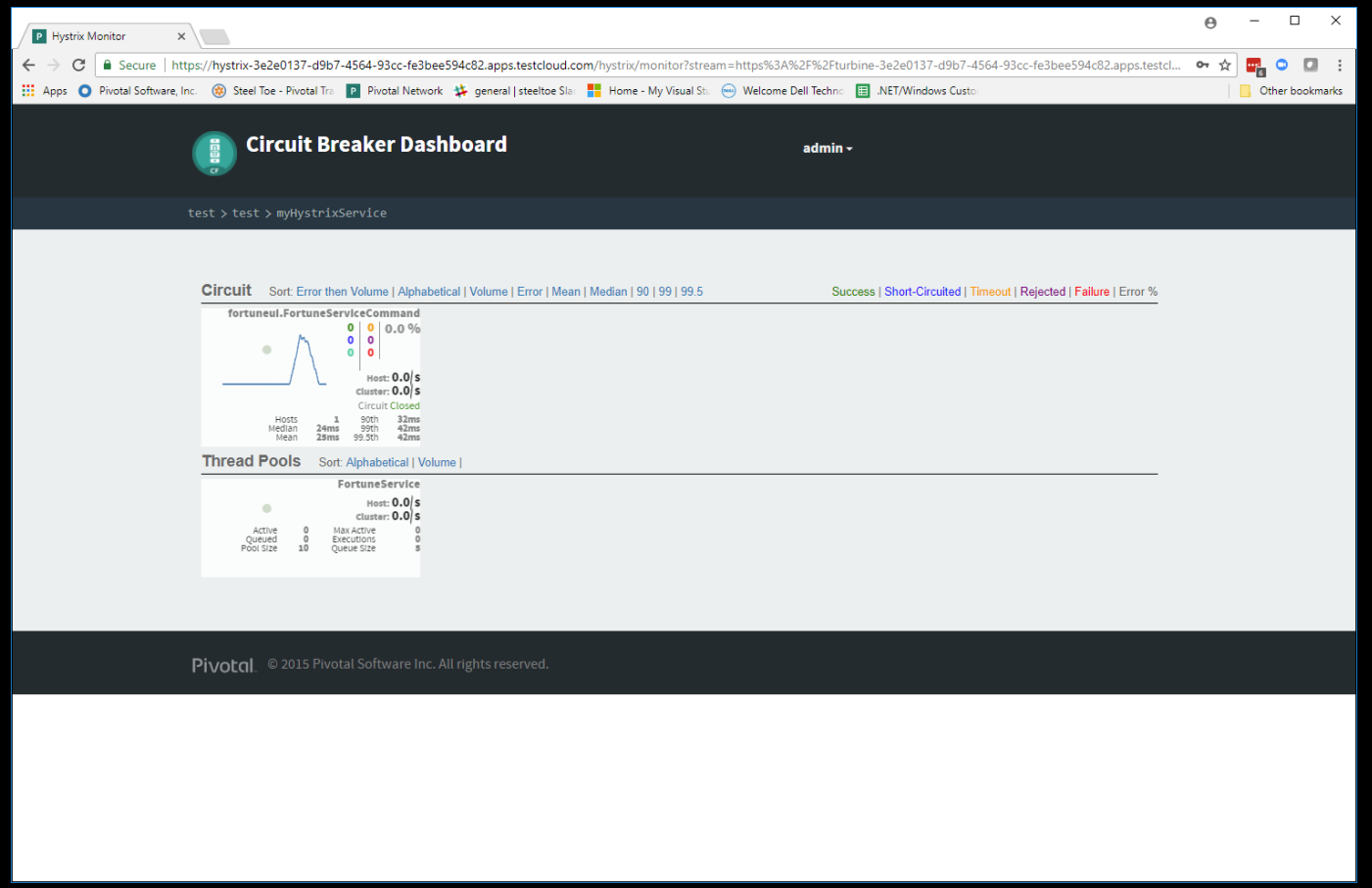
# Hystrix - Self-signed Certificates

- Communications between Steeltoe Hystrix Metrics Stream client and Spring Cloud Hystrix Dashboard on Cloud Foundry may use TLS/SSL
  - Default behavior for RabbitMQ client is to validate server certificate
- Some Cloud Foundry installations will be using self-signed certificates
  - Validation will typically fail unless installation has been configured properly
- Can disable validation using setting
  - `hystrix:stream:validate_certificates` – set to false to disable

```
hystrix:
  stream:
    validate_certificates: false
```

# Spring Cloud Services Hystrix Dashboard

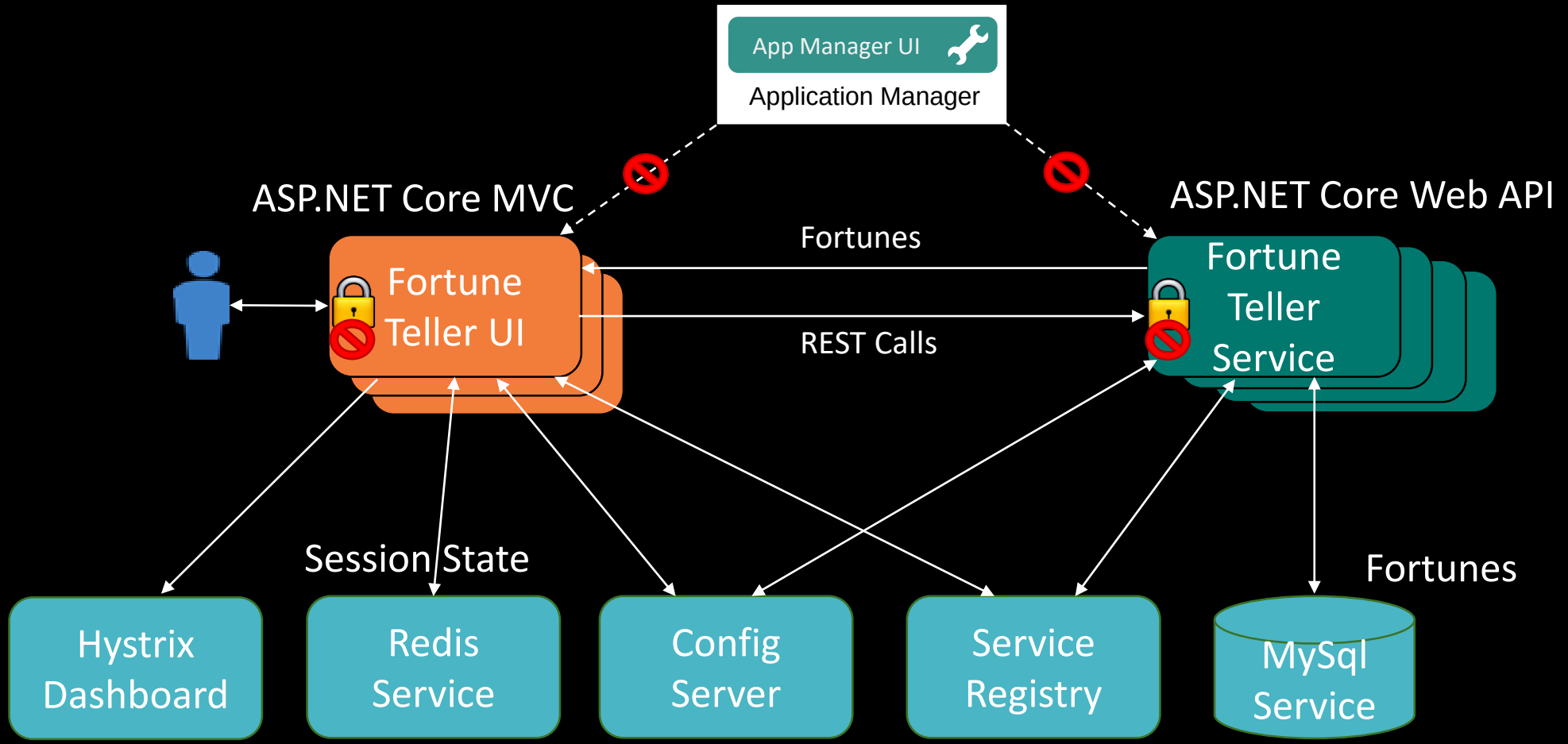
- Determine Hystrix Dashboard URL using command line
  - `cf service myHystrixService`



# Lab9 – Fault Tolerance and Monitoring

- Change Fortune Teller UI to use a `Netflix Hystrix Command` to wrap Fortune Teller Service REST requests
  - Implement a fallback function that handles request failures
- Change Fortune Teller UI to report command metrics/status to the Hystrix dashboard
- Kill Fortune Teller service and see that you can still get Fortunes
- View status in Hystrix Dashboard as you induce failures
- Follow Lab Description
  - Description: https://github.com/SteeltoeOSS/Workshop/tree/master/Lab09
  - Code: /Workshop/Start/FortuneTeller.sln

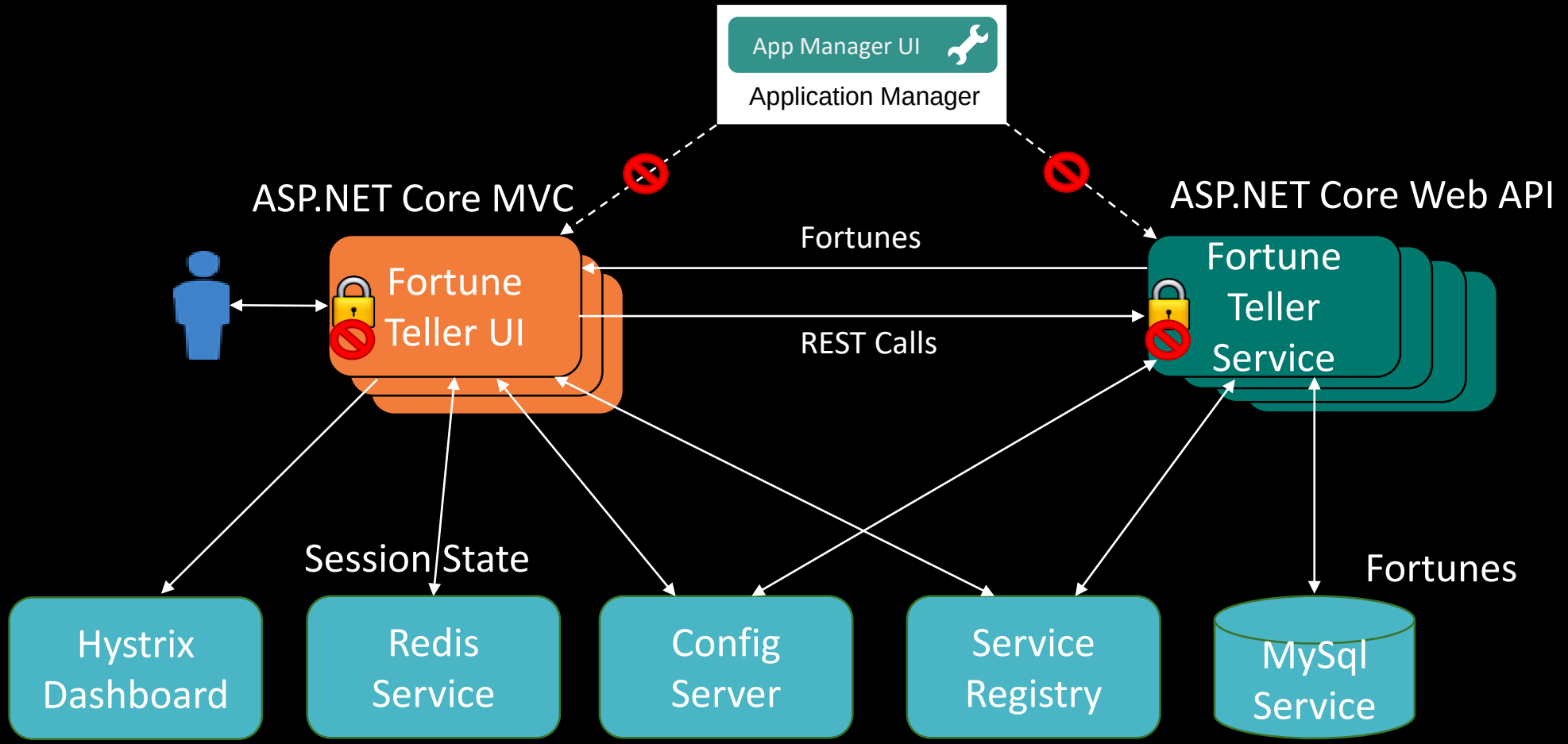
# Fortune Teller App – After Lab9



# Steeltoe Security

## ASP.NET CORE, OAUTH, ACCESS TOKENS, JAVA WEB TOKENS (JWT)

# Fortune Teller App – After Lab9



# Steeltoe Security Provider Overview

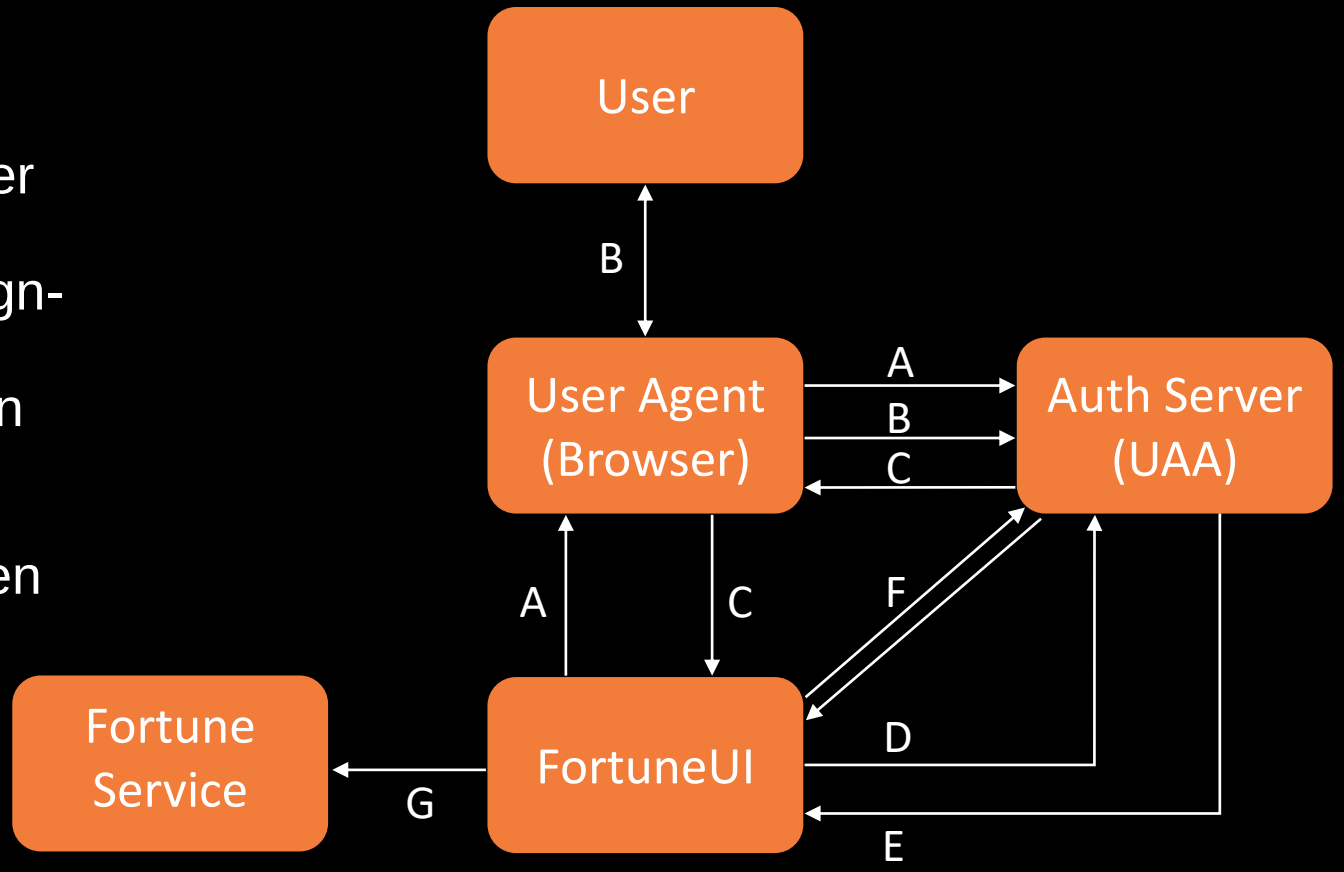
- Security providers that simplify using Cloud Foundry based security services
  - Enable using UAA Server and/or Pivotal Single Sign-on service for Authentication and Authorization services
- Security Providers
  - OAuth2
  - Java Web Tokens (JWT)
- Application type support
  - ASP.NET - MVC, WebForm, WebAPI, WCF
  - ASP.NET Core (.NET Framework and .NET Core)
  - Console apps (.NET Framework and .NET Core)

# Steeltoe OAuth2 Provider

- Enables applications to implement an OAuth 2.0 Authorization Code grant flow using security services provided by Cloud Foundry
  - UAA Server
  - Pivotal Single Sign-on service
- Usage
  - Configure user credentials and applications in UAA Server
  - Add CloudFoundry config provider to Configuration builder (i.e. `AddCloudFoundry()`)
    - Not needed if already using Steeltoe Config Server client (i.e. `AddConfigServer()` )
  - Add NuGet references to project
  - Add Authentication and Authorization services to container
  - Add Authentication middleware to pipeline
  - Apply Authorization attributes
  - Implement access token forwarding as needed
  - Configure Oauth provider settings
  - Create & bind OAuth service instance for usage on Cloud Foundry

# Understanding OAuth2 Authorization Code Grant Flow

- A - Application redirects browser to Authorization Server
  - Applications Client Id
- B- User authenticates using Auth Server login screen
- C - Auth Server redirects browser to sign-in endpoint with authorization code
- D - Application exchanges authorization code
  - Applications Client Id and Client Secret
- E - Auth Server returns an Access Token
- F - Application uses the Auth Server to decode Token
  - Claims in token used to build identity
- G - Application forwards access token



# OAuth2 Provider – Configure User Credentials & Application

- Configure User credentials
  - Add user and credentials
    - e.g. “read.fortunes”
- Configure application
  - Applications client id
  - Applications client secret
  - Scopes
- Already done for lab

```
> # Configure Users Credentials
>
>uaac target uaa.system.testcloud.com
>
>uaac token client get admin –s ADMIN_CLIENT_SECRET
>
>uaac group add read.fortunes
>
>uaac user add dave –-given_name Dave –-family_name Tillman –-emails dave@testcloud.com
  –-password myPassword
>
>uaac member add read.fortunes dave
>
> # Configure Application
>uaac client add myApp
  –-scope cloud_controller.read,cloud_controller_service_permissions.read,openid,read.fortunes
  --authorized_grant_types authorization_code,refresh_token
  --authorities uaa.resource
  --redirect_uri http://fortuneui-*-*.testcloud.com/signin-cloudfoundry
  --autoapprove cloud_controller.read,cloud_controller_service_permissions.read,openid,read.fortunes
  --secret mySecret
```

# OAuth2 Provider – Add NuGet References

- OAuth2 Provider
  - ASP.NET Core - `Steeltoe.Security.Authentication.CloudFoundryCore`
  - ASP.NET 4 OWIN - `Steeltoe.Security.Authentication.CloudFoundryOwin`
  - Namespace - `#using Steeltoe.Security.Authentication.CloudFoundry;`

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />
    <PackageReference Include="Steeltoe.Security.Authentication.CloudFoundryCore" Version= "x.y.z" />
<ItemGroup>

</Project>
```

# OAuth2 Provider – Add Authentication and Authorization

- Add & configure ASP.NET Core Authentication services
  - Default Scheme – Cookie
  - Default Challenge – Cloud Foundry
- Add Cookie Authentication scheme
  - Configure access denied path
- Add CloudFoundry OAuth2 Authentication scheme
  - `AddCloudFoundryOAuth(config)`
- Add ASP.NET Core Authorization services
  - Configure Authorization policies as needed
- Add ASP.NET Core Authentication middleware to request processing pipeline
  - `UseAuthentication()`

```
public class Startup
{
  public IConfiguration Configuration { get; }
  public void ConfigureServices(IServiceCollection services)
  {
    services.AddOptions();

    services.AddAuthentication((options) => {
      options.DefaultScheme=CookieAuthenticationDefaults.AuthenticationScheme;
      options.DefaultChallengeScheme=CloudFoundryDefaults.AuthenticationScheme
    })
    .AddCookie((options) => {
      options.AccessDeniedPath = new PathString("/Fortunes/AccessDenied");
    })
    .AddCloudFoundryOAuth(Configuration);

    services.AddAuthorization(options => {
      options.AddPolicy("read.fortunes",
         policy => policy.RequireClaim("scope", "read.fortunes"));
    });
  }

  public void Configure(IApplicationBuilder app, IHostingEnvironment env)
  {
    .....
    app.UseMvc();

    app.UseAuthentication();
  }
}
```

# OAuth2 Provider – Apply Authorization Attributes

- Add Authorize attributes to controller actions to enable security
  - Optionally specify required policies
- Non authenticated users will be “challenged” and redirected to Cloud Foundry UAA to login

```
public class FortunesController : Controller
{
  [Authorize(Policy = "read.fortunes")]
  public async Task<IActionResult> RandomFortune()
  {
    var fortune = await _fortunes.RandomFortuneAsync();
    HttpContext.Session.SetString("MyFortune", fortune.Text);
    return View(fortune);

  }

  [HttpGet]
  [Authorize]
  public IActionResult Login()
  {
    return RedirectToAction(nameof(FortunesController.Index), "Fortunes");
  }
}
```

# OAuth2 Provider – Implement Token Forwarding

- Add `HttpContextAccessor` to service container
  - Inject into any required services
- Access the users OAuth2 Bearer token from `HttpContext`
  - Add token to outgoing request headers

```
public class Startup
{
  public IConfiguration Configuration { get; }
  public void ConfigureServices(IServiceCollection services)
  {
    services.AddOptions();

    services.AddSingleton<IHttpContextAccessor, HttpContextAccessor>();
  }
}
```

```
public class FortuneServiceClient : IFortuneService
{
  IHttpContextAccessor _reqContext;
  public FortuneServiceClient(…, IHttpContextAccessor context)
  {
    _reqContext = context;
  }
  private async Task<HttpClient> GetClientAsync()
  {
    var client = new HttpClient(_handler, false);
    var token = await _reqContext.HttpContext.GetTokenAsync("access_token");
    client.DefaultRequestHeaders.Authorization =
              new AuthenticationHeaderValue("Bearer", token);
    return client;
 }
}
```

# OAuth2 Provider - Using on Cloud Foundry

- Create instance of OAuth2 service using CF CLI
  - `cf cups myOAuthService -p "{"client_id": "myTestApp","client_secret": "myTestApp","uri": "uaa://login.system.testcloud.com"}"`
- Bind instance to applications
  - `` `cf bind-service appName myOAuthService` ``
  - Also specify binding in `manifest.yml`
- Ensure you have added CloudFoundry Configuration provider or are using the Config Server provider
  - Service bindings added to configuration
- Steeltoe OAuth2 provider detects binding
  - Overrides any other settings with binding information

# OAuth2 Provider - Self-signed Certificates

- Communications between the Steeltoe OAuth2 provider and the UAA on Cloud Foundry uses TLS/SSL
  - Default behavior for provider is to validate server certificate
- Some Cloud Foundry installations will be using self-signed certificates
  - Validation will typically fail unless installation has been configured properly
- Can disable validation using setting
  - `security:oauth2:client:validate_certificates` – set to false to disable

```
security:
  oauth2:
    client:
      validate_certificates: false
```

# Steeltoe JWT Provider

- Enables using OAuth2 tokens issued by Cloud Foundry Security services for securing access to REST resources/endpoints
- Usage
  - Add NuGet reference to project
  - Add CloudFoundry config provider to Configuration builder (i.e. `AddCloudFoundry()`)
    - Not needed if already using Steeltoe Config Server client (i.e. `AddConfigServer()` )
  - Add Authentication and Authorization services to container
  - Add Authentication middleware to pipeline
  - Apply Authorization attributes
  - Implement access token forwarding as needed
  - Create & bind OAuth service instance for usage on Cloud Foundry

# JWT Provider – Add NuGet References

- JWT Provider
  - ASP.NET Core - `Steeltoe.Security.Authentication.CloudFoundryCore`
  - ASP.NET 4 WCF - `Steeltoe.Security.Authentication.CloudFoundryWcf`
  - Namespace - `#using Steeltoe.Security.Authentication.CloudFoundry;`

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />
    <PackageReference Include="Steeltoe.Security.Authentication.CloudFoundryCore" Version= "x.y.z" />
<ItemGroup>

</Project>
```

# JWT Provider – Add Authentication and Authorization

- Add and configure ASP.NET Core Authentication services
  - Default Scheme = JWT Scheme
- Add CloudFoundry JWT Authentication scheme
  - `AddCloudFoundryJwtBearer(config)`
- Add ASP.NET Core Authorization services
  - Configure Authorization policies as needed
- Add ASP.NET Core Authentication middleware to request processing pipeline
  - `UseAuthentication()`

```
public class Startup
{
  public IConfiguration Configuration { get; }
  public void ConfigureServices(IServiceCollection services)
  {
    services.AddOptions();

    services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
                .AddCloudFoundryJwtBearer(Configuration);

    services.AddAuthorization(options => {
       options.AddPolicy("read.fortunes",
         policy => policy.RequireClaim("scope", "read.fortunes"));
    });
  }

  public void Configure(IApplicationBuilder app, IHostingEnvironment env)
  {
    .....
    app.UseMvc();

    app.UseAuthentication();
  }
}
```

# JWT Provider – Apply Authorization Attributes

- Add Authorize attributes to controller actions to enable security

```
public class FortunesController : Controller
{
  [HttpGet("all")]
  [Authorize(Policy = "read.fortunes")]
  public async Task<List<Fortune>> AllFortunesAsync()
  {
   . . . .
  }

  [HttpGet("random")]
  [Authorize(Policy = "read.fortunes")]
  public async Task<Fortune> RandomFortuneAsync()
  {
   . . . .
  }
}
```

# JWT Provider - Using on Cloud Foundry

- Create instance of OAuth2 service using CF CLI
  - `cf cups myOAuthService -p "{"client_id": "myTestApp","client_secret": "myTestApp","uri": "uaa://login.system.testcloud.com"}"`
- Bind instance to applications
  - `` `cf bind-service appName myOAuthService` ``
  - Also specify binding in `manifest.yml`
- Ensure you have added CloudFoundry Configuration provider or are using the Config Server provider
  - Service bindings added to configuration
- Steeltoe JWT provider detects binding
  - Overrides any other settings with binding information

# JWT Provider - Self-signed Certificates

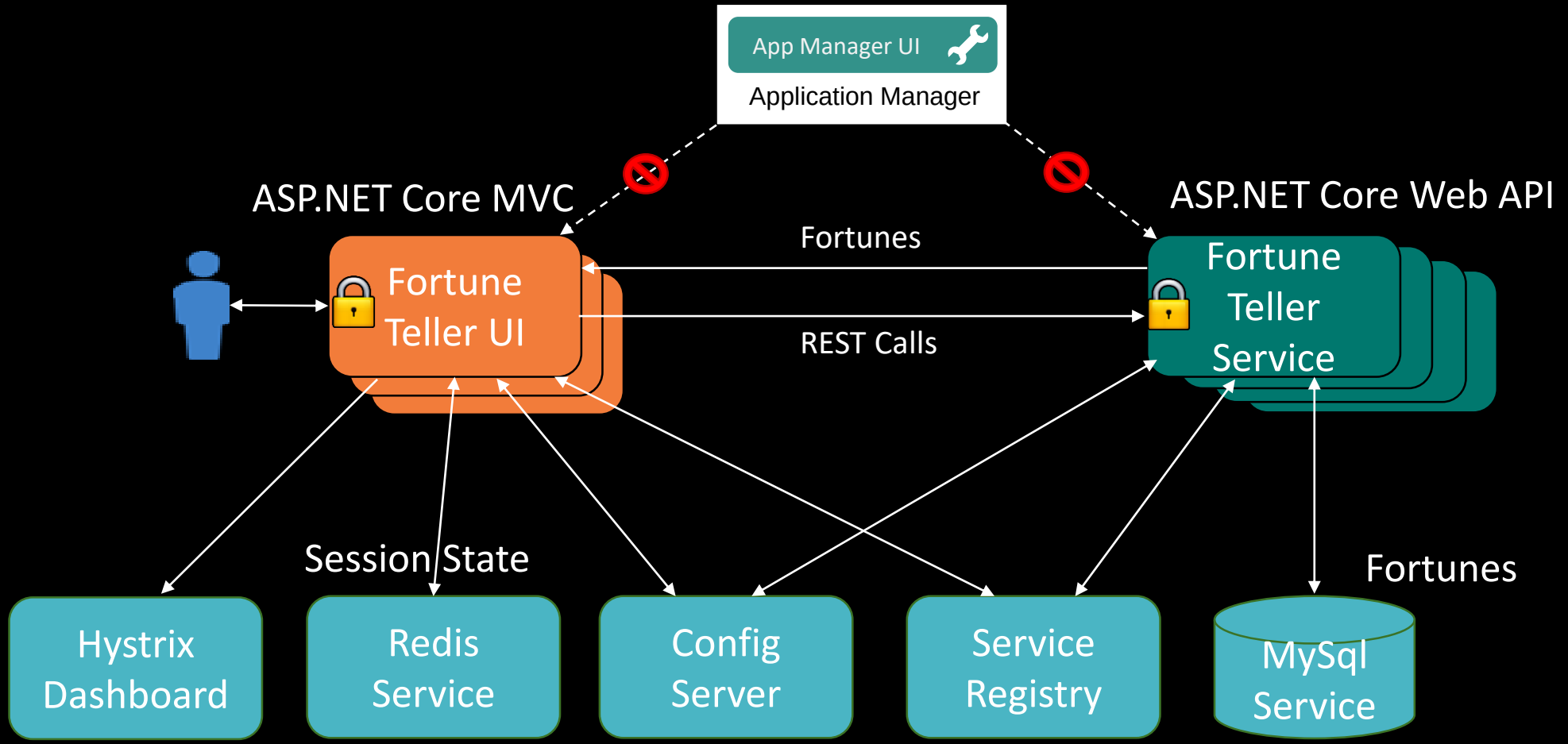
- Communications between the Steeltoe JWT provider and the UAA on Cloud Foundry uses TLS/SSL
  - Default behavior for provider is to validate server certificate
- Some Cloud Foundry installations will be using self-signed certificates
  - Validation will typically fail unless installation has been configured properly
- Can disable validation using setting
  - `security:oauth2:client:validate_certificates` – set to false to disable

```
security:
  oauth2:
    client:
      validate_certificates: false
```

# Lab10 – Securing Service Endpoints

- Change Fortune Teller Service to secure the REST endpoints by requiring valid OAuth Bearer tokens with `read.fortunes` permissions to access Fortunes
- Change Fortune Teller UI to require users to authenticate using the Cloud Foundry UAA before fetching a Fortune and to have `read.fortunes` permissions to access Fortunes
- Follow Lab Description
  - Description: https://github.com/SteeltoeOSS/Workshop/tree/master/Lab11
  - Code: /Workshop/Start/FortuneTeller.sln

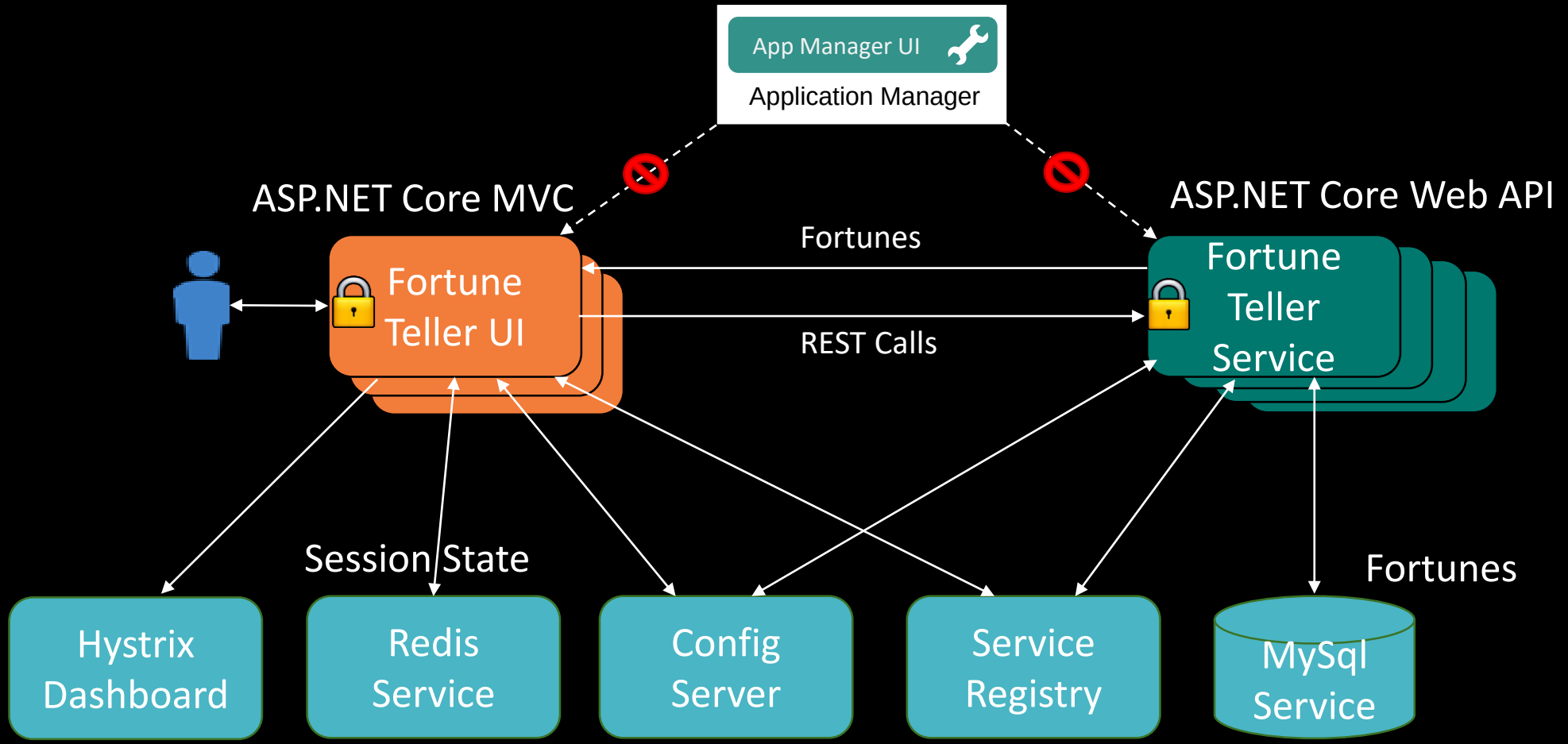
# Fortune Teller App – After Lab10



# Steeltoe Management

## ASP.NET CORE, MANAGEMENT, ENDPOINTS, TRACE, LOGS,

# Fortune Teller App – After Lab10



# Steeltoe Management Overview

- A set of frameworks you can use to add monitoring and management features to your application
  - Typically expose management features as HTTP REST endpoints using management actuators
    - Seamlessly integrates with Pivotal Apps Manager
  - Can expose endpoints using any technology
- Several Endpoints
  - Health
  - App Info
  - Loggers
  - Trace
  - Cloud Foundry
- Application type support
  - ASP.NET Core (.NET Framework and .NET Core)

# Steeltoe Health Endpoint

- Used to check the health of an application
  - Health is collected from all `IHealthContributor`
  - Final application health is computed by an `IHealthAggregator`
- Usage
  - Add NuGet reference to project
  - Create any needed custom `IHealthContributor`
  - Add all `IHealthContributor` to service container
  - Add Health actuator to service container and pipeline

# Health Endpoint – Add NuGet References

- Health Endpoint
  - Basic functionality - `Steeltoe.Management.Endpoint`
  - ASP.NET Core Actuator - `Steeltoe.Management.EndpointCore`
  - All ASP.NET Core Actuators - `Steeltoe.Management.CloudFoundryCore`

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />
    <PackageReference Include="Steeltoe.Management.CloudFoundryCore" Version= "x.y.z" />
<ItemGroup>

</Project>
```

# Health Endpoint – Create Custom HealthContributor

- Define class which derives from `IHealthContributor`
  - Give it an ID
- Create `Health()` method to calculate returned `Health`
  - Result put in `Health.Status`
  - Provide more health information in `Health.Details`
    - `status`
    - `error`

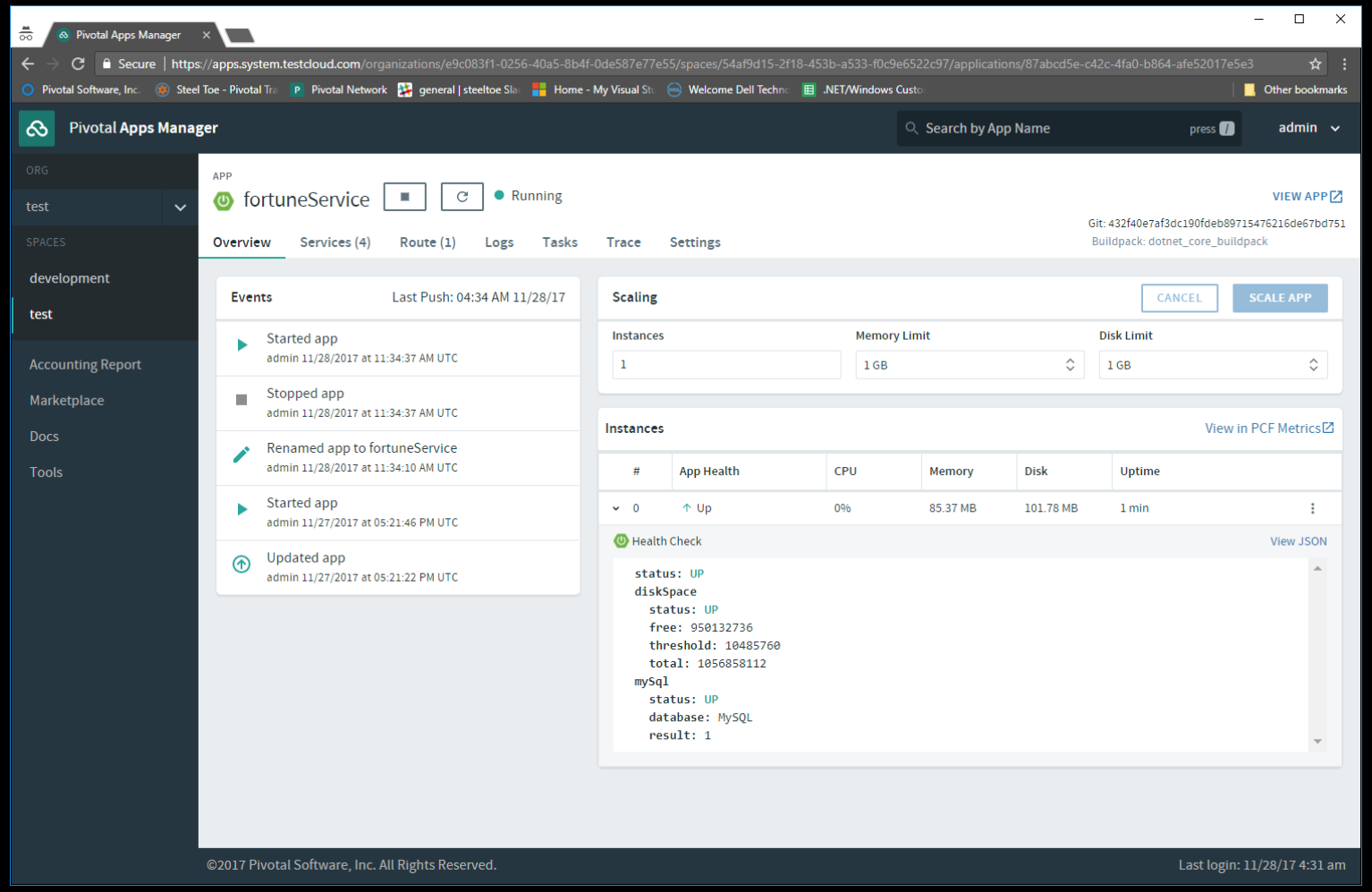
```
public class MySqlHealthContributor : IHealthContributor
{
  FortuneContext _context;
  public MySqlHealthContributor(FortuneContext dbContext)
  {
    _context = dbContext;
  }
  public string Id { get; } = "mySql";
  public Health Health()
  {
    Health result = new Health();
    result.Details.Add("database", "MySQL");
    MySqlConnection _connection = null;
    try {
       _connection = _context.Database.GetDbConnection() as MySqlConnection;
       if (_connection != null) {
         _connection.Open();
         MySqlCommand cmd = new MySqlCommand("SELECT 1;", _connection);
         var qresult = cmd.ExecuteScalar();
         result.Details.Add("result", qresult);
         result.Details.Add("status", HealthStatus.UP.ToString());
         result.Status = HealthStatus.UP;
       }
    } catch (Exception e) {
       result.Details.Add("error", e.GetType().Name + ": " + e.Message);
       result.Details.Add("status", HealthStatus.DOWN.ToString());
       result.Status = HealthStatus.DOWN;
    } finally { if (_connection != null) _connection.Close(); }
    return result;
  }
}
```

# Health Endpoint – Add Health Actuator

- Add any custom `IHealthContributor` to container
- Add Health endpoint actuator or all actuators
  - `AddHealthActuator`
  - `AddCloudFoundryActuators`
- Add Health endpoint middleware to request pipeline
  - `UseHealthActuator`
  - `UseCloudFoundryActuators`

```
public class Startup
{
  public IConfiguration Configuration { get; }

  public void ConfigureServices(IServiceCollection services)
  {
    services.AddOptions();

    services.AddScoped<IHealthContributor, MySqlHealthContributor>();

    // services.AddHealthActuator(Configuration);
    services.AddCloudFoundryActuators(Configuration);
}

  public void Configure(IApplicationBuilder app, IHostingEnvironment env)
  {
    .....

    // app.UseHealthActuator();
    app.UseCloudFoundryActuators();

    app.UseMvc();

}

}
```

# Health Endpoint - Using on Cloud Foundry



# Steeltoe App Info Endpoint

- Used to gather general application information
  - Info is collected from all `IInfoContributor`
    - `GitInfoContributor` – returns `git.properties` file
    - `AppSettingsInfoContributor` – returns appsettings info
- Usage
  - Add NuGet references to project
  - Use `GitInfo` in project to create `git.properties`
  - Add Info actuator to service container and pipeline

# App Info Endpoint – Add NuGet References

- App Info Endpoint
  - Basic functionality - `Steeltoe.Management.Endpoint`
  - ASP.NET Core Actuator - `Steeltoe.Management.EndpointCore`
  - All ASP.NET Core Actuators - `Steeltoe.Management.CloudFoundryCore`
- Use GitInfo to provide Git information during MSBuild

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />
    <PackageReference Include="Steeltoe.Management.CloudFoundryCore" Version= "x.y.z" />
    <PackageReference Include="GitInfo" Version="2.0.1" />
<ItemGroup>

</Project>
```

# App Info Endpoint – Using GitInfo

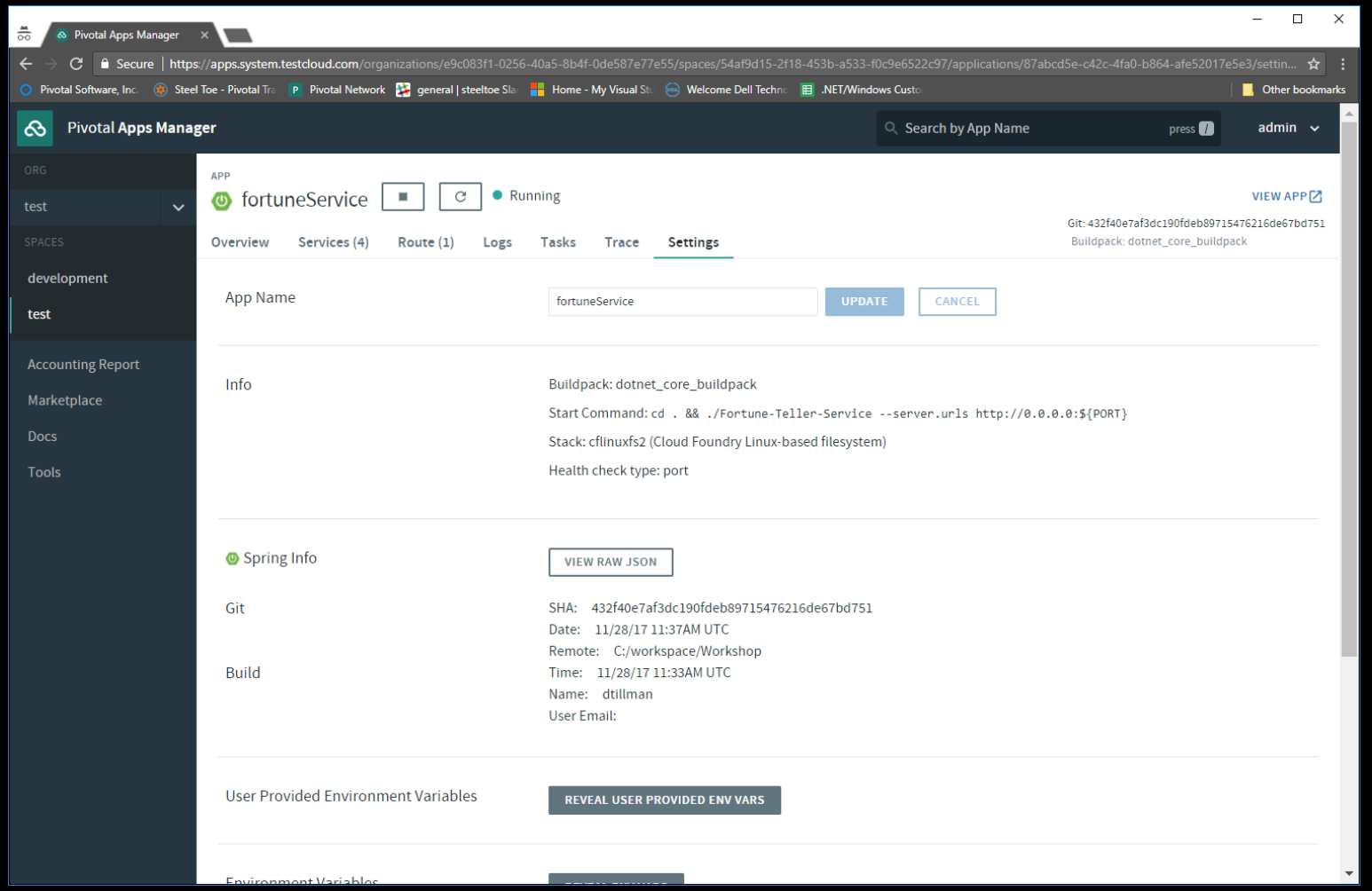
```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />
    <PackageReference Include="Steeltoe.Management.CloudFoundryCore" Version= "x.y.z" />
    <PackageReference Include="GitInfo" Version= "x.y.z" />
    <None Include="git.properties">
      <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
    </None>
<ItemGroup>
 <Target Name="_GitProperties" AfterTargets="CoreCompile">
    <WriteLinesToFile File="git.properties" Lines="git.remote.origin.url=$(GitRoot)" Overwrite="true" />
    <WriteLinesToFile File="git.properties" Lines="git.build.version=$(GitBaseVersion)" Overwrite="false" />
    <WriteLinesToFile File="git.properties" Lines="git.commit.id.abbrev=$(GitCommit)" Overwrite="false" />
    <WriteLinesToFile File="git.properties" Lines="git.commit.id=$(GitSha)" Overwrite="false" />
    <WriteLinesToFile File="git.properties" Lines="git.tags=$(GitTag)" Overwrite="false" />
    <WriteLinesToFile File="git.properties" Lines="git.branch=$(GitBranch)" Overwrite="false" />
    <WriteLinesToFile File="git.properties" Lines="git.build.time=$([System.DateTime]::Now.ToString('O'))" Overwrite="false" />
    <WriteLinesToFile File="git.properties" Lines="git.build.user.name=$([System.Environment]::GetEnvironmentVariable('USERNAME'))" Overwrite="false" />
    <WriteLinesToFile File="git.properties" Lines="git.build.host=$([System.Environment]::GetEnvironmentVariable('COMPUTERNAME'))" Overwrite="false" />
  </Target>
</Project>
```

# App Info Endpoint – Add Info Actuator

- Add any custom `IInfoContributor` to container
- Add Info endpoint actuator or all actuators
  - `AddInfoActuator`
  - `AddCloudFoundryActuators`
- Add Info endpoint middleware to request pipeline
  - `UseInfoActuator`
  - `UseCloudFoundryActuators`

```
public class Startup
{
  public IConfiguration Configuration { get; }

  public void ConfigureServices(IServiceCollection services)
  {
    services.AddOptions();

    // services.AddInfoActuator(Configuration);
    services.AddCloudFoundryActuators(Configuration);
}

  public void Configure(IApplicationBuilder app, IHostingEnvironment env)
  {
    .....

    // app.UseInfoActuator();
    app.UseCloudFoundryActuators();

    app.UseMvc();

}

}
```

# App Info Endpoint - Using on Cloud Foundry



# Steeltoe Loggers Endpoint

- Used to view and configure the active loggers and their logging levels at runtime
  - Must use the Steeltoe Logging provider
- Usage
  - Add NuGet references to project
  - Replace Console logging provider with Steeltoe provider
  - Add Logger actuator to service container and pipeline

# Loggers Endpoint – Add NuGet References

- Loggers Endpoint
  - Basic functionality - `Steeltoe.Management.Endpoint`
  - ASP.NET Core Actuator - `Steeltoe.Management.EndpointCore`
  - All ASP.NET Core Actuators - `Steeltoe.Management.CloudFoundryCore`

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />
    <PackageReference Include="Steeltoe.Management.CloudFoundryCore" Version= "x.y.z" />
<ItemGroup>

</Project>
```

# Loggers Endpoint – Use Steeltoe Logging Provider

- Add while building `WebHost`
  - In `ConfigureLogging()`
  - Use `AddDynamicConsole()` on provided `LoggingBuilder`
- `DynamicConsole` is a simple wrapper around standard Console provider
  - Configure using Logging:Console

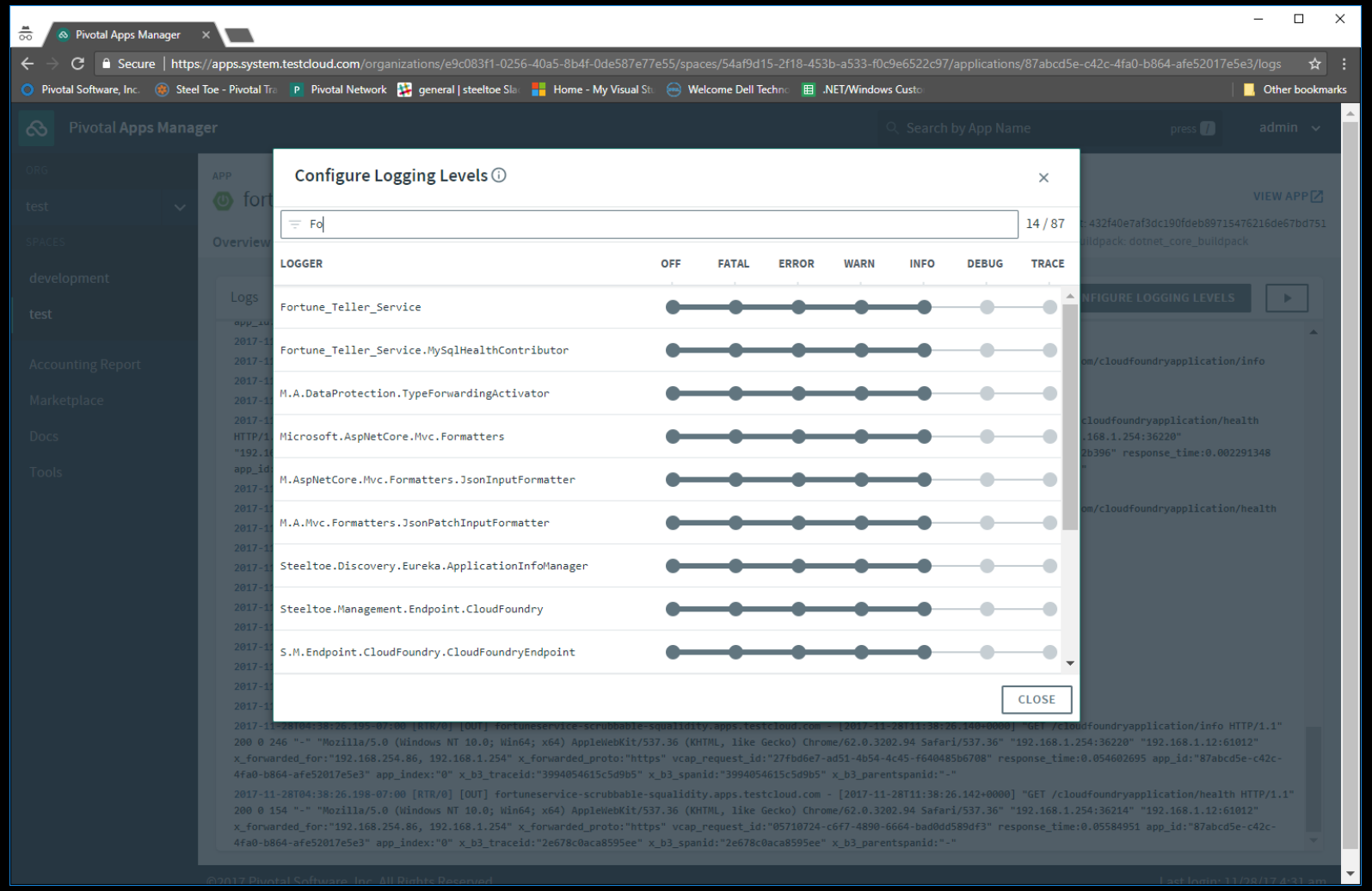
```
public static IWebHost BuildWebHost(string[] args)
{
  var builder = new WebHostBuilder()
    .UseKestrel()
    .UseIISIntegration()
    .ConfigureLogging((hostingContext, logging) =>
    {
      logging.AddConfiguration(
           hostingContext.Configuration.GetSection("Logging"));
      logging.AddDebug();

      logging.AddDynamicConsole(hostingContext.Configuration);

    })
 . . . .
}

public class Startup
{
  public IConfiguration Configuration { get; }
  public void ConfigureServices(IServiceCollection services)
  {
     services.AddOptions();
     services.ConfigureCloudFoundryOptions(Configuration)
  }
}
```

# Loggers Endpoint – Add Logger Actuator

- Add Logger endpoint actuator or all actuators
  - `AddLoggerActuator`
  - `AddCloudFoundryActuators`
- Add Logger endpoint middleware to request pipeline
  - `UseLoggerActuator`
  - `UseCloudFoundryActuators`

```
public class Startup
{
  public IConfiguration Configuration { get; }

  public void ConfigureServices(IServiceCollection services)
  {
    services.AddOptions();

    // services.AddLoggerActuator(Configuration);
    services.AddCloudFoundryActuators(Configuration);
}

  public void Configure(IApplicationBuilder app, IHostingEnvironment env)
  {
    .....

    // app.UseLoggerActuator();
    app.UseCloudFoundryActuators();

    app.UseMvc();

}

}
```

# Loggers Endpoint - Using on Cloud Foundry



# Steeltoe Trace Endpoint

- Used to view the last several requests and responses handled by an application
- Usage
  - Add NuGet references to project
  - Add Trace actuator to service container and pipeline

# Trace Endpoint – Add NuGet References

- Trace Endpoint
  - Basic functionality - `Steeltoe.Management.Endpoint`
  - ASP.NET Core Actuator - `Steeltoe.Management.EndpointCore`
  - All ASP.NET Core Actuators - `Steeltoe.Management.CloudFoundryCore`

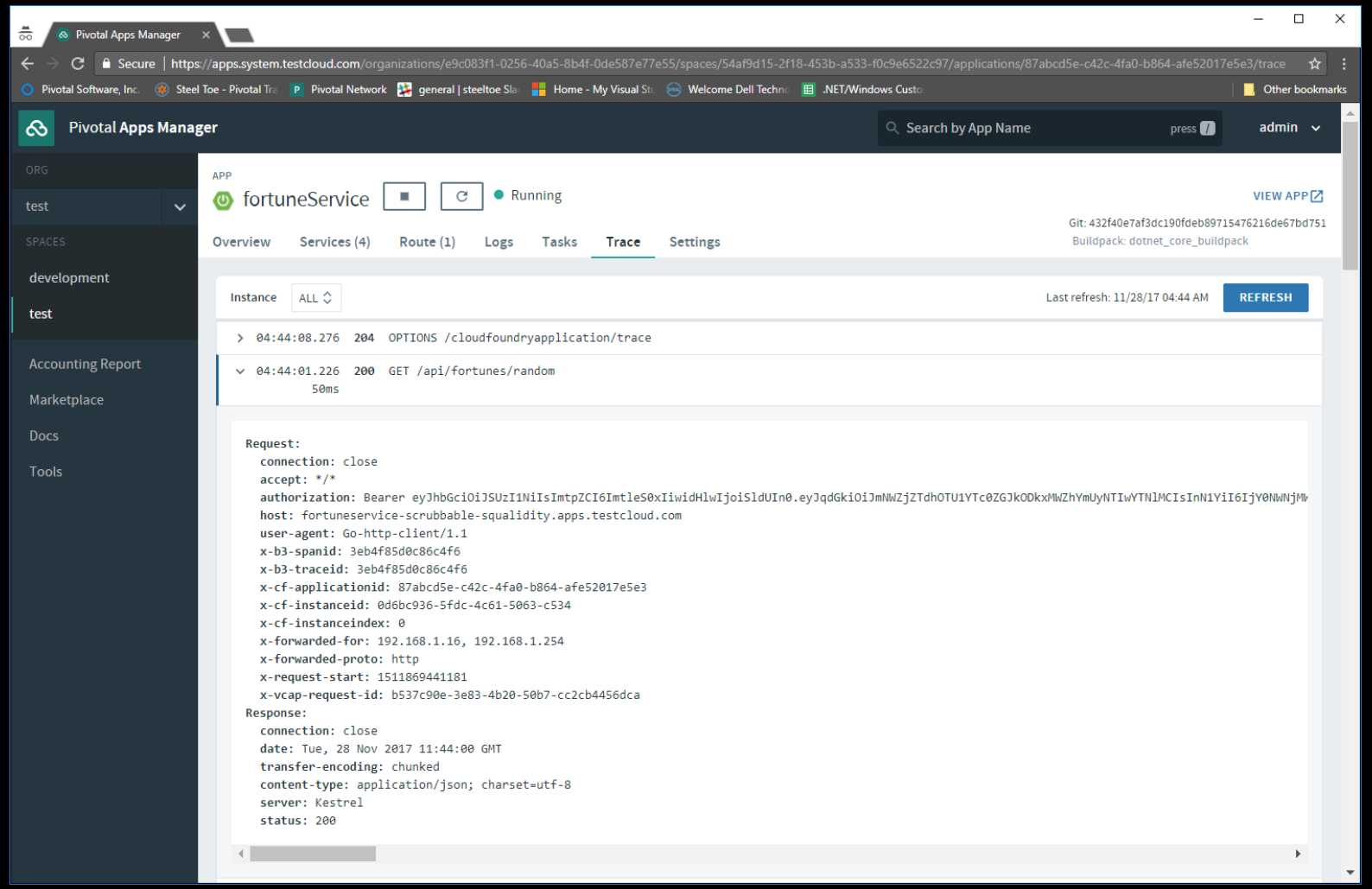
```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />
    <PackageReference Include="Steeltoe.Management.CloudFoundryCore" Version= "x.y.z" />
<ItemGroup>

</Project>
```

# Trace Endpoint – Add Trace Actuator

- Add Trace endpoint actuator or all actuators
  - `AddTraceActuator`
  - `AddCloudFoundryActuators`
- Add Trace endpoint middleware to request pipeline
  - `UseTraceActuator`
  - `UseCloudFoundryActuators`

```
public class Startup
{
  public IConfiguration Configuration { get; }

  public void ConfigureServices(IServiceCollection services)
  {
    services.AddOptions();

    // services.AddTraceActuator(Configuration);
    services.AddCloudFoundryActuators(Configuration);
}

  public void Configure(IApplicationBuilder app, IHostingEnvironment env)
  {
    .....

    // app.UseTraceActuator();
    app.UseCloudFoundryActuators();

    app.UseMvc();

}

}
```

# Trace Endpoint - Using on Cloud Foundry



# Steeltoe Cloud Foundry Endpoint

- Used to enable integration of the management endpoints with the Pivotal Apps Manager
  - Exposes query endpoint which returns the Ids and links to all enabled management endpoints
  - Adds security middleware to the request pipeline which requires using security tokens acquired from the UAA to access endpoints
- Usage
  - Add NuGet references to project
  - Configure settings
  - Add CloudFoundry actuator to service container and pipeline

# Cloud Foundry Endpoint – Add NuGet References

- Trace Endpoint
  - Basic functionality - `Steeltoe.Management.Endpoint`
  - ASP.NET Core Actuator - `Steeltoe.Management.EndpointCore`
  - All ASP.NET Core Actuators - `Steeltoe.Management.CloudFoundryCore`

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />
    <PackageReference Include="Steeltoe.Management.CloudFoundryCore" Version= "x.y.z" />
<ItemGroup>

</Project>
```

# Cloud Foundry Endpoint – Endpoint Settings

- For integration with Pivotal Apps Manager, you must configure the management query endpoint to `/cloudfoundryapplication`
  - Use `management:endpoints:path`
- You may need to disable certificate validation depending on your installation of Cloud Foundry
  - `management:endpoints:cloudfoundry:validateCertificates` – set to false to disable

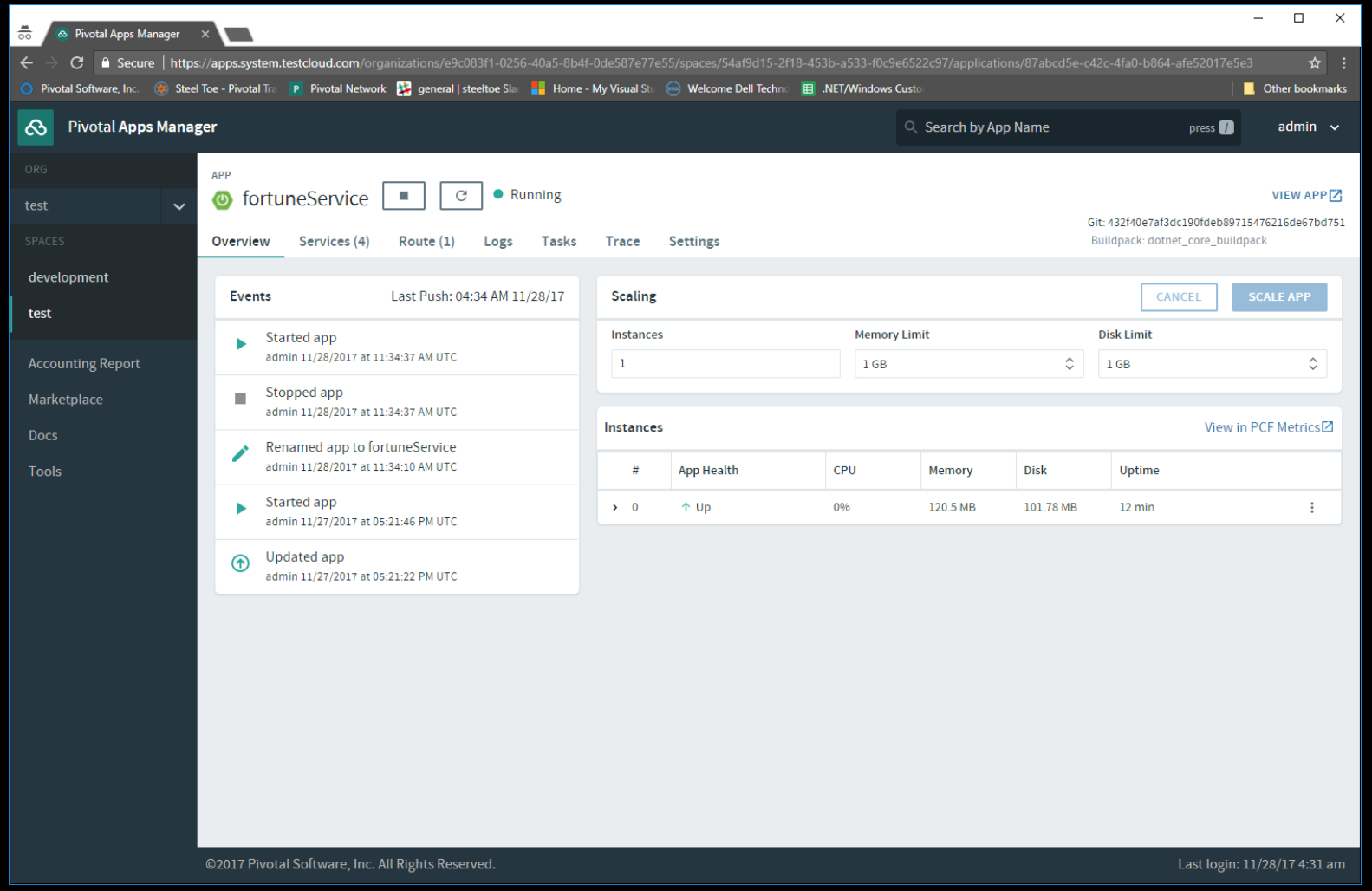
```
management:
  endpoints:
    path: /cloudfoundryapplication
    cloudfoundry:
      validateCertificates: false
```

# CloudFoundry Endpoint – Add Actuators

- Add CloudFoundry endpoint actuator or all actuators
  - `AddCloudFoundryActuator`
  - `AddCloudFoundryActuators`
- Add CloudFoundry endpoint middleware to request pipeline
  - `UseCloudFoundryActuator`
  - `UseCloudFoundryActuators`

```
public class Startup
{
  public IConfiguration Configuration { get; }

  public void ConfigureServices(IServiceCollection services)
  {
    services.AddOptions();

    // services.AddTraceActuator(Configuration);
    // services.AddLoggerActuator(Configuration);
    // services.AddCloudFoundryActuator(Configuration);

    services.AddCloudFoundryActuators(Configuration);
}

  public void Configure(IApplicationBuilder app, IHostingEnvironment env)
  {
    .....
    // app.UseCloudFoundryActuator();
    // app.UseTraceActuator();
    // app.UseLoggerActuator();

    app.UseCloudFoundryActuators();

    app.UseMvc();

}

}
```

# Cloud Foundry Endpoint - Using on Cloud Foundry



# Lab11 – Production Monitoring and Management

- Add several management endpoints to both applications to give us visibility into its operation while running in production
  - Git build information
  - Adjustable logging levels
  - Application specific health checks
  - Trace of last 100 requests
- Follow Lab Description
  - Description: https://github.com/SteeltoeOSS/Workshop/tree/master/Lab11
  - Code: /Workshop/Start/FortuneTeller.sln

# Fortune Teller App – Done!