Pivotal
Cloud Foundry®

.NET Cloud-Native Bootcamp

Hands-on exercises

Exercise #1 - Set up

In this exercise, we'll set up our workstation and cloud environment so that we're ready to build and run modern .NET applications.

Create Pivotal Web Services account

Here we set up an account on the hosted version of Pivotal Cloud Foundry, called Pivotal Web Services.

1. Go to http://run.pivotal.io/ and choose "sign up for free."
2. Click "create account" link on sign up page.
3. Fill in details.
4. Go to email account provided and click on verification email link.
5. Click on "claim free trial" link and provide phone number.
6. Validate your account and create your organization.

Install Cloud Foundry command line interface

You can interact with Cloud Foundry via Dashboard, REST API, or command line interface. Here, we install the CLI and ensure it's configured correctly.

1. Go to https://github.com/cloudfoundry/cli/releases and find the installer for your workstation. Note that for Mac users, you can download this script (http://bit.ly/2stikFz) that installs all the prerequisites via Homebrew!
2. Download the installer and run it.
3. Confirm that it installed successfully by going to a command line, and typing in **`cf -v`**

Install .NET Core and Visual Studio Code

.NET Core represents a modern way to build .NET apps, and here we make sure we have everything needed to build ASP.NET Core apps.

1. Visit https://www.microsoft.com/net/core and choose the .NET Core download for your machine.
2. Install .NET Core.
3. Confirm that it installed correctly by opening a command line and typing **`dotnet --version`**
4. Go to https://code.visualstudio.com/ and download the Visual Studio Code editor to your workstation. Run the installer.
5. Open Visual Studio Code and go to **View** → **Extensions**
6. Search for "C#" and choose the top "C# for Visual Studio Code" option and click "Install." This gives you type-ahead support for C#.

Create an ASP.NET Core project

Visual Studio Code makes it easy to build new ASP.NET Core projects. We'll create a sample project just to prove we can!

1. Within Visual Studio Code, go to **View** → **Integrated Terminal**. The Terminal gives you a shell interface without leaving Visual Studio Code.
2. Navigate to a location where you'll store your project files (e.g. C:\temp).
3. In the Terminal window, type in **`dotnet new mvc`** to create a new ASP.NET Core MVC project.
4. In Visual Studio Code, click **File** → **Open** and navigate to the directory containing the new ASP.NET Core project.
5. Observe the files that were automatically generated. Re-open the Terminal window.
6. In the Terminal window, type **`dotnet restore`** to load all the dependent packages.
7. Start the project by typing **`dotnet run`** and visiting http://localhost:5000. To stop the application, enter Ctrl+C.

Deploy ASP.NET Core application to Cloud Foundry

Let's push an app! Here we'll experiment with sending an application to Cloud Foundry.

1. In the Terminal, type in **`dotnet publish --configuration release`** to create the deployable artifacts. Those artifacts show up in a **Debug** → **release** folder in your project directory.
2. In Visual Studio Code, go to **View** → **Extensions**.
3. Search for "Cloudfoundry" and install "Cloudfoundry Manifest YML support" extension. This gives type-ahead support for Cloud Foundry manifest files.
4. In Visual Studio Code, create a new file called manifest.yml at base of your project.
5. Open the `manifest.yml` file, and type in the following (notice the typing

assistance from the extension):

```
---
applications:
- name: core-cf-[enter your name]
instances: 1
memory: 256M
path: ./bin/release/netcoreapp1.1/publish
```

6. In the Terminal, type in **`cf login -a api.run.pivotal.io`** and provide your credentials. Now you are connected to Pivotal Web Services.
7. Enter **`cf push`** into the Terminal, and watch your application get bundled up and deploy to Cloud Foundry.
8. In Pivotal Cloud Foundry Apps Manager (https://console.run.pivotal.io), see your app show up, and visit the app's URL.

Instantiate Spring Cloud Services instances

Spring Cloud Services wrap up key Spring Cloud projects with managed capabilities. Here we create a pair of these managed services.

1. In Pivotal Cloud Foundry Apps Manager, click on your "space" on the left, and switch to the "Services" tab. Note that all of these activities can also be done via the CF CLI.
2. Click "Add Service."
3. Type "Spring" into the search box to narrow down the choices.
4. Select "Service Registry" and select the default plan.
5. Provide an instance name and do not choose to bind the service to any existing applications. Click "Add." This service will take a couple of minutes to become available.
6. Repeat step 3 above and choose "Config Server" from the marketplace.
7. Choose the default plan, provide an instance name, and click "Add." Wait a couple minutes before expecting to see this service fully operational.
8. Return to your default space in Apps Manager, click "Services", choose Service Registry, and click the "manage" link. This takes you to the Eureka dashboard.
9. Return again to the default space, click "Service", choose Config Server, and click the "manage" link. Nothing here just yet!

Exercise #2 - Working with a Configuration Store

In this exercise, we get hands-on with a Git-backed config store and build an app that consumes it.

Point Config Server to Git repository

1. From the Terminal within Visual Studio Code, enter in the following command. This tells the Config Server where to get its configurations from: **`cf update-service <your config server name> -c '{"git": { "uri":"https://github.com/rseroter/cloud-native-net-configs"}}'`**
2. From Apps Manager, navigate to the “Services” view, manage the Config Server instance, and notice the git URL in the configuration.

Create ASP.NET Core Web API that points to Config Server

Here we create a brand new microservice and set it up with the Steeltoe libraries that pull configuration values from our Spring Cloud Services Config Server.

1. In the Visual Studio Code Terminal window, navigate up one level and create a new folder (bootcamp-webaapi) to hold our data microservice.
2. From within that folder, run the **`dotnet new webapi`** command in the Terminal. This uses the “web api” template to create scaffolding for a web service.
3. From the Terminal enter **`dotnet add package Microsoft.Extensions.Options`** then **`dotnet add package Microsoft.Extensions.Configuration`** and finally **`dotnet add package Pivotal.Extensions.Configuration.ConfigServer`**
4. Enter **`dotnet restore`** into the Terminal. This actually retrieves the packages and puts them into the project folder.
5. Open the newly created folder in Visual Studio Code.
6. In the Controllers folder, create a new file named `ProductsController.cs.`
7. Enter the following code. It defines a new API controller, specifies the route that handles, it, and an operation we can invoke.

```
using Microsoft.AspNetCore.Mvc;

using Microsoft.Extensions.Options;

using System.Collections.Generic;

using Pivotal.Extensions.Configuration.ConfigServer;

using Microsoft.Extensions.Configuration;
```

```
namespace bootcamp_demo2_practice.Controllers
{
   [Route("api/[controller]")]
   public class ProductsController: Controller
   {
       public ProductsController(IConfigurationRoot config) {
           Config = config;
       }
       private IConfigurationRoot Config {get; set;}

       // GET api/products
       [HttpGet]
       public IEnumerable<string> Get()
       {
           System.Console.WriteLine("connection string is " +
              Config["productdbconnstring"]);
           System.Console.WriteLine("Log level from config is " +
              Config["loglevel"]);
           return new string[] { "product1", "product2" };
       }
   }
}
```

8. Next, we add what's needed to make our ASP.NET Core application retrieve the configuration data from Cloud Foundry and the Config Server. Enter **`dotnet add package Microsoft.Extensions.Configuration.CommandLine`**
9. Then enter **`dotnet restore`** to actually retrieve the package.
10. Go to `appsettings.json` file, and edit to include the application name and cloud config name. This maps to the configuration file read from the server.

```
{
 "Logging": {
   "IncludeScopes": false,
   "LogLevel": {
     "Default": "Warning"
   }
 },
 "spring": {
   "application": {
     "name": "branch1"
   },
   // determines the name of the files pulled; explicitly set to avoid env
variable overwriting it
   "cloud": {
     "config": {
       "name": "branch1"
     }
   }
 }
}
```

11. Open `Startup.cs` class and add the pieces that add the configuration store as a provider. First, add a "using" reference to `Pivotal.Extensions.Configuration` so that your complete block of using statements looks like:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using Pivotal.Extensions.Configuration;
```

12. Next set up the "builder" variable to pull in all the parameters from (1) our environment variables, (2) the `appsettings.json` file, and (3) our config store. This happens in the "Startup" operation.

```
public Startup(IHostingEnvironment env)
{
  //order matters here;
  //if env variables last, then they overwrite anything in JSON files
  var builder = new ConfigurationBuilder()
       .SetBasePath(env.ContentRootPath)
       .AddEnvironmentVariables()
       .AddJsonFile("appsettings.json", optional: false, reloadOnChange: true)
       .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true)
       .AddConfigServer(env);
  Configuration = builder.Build();
}
```

13. Still in the `Startup.cs` class, add the following code to the `ConfigureServices` operation:

```
public void ConfigureServices(IServiceCollection services)
{
     // Add framework services.
     services.AddMvc();

     services.AddConfigServer(Configuration);
 }
```

14. In the `Program.cs` class file, add a “using” reference to `Microsoft.Extensions.Configuration`.

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;

using Microsoft.Extensions.Configuration;
```

15. Next, create a `ConfigurationBuilder` object, and then add it to the host with the following code:

```
public static void Main(string[] args)
{
   var config = new ConfigurationBuilder()
      .AddCommandLine(args)
      .Build();

   var host = new WebHostBuilder()
      .UseKestrel()
      .UseConfiguration(config)
      .UseContentRoot(Directory.GetCurrentDirectory())
      .UseIISIntegration()
      .UseStartup<Startup>()
      .Build();
   host.Run();
 }
```

16. Add a `manifest.yml` file to the base of the project. This tells Cloud Foundry how to deploy your app. Enter:

```
---
applications:
- name: core-cf-microservice-<enter your name>
instances: 1
memory: 256M
path: ./bin/release/netcoreapp1.1/publish
# determines which environment to pull configs from
env:
   ASPNETCORE_ENVIRONMENT: dev
services:
- <your config server instance name>
```

Deploy ASP.NET Core application to Cloud Foundry

Get this application into Cloud Foundry!

1. Run a **`dotnet publish --configuration release`** command to generate the artifacts.
2. Execute a **`cf push`**.

Observe behavior when changing application name or label

See how configurations are pulled and used.

1. Hit the products microservice by going to **https://[your PWS URL]/api/products** and seeing a result.
2. Go to the "Logs" view in Apps Manager and see the connection string and log level written out.
3. Go back to the source code and change the application name and cloud config name in the `appsettings.json` to "branch3", and in the `manifest.yml` change the environment to "qa." This should resolve to a different configuration file in the GitHub repo, and load different values into the app's configuration properties.
4. Re-publish the app (**`dotnet publish --configuration release`**), redeploy the app (**`cf push`**), hit the API endpoint, and observe the different values logged out.

Exercise #3 - Using a Service Registry

This exercise helps us understand how to register our microservices with the Spring Cloud Services Registry, and also discover those services at runtime.

Update previous service to register itself with the Service Registry

Modify our products microservice to automatically register itself upon startup.

1. In the existing microservice project, add a Nuget package dependency by entering **`dotnet add package Pivotal.Discovery.Client`** into the Terminal. Then execute a **`dotnet restore`** command. If you weren't using Spring Cloud Services, you could use a vanilla Steeltoe package for service discovery.
2. Open the `appsettings.json` file and add this block *after* the "spring" block:

```
{
 "Logging": {
   "IncludeScopes": false,
   "LogLevel": {
     "Default": "Warning"
   }
 },
 "spring": {
   "application": {
     "name": "branch1"
   },
   "cloud": {
     "config": {
       "name": "branch1"
     }
   }
 },
 "eureka": {
   "client": {
     "shouldRegisterWithEureka": true,
     "shouldFetchRegistry": true
   }
 }
}
```

3. In `Startup.cs`, add a new using statement for `Pivotal.Discovery.Client`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using Pivotal.Extensions.Configuration;
using Pivotal.Discovery.Client;
```

4. In that class, update the `ConfigureServices` method to include: `services.AddDiscoveryClient(Configuration).`

```
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddMvc();
    services.AddConfigServer(Configuration);
    services.AddDiscoveryClient(Configuration);
 }
```

5. In the same class, update the `Configure` method to include: `app.UseDiscoveryClient().`

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env,
ILoggerFactory loggerFactory)
{
     loggerFactory.AddConsole(Configuration.GetSection("Logging"));
     loggerFactory.AddDebug();

     app.UseMvc();
     app.UseDiscoveryClient();
 }
```

6. Update manifest so that the microservice also binds to the Service Registry upon push.

```
---
applications:
- name: core-cf-microservice-<enter your application name>
instances: 1
memory: 256M
path: ./bin/release/netcoreapp1.1/publish
# determines which environment to pull configs from
env:
   ASPNETCORE_ENVIRONMENT: qa
services:
- <your configuration service name>
- <your registry name>
```

7. Publish the application, and then push the application. Go “manage” the Service Registry instance from within Apps Manager. Notice our service is now listed!

Download and open up front-end ASP.NET Core application

Configure a pre-built front end app so that it discovers our products microservice.

1. Go to https://github.com/rseroter/cloud-native-net-sampleui to download the pre-built front-end application.
2. Download the code into a folder on your machine.
3. Open the project in Visual Studio Code.
4. Observe in `bootcamp-core-ui.csproj` that the project references the discovery package. See in `appsettings.json` that this app does NOT register itself with Eureka, but just pulls the registry. Also see in the `Startup.cs` file that it loads the discovery service.

Update front-end application to pull services from the registry and use them

Replace placeholder values so that your app talks to your own microservice.

1. Open the `HomeController.cs` file in the Controllers folder.
2. Go to the `Index()` operation and replace the placeholder string with the microservice’s application name (**NOT THE URL!**) from your Service Registry.

```
[Route("/home")]
public IActionResult Index() {

   var client = GetClient();
   var result = client.GetStringAsync("https://<replace
me>/api/products").Result;
```

```
   ViewData["products"] = result;
    return View();
 }
```

3. Go to the `manifest.yml` file and replace the topmost placeholder values for “application name” and in the services section, set the registry service name.
4. Package the application by entering **`dotnet publish --configuration release`** into the Terminal.
5. Enter **`cf push`** into the Terminal and see your application get deployed to Cloud Foundry.
6. After the application successfully deploys, go to https://<your application name>.cfapps.io/Home and you should see the web page with products retrieved from your data microservices.

Exercise #4 - Flexing and Breaking Things

It's fun to try out the platform itself and see how it works in certain situations. Here we trigger autoscaling, and observe failure recovery.

Configure and test out autoscaling policies

Autoscaling is a key part of any solid platform. Here we create a policy, and trigger it!

1. Download the .NET Core project located at https://github.com/rseroter/cloud-native-net-loadtest.
2. Open the project in Visual Studio Code.
3. Update the `Program.cs` file with a pointer to your API microservice URL.
4. In Apps Manager, go to the Marketplace and provision the **Autoscaler** service.
5. Bind the autoscaler service to your microservice by viewing the autoscaler service, add choosing to add a bound app.
6. Click the "manage" link on the autoscaler page to set up a scaling policy.
7. Click "edit" on Instance Limits and set it to have a minimum of 1, and maximum of 3.
8. Edit the scaling rules and set an **HTTP Throughput** policy that scales down if less than 1 request per second, and scales up if more than 4 requests per second.
9. Enable the policy by sliding the toggle at the top of the policy definition. Save the policy.
10. Start the "load test" .NET Core project on your machine that repeatedly calls your microservice. Start it by entering **`dotnet run`** in the Terminal while pointing at that application folder.
11. On the overview page of your microservice in Apps Manager, observe a second instance come online shortly. This is in response to the elevated load, and your autoscaling policy kicking in. Also notice a new "event" added to the list.
12. Stop the load testing app (Ctrl+C), and watch the application scale back down to a single instance within 30 seconds.

Update microservice with new "fault" endpoint and logs

Let's add a new microservice endpoint that purposes crashes our app. We can see how the platform behaves when an instance disappears.

1. Return to your products API microservice in Visual Studio Code.
2. Create a new controller named `FailureController.cs` with the following content:

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Options;
using System.Collections.Generic;
using Pivotal.Extensions.Configuration.ConfigServer;
using Microsoft.Extensions.Configuration;

namespace bootcamp_demo2_practice.Controllers
{
   [Route("api/[controller]")]
   public class FailureController: Controller
   {
       public FailureController(IConfigurationRoot config) {
           Config = config;
       }

       private IConfigurationRoot Config {get; set;}
        // GET api/failure
       [HttpGet]
       public ActionResult TriggerFailure(string action)
       {
           System.Console.WriteLine("bombing out ...");
           //purposely crash
           System.Environment.Exit(100);
           return null;
       }
   }
}
```

3. Publish the app, and then deploy the updated service to Cloud Foundry.
4. Confirm that “regular” URL still works fine: https://<your app name>.cfapps.io/api/products
5. Now send a request to https://<your app name>.cfapps.io/api/failure
6. See in Applications Manager that the app crashes, and the platform quickly recovers and spins up a new instance.
7. Visit the “logs” view and see that logs were written out.