# DEMO 1 - POOR MAN FEATURE FLAG

Demo Instructions

**DEMO: POOR MAN’S FEATURE FLAGS - Using MVC Web App - PoorMansFeatureFlag**

Goal: Use feature flags to hide text on the About page when flag is turned off.

 Show how we have the feature flag “MyFeature” hard-coded in the HomeController.cs file. Run the app and it should display the message.

Open web.config and add a key to drive the feature:

**<add key="MyFeature" value="true"/>**

 Open the **Controllers/HomeController.cs** file. We are going to use the ViewBag to tell status of the feature from the web.config

**Using System.Configuration**

//Add this to the About() method and the Contact() Method

**ViewBag.MyFeature = bool.Parse(ConfigurationManager.AppSettings["MyFeature"]);**

 Open About View and Contact Views and add the following code

**@if (ViewBag.MyFeature)**

**{}**

 Run the app. You should be able to see the strings

Modify the web.config to turn the feature off

 Run the app. You should not be able to see the strings

**Now let’s look at why this isn’t a great way to do feature flags**

 Go to HomeController.cs and change the “About” AppSetting to **MyFeatures.** Run the app, You will get a null exception when you navigate to the page.

Fix the code and enable the feature in the web.config to set things back to working

**LET'S REMOVE THE FEATURE FLAG CODE AND SEE WHAT HAPPENS**

Delete the key from the web.config

Remove code from HomeController.cs

Run application. Go to the About page and it will fail.The problem is we didn’t clean up all the references to it.The issue with the poor man approach is nothing is strongly typed, and you don’t get compile time checks.

Goal: Show how the FeatureToggle Framework works by hiding the Link to the About page

# DEMO 2 – Feature Toggle Framework

 I’ve already installed the FeatureToggle Framework using NuGet.

Open Manage Nuget Packages in Visual Studio and show it installed.

Let’s start with the AlwaysOnFeatureToggle flag. Explain how this flag works, and how it is compiled into the code directly.

**CREATE THE MYALWAYSONFEATURETOGGLE CLASS**

Only way to turn this on or off is to re-release a new assembly

Create a new folder called **Flags**

Create a new class called **MyFeatureToggle**

Inherit from **FeatureToggle.AlwaysOnFeatureToggle**

Open **Views/Shared/\_Layout.cshtml**

Wrap the **About** link in the following code

@if (new {MYAPPNAMEHERE].Flags.MyFeatureToggle().FeatureEnabled)

{}

Run the app and show that About is there.

Change inheritence to AlwaysOffFeatureToggle, recompile, and rerun

**CREATE A SIMPLE FEATURE TOGGLE**

Enable/Disable based off a configuration setting

Change the Inheritance on the class to be SimpleFeatureToggle

Run the app and it will error out because we haven't updated the app.config. That strong typing coming in handy

Add the following key to web.config

**<add key ="FeatureToggle.MyFeatureToggle" value="true"/>**

Run and should see About

Change to false

Run and should not see about

**CREATE A RANDOM FEATURE TOGGLE**

Change the Inheritance on the class to be RandomFeatureToggle

Remove web.config values.Run multiple times to see changes

# DEMO 3 – Feature Flags Using Azure App Configuration

Let’s get to it. In this demo, I’m going to make use of the Azure App Configuration Framework to keep our app settings and enable feature flags in my ASP.NET .NET Core application.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

* This is a simple .NET Core MVC App that I created
* Open the TestFeatureFlagAppConfig.csproj file and show the two references I added for App Config and Feature Management
* Open portal.azure.com, show how you can create an app configuration.
* Open feature-flag-demo-appconfig. Show access keys. Show Feature Flag
* dotnet user-secrets set ConnectionStrings:AppConfig <your\_connection\_string>
  + Connection string to access my App Configuration store
* Program.cs
  + Specifying where to get config file from
* Startup.cs
  + Turned on FeatureManagement  
    Update the Configure method to add a middleware to allow the feature flag values to be refreshed at a recurring interval while the ASP.NET Core web app continues to receive requests
* MyFeatureFlag.cs
  + Enumaration that is a list of my feature flags. Makes for easy reference
* BetaController.cs
* \_viewImports.cshtml
* \_Layout.cshtml
* Index.cshtml

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

DEMO 3 – LaunchDarkly and Azure Pipelines

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Demo Instructions

(This demo is based off the <https://www.azuredevopslabs.com/labs/vstsextend/launchdarkly/> HOL)

<https://docs.launchdarkly.com/docs/visual-studio-team-services-extension>

Azure DevOps Team Project being used is devopsmickey/LaunchDarklyHOL

LaunchDarkly is a continuous delivery platform that provides feature flags as a service and can manage your feature flags at scale. You can make use of its integration with Azure DevOps to tie feature flag rollouts to work items, as well as manage feature flags as part of a release pipeline.

In this demo we are going to:

* Implement a LaunchDarkly Feature Flag in an MVC Application
* Integrate Launch Darkly with Azure DevOps
* Rollout LaunchDarkly Feature Flag as part of an Azure Pipelines release

STEP 1: Create a Feature Flag in LaunchDarkly

* Log into LaunchDarkly
* Create a new feature flag: “Member Portal #”
* Get the Production SDK Key

STEP 2: Add Feature Flag to Web Application

* Turn on Work Item Management in Team Project settings under repository
* Clone the repo in Visual Studio
* Check out the launch-darkly branch and open the PartsUnlimited.sln
* Install the LaunchDarkly NuGet Package
* Restore all packages and build the app (dotnet restore)
* Run the app and you should see the Member Portal in top right
* Replace code in HomeController with snippet from HOL. This adds the LDClient object and adds the if statement that uses the launch darkly feature flag. Do the same with AccountController
* Modify Views\Shared\\_Layout.chtml, line 55 with @if (ViewBag.togglevalue == true) { @await Html.PartialAsync("\_Login") }
* Add LaunchDarkly SDK key to HomeController and AccountController
* HomeController starts with initializing a static LaunchDarkly client. And the methods to view MemberPOrtal are modified to check the feature flag is on or off. The \_Layout.cshrml page checks the toggle value and renders the link if it is turned on.
* When you request a feature flag, you need to pass in a user object. So we are initializing the user object in the begtinning. This will be used to check whether a user with a specified key exists in launchdarkly or not. In real time it might be from a logged in user, or a database, for example.
* We then call BoolVariation to check the value. If it is true, we set a view data value to its value.
* Run the application. Turn the flag on manually and run again. Turn the flag off.
* In team explorer, navigate to changes and commit/push the code, associating it with the work item
* Open the work item and you can see the commit associated with it.

STEP 3: AUTOMATICALLY Rollout LaunchDarkly feature flags during release

* First, we need a launchdarkly access token to integrate with Azure DevOps.
* Azure DevOps | Project Settings | Pipelines | Service Connections | New LaunchDarkly Service Connection and create a new service connection
* Open work item and view Launch Darkly tab, select the feature flag
* Configure the release pipeline
* Execute the build and it should also release and enable the flag