# Context.cs Notes and Queries

Contents

[Context.cs Notes and Queries 1](#_Toc5605484)

[Overview 1](#_Toc5605485)

[Context Class vs Driver Class 1](#_Toc5605486)

[ContextFactory 3](#_Toc5605487)

[Constructor 3](#_Toc5605488)

[internal TestUser LoggedInUser 3](#_Toc5605489)

[internal IWebDriver Driver 4](#_Toc5605490)

[internal IWait<IWebDriver> DriverWait 4](#_Toc5605491)

[internal IWebDriver ChromeDriver 4](#_Toc5605492)

[public static IWebDriver GetDriverForKey 4](#_Toc5605493)

[FeatureContext.Current 5](#_Toc5605494)

[TestDataRepository 5](#_Toc5605495)

[public TestUser GetUser(string userKey) 5](#_Toc5605496)

[ReadFromFile<TestUser> 7](#_Toc5605497)

## Overview

Contains the following variables and the methods to set/get then:

RootUrl,

TestDataRepository,

MockTestDataRepository,

RunSettings

## Context Class vs Driver Class

It looks like the main difference between this Context class and what could reasonably just go in a Driver class is the TestDataRepository and MockDataRepository. Aside from these, it looks like most/all of it could go into a Driver class

The static ContextFactory class will create a static instance (called instance) of the non –static Context class:

public static class ContextFactory

{

internal static Context Instance { get; } = new Context();

}

internal class Context

{…}

This is equivalent to the static OpenBrowser() method which creates a static instance (called driver) of the non-static ChomeDriver class

public static IWebDriver driver;

public static void OpenBrowser(string selectedBrowser)

{

switch (selectedBrowser.ToLower())

{

case "chrome":

driver = new ChromeDriver();

…}

We could (I think) have RootUrl and ApiRootUrl as static class variables within Driver class that are instantiated in the OpenBrowser() method, just like we do with

public static WebDriverWait wait;

The code to instantiate them is the same as we have in RunSetttings.cs (which is a wrapper class to abstract this functionality)

public string WebRoot => (TestContext.Properties["WebRoot"] ?? "http://porapp01qa/").ToString();

public string ApiRoot => (TestContext.Properties["ApiRoot"] ?? "http://porapp01qa:81/api/").ToString();

(Does the lamba operator/expression really and anything here? Is it just a short hand way for writing this?

Is this example of Expression-Bodied Properties?

public string webRoot;

public void SetWebRoot()

webRoot = (TestContext.Properties["WebRoot"] ?? "http://porapp01qa/").ToString();

}

## ContextFactory

public static class ContextFactory

{

internal static Context Instance { get; } = new Context();

}

This is the separate factory class to expedite the creation of instances of Context

Equivalent to this?

internal static Context MyContextInstance()

{

Context instance = new Context();

return instance;

}

I think the version in the Solution is a Property so you can just call Context.

Rather than having to have another step (perhaps in an Initialisation step in WebHooks) that creates the instance (like I do for driver in my framework )

## Constructor

internal class Context

{

internal string RootUrl;

internal TestDataRepository TestDataRepository;

internal MockDataRepository MockTestDataRepository;

internal RunSettings

}

Just creates instances of each of the class variables.

Instances of the above objects and their variables (properties?) are created when calling the constructor.

## internal TestUser LoggedInUser

internal TestUser LoggedInUser

{

get { return FeatureContext.Current.GetUserForKey("LoggedInUser"); }

set { FeatureContext.Current["LoggedInUser"] = value; }

}

LoggedInUser is a variable (property? What is difference? Property some kind for wrapper around a variable?) of type TestUser.

It has getter and setter (functions?) for getting and setting its value.

Could same could be achieved by having a LoggedInUser variable and getLoggedInUser() and setLoggedInUser() methods for getting and setting value?

## internal IWebDriver Driver

As above, but for IWebDriver.

Normally I would create this as a static instance of (e.g.) ChromeDriver from a method in the static Driver class.

Here it seems to be a property within Context class (equivalent would be Driver class). Instead of being static, it is retrieved on demand via FeatureContext.Current.GetDriverForKey("Driver")

Is this a better approach that a static variable?

## internal IWait<IWebDriver> DriverWait

internal IWait<IWebDriver> DriverWait => new WebDriverWait(Driver, new TimeSpan(0, 0, 20));

Just creates a new WebDriverWait.

How is this different to

internal WebDriverWait aDriverWait = new WebDriverWait(Driver, new TimeSpan(0, 0, 20));

Except that you have to make the Driver static?

## internal IWebDriver ChromeDriver

internal IWebDriver ChromeDriver

{

get { return FeatureContext.Current.GetDriverForKey("ChromeDriver"); }

set { FeatureContext.Current["ChromeDriver"] = value; }

}

Same as LoggedInUser, but for ChromeDriver

## public static IWebDriver GetDriverForKey

public static IWebDriver GetDriverForKey(this FeatureContext featureContext, string key)

{

return FeatureContext.Current.ContainsKey(key) ? FeatureContext.Current[key] as IWebDriver : null;

}

Pass in the key value that corresponds to the type of driver you want (e.g. Chrome) and method will return that driver.

## FeatureContext.Current

A set of global variables/properties(?) that exist for the duration of the current context. There are Set and Get methods for each of them in Context class.

# TestDataRepository

TestDataRepository seems to be a custom class.

It implements a customer ITestDataRepository, but this does not look like it really adds anything and the interface could be removed?

The same seems to be true for MockDataRepository

## public TestUser GetUser(string userKey)

public TestUser GetUser(string userKey)

{

var users = \_xlsxService.ReadFromFile<TestUser>($"{\_dataFolder}Users.xlsx");

var selectedUser = users.FirstOrDefault(u => u.UserKey == userKey);

if (selectedUser == null)

{

throw new ArgumentNullException(@"User", new Exception($"User {userKey} not found!"));

}

return \_xlsxService.ReadFromFile<TestUser>($"{\_dataFolder}Users.xlsx").First(u => u.UserKey == userKey);

}

Takes the given user (userKey). Reads the users.xlsx file into a collection.

Returns the first entry on the file that matches userKey

NOTE: The following line appears to be unnecessary. Could just return userKey?

return \_xlsxService.ReadFromFile<TestUser>($"{\_dataFolder}Users.xlsx").First(u => u.UserKey == userKey);

}

### ReadFromFile<TestUser>

ReadFromFile will return some kind of IEnumberable (like a collection) and <TEntity> tells it what type of IEnumberable to return, in this case a TestUser type