# CTX Notes

Table of Contents

[CTX Notes 1](#_Toc8216175)

[Context Class vs Driver Class 1](#_Toc8216176)

[Context Class 2](#_Toc8216177)

[ContextFactory Class 3](#_Toc8216178)

[TestDataRepository Class 3](#_Toc8216179)

[GetUser 3](#_Toc8216180)

[The Lamda Expression 4](#_Toc8216181)

[TestDataRepository 4](#_Toc8216182)

## 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 do 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?

public string webRoot;

public void SetWebRoot()

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

}

## Context Class

It looks like the Context class contains a handful of its own class variables:

internal string RootUrl;

internal TestDataRepository TestDataRepository;

internal MockDataRepository MockTestDataRepository;

internal RunSettings RunSettings;

And is mainly made up of methods for setting and getting FeatureContext variables which are effectively global variables that exist for duration of the running of the current Feature.

**For example:**

internal TestUser LoggedInUser

{

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

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

}

LoggedInUser is a property object. Its values is set like this

\_ctx.LoggedInUser = user;

Where \_ctx is the instance of a Context class (created by calling the ContextFactory class which takes care of the creation of Context class instances)

Its value (and its sub values as LoggedInUser is an object with its own members) are accessed like this:

\_accountGasPage.SearchAccountAndSite(**ContextFactory.Instance.LoggedInUser.AccountNum**, ContextFactory.Instance.LoggedInUser.SiteRefNum);

## ContextFactory Class

public static class ContextFactory

{

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

}

This appears to be a class that is just used to take care of the construction of a Context instance. However, since the constructor for Context does not take any parameters, does this add any value?

The above code seems to be equivalent to this:

public static class ContextFactory

{

internal static Context MyContextInstance()

{

Context instance;

return instance = new Context();

}

}

Don’t know what the internal access modifier does

## TestDataRepository Class

### GetUser

public TestUser GetUser(string userKey)

{…}

The constructor basically seems to go through a lot of complicated stuff (need help with detail of this) to work out the folder that contains the users file.

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

F12 on ReadFromFile, you will get

public interface IFileReader : IDisposable

{

IEnumerable<TEntity> ReadFromFile<TEntity>(string filePath) where TEntity : new();

}

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

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

This will get the first element in the users collection (created above) where the UserKey equals the userKey value supplied when GetUser() is called.

### The Lamda Expression

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

The parameter ‘u’ will be the same as the type that calls it, in this case users, which is of type TestUser.

It is passed into the anonymous function represent by ‘u.UserKey == userKey’.

This function simply returns true or false depending on the outcome of the expression.

public static TSource FirstOrDefault<TSource>(this IEnumerable<TSource> source, Func<TSource, bool> predicate)

FirstOrDefault takes in a source of type TSource (which will be the type of the object that calls it (this), in this case users, which is of type TestUser.

It also takes in an anonymous function, that is given the name of predicate within the FirstOrDefault method.

This function also takes in a source of type TSource and returns a Boolean.

## TestDataRepository

It extends ITestDataRepository, but I don’t see any advantage in having this interface.

The constructor is taken up with calculating then setting the value of the \_dataFolder variable.

The GetUser(string userKey) method is the key to this class.

It uses an XLSX utility, seemly found in:

GazpromEnergy.Shared.Integration.Interface.Services.Files;

and

GazpromEnergy.Shared.Integration.Services.Excel;

To read the Users.xlsx file found in the \_dataFolder.

<TestUser> indicates what the data it is ready should look like.

Users will be a collection of TestUser objects.

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

This will get the first element in users that meets the condition of the UserKey of the element matching the userKey passed into the method.

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

I think that this code:

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);

could be simplified to:

if (selectedUser == null)

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

return selectedUser;

### How to use GetUser()

var user = \_ctx.TestDataRepository.GetUser(userEmail);

\_ctx is an instance of the Context class which is a wrapper around TestDataRepository

If you did not have this, the code would simply be:

var user = TestDataRepository.GetUser(userEmail);

if TestDataRepository was changed into a static class.

or

tdr. .GetUser(userEmail);

if tdr was an instance of TestDataRepository