OPTIMUS - GETTING STARTED

WITH FAQ'S

Abstract

Guide to get started with Optimus (Automation Framework) and provide answers to FAQ's

Shivam Mishra

Contents

Getting Started

Purpose

Guide to get started with Optimus (Automation Framework) and provide answers to FAQ’s

Audience

Project solutions teams who are looking to use Optimus to write more maintainable test cases for any solution of masterworks.

Optimus Framework Components

We have 3 main projects inside Optimus –

1. **AutomationStart** – It contains logic of how the test cases are going to run. It is also the start up project and is a console application. We will focus on the configurations folder as it contains all the xml files that defines how automation is going to be configured.
2. **MWPlatform** – It contains all the framework related code. Examples – XML Helpers, Database utilities, IO utilities, Selenium wrappers, Logic for reading from list page in the product and other common function that is used in the entire product
3. **ProductTestBed** – It contains all the test cases that are written for the product.

How to configure Optimus for a Solutions?

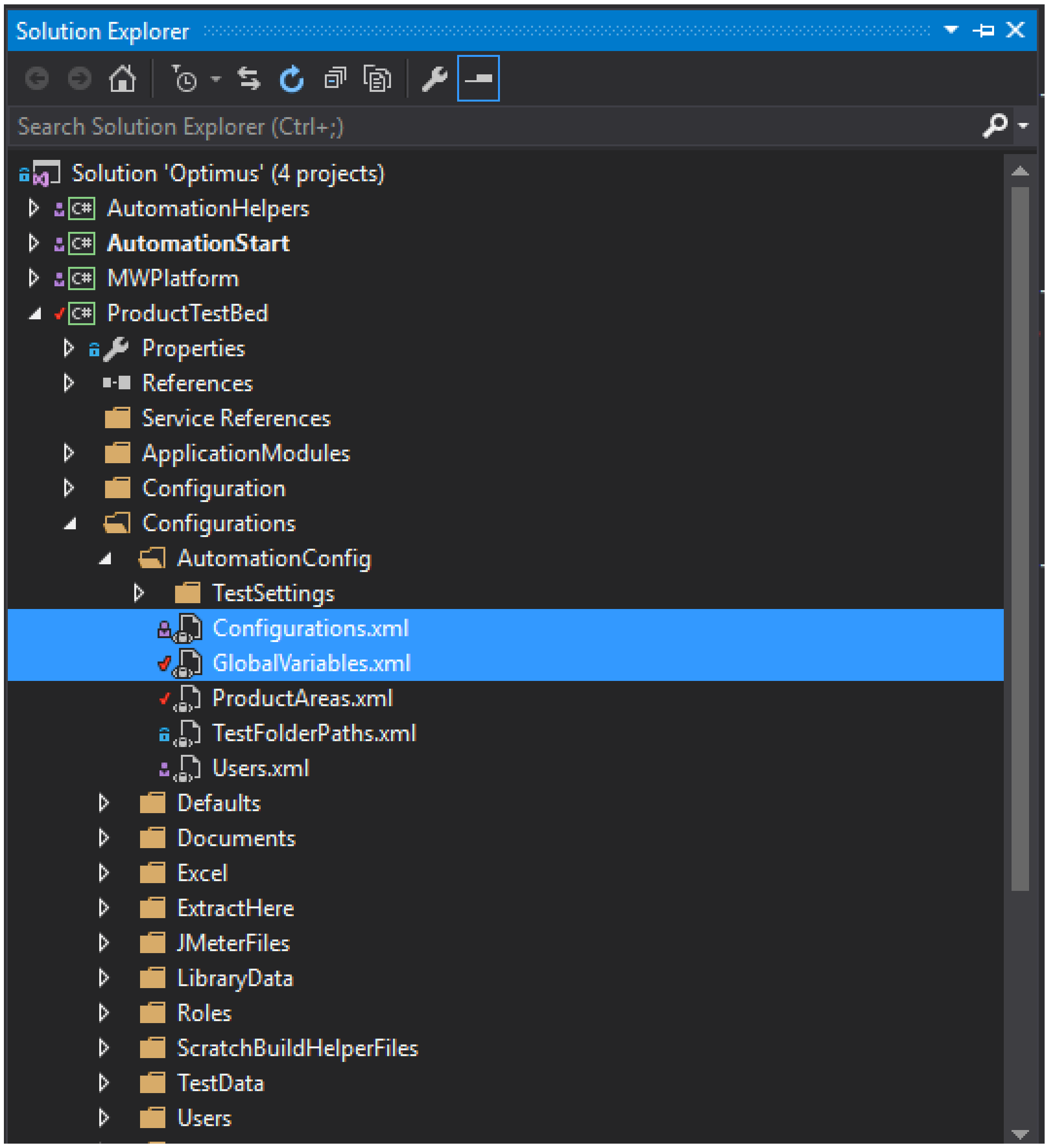
GlobalVariables.xml and Configurations.xml

It contains all the settings like build details, server details, new website details, configurations, email settings, automation dashboard settings, etc. Configurations are split up into two files. i.e. GlobalVariables.xml and Configurations.xml. The basis of this splitting is that there are few settings that we change very frequently, which are put in GlobalVariables.xml file. The others, we seldom change, and they are grouped in Configurations.xml file.

The diagram (figure 1) below shows the location of both the xml files.

These xml files are placed in the Test Bed because we can have different configurations for each of the solutions teams.

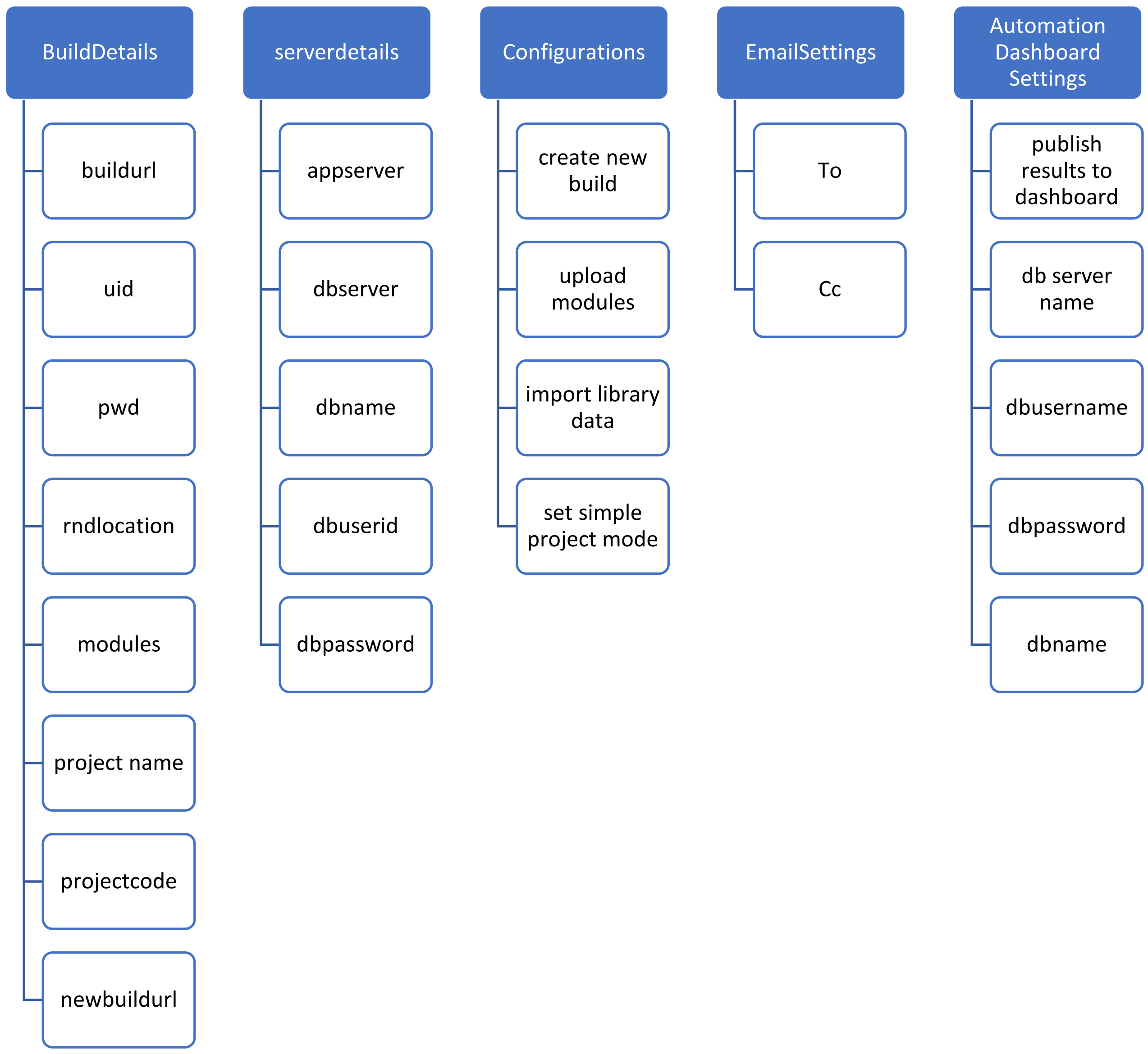
The entire configurations folder is copied to the bin folder of the executor as a part of the post build event so that the executor can use it.



```
Solution 'Optimus' (4 projects)
  AutomationHelpers
  AutomationStart
  MWPlatform
  ProductTestBed
    Properties
    References
    Service References
    ApplicationModules
    Configuration
    Configurations
      AutomationConfig
        TestSettings
        Configurations.xml
        GlobalVariables.xml
        ProductAreas.xml
        TestFolderPaths.xml
        Users.xml
      Defaults
      Documents
      Excel
      ExtractHere
      JMeterFiles
      LibraryData
      Roles
      ScratchBuildHelperFiles
      TestData
      Users
```

Snapshot of GlobalVariables.xml file (figure 1)

Few important settings in these files are displayed in the chart below and we will discuss each one them in brief.



Some important settings in GlobalVariables.xml file Snapshot of GlobalVariables.xml file (figure 2)

Build Details

- **Buildurl** – it is the URL of the instance on which the automation will run.
- **Uid** – user id that will be used to log in to masterworks
- **Pwd** – password of the user which will be used to login
- **Rndlocation** – location where we pick up the latest zip files to upload in the product
- **Modules** – list of modules which are separated by commas which are to be uploaded in the exact same order.
- **Projectname/projectcode** – name and code of the project on which we will run our tests. If it is a create project test case, then the project name and project code must be overwritten so that other tests can make use of this to execute.
- **Newbuildurl** – when we create a new build, this tag is used to write the URL of the newly deployed build.

Server Details

It contains the details of the machine where the product to be tested is hosted. It has the server and the database details. The database details could be used for executing SQL queries on the server.

- **Appserver** – machine where the product is deployed
- **Dbserver** – machine where the database of the product is hosted
- **Dbname** – name of the database that the product is using
- **Dbuserid** – user id of the database
- **Dbpassword** – password for the database server

Configurations

- **Createnewbuild** – if set to yes, then a new build will be created and newbuildurl and buildurl will be replaced with the URL of this new build and we will use this to run all our test cases.
- **Upload modules** – if set to yes, then we upload all the modules listed in the modules tag in the exact same order.
- **Import library data** – if set to yes, then all the library data is imported into the database and we can use this data to run our tests.
- **Set simple project mode** – if set to yes, then we don't use the planning module (without planning runs)

Email Settings

It is used to specify the stake holders of the automation run who will be receiving the reports of the automation runs via emails. You can set it to your own email for development purposes.

- **To** – To field for sending out the email reports. It is not a mandatory field and if not provided or incorrectly provided, then you will not receive the email reports of the automation runs.
- **Cc** – CC field to send the emails.

Automation Dashboard Settings

This setting is used to configure whether the report will be published on the automation dashboard.

- **Publish results to dashboard** – if set to yes, then the automation report will be published to the automation dashboard.
- **Db server name** – server name where the dashboard database is present.
- **Db user name** – user name to use the DB server of the dashboard
- **Db password** – password to login into the dashboard DB server
- **Dbname** – name of the database which the dashboard is using

You can opt to publish the results of Daily Smokes and weekly automation runs on to the dashboard. Do not publish the results of testing and development reports on the dashboard.

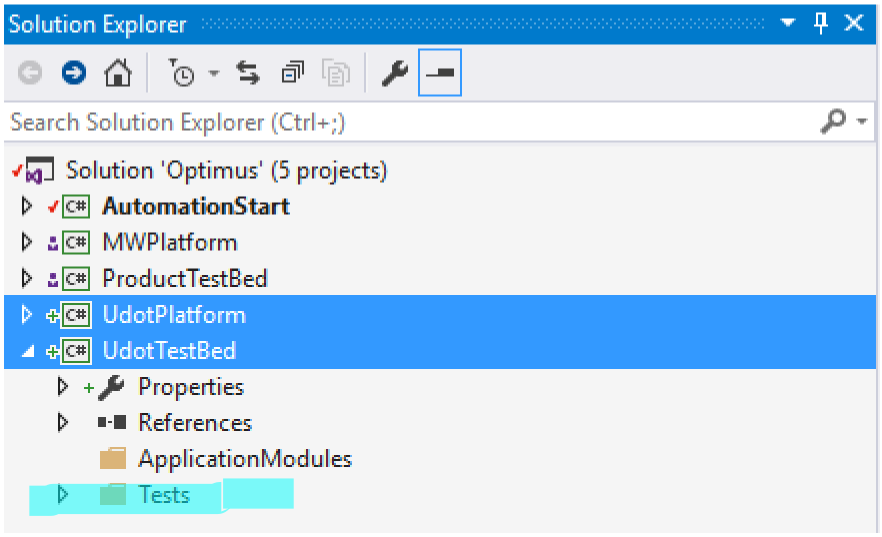
For sample settings of GlobalVariables.xml file, you can refer Optimus and modify these settings to run automation on your builds.

FAQ's

Where can I write the test cases customized for my product?

Each solution team will have a platform assembly and a test bed. You can write all the generic utilities which is not available in Optimus and which is customized for your product in [TeamName]Platform and all the test cases in [TeamName]TestBed where team name is your team name. For instance, UdotPlatform and UdotTestBed for Udot Team.

Here is a snapshot that shows the situation.



(figure 3)

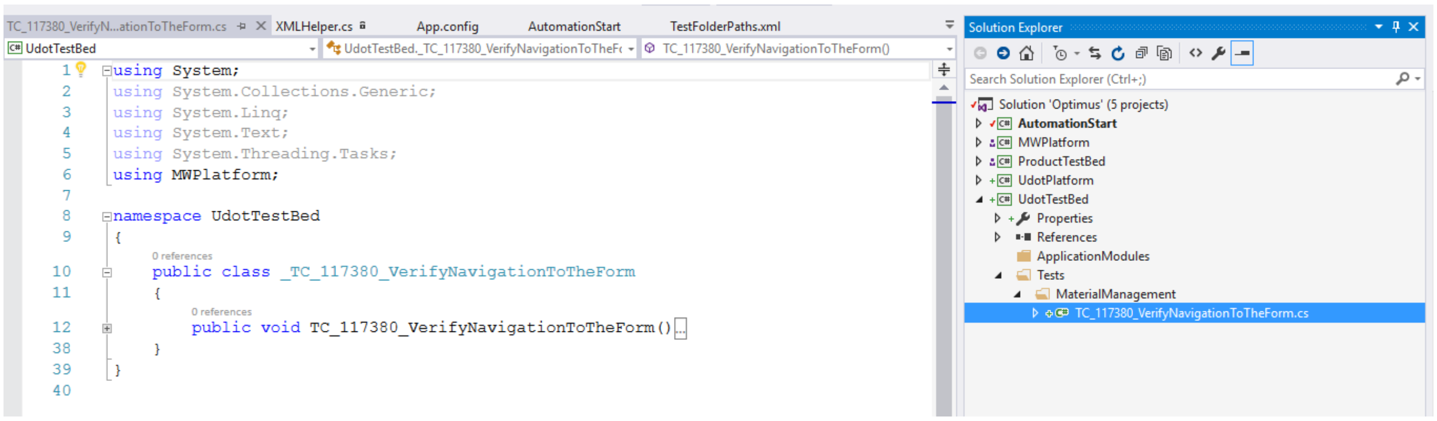
You can see the platform and the testbed highlighted in the above snapshot. The Test cases will be written in the Tests folder under UdotTestBed. You are free to create a folder structure inside the tests folder to keep your code organized. Application modules are used to extract the logic of the action to be performed in each form for reusability and maintainability purposes. You can follow the structure that we have followed in ProductTestBed and MWPlatform.

You will have to write post build events to copy the DLL's and PDB file of your test bed and platform to the bin folder of the AutomationStart. This is done because the executor does not have a direct reference to the test bed, and it loads the dll's of the test bed dynamically so that the framework is can be delivered to various solutions teams. Its like plug and play. Just put your DLL's and PDB's of the test bed, copy your configurations and we will execute the test cases for you, and take care of reporting and other tasks.

What are the conventions that must be followed to create a test case?

- Class containing the test case will be named as the test case preceded by an underscore.
- The function containing a test case will have the same name as the test case
- Test data file name should be the same as the test case name.
- Test settings file must be placed in the TestSettings folder
- Product areas, configurations, globalVariables, Users, and TestFolderPaths files must be present in the AutomationConfig folder
- Test data files must be placed in the test data folder
- Tests must be written in the assembly and namespace that is specified in the TestFolderPaths File

Consider this scenario where we are writing the tests for UDOT project. Observe the Assembly name and the namespace under which the test is written. Also note the convention for naming the test class and test method.


TC_117380_VerifyN...ationToTheForm.cs
XMLHelper.cs
App.config
AutomationStart
TestFolderPaths.xml
UdotTestBed
UdotTestBed._TC_117380_VerifyNavigationToTheFo
TC_117380_VerifyNavigationToTheForm()
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'Optimus' (5 projects)
AutomationStart
MWPlatform
ProductTestBed
UdotPlatform
UdotTestBed
Properties
References
ApplicationModules
Tests
MaterialManagement
TC_117380_VerifyNavigationToTheForm.cs

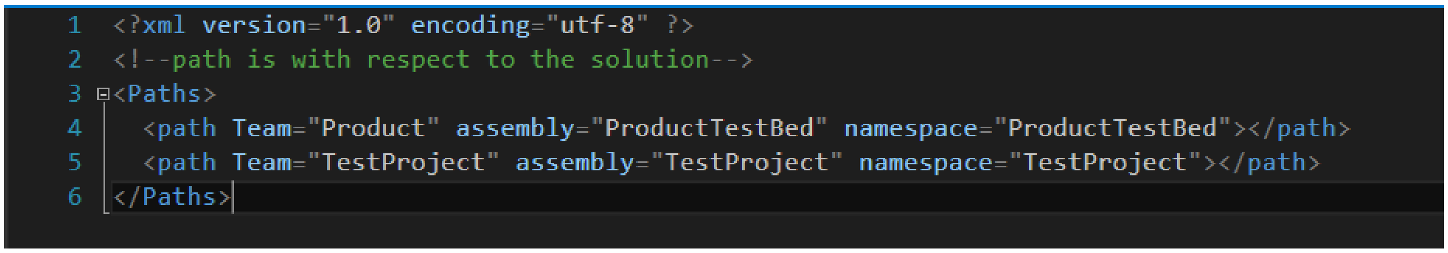
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using MWPlatform;

namespace UdotTestBed
{
    0 references
    public class _TC_117380_VerifyNavigationToTheForm
    {
        0 references
        public void TC_117380_VerifyNavigationToTheForm()...
    }
}
```

(figure 4)

These conventions are important because we automatically pick up the test case that you specify to run in the XML files. You don't have to put any entry for that to happen anywhere else other than the xml file. The system is designed to pick the tests which follow these conventions.

Test folder Paths file



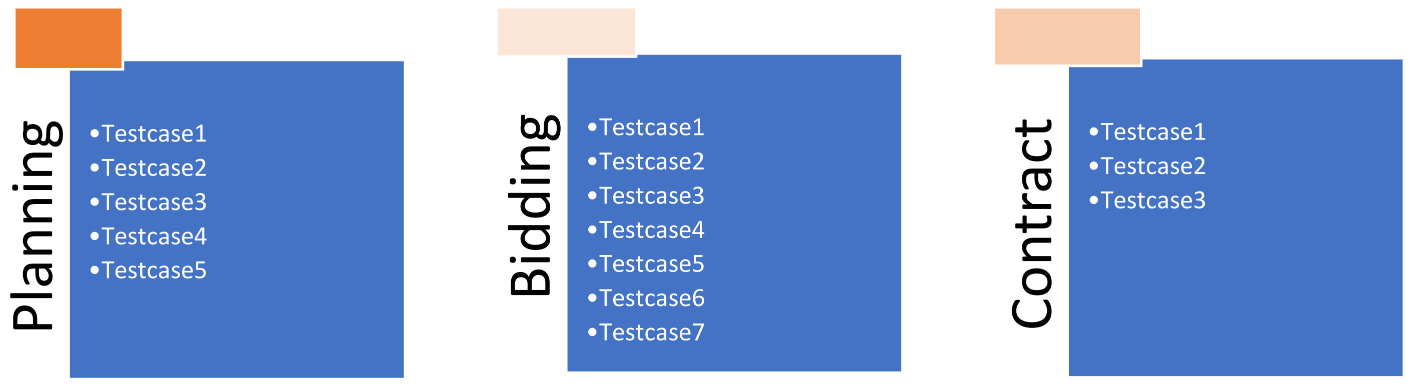
```xml
<?xml version="1.0" encoding="utf-8" ?>
<!--path is with respect to the solution-->
<Paths>
  <path Team="Product" assembly="ProductTestBed" namespace="ProductTestBed"></path>
  <path Team="TestProject" assembly="TestProject" namespace="TestProject"></path>
</Paths>
```

How to execute the test cases?

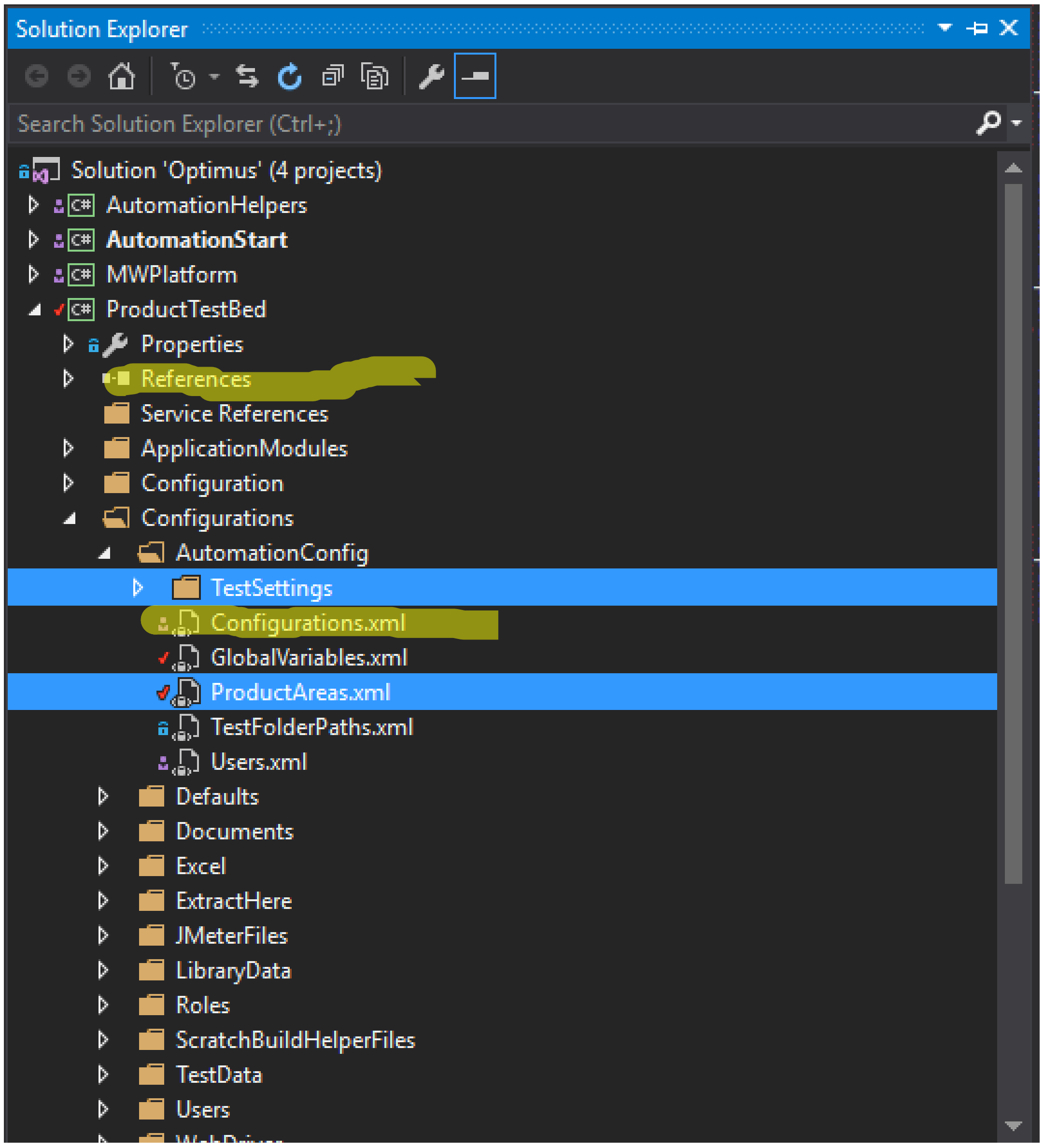
Before we talk about how to run the tests let's see what is the logic which is used to execute the test cases.

We have different module in masterworks. Each one corresponds to a product area. Each product area can have n number of test cases. All the product areas are listed in ProductAreas.xml file. Each product area is listed in the xml file with a unique id and has a team name associated with it. If the team name is not specified, then the default value of the team that is specified in the App.config file in AutomationStart will be picked. To override this, you must specify the TeamName attribute with the required value.

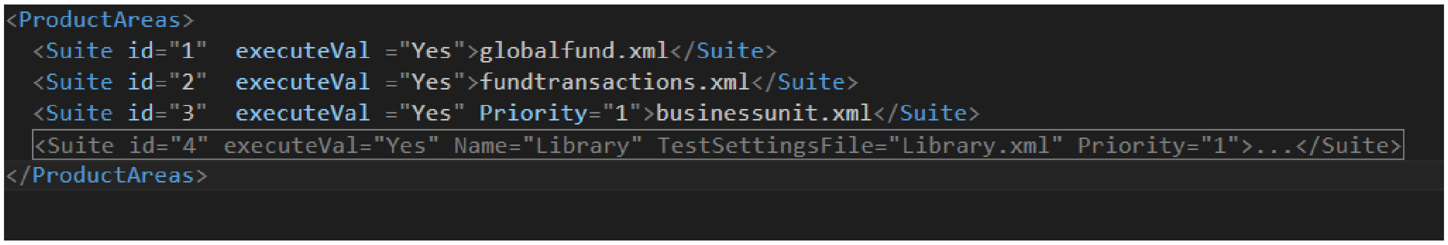
The text of the Suite tag in the xml file is another xml file which lists all the test cases that belongs to that Suite. Those files are present in the TestSettings folder



Product Areas and Test Cases (figure 5)



The figure below shows the location of ProductAreas.xml file and the TestSettings Folder (figure 6)



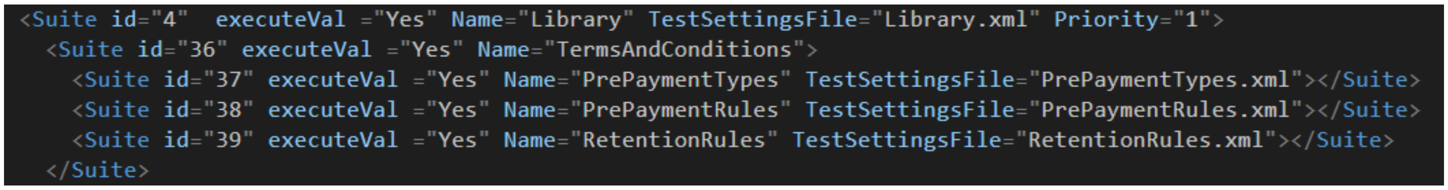
```xml
<ProductAreas>
  <Suite id="1"  executeVal ="Yes">globalfund.xml</Suite>
  <Suite id="2"  executeVal ="Yes">fundtransactions.xml</Suite>
  <Suite id="3"  executeVal ="Yes" Priority="1">businessunit.xml</Suite>
  <Suite id="4" executeVal="Yes" Name="Library" TestSettingsFile="Library.xml" Priority="1">...</Suite>
</ProductAreas>
```

Product Areas xml file (figure 7)

We can see in the snapshot that Suites have no TeamName specified, so the executor will look for those test cases in ProductTestBed dll, because the default value for the team is Product as specified in the App.config file in AutomationStart. ExecuteVal of each suite specifies whether to pick up that suite for execution. If set to no, that suite will be ignored for execution.

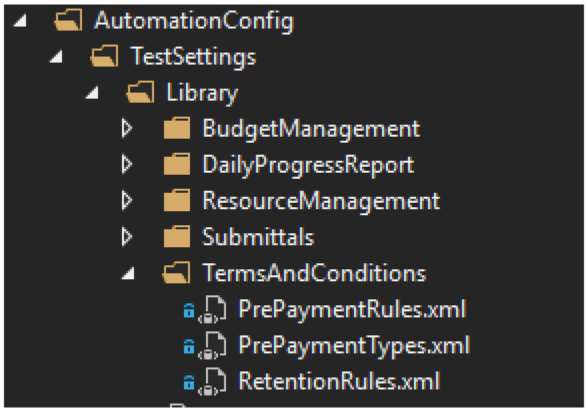
The test setting file can be specified either as a node inner text or as an attribute 'TestSettingsFile'. These settings files should be present directly in TestSettings folder under AutomationConfig.

If you want to organize the hierarchy level of the test settings files, then you can nest the suites in the Product Areas xml file and provide a Name attribute for each level. The naming will define the folder hierarchy of the test settings xml file. For example, consider the library Suite shown in the diagram below.



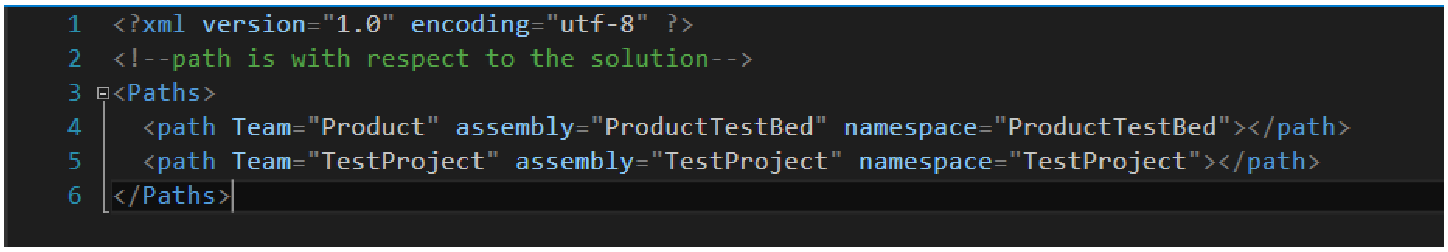
```xml
 <Suite id="4"  executeVal ="Yes" Name="Library" TestSettingsFile="Library.xml" Priority="1">
   <Suite id="36" executeVal ="Yes" Name="TermsAndConditions">
     <Suite id="37" executeVal ="Yes" Name="PrePaymentTypes" TestSettingsFile="PrePaymentTypes.xml"></Suite>
     <Suite id="38" executeVal ="Yes" Name="PrePaymentRules" TestSettingsFile="PrePaymentRules.xml"></Suite>
     <Suite id="39" executeVal ="Yes" Name="RetentionRules" TestSettingsFile="RetentionRules.xml"></Suite>
   </Suite>
```

It has a sub-suite called terms and conditions and that sub-suite has test settings file. Observe how the Name attribute for each of the suite and sub suite is done. Now look at the folder structure in the test settings.



```
AutomationConfig
  TestSettings
    Library
      BudgetManagement
      DailyProgressReport
      ResourceManagement
      Submittals
      TermsAndConditions
        PrePaymentRules.xml
        PrePaymentTypes.xml
        RetentionRules.xml
```

You can imagine it as Library Suite has a sub suite i.e. Terms and conditions and that sub-suite has test cases. Same situation is replicated at both places, ProductAreas xml file and the folder structure.



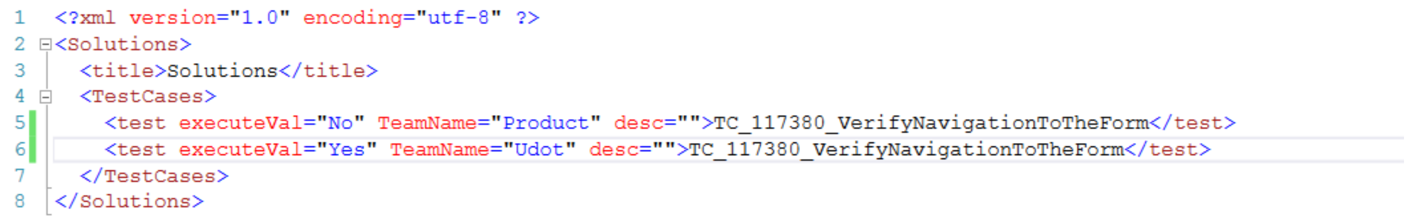
```xml
<?xml version="1.0" encoding="utf-8" ?>
<!--path is with respect to the solution-->
<Paths>
  <path Team="Product" assembly="ProductTestBed" namespace="ProductTestBed"></path>
  <path Team="TestProject" assembly="TestProject" namespace="TestProject"></path>
</Paths>
```

Snapshot of the TestFolderPaths.xml (figure 8)

The DLL's of the teams that you specify in this file (Test folder paths) will be loaded dynamically in the runtime. These dll's will have the tests for their respective solutions. If you miss this step, then your tests will not be executed as the executor will not have an idea of the location of the test cases that you will specify in the settings file.

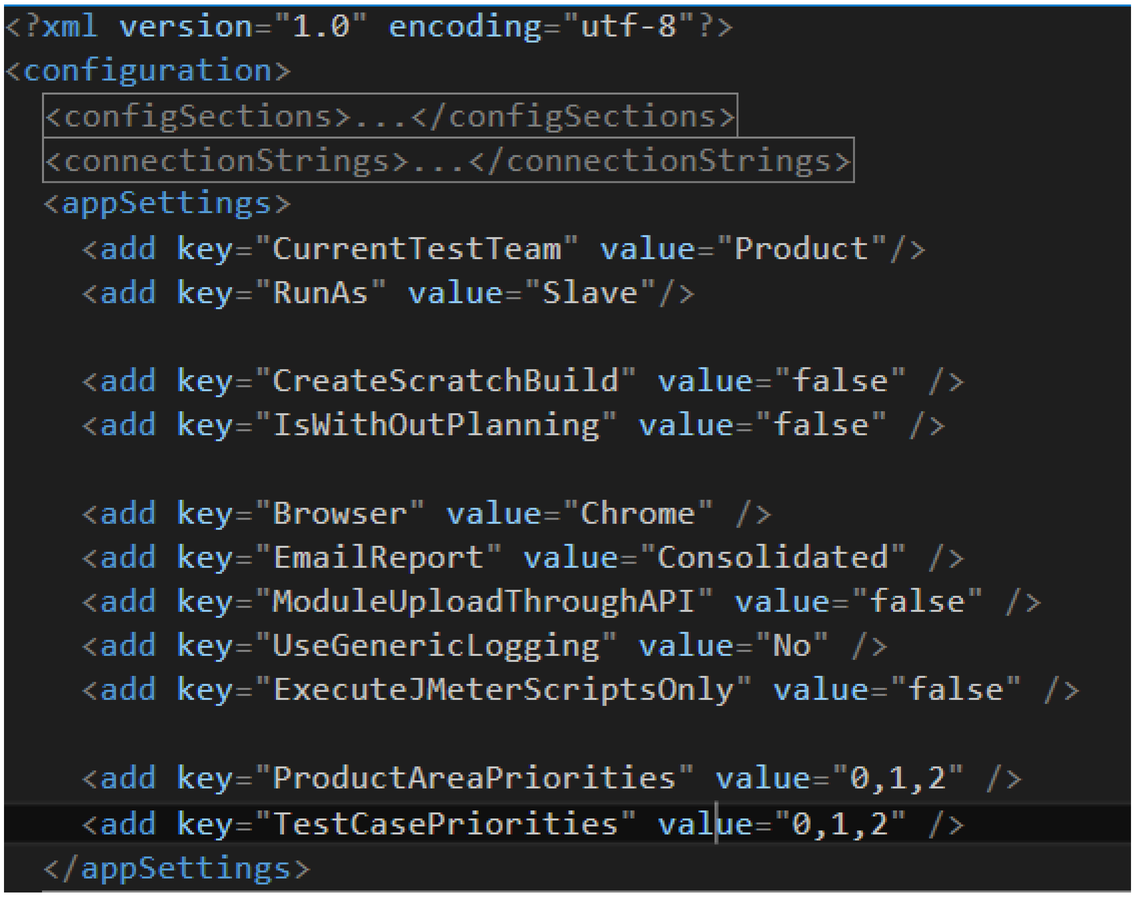
You will also have to provide the correct assembly and namespace under which your tests are written, so the executor knows where to pick the tests from, since it calls your test methods by reflection.

Even the test cases can have the TeamName attribute to specify the team to which the test case belongs to and it will override the value that was specified in the product Areas xml file. This feature helps you to use some tests from another solution (like the product tests) if it is exactly same, so you don't have to duplicate the test cases.



```xml
<?xml version="1.0" encoding="utf-8" ?>
<Solutions>
  <title>Solutions</title>
  <TestCases>
    <test executeVal="No" TeamName="Product" desc="">TC_117380_VerifyNavigationToTheForm</test>
    <test executeVal="Yes" TeamName="Udot" desc="">TC_117380_VerifyNavigationToTheForm</test>
  </TestCases>
</Solutions>
```

Snapshot of the test settings xml file (figure 9)



```xml
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>...</configSections>
  <connectionStrings>...</connectionStrings>
  <appSettings>
    <add key="CurrentTestTeam" value="Product"/>
    <add key="RunAs" value="Slave"/>

    <add key="CreateScratchBuild" value="false" />
    <add key="IsWithOutPlanning" value="false" />

    <add key="Browser" value="Chrome" />
    <add key="EmailReport" value="Consolidated" />
    <add key="ModuleUploadThroughAPI" value="false" />
    <add key="UseGenericLogging" value="No" />
    <add key="ExecuteJMeterScriptsOnly" value="false" />

    <add key="ProductAreaPriorities" value="0,1,2" />
    <add key="TestCasePriorities" value="0,1,2" />
  </appSettings>
```

App.Config file in AutomationStart

Logic for executing the test cases –

The executor picks up a product area from the ProductArea.xml file and executes all the test cases that are present in that product area and moves on to the next one.

If you want a specific product area to be run, you can pass the id of the product area via the command line. In this case, the executor will pick up only the specified product area and will execute only that area. Multiple areas can be specified in the command line separated by spaces

Now let's see how does the executor finds the location of the test case file –

We have a configuration file called TestFolderPaths. For all the teams, it has attributes that specifies the physical location of the test case files, the assembly and the namespace in which the test cases are written. Using this information and the naming conventions of the test cases which are specified above, the executor locates the test case and starts its execution.

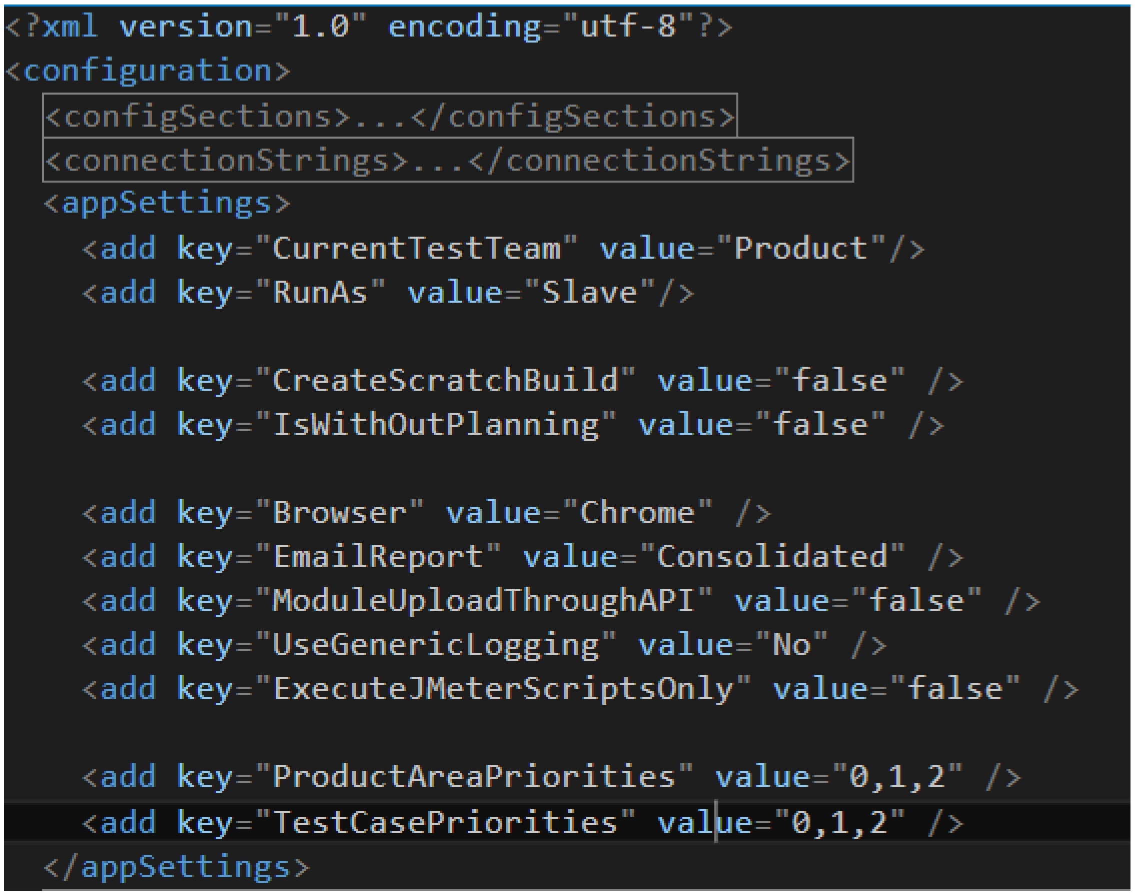
I have multiple product areas, and I want to run only few of them. How can I do that?

This can be done by passing the id of a specific product area via the command prompt. If you look closely at the ProductArea.xml file, you will see that each of the product area has a unique id associated with it (refer figure 7). We make use of this id to select the product area(s) to run. Multiple product areas can be also be configured to run by passing more than one id's from the command prompt separated by spaces.

We can pass command line arguments either from the command prompt or from visual studio using the IDE itself.

Can I run the tests on different browsers?

We have App.config file in AutomationStart assembly where we have a field to specify a browser. It could be chrome, Firefox or i.e. Here is a snapshot of the file



```xml
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>...</configSections>
  <connectionStrings>...</connectionStrings>
  <appSettings>
    <add key="CurrentTestTeam" value="Product"/>
    <add key="RunAs" value="Slave"/>

    <add key="CreateScratchBuild" value="false" />
    <add key="IsWithOutPlanning" value="false" />

    <add key="Browser" value="Chrome" />
    <add key="EmailReport" value="Consolidated" />
    <add key="ModuleUploadThroughAPI" value="false" />
    <add key="UseGenericLogging" value="No" />
    <add key="ExecuteJMeterScriptsOnly" value="false" />

    <add key="ProductAreaPriorities" value="0,1,2" />
    <add key="TestCasePriorities" value="0,1,2" />
  </appSettings>
```

Figure 10

Priorities of Test cases and Product Areas to be run

You can specify the priority of test cases and product areas that you want to be picked up for execution in the App.config file in AutomationStart. Multiple priorities can be specified here separated by comma (,), as this only says the priority of product areas or test cases to be picked up for automation.

Following are the allowed priority values –

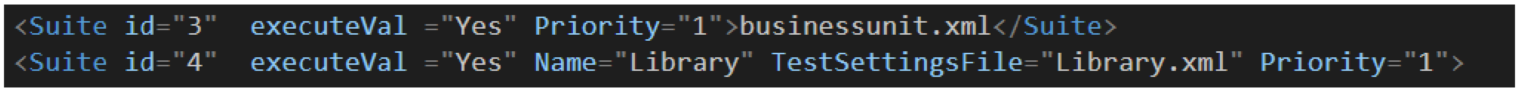
0 – High

1 – Medium

2 – Low

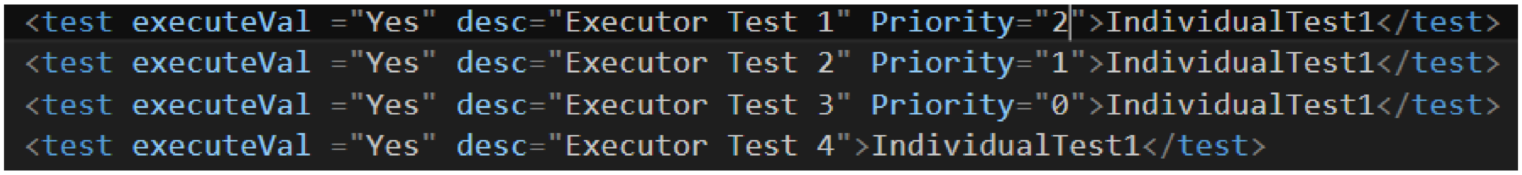
You can specify the Priority for each Product area and the test case in their respective nodes. If you do not specify the priority attribute, it will be considered as Medium by default.

Product Area -



```
<Suite id="3"  executeVal ="Yes" Priority="1">businessunit.xml</Suite>
<Suite id="4"  executeVal ="Yes" Name="Library" TestSettingsFile="Library.xml" Priority="1">
```

TestSettings-



```
 <test executeVal ="Yes" desc="Executor Test 1" Priority="2">IndividualTest1</test>
 <test executeVal ="Yes" desc="Executor Test 2" Priority="1">IndividualTest1</test>
 <test executeVal ="Yes" desc="Executor Test 3" Priority="0">IndividualTest1</test>
 <test executeVal ="Yes" desc="Executor Test 4">IndividualTest1</test>
```

If a product area that has a dependency on another area, then that area will be picked up for execution (if not executed earlier in the same Run) without considering its priority and only the critical tests of that area will be executed.

How can I report test results/logs?

You can use Utilities.Reports class in MWPlatform to start logging and end logging. You can refer any test case in platform test bed to see how to log success, failure, info, error and skip.

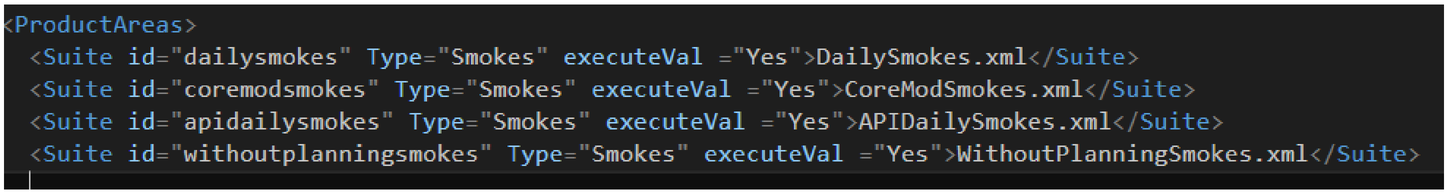
Can I get separate Reports for each product area?

Yes, just configure the EmailReport Setting in the same file and set it to Individual. You will then start receiving one report file for each product area. If it is set to consolidated, then you will receive one report in which all the tests will be clubbed.

What is Full pass and daily smokes and how can I Run them?

Full pass is a run where all the tests that are written in the system for the product will run. You can run full pass by setting the executeVal of all the productAreas and all the tests as yes. All the suites of Type Smokes will be ignored in the full pass run.

Daily smokes are a set of test cases that can check the sanity of the build. It need not contain all the detailed test cases, it only has those test cases that will ensure the sanity if the build. All the smokes suites have an attribute ‘Type’ defined as smokes. You can run smokes by passing the id of the suite as a command line argument. The figure below shows the situation described.



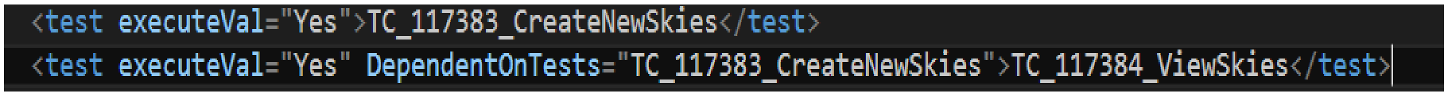
```
<ProductAreas>
  <Suite id="dailysmokes" Type="Smokes" executeVal ="Yes">DailySmokes.xml</Suite>
  <Suite id="coremodsmokes" Type="Smokes" executeVal ="Yes">CoreModSmokes.xml</Suite>
  <Suite id="apidailysmokes" Type="Smokes" executeVal ="Yes">APIDailySmokes.xml</Suite>
  <Suite id="withoutplanningsmokes" Type="Smokes" executeVal ="Yes">WithoutPlanningSmokes.xml</Suite>
```

For instance, if you want to run daily smokes, you just pass dailysmokes as a command line argument.

How can I handle the dependency of the test cases?

If a test case fails, then all the tests which were dependent on the former would also fail and all of them would be marked as failed in the reports. A better way to present the picture would be to skip the tests which were dependent on a case and that -case failed and mark them Skipped in the reports. This can be achieved with the help of **DependentOnTests** attribute in the tests xml file.

Here is an example of the same

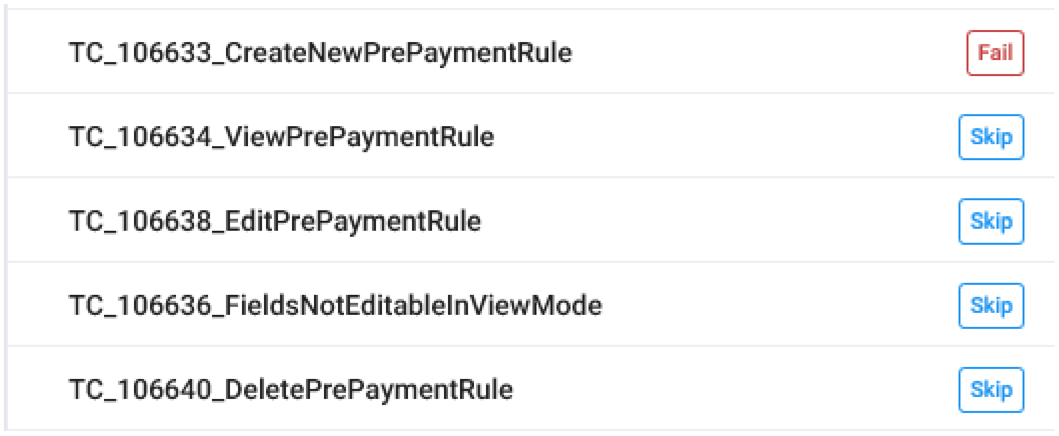


```
<test executeVal="Yes">TC_117383_CreateNewSkies</test>
<test executeVal="Yes" DependentOnTests="TC_117383_CreateNewSkies">TC_117384_ViewSkies</test>
```

Figure 11

In the above example, a record can be viewed only if it is created and hence the dependency. Observe how the dependencies are injected in the test cases.

Here is a sample report files with test cases skipped because of dependency issues.



TC_106633_CreateNewPrePaymentRule	Fail
TC_106634_ViewPrePaymentRule	Skip
TC_106638_EditPrePaymentRule	Skip
TC_106636_FieldsNotEditableInViewMode	Skip
TC_106640_DeletePrePaymentRule	Skip

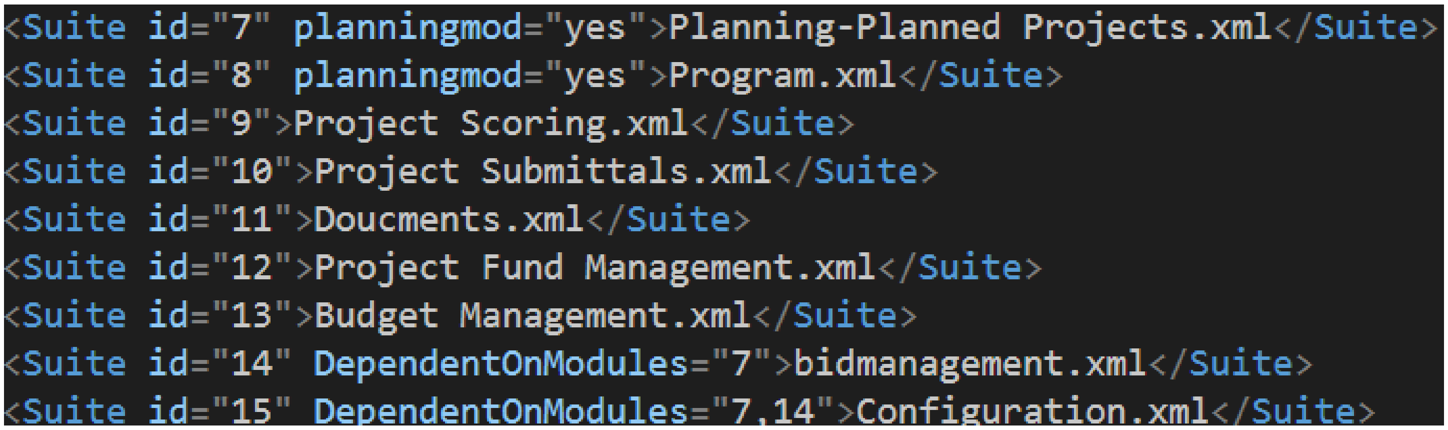
Figure 12

What if the modules are dependent?

You can add an attribute called **DependentOnModules** for handling the module dependencies. If a module fails, then all the other Product areas which are dependent on the former will be skipped in execution and all the tests in that product area will be marked Skipped in the Report.

Value for **DependentOnModules** attribute could be one or more ID's of product area(s) which are separated by comma.

Example of marking dependencies in the modules (figure 13)



```
<Suite id="7" planningmod="yes">Planning-Planned Projects.xml</Suite>
<Suite id="8" planningmod="yes">Program.xml</Suite>
<Suite id="9">Project Scoring.xml</Suite>
<Suite id="10">Project Submittals.xml</Suite>
<Suite id="11">Doucments.xml</Suite>
<Suite id="12">Project Fund Management.xml</Suite>
<Suite id="13">Budget Management.xml</Suite>
<Suite id="14" DependentOnModules="7">bidmanagement.xml</Suite>
<Suite id="15" DependentOnModules="7,14">Configuration.xml</Suite>
```

Figure 13

There is one more functionality of this attribute. If the modules on which a certain module is dependent on hasn't been executed in the current run, then that module will be picked for execution first and then the current module will be executed. For instance, consider the diagram above. Module 14 is dependent on module 7. If you want to execute only module 14 (by passing the command line arguments), then the executor will pick up module 7 and execute only the **critical tests** of that module so that the dependency for module 14 is resolved and then execute module 14.

Where can I put my test data?

You can create test data xml files in the TestData folder in AutomationConfig under the Test Bed.

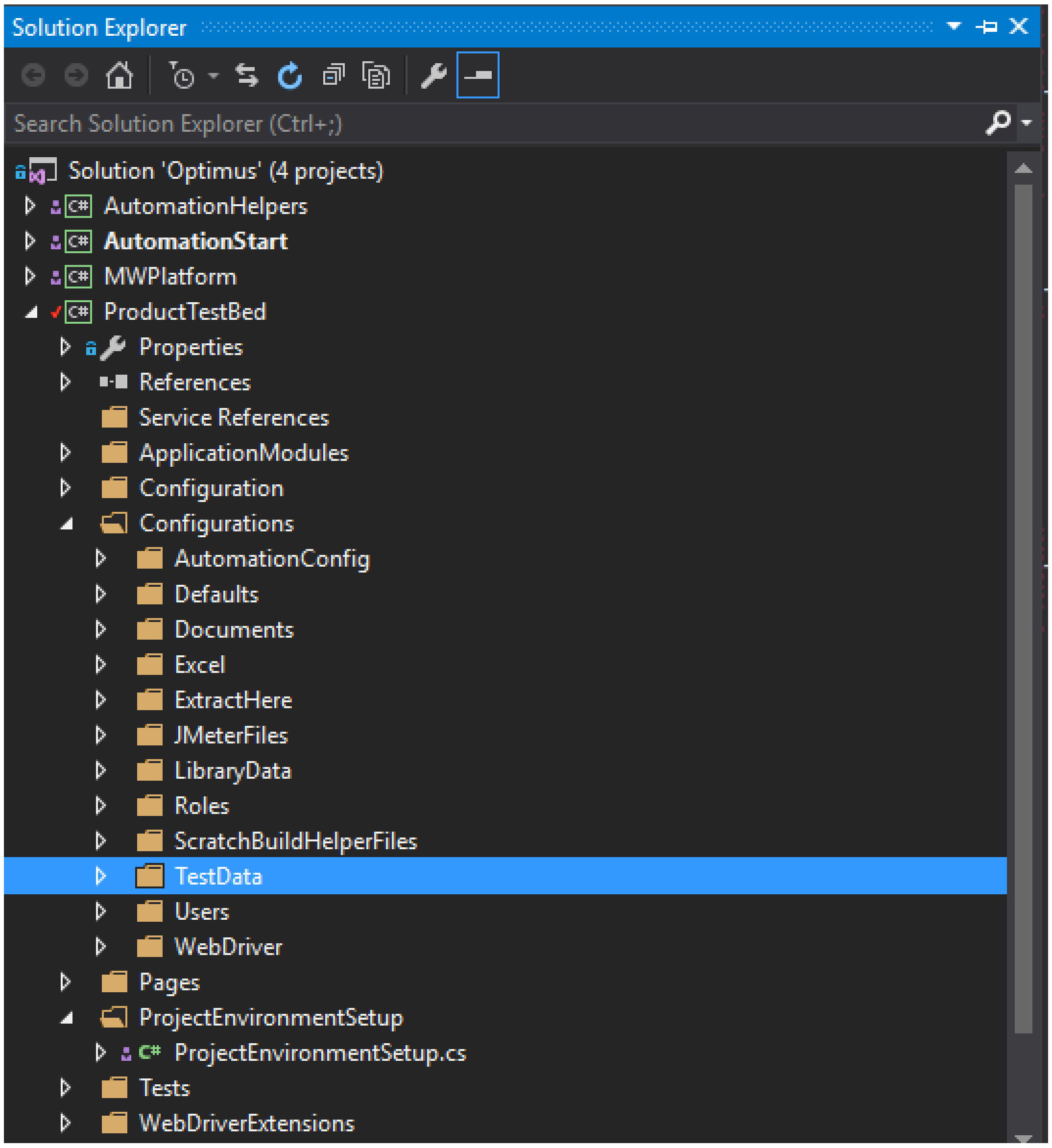


Figure 14

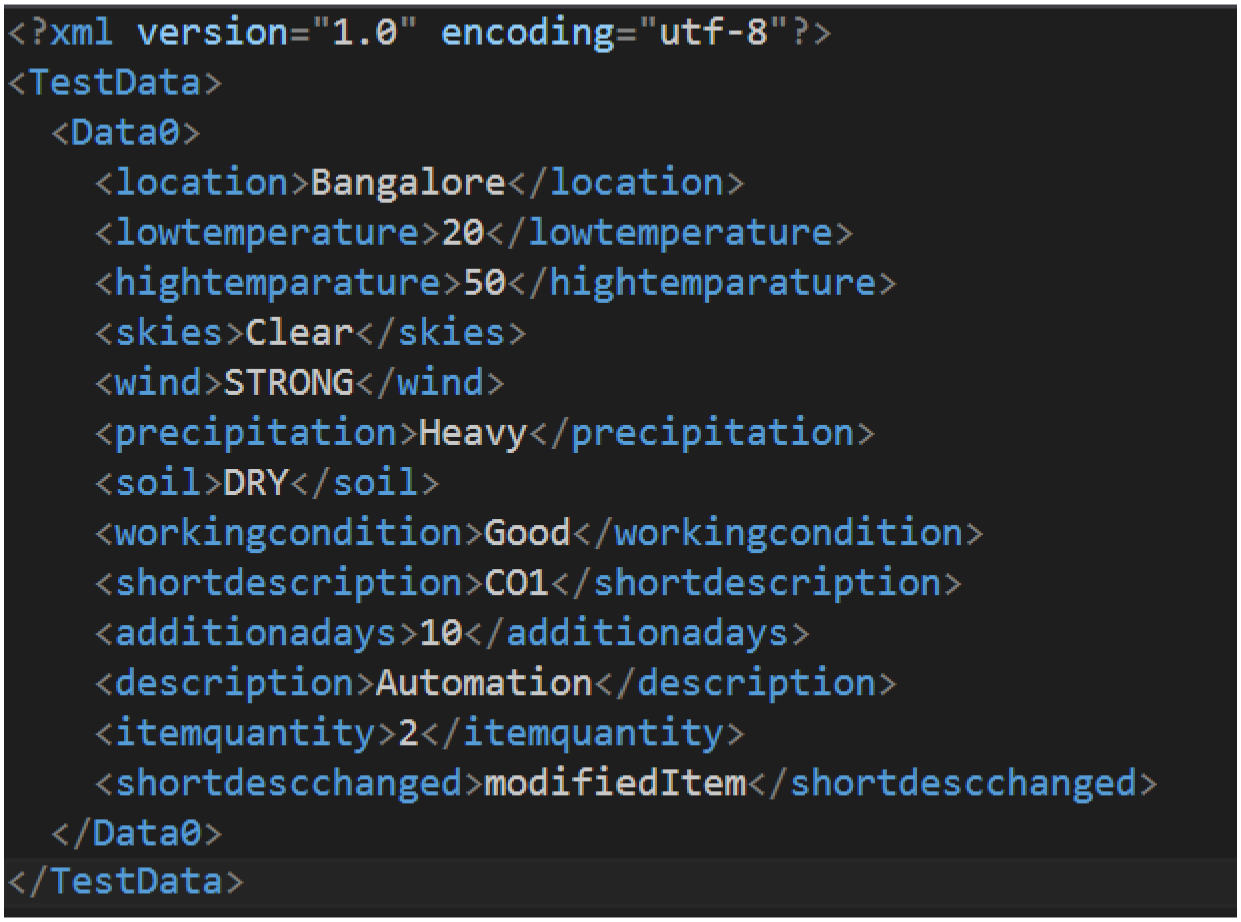
What is the Format for TestData xml files?

Some of the guidelines should be strictly followed while specifying the test data file. They are as follows –

1. Name of the testdata file will be same as the testcase and will be small caps only.
2. Root tag of the test data file is TestData and is case sensitive
3. Multiple sets of data can be specified in different datasets like Data0, Data1, Data2 and so on
4. All the tags in a dataset should be small caps strictly.

If the test case name is TC_117383_CreateNewSkies then the test data file name is tc_117383_createnewskies.xml

Here is an example of a test data file.



```xml
<?xml version="1.0" encoding="utf-8"?>
<TestData>
  <Data0>
    <location>Bangalore</location>
    <lowtemperature>20</lowtemperature>
    <hightemparature>50</hightemparature>
    <skies>Clear</skies>
    <wind>STRONG</wind>
    <precipitation>Heavy</precipitation>
    <soil>DRY</soil>
    <workingcondition>Good</workingcondition>
    <shortdescription>CO1</shortdescription>
    <additionadays>10</additionadays>
    <description>Automation</description>
    <itemquantity>2</itemquantity>
    <shortdescchanged>modifiedItem</shortdescchanged>
  </Data0>
</TestData>
```

Figure 15

How can I Read/Write data to the test data files?

You can make use of the TestData class in MWPlatform namespace. It has a method **Read** which accepts parameter name that is to be read and an optional data set number from where that parameter should be read. If the data set number is not provided, then by default it will read the value from the first dataset.

Where are the types of logs in Optimus?

We have the following logs for each run –

1. AutomationStart Logs – Logs for the Executor
2. XML Logs – xml logs for all the test cases
3. Consolidated report – HTML report depicting only an overview of the current run
4. Detailed report – Detailed HTML report that has logs for each test case.
5. Database Logs – XML file that has all the database exceptions generated for each test case.

Where are the Logs Generated?

All the logs are generated in the Logs folder in the bin

Automation dashboard – to track automation progress

It is a tool that is used to keep a track of the automation progress. URL – http://autodash.aurigo.net/

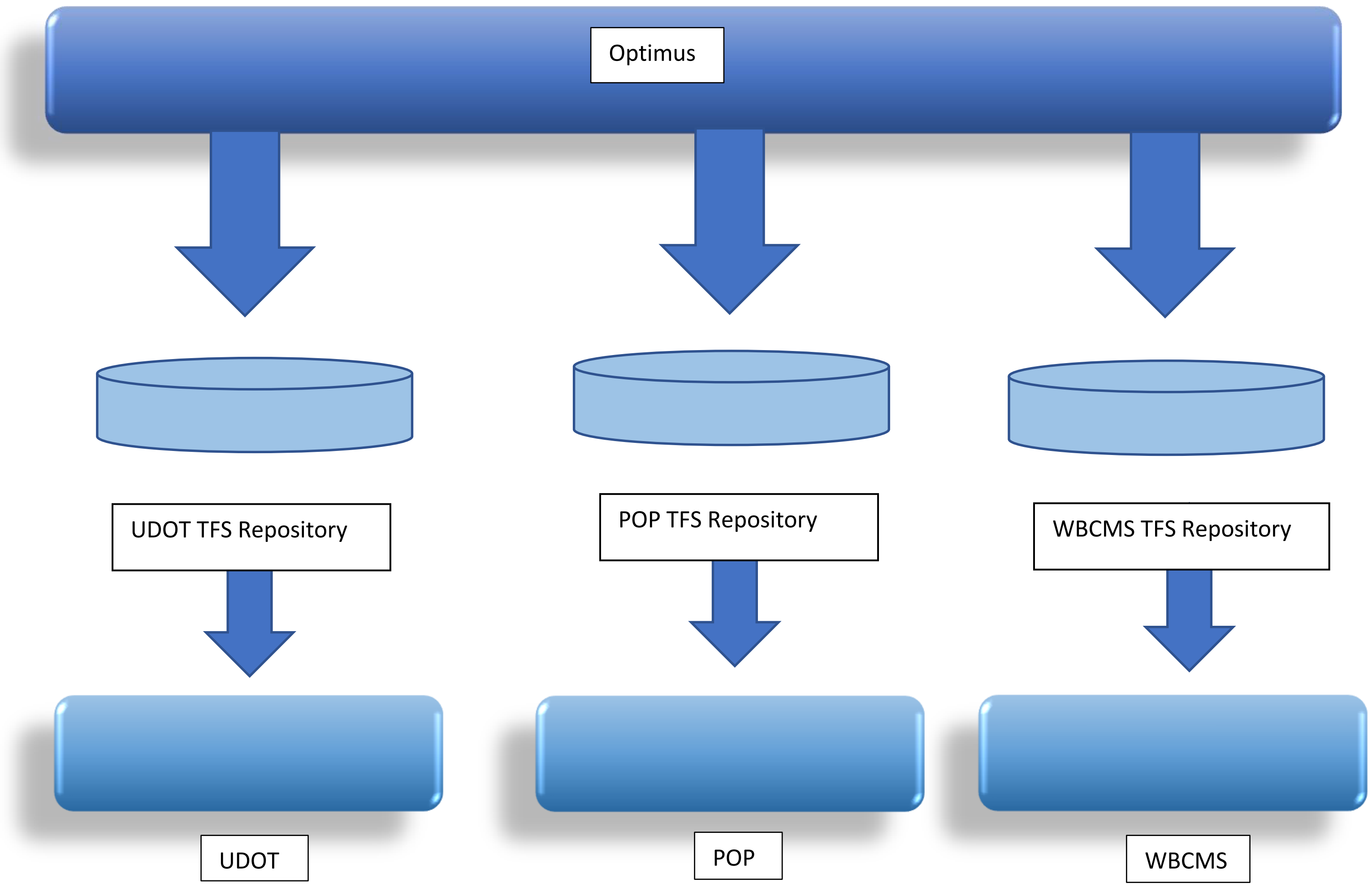
On the landing page, all the teams and their progress for the past three weeks are displayed and the last column shows the total number of test cases in TFS and how many of them are automated.

When you select a team, you can see the product area wise test case numbers and charts for the status of each product area. You can select a different product area from the dropdown. You can update the weekly progress for your team using excel sheets and the progress will be displayed in the weekly progress tab. Credentials for uploading data will be provided by us. In future, all the tests and their automation statuses will be fetched directly from TFS and you don't have to go through uploading excel sheets every week.

To align with this, mark the testcase as Automated in TFS and update the Automation Id.

The dashboard has a section to display the results of the previous automation runs, if we published them here. We have already discussed how to publish the reports on the dashboard.

Versioning



Each solution team will maintain their own branch of code. You have all the freedom to modify/add files to the platform and the TestBed Assemblies of your team, and you will strictly not be making any change to the AutomationStart, MWPlatform and the ProductTestBed. You can use the tests though in ProductTestBed by marking a test case or a product area to be of Product Team as we discussed earlier, also you can reference MWPlatform and use all the methods written in the library but if any modifications are needed, then you must write extension methods or custom utilities in your platform and use them in your tests.

This will help us to develop AutomationStart , MWPlatform, and ProductTestBed independently and thus deliver them to you without any collisions.

You can modify any of the xml files to suit your requirements in your own Test assembly.

If you find a bug in the framework or if you need any feature which could be integrated in Optimus, then you can discuss the issue with us and we will do the needful.

API Testing

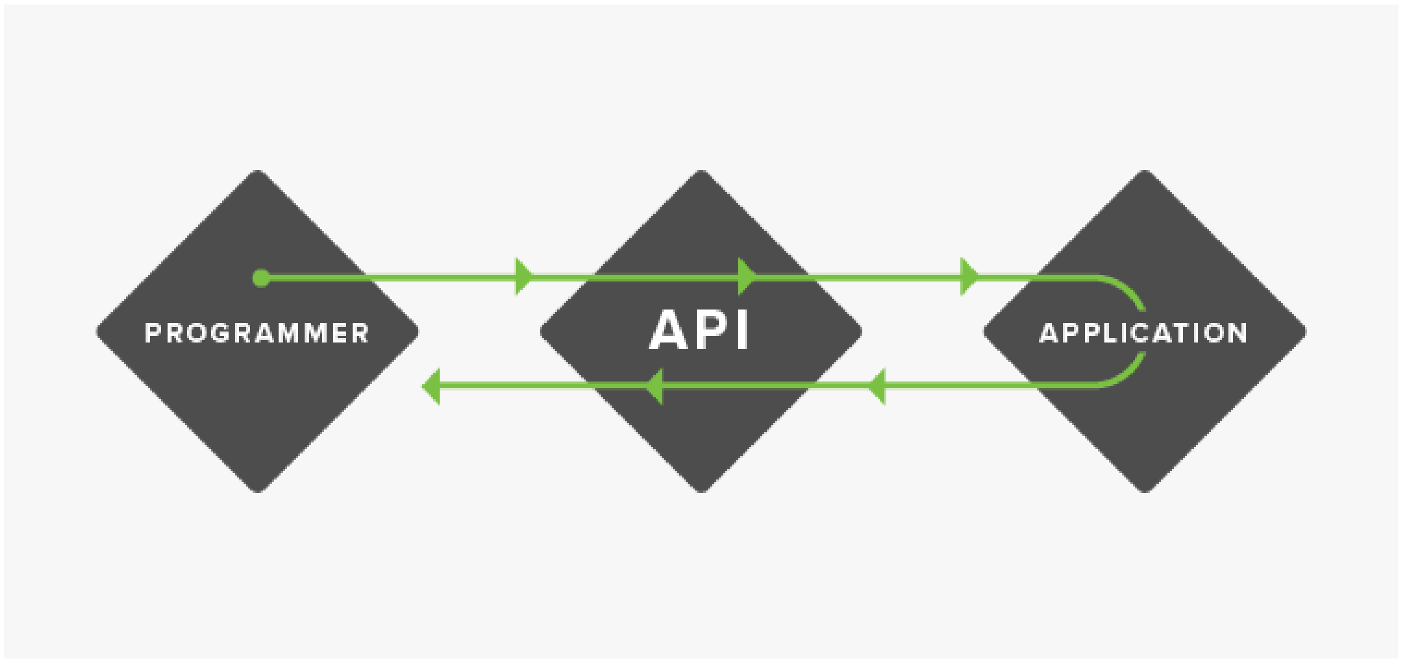
What is an API?

“API is a precise specification written by providers of a service that programmers must follow when using that service,” he says. “It describes what functionality is available, how it must be used and what formats it will accept as input or return as output. In recent years, the term API colloquially is used to describe both the specification *and* service itself, e.g., the Facebook Graph API.”

API Analogy

Every time you want to access a set of data from an application, you have to call the API. But there is only a certain amount of data the application will let you access, so you have to communicate to the operator in a very specific language—a language unique to each application.

To help visualize this concept, imagine an API as the middleman between a programmer and an application. This middleman accepts requests and, if that request is allowed, returns the data. The middleman also informs programmers about everything they can request, exactly how to ask for it and how to receive it.



Idea of API Test Cases –

API testing is a type of software testing that involves testing application programming interfaces (APIs) directly and as part of integration testing to determine if they meet expectations for functionality, reliability, performance, and security. Since APIs lack a GUI, API testing is performed at the message layer. API testing is now considered critical for automating testing because APIs now serve as the primary interface to application logic and because GUI tests are difficult to maintain with the short release cycles and frequent changes commonly used with Agile software development and DevOps.

API Testing Overview

API testing involves testing APIs directly (in isolation) and as part of the end-to-end transactions exercised during integration testing. Beyond RESTful APIs, these transactions include multiple types of endpoints such as web services, ESBs, databases, mainframes, web UIs, and ERPs. API testing is performed on APIs that the development team produces as well as APIs that the team consumes within their application (including third-party APIs).

API testing is used to determine whether APIs return the correct response (in the expected format) for a broad range of feasible requests, react properly to edge cases such as failures and unexpected/extreme inputs, deliver responses in an acceptable amount of time, and respond securely to potential security attacks. Service virtualization is used in conjunction with API testing to isolate the services under test as well as expand test environment access by simulating APIs/services that are not accessible for testing.

API testing commonly includes testing REST APIs or SOAP web services with JSON or XML message payloads being sent over HTTP, HTTPS, JMS, and MQ. It can also include message formats such as SWIFT, FIX, EDI and similar fixed-length formats, CSV, ISO 8583 and Protocol Buffers being sent over transports/protocols such as TCP/IP, ISO 8583, MQTT, FIX, RMI, SMTP, TIBCO Rendezvous, and FIX.

API’s in Masterworks - Getting Started

Audience

This document comprises the list of APIs used to communicate with **Aurigo Masterworks®** web application. It helps developers to understand the characteristics and functions of APIs.

Introduction

Masterworks Application Programming Interface (API) is a set of rules and specifications that the Masterworks web application can follow to communicate or interface with external software applications using web services.

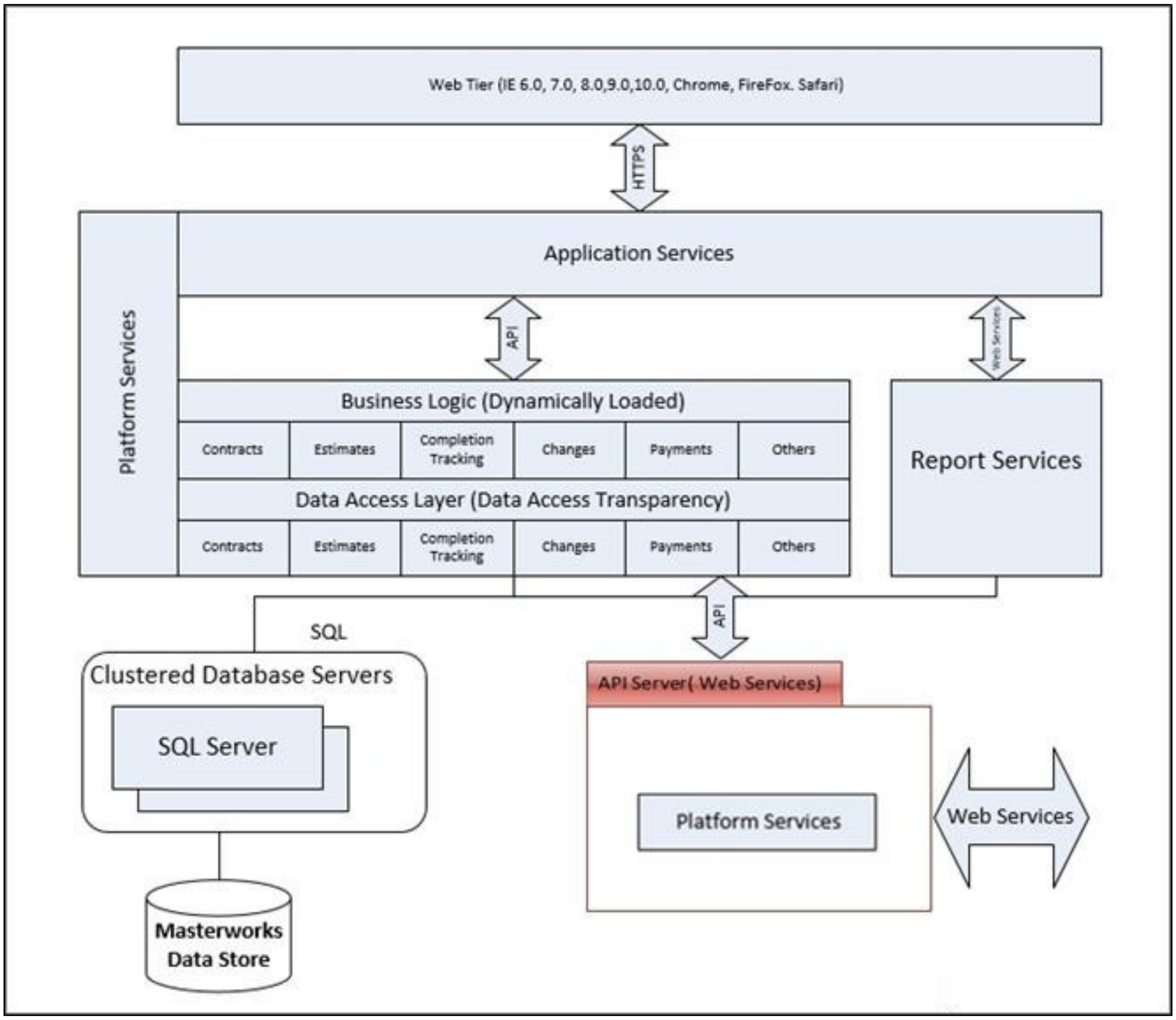


FIGURE 17: MASTERWORKS LOGICAL ARCHITECTURE REPRESENTATION

Masterworks utilizes web service based APIs to integrate seamlessly with existing IT infrastructure and can be used by any application; APIs use industry-standard protocols such as SOAP, XML, and HTTP.

Masterworks APIs are completely RESTful. They are authenticated through Hypertext Transfer Protocol (HTTP) request message headers and the response is in JavaScript Object Notation (JSON) format.

Classification

For ease of understanding, Masterworks APIs are classified as follows:

- Authentication APIs
- Generic APIs
- Document Management APIs
- Workflow Related APIs

The following sections describe each API, its functions and parameters. Examples are stated wherever necessary.

Authentication APIs

Login

Description

To use the Masterworks web application, users will have to authenticate by submitting a login request using a username and password. This method returns the confirmation token that is required for repeated requests.

Function

Method	Data Type	Verb	Request Body
[Base_URI]/api/Login	JSON/FromBody	POST	Content-Type: application/x-www-form-urlencoded

Parameters

Input	Output

{
"username":"username",
"password":"Passowrd",
"c":"nc",
"role":""
}
	{
	"ErrorCode":"0",
	"ErrorMessage":"Error Message Code: 0",
	"Token":"77f658f5-e9bc-4999-962c-1cdb74dadb51",
	"RoleName":"Administrator",
	"FirstName":"Joe",
	"LastName":"Tam",
	"MiddleName":"",
	"UID":"31",
	"UserId":"Joe",
	"RoleArray":{"1":"Administrator"}
	}

Logout

Description

Helps the user to exit the application and deletes the confirmation token that was obtained during login.

Function

Method	Data Type	Verb	Request Body
[Base_URI]/api/Logout		POST	Accept: application/json, text/javascript, */*; q=0.01 **Authorization-Token: [[Auth Token]]** Content-Type: application/x-www-form-urlencoded

Parameters

Input	Output

	{
	"ErrorCode":"0",
	"ErrorMessage":"Invalid Username/Password",
	"Token":null,
	"RoleName":null
	}

Generic APIs

Generic APIs are used to create , update, delete, and read data records in various functional modules.

GetProjectDetails

Description

Retrieves the list of all the projects that the current user is invited to.

Function

Method	Data Type	Verb	Request Body
api/ Module/GetProjectDetails		GET	Accept: application/json, text/javascript, */*; q=0.01 **Authorization-Token: [[Auth Token]]** Content-Type: application/x-www-form-urlencoded

Parameters

Input	Output
	`{ "Rowdata": [{ "ProjectId": 531, "ProjectName": "PRK2704", "ProjectCode": "PRK2704", "DateCreated": "2016-04-27T00:00:00", "StatusName": "Planning", "Owner": "Rakesh", "StartDate": "2016-04-27T00:00:00", "EndDate": "2016-04-27T00:00:00", "StatusId": 1, "IsActive": true, "TemplateId": 0, "AUR_ModifiedOn": "2016-04-27T14:08:41.633", "CreateDate": "2016-04-27T00:00:00", "TotalScore": null, "DISPLAYTEXT": "Active", "ProjectMode": "Live", "RowNum": 1, "PageCount": 0, "NoRecMsgStyle": null, "DetailsData": "{\"XPROJCT\":{\"AUR_ModifiedBy\":\"\",\"AUR_ModifiedOn\":\"4/27/2016 2:08:41 PM\",\"ProjectID\":\"531\",\"ProjectCode\":\"PRK2704\",\"ProjectName\":\"PRK2704\",\"Description\":\"\",\"CreateDate\":\"4/27/2016 12:00:00 AM\",\"Owner\":\"Rakesh\",\"StartDate\":\"4/27/2016 12:00:00 AM\",\"EndDate\":\"4/27/2016 12:00:00 AM\",\"StatusId\":\"1\",\"IsActive\":\"True\",\"IsPublish\":\"True\",\"ProjectModules\":{\"Row\":[{\"ID\":\"7685\",\"ProjectId\":\"531\",\"ModuleId\":\"BDGMGMT\",\"PlatformKey\":\"W_W_WEB\"},{\"ID\":\"7686\",\"ProjectId\":\"531\",\"ModuleId\":\"CHKPRJL\",\"PlatformKey\":\"W_W_WEB\"},{\"ID\":\"7687\",\"ProjectId\":\"531\",\"ModuleId\":\"CONTMGT\",\"PlatformKey\":\"W_W_WEB\"},{\"ID\":\"7688\",\"ProjectId\":\"531\",\"ModuleId\":\"ESTMATE\",\"PlatformKey\":\"W_W_WEB\"},{\"ID\":\"7689\",\"ProjectId\":\"531\",\"ModuleId\":\"FNDPRJT\",\"PlatformKey\":\"W_W_WEB\"},{`

Input	Output
	\"ID\":\"7690\",\"ProjectId\":\"531\",\"ModuleId\":\"FNDPTRN\",\"PlatformKey\":\"W_W_WEB\"},{\"ID\":\"7691\",\"ProjectId\":\"531\",\"ModuleId\":\"FNDRULE\",\"PlatformKey\":\"W_W_WEB\"},{\"ID\":\"7692\",\"ProjectId\":\"531\",\"ModuleId\":\"PRJSCNG\",\"PlatformKey\":\"W_W_WEB\"},{\"ID\":\"7693\",\"ProjectId\":\"531\",\"ModuleId\":\"SAEEFRM\",\"PlatformKey\":\"W_W_WEB\"},{\"ID\":\"7694\",\"ProjectId\":\"531\",\"ModuleId\":\"SUBMTAL\",\"PlatformKey\":\"W_W_WEB\"},{\"ID\":\"7695\",\"ProjectId\":\"531\",\"ModuleId\":\"Test111\",\"PlatformKey\":\"W_W_WEB\"},{\"ID\":\"7696\",\"ProjectId\":\"531\",\"ModuleId\":\"Venu@22\",\"PlatformKey\":\"W_W_WEB\"},{\"ID\":\"7697\",\"ProjectId\":\"531\",\"ModuleId\":\"Venu222\",\"PlatformKey\":\"W_W_WEB\"}]},\"Contracts\":{\"Row\":{\"ID\":\"483\",\"Code\":\"PRK2704\",\"Name\":\"PRK2704\",\"ProjectID\":\"531\",\"Desc\":\"\",\"Days\":\"0\",\"StartDt\":\"4/27/2016 12:00:00 AM\",\"ClosureDt\":\"4/27/2016 12:00:00 AM\",\"StatusID\":\"1\",\"MeasSysID\":\"1\",\"Locked\":\"Y\",\"OrigDays\":\"1\",\"IsDeleted\":\"N\",\"CreatedBy\":\"1\",\"CreatedOn\":\"4/27/2016 2:08:41 PM\",\"OrigContAmt\":\"2.0000\",\"OrigContBaseCost\":\"2.00000000\",\"GroupType\":\"L\",\"EstimateID\":\"\",\"BidderID\":\"\",\"ERPCurrency\":\"$\"}}}}" }], "InstanceIDS": [{ "InstanceId": "531" }] }

GetProjectModules

Description

Retrieves the list of metadata information of offline enabled project modules. For example, in the **Request for Information** (RFI) module, this method generates RFI MetaData.

Function

Method	Data Type	Verb	Request Body
[Base_URI]/api/ Module/GetProjectModules? moduleId=xprojct&projectId= [[ProjectId]]&contractId=[[Co ntractId]]&modulesTemplate =&lastSyncedAt=[[datetime or empty]]		GET	Accept: application/json, text/javascript, */*; q=0.01 **Authorization-Token: [[Auth Token]]** Content-Type: application/x-www-form-urlencoded

Parameters

Input (JSON body)	Output

{
"sort":[string],
"filter":[string],
"pid":"[string]",
"parentid":"[string]",
"take":int,"skip":int,
"page":int,"pageSize":int,
"additionalParameters"
{
"ModuleID":"[string]",
"TableName":"[string]",
"PrimaryKeyName":"[string]",
"WhereCond":"[string]"
},
"requestParams":"
{
"pid":"[string]",
"parentid":"[string]","context":"[string]",
"instanceid":"0"
}
}
	ListDetails : [
	{
	"ProjectId": 531,
	"ProjectName": "PRK2704",
	"ProjectCode": "PRK2704",
	"DateCreated": "/Date(1461740400000)/",
	"StatusName": "Planning",
	"Owner": "Rakesh",
	"StartDate": "/Date(1461740400000)/",
	"EndDate": "/Date(1461740400000)/",
	"StatusId": 1,
	"IsActive": true,
	"TemplateId": 0,
	"AUR_ModifiedOn": "/Date(1461791321633)/",
	"CreateDate": "/Date(1461740400000)/",
	"TotalScore": null,
	"DISPLAYTEXT": "Active",
	"ProjectMode": "Live",
	"RowNum": 1,
	"PageCount": 0,
	"NoRecMsgStyle": null
	}
	], DrawerTemplate : {
	"CONMODS": {
	"ProjectID": 531,
	"ID": "",
	"Modules": {
	"Row": [
	{
	"ID": null,
	"ProjectID": "531",
	"ContractID": "0",
	"ModuleName": "Daily Progress Report",
	"ModuleID": "CONTDWR",

Input (JSON body)	Output

```
          "URL": null
        },
        {
          "ID": null,
          "ProjectID": "531",
          "ContractID": "0",
          "ModuleName": "Documents",
          "ModuleID": "DOCMGMT",
          "URL": null
        }
      ]
    },
    "ModulesPermission": [
      {
        "ModuleId": "CONTDWR",
        "Permission": [
          "C",
          "D",
          "E",
          "V"
        ]
      },
      {
        "ModuleId": "DOCMGMT",
        "Permission": [

        ]
      }
    ]
  }
} , ListTemplates{
  "CONMODS": {
    "ProjectID": 531,
    "ID": "",
    "Modules": {
      "Row": [
        {
          "ID": null,
          "ProjectID": "531",
          "ContractID": "0",
          "ModuleName": "Daily Progress Report",
          "ModuleID": "CONTDWR",
          "URL": null
        },
        {
          "ID": null,
          "ProjectID": "531",
          "ContractID": "0",
          "ModuleName": "Documents",
          "ModuleID": "DOCMGMT",
```

Input (JSON body)	Output
	```
	        "URL": null
	       }
	      ]
	    },
	    "ModulesPermission": [
	      {
	        "ModuleId": "CONTDWR",
	        "Permission": [
	          "C",
	          "D",
	          "E",
	          "V"
	        ]
	      },
	      {
	        "ModuleId": "DOCMGMT",
	        "Permission": [

	        ]
	      }
	    ]
	  }
	}, FormTemplates  : {
	  "CONMODS": {
	    "ProjectID": 531,
	    "ID": "",
	    "Modules": {
	      "Row": [
	        {
	          "ID": null,
	          "ProjectID": "531",
	          "ContractID": "0",
	          "ModuleName": "Daily Progress Report",
	          "ModuleID": "CONTDWR",
	          "URL": null
	        },
	        {
	          "ID": null,
	          "ProjectID": "531",
	          "ContractID": "0",
	          "ModuleName": "Documents",
	          "ModuleID": "DOCMGMT",
	          "URL": null
	        }
	      ]
	    },
	    "ModulesPermission": [
	      {
	        "ModuleId": "CONTDWR",
	        "Permission": [
	```

Input (JSON body)	Output

	"C",
	"D",
	"E",
	"V"
	]
	},
	{
	"ModuleId": "DOCMGMT",
	"Permission": [
	
	]
	}
	]
	}
	}

GetAllInstanceDetails

Description

Retrieves details of the specified instance of the specified module. For example, details of a form can be retrieved using the module ID and instance ID of the specified instance.

Function

Method	Data Type	Verb	Request Body
[Base_URI]/api/Module/GetAllInstanceDetails		GET	Accept: application/json, text/javascript, */*; q=0.01 **Authorization-Token: [[Auth Token]]** Content-Type: application/x-www-form-urlencoded

Parameters

Input	Output

moduleId=[[moduleid]]
&
pid=[[pid]]
&
parentid=[[parentid]]
&
jsonParameters=[[empty]]
&
lastSyncedAt=[[datetimestamp or empty]]
	Generate instance details for a particular instance.

CreateOrUpdateInstance

Description

Adds an instance to the specified module or modifies an existing instance.

Function

Method	Data Type	Verb	Request Body
[Base_URI]/api/Module/CreateOrUpdateInstance		POST	Accept: application/json, text/javascript, */*; q=0.01 **Authorization-Token: [[Auth Token]]** Content-Type: application/x-www-form-urlencoded

Parameters

Input	Output

moduleId=[ModuleId]
&jsonReqParams
=
{
"pid":"string",
"parentid":"string",
"context":"string",
"instanceid":"string",
"offid":"string",
"datamode":"[string]on"
}
/ FormBody Data for the instance
	Add an instance in the respective module.

DeleteInstance

Description

Deletes the specified instance of the specified module. Instances to be deleted are identified using the combination of the module ID and the instance ID.

Function

Method	**Data Type**	**Verb**	**Request Body**
[Base_URI]/api/Module/ DeleteInstance		POST	Accept: application/json, text/javascript, */*; q=0.01 **Authorization-Token: [[Auth Token]]** Content-Type: application/x-www-form-urlencoded

Parameters

Input	**Output**
moduleId=[ModuleId] & ids="string" / FormBody Data for the instance	Delete an instance from the respective module

Document Management APIs

GetFilesAndFolder

Description

Retrieves all the files and folders from the specified project. This helps the user to perform further actions such as create, delete, download or upload files, and create and delete folders.

Function

Method	Data Type	Verb	Request Body
[Base_URI]/api/Document/GetFilesAndFolder	moduleId:string&id=string[ProjectId]&jsonParameters=string[{}]	GET	Accept: application/json, text/javascript, */*; q=0.01 **Authorization-Token: [[Auth Token]]** Content-Type: application/x-www-form-urlencoded

Parameters

Input	Output

moduleId=conmods
&
id=[ProjectId]
&
jsonParameters={}
	{
	"Result":
	[
	{
	"ID":"int",
	"Name":"String",
	"Type":"String",
	"StorageID":"String",
	"LinkID":int,
	"HasSubFolders":bool,
	"ItemType":"String",
	"ParentID":int,
	"NoRecMsgStyle":String,
	"requestFolderId":"String"
	},
	{
	}
	],
	"Error":String,
	"Total":uint
	}

Upload

Description

Uploads a document to the specified folder in the specified project. A description can also be added to the uploaded document.

Function

Method	Data Type	Verb	Request Body
[Base_URI]/api/Document/ Upload	folderId:int projectId: int description:string	GET	Accept: application/json, text/javascript, */*; q=0.01 **Authorization-Token: [[Auth Token]]** Content-Type: application/x-www-form-urlencoded

Parameters

Input	Output
folderId=folderId & projectId=projectId & description=desc	

DeleteAttachment

Description

Deletes the specified files uploaded as attachments in a form. The attachment to be deleted is identified using the link ID - auto-generated unique identifier assigned to an attachment. Link ID can be obtained using the **GetFilesAndFolder** method.

Function

Method	Data Type	Verb	Request Body
[Base_URI]/api/Document/ DeleteAttachment	linkIds : string	GET	Accept: application/json, text/javascript, */*; q=0.01 **Authorization-Token: [[Auth Token]]** Content-Type: application/x-www-form-urlencoded

Parameters

Input	Output
linkIds="linkIds"	

Download

Description

Downloads the specified file. The file to be downloaded is identified using the doc ID.

Function

Method	Data Type	Verb	Request Body
[Base_URI]/api/Document/Download	docID:int		Accept: application/json, text/javascript, */*; q=0.01 **Authorization-Token: [[Auth Token]]** Content-Type: application/x-www-form-urlencoded

Parameters

Input	Output
docID=docID	

DeleteFolder

Description

Deletes the specified folder. The folder to be deleted is identified using the folder ID.

Function

Method	**Data Type**	**Verb**	**Request Body**
[Base_URI]/api/Document/DeleteFolder	folderId:int		Accept: application/json, text/javascript, */*; q=0.01 **Authorization-Token: [[Auth Token]]** Content-Type: application/x-www-form-urlencoded

Parameters

Input	**Output**
folderId=folderId	

DeleteDocument

Description

Deletes the specified document. The document to be deleted is identified using the link ID, project ID, and parent ID.

Function

Method	Data Type	Verb	Request Body
[Base_URI]/api/Document/ DeleteDocument	linkId:int pid:int parentId:int		

Parameters

Input	Output
linkId="linkId" & pid=projectid & parentId=parent id	

Workflow Related APIs

Workflow

Description

Retrieves the workflow actions for the specified instance that is pending on the user.

Function

Method	Data Type	Verb	Request Body
[Base_URI]/api/ Workflow	moduleId:string forminstanceid:int jsonParameters:string	GET	Accept: application/json, text/javascript, */*; q=0.01 **Authorization-Token: [[Auth Token]]** Content-Type: application/x-www-form-urlencoded

Parameters

Input	Output

moduleId=ModuleId
&
forminstanceid=InstanceID
&
jsonParameters=
{
"pid":ProjectId,
"parentid":ParentId
}
	[
	{
	"Guid":"WorkFlowGuid",
	"ActionId":"ActionID",
	"ActionName":"Action Name"
	}
	]

Workflow History

Description

Retrieves the workflow history of the specified instance. Workflow history provides information on the status of the instance, date created, user, notes, workflow action, due date override, user who has performed the action and the action date.

Function

Method	Data Type	Verb	Request Body
[Base_URI]/api/ Workflow	moduleId:string forminstanceid:int workflowInstanceGuid: string jsonParameters:string	GET	Accept: application/json, text/javascript, */*; q=0.01 **Authorization-Token: [[Auth Token]]** Content-Type: application/x-www-form-urlencoded

Parameters

Input	Output

moduleId=ModuleId
 &
 forminstanceid=InstanceID
 &
 workflowInstanceGuid=WORKFLOWGUID
 &
 jsonParameters=
 {
 "pid":"pid",
 "parentid":"parentid",
 "formid":"formid",
 "forminstanceid":"forminstanceid",
 "workflowinstanceguid":"workflowinstanceguid",
 "actionbar":"true",
 "instanceid":"0"
 }
	[
	 {
	 "ID":INT,
	 "Status":"string",
	 "PendingOn":"string",
	 "DateCreated":"Date",
	 "User":"string",
	 "ActionNotes":"string",
	 "Action":"string",
	 "DueDateOverride":null,
	 "ActionUser":"string",
	 "ActionDate":"Date"
	 }
	]

PerformAction

Description

Performs the specified workflow action on the specified instance of the module.

Function

Method	Data Type	Verb	Request Body
[Base_URI]/api/Workflow /PerformAction	moduleId:string forminstanceid: int actionId: string actionName:string	POST	Accept: application/json, text/javascript, */*; q=0.01 **<u>Authorization-Token: [[Auth Token]]</u>** Content-Type: application/x-www-form-urlencoded

Parameters

Input	Output

moduleId=Module ID
&
formInstanceId= Instance ID
&
actionId=Action ID
&
actionName=Action Name
	{
	"Result":
	{
	"Success":"Message"
	}
	}

How can I Write API Test Cases using Optimus?

Making API Calls

We have added a new project in Optimus i.e. AutomationHelpers. It has a class called APIClient.cs that will help you to make all the API requests. It is present in APIHelper.cs file

MakeRequest function in APIClient.cs file will make a request to the specified API and return the response back to you.

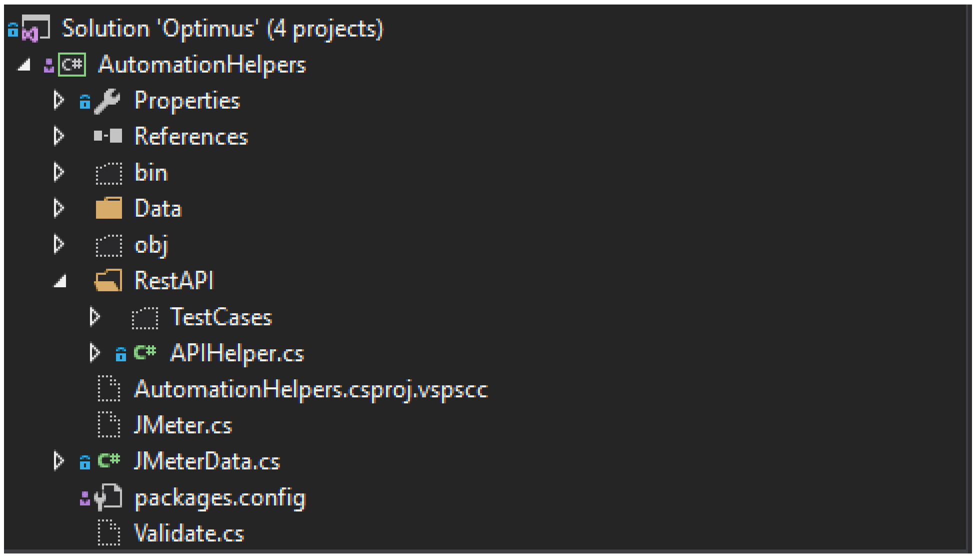


Figure 18

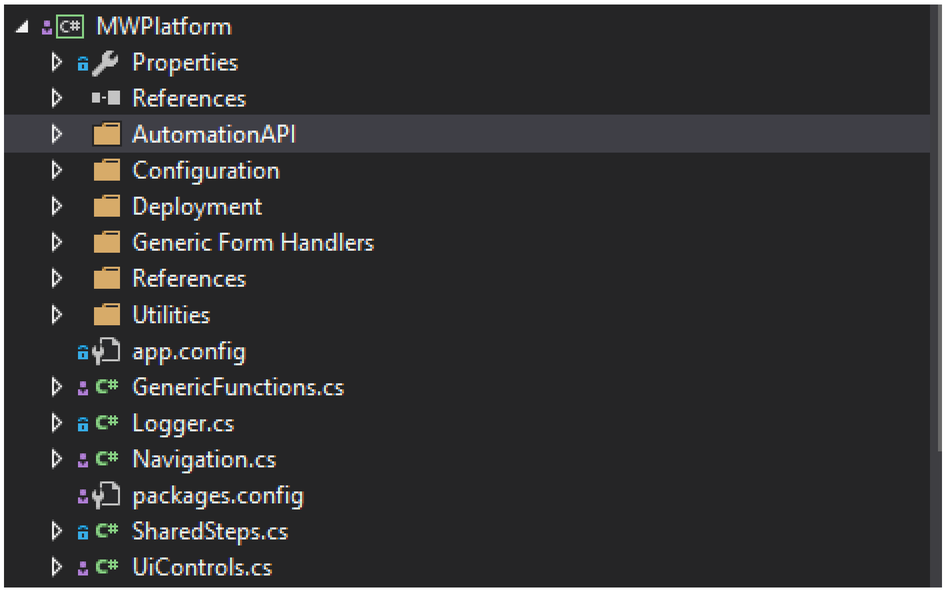
The Flow

We have to first make a call to the Login API and get the authentication token. We will be using this token to make request to any other API.

All the API's in Masterworks have their separate classes. So while making a request to an API, all you have to do is create an instance of that class and make a call to the MakeRequest function along with the authentication token and the resource path for that API(present in the API class itself) and get the response back.

Where are all the files related to API's present?

Its Present in the AutomationAPI folder inside the MWPlatform Project. All the API things are present under the namespace **AutomationAPI**.

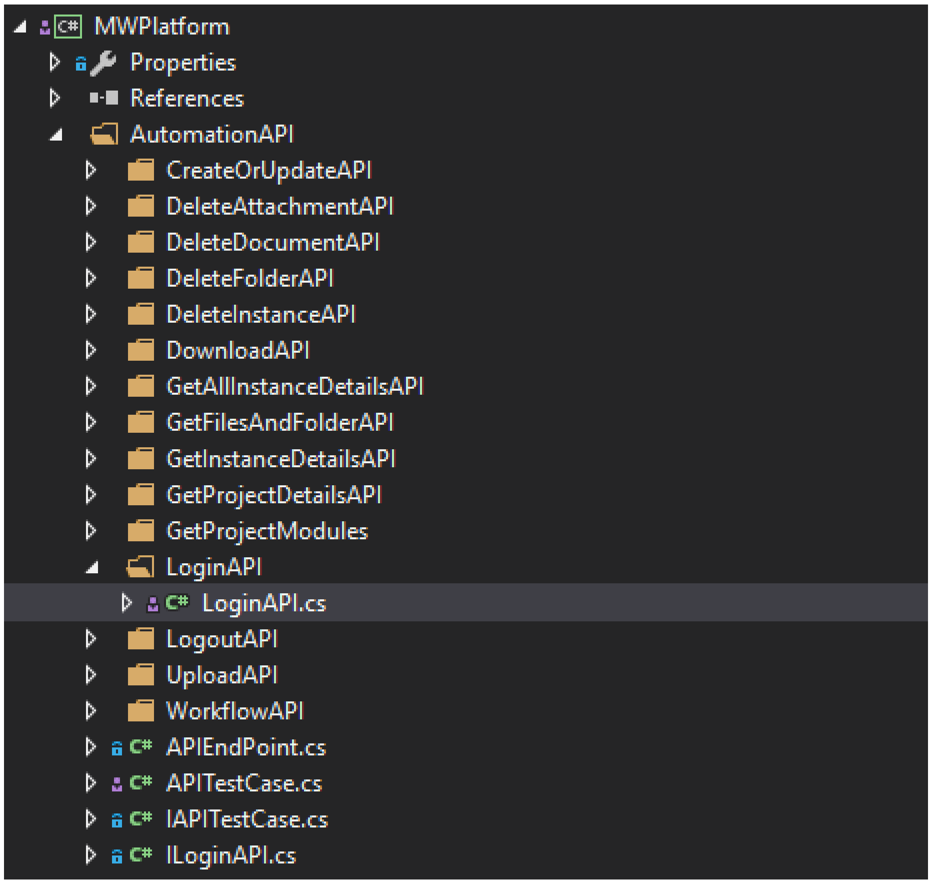


```
MWPlatform
    Properties
    References
    AutomationAPI
    Configuration
    Deployment
    Generic Form Handlers
    References
    Utilities
    app.config
    GenericFunctions.cs
    Logger.cs
    Navigation.cs
    packages.config
    SharedSteps.cs
    UiControls.cs
```

Figure 19

How to Login and Get the Authentication Token?

The is class named LoginAPI does the job for you. All you have to do is call the LoginAndGetToken function. This function will return you the authentication token and you can use this to make the subsequent requests.



```
MWPlatform
    Properties
    References
    AutomationAPI
        CreateOrUpdateAPI
        DeleteAttachmentAPI
        DeleteDocumentAPI
        DeleteFolderAPI
        DeleteInstanceAPI
        DownloadAPI
        GetAllInstanceDetailsAPI
        GetFilesAndFolderAPI
        GetInstanceDetailsAPI
        GetProjectDetailsAPI
        GetProjectModules
        LoginAPI
            LoginAPI.cs
        LogoutAPI
        UploadAPI
        WorkflowAPI
        APIEndPoint.cs
        APITestCase.cs
        IAPITestCase.cs
        ILoginAPI.cs
```

Figure 20

Writing API Test Cases

You don't have to worry about start logging and end logging for api test cases. It is handled at the executor level, but you will have to log any other information that is necessary for the test case.

Error handling is also taken care at the executor level, so NO TRY CATCH blocks in the test case. You have to only write your logic and let the framework take care of other things.

To write an API test case, just inherit your class from APITestCase which is present in AutomationAPI namespace. This will give you the authorization token and access to a lot of other useful properties like apiClient(used for making API requests), PID, Parent ID, Instance ID, Headers (content type and auth token will be set as a part of the initialization process). Using this, you only have to create an instance of your API class, make request and validate the response.

When you create an instance of an API class (e.g. CreateOrUpdateAPI), it expects a few parameters that it uses to construct the resource path for that API. The class then exposes the ResourcePath property that is used to make API calls.

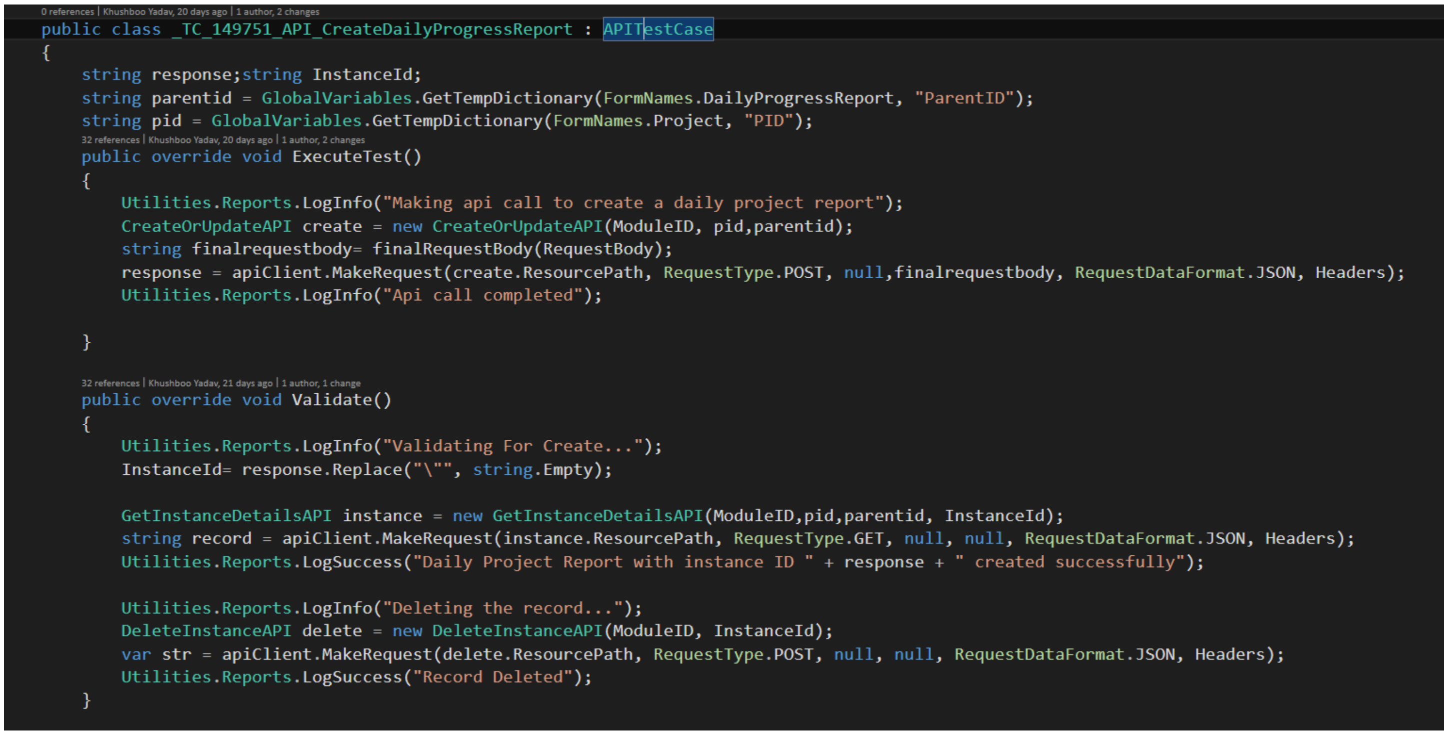
You will have to override two methods when you inherit from APITestCase class.

1. ExecuteTest – Write your logic to make a request to the API here
2. Validate – Write your validation logic here

In case you don't want the validate function, you can still write everything in the ExecuteTest function, and keep the Validate function empty.

The naming of the API Test class will follow the older conventions of naming a test class.

Here is snapshot of a sample API test case –



```
0 references | Khushboo Yadav, 20 days ago | 1 author, 2 changes
public class _TC_149751_API_CreateDailyProgressReport : APITestCase
{
    string response;string InstanceId;
    string parentid = GlobalVariables.GetTempDictionary(FormNames.DailyProgressReport, "ParentID");
    string pid = GlobalVariables.GetTempDictionary(FormNames.Project, "PID");
    32 references | Khushboo Yadav, 20 days ago | 1 author, 2 changes
    public override void ExecuteTest()
    {
        Utilities.Reports.LogInfo("Making api call to create a daily project report");
        CreateOrUpdateAPI create = new CreateOrUpdateAPI(ModuleID, pid,parentid);
        string finalrequestbody= finalRequestBody(RequestBody);
        response = apiClient.MakeRequest(create.ResourcePath, RequestType.POST, null,finalrequestbody, RequestDataFormat.JSON, Headers);
        Utilities.Reports.LogInfo("Api call completed");

    }

    32 references | Khushboo Yadav, 21 days ago | 1 author, 1 change
    public override void Validate()
    {
        Utilities.Reports.LogInfo("Validating For Create...");
        InstanceId= response.Replace("\"", string.Empty);

        GetInstanceDetailsAPI instance = new GetInstanceDetailsAPI(ModuleID,pid,parentid, InstanceId);
        string record = apiClient.MakeRequest(instance.ResourcePath, RequestType.GET, null, null, RequestDataFormat.JSON, Headers);
        Utilities.Reports.LogSuccess("Daily Project Report with instance ID " + response + " created successfully");

        Utilities.Reports.LogInfo("Deleting the record...");
        DeleteInstanceAPI delete = new DeleteInstanceAPI(ModuleID, InstanceId);
        var str = apiClient.MakeRequest(delete.ResourcePath, RequestType.POST, null, null, RequestDataFormat.JSON, Headers);
        Utilities.Reports.LogSuccess("Record Deleted");
    }
```

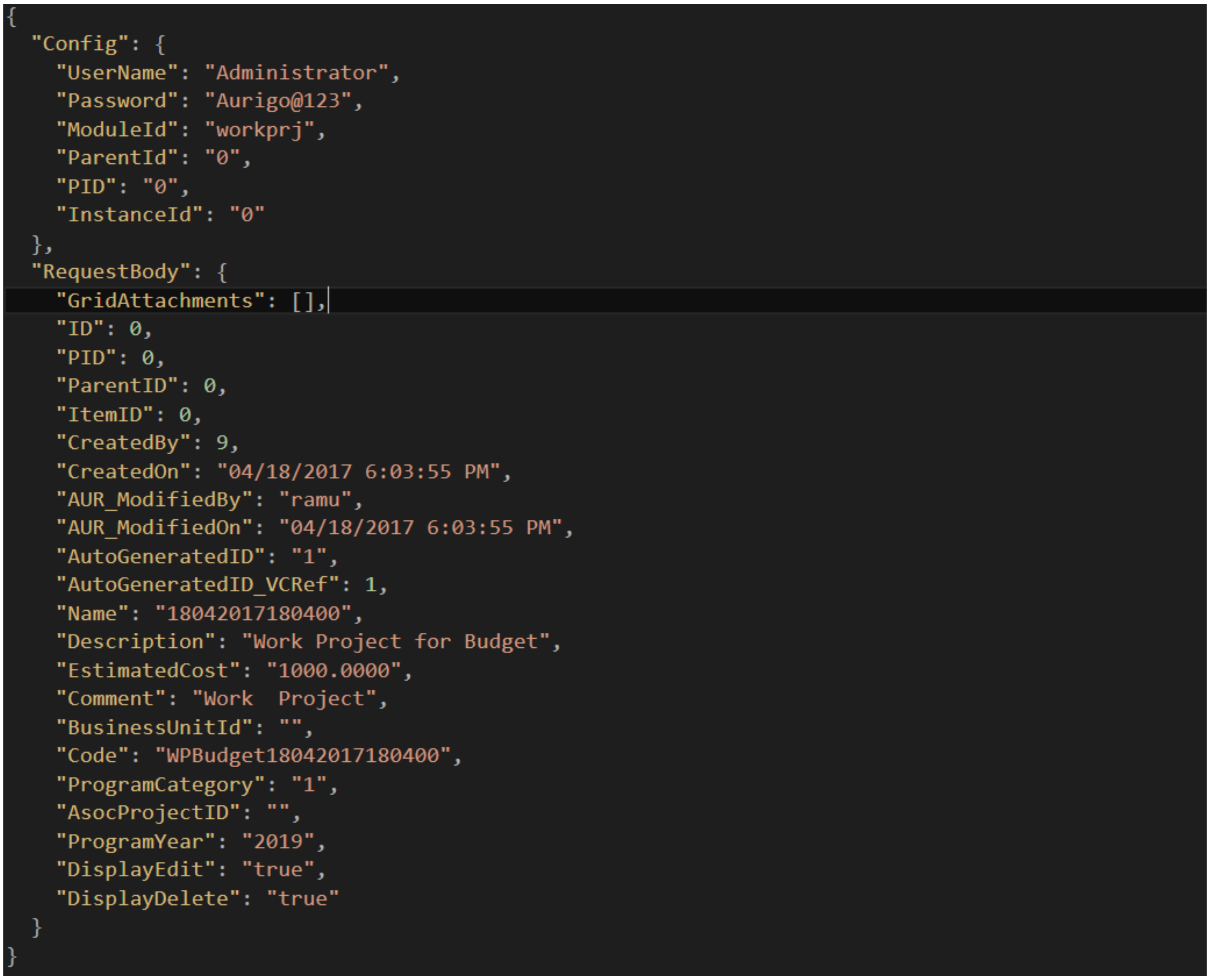
Figure 21

TestData for API Test Cases

Unlike the regular test cases, the API tests will have the test data in a json format. The test data files should be present in the test data folder itself. The name of the test data file will same as the name of the test case in lower case and will have a '.json' extension.

Here is a sample test data file for an API test case (figure 22).

It has 2 main elements, one is the config element that will hold all the information that will be required for executing the test case like username and password for login, pid, parentId, module id, etc. The other element is the Request body that will be sent as it is to the API request.



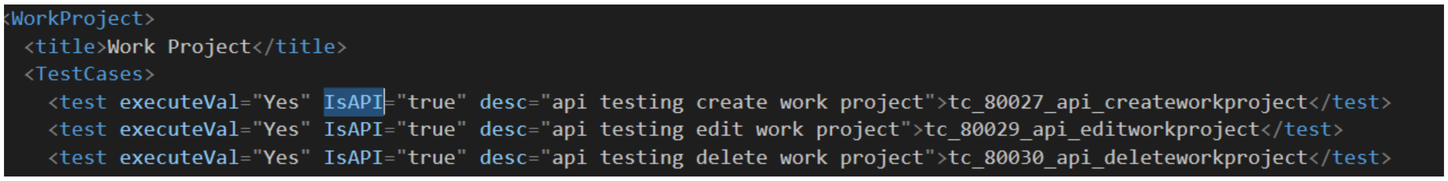
```
{
  "Config": {
    "UserName": "Administrator",
    "Password": "Aurigo@123",
    "ModuleId": "workprj",
    "ParentId": "0",
    "PID": "0",
    "InstanceId": "0"
  },
  "RequestBody": {
    "GridAttachments": [],
    "ID": 0,
    "PID": 0,
    "ParentID": 0,
    "ItemID": 0,
    "CreatedBy": 9,
    "CreatedOn": "04/18/2017 6:03:55 PM",
    "AUR_ModifiedBy": "ramu",
    "AUR_ModifiedOn": "04/18/2017 6:03:55 PM",
    "AutoGeneratedID": "1",
    "AutoGeneratedID_VCRef": 1,
    "Name": "18042017180400",
    "Description": "Work Project for Budget",
    "EstimatedCost": "1000.0000",
    "Comment": "Work  Project",
    "BusinessUnitId": "",
    "Code": "WPBudget18042017180400",
    "ProgramCategory": "1",
    "AsocProjectID": "",
    "ProgramYear": "2019",
    "DisplayEdit": "true",
    "DisplayDelete": "true"
  }
}
```

Figure 22

Executing an API Test case

Everything remains the same except for one thing –

You have to specify that a test case is an API Test by providing the ‘IsAPI’ attribute as true in the test settings file.



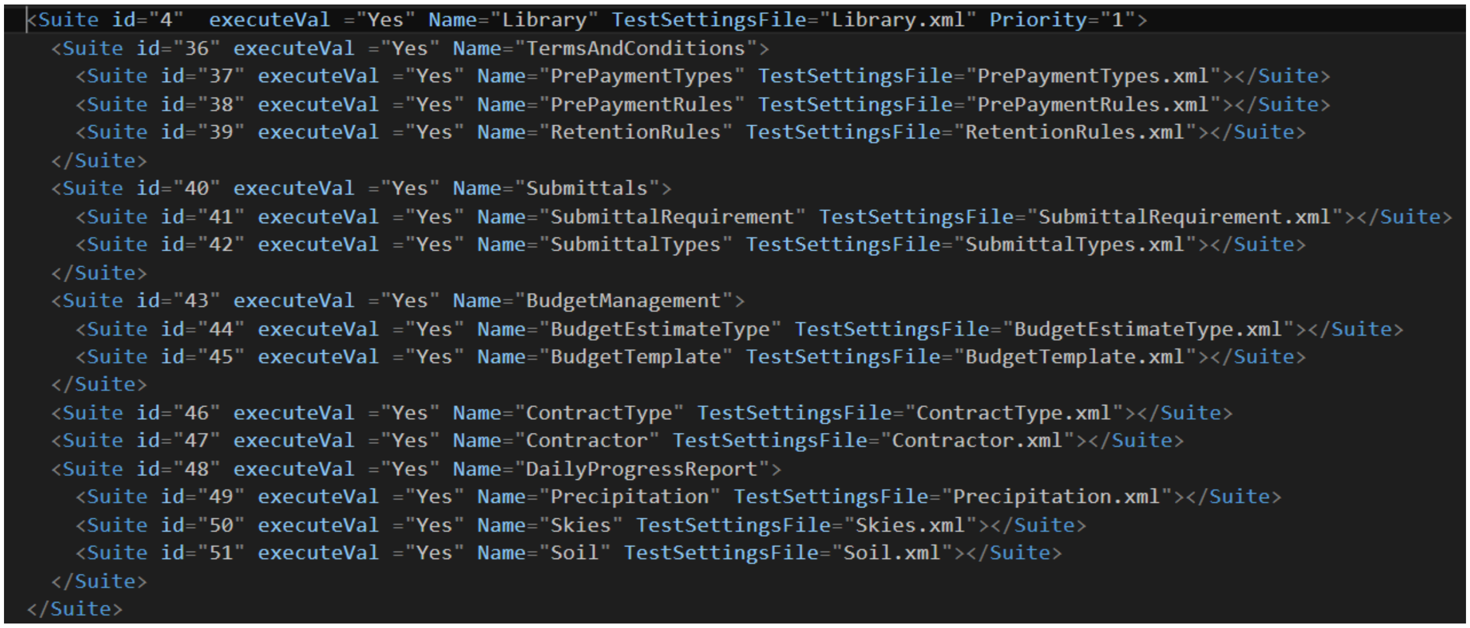
```
<WorkProject>
  <title>Work Project</title>
  <TestCases>
    <test executeVal="Yes" IsAPI="true" desc="api testing create work project">tc_80027_api_createworkproject</test>
    <test executeVal="Yes" IsAPI="true" desc="api testing edit work project">tc_80029_api_editworkproject</test>
    <test executeVal="Yes" IsAPI="true" desc="api testing delete work project">tc_80030_api_deleteworkproject</test>
```

Figure 23

The description that you specify here will be used for reporting

Breaking down suites into sub-suites

We only have one node for each suite in product areas xml file. But with the latest release, we can break down suites into a number of sub suites here is a sample Library suite that is broken down into sub suites –



```
<Suite id="4"  executeVal ="Yes" Name="Library" TestSettingsFile="Library.xml" Priority="1">
  <Suite id="36" executeVal ="Yes" Name="TermsAndConditions">
    <Suite id="37" executeVal ="Yes" Name="PrePaymentTypes" TestSettingsFile="PrePaymentTypes.xml"></Suite>
    <Suite id="38" executeVal ="Yes" Name="PrePaymentRules" TestSettingsFile="PrePaymentRules.xml"></Suite>
    <Suite id="39" executeVal ="Yes" Name="RetentionRules" TestSettingsFile="RetentionRules.xml"></Suite>
  </Suite>
  <Suite id="40" executeVal ="Yes" Name="Submittals">
    <Suite id="41" executeVal ="Yes" Name="SubmittalRequirement" TestSettingsFile="SubmittalRequirement.xml"></Suite>
    <Suite id="42" executeVal ="Yes" Name="SubmittalTypes" TestSettingsFile="SubmittalTypes.xml"></Suite>
  </Suite>
  <Suite id="43" executeVal ="Yes" Name="BudgetManagement">
    <Suite id="44" executeVal ="Yes" Name="BudgetEstimateType" TestSettingsFile="BudgetEstimateType.xml"></Suite>
    <Suite id="45" executeVal ="Yes" Name="BudgetTemplate" TestSettingsFile="BudgetTemplate.xml"></Suite>
  </Suite>
  <Suite id="46" executeVal ="Yes" Name="ContractType" TestSettingsFile="ContractType.xml"></Suite>
  <Suite id="47" executeVal ="Yes" Name="Contractor" TestSettingsFile="Contractor.xml"></Suite>
  <Suite id="48" executeVal ="Yes" Name="DailyProgressReport">
    <Suite id="49" executeVal ="Yes" Name="Precipitation" TestSettingsFile="Precipitation.xml"></Suite>
    <Suite id="50" executeVal ="Yes" Name="Skies" TestSettingsFile="Skies.xml"></Suite>
    <Suite id="51" executeVal ="Yes" Name="Soil" TestSettingsFile="Soil.xml"></Suite>
  </Suite>
</Suite>
```

Figure 24

Now each suite has **executeVal** and it works in the same fashion as for the test cases, i.e. a product area will not be executed, if the executeVal is set to no.

The name attribute of the suite will be used for reporting purposes as well as for defining the folder structure of the test settings file inside the TestSettings folder. So now we don't have a flat test settings folder. We have nested folders according to the hierarchy specified in the ProductAreas.xml file. For instance, PrePaymentTypes is a sub suite of TermsAndConditions which in turn is a sub suite of Library. So, the test settings file for PrepayemtnTypes will be found in Library -> TermsAndConditions -> PrePaymentTypes.xml. Here is a snapshot of the scenario.

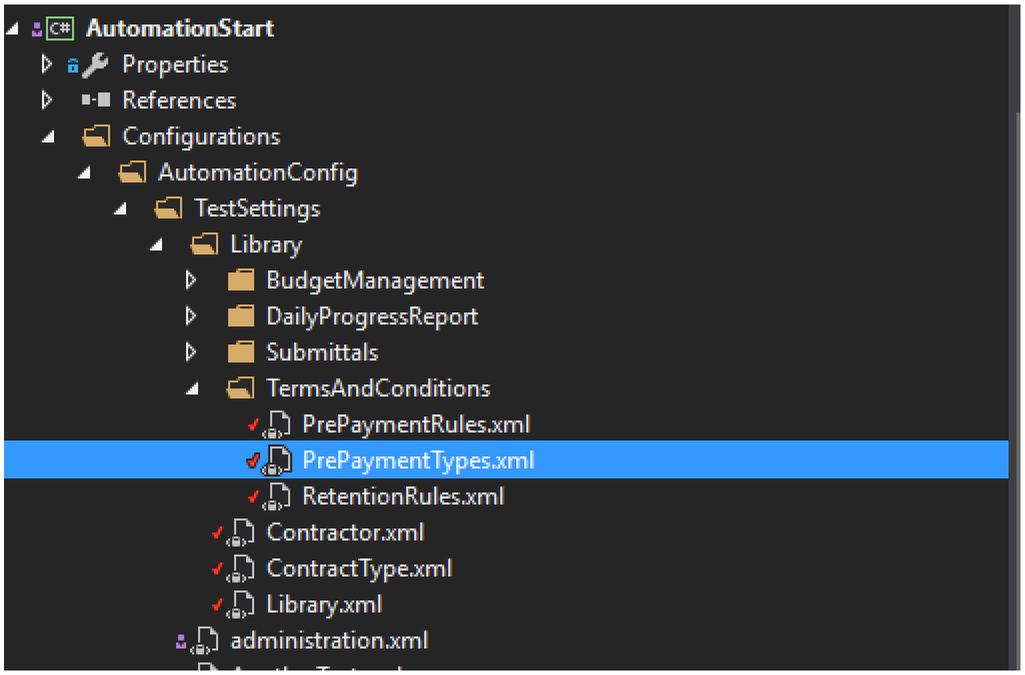


Figure 25

The TestSettingsFile attribute specifies the test settings file for that suite.

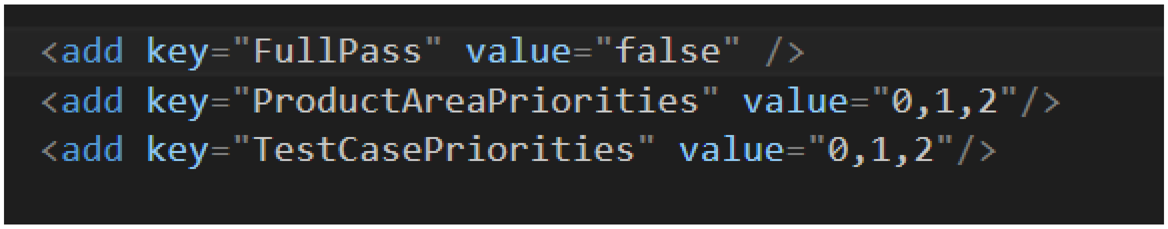
A suite may or may not have a test settings file. If the suite doesn't have a test settings file then we don't have to specify the TestSettingsFile attribute, but the Name attribute is compulsory as it will be used for reporting and finding the location of the sub suites' test setting files.

We can choose to run a sub suite by passing the id of that suite via the command line arguments. If that suite has further sub suites, then all of them will be considered for the current automation run(only areas that have executeVal as yes).

Parent Suite's executeVal will override all the children suites executeVal attribute. So if Library has executeVal as no and we want to execute Library, all the sub suites of library will be ignored for execution.

Priorities of ProductAreas and Test Cases

We can specify the priorities of product areas and test cases that we want to execute in App.config file.



```
<add key="FullPass" value="false" />
<add key="ProductAreaPriorities" value="0,1,2"/>
<add key="TestCasePriorities" value="0,1,2"/>
```

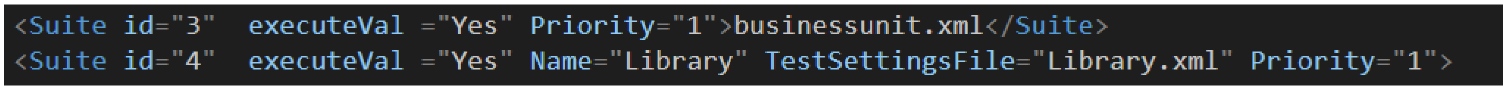
Figure 26

Priorities have 3 values
0 – High
1 – Medium
2 – Low

Multiple priorities can be specified here separated by comma(,) , as this only says the priority of product areas or test cases to be picked up for automation.

We will be picking up only those product areas and test cases for execution that have priorities defined in App.config file.

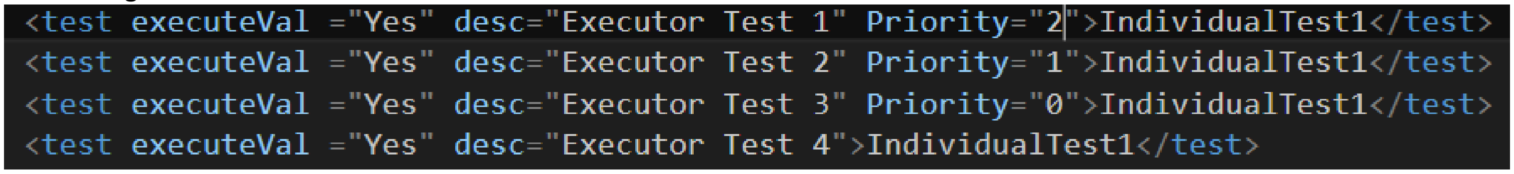
We can specify the priority of a product area or a testcase by adding a Priority attribute to the node in their xml files.
Product Area -



```
<Suite id="3"  executeVal ="Yes" Priority="1">businessunit.xml</Suite>
<Suite id="4"  executeVal ="Yes" Name="Library" TestSettingsFile="Library.xml" Priority="1">
```

Figure 27

TestSettings-



```
<test executeVal ="Yes" desc="Executor Test 1" Priority="2">IndividualTest1</test>
<test executeVal ="Yes" desc="Executor Test 2" Priority="1">IndividualTest1</test>
<test executeVal ="Yes" desc="Executor Test 3" Priority="0">IndividualTest1</test>
<test executeVal ="Yes" desc="Executor Test 4">IndividualTest1</test>
```

Figure 28

A product area or a test case should have only one priority defined.
If we don't specify a priority, then it is taken as Medium by default.
If a product area that has a dependency on another area, then that area will be picked up for execution (if not executed earlier in the same Run) without considering its priority and only the critical tests of that area will be executed.