

"The most strategic  
developer surface area  
for us is Office 365"

—Satya Nadella



# Strategik

## CORE FRAMEWORK

---

*Dr Adrian Colquhoun*

*June 2015*

---

## SUMMARY

The Strategik Core Framework is our open source library for creating solution definitions, provisioning and helper code for SharePoint 2013 and Office 365. It is the foundation of our solution development efforts, containing much of the plumbing that we would otherwise need to create for every custom solution we build. Our Core Framework features best practice client object model and REST API code. Wherever possible, our code is a wrapper to (and extends the capabilities of) the Microsoft Office Dev Patterns & Practices library. We are in ongoing conversations with the PnP team to ensure alignment between their code and our framework.

This document describes the structure and rationale for the Core Framework. It outlines its key features, how it works, how we develop it and its future development roadmap. The current version of this document supports the minimum viable product (MVP) release scheduled for October 2015.

## CONTENTS

Summary .....	2
Document History .....	4
Introducing the Core Framework.....	5
Overview .....	5
Key Features.....	6
Overview .....	6
Definitions .....	7
Helpers .....	9
Provisioning .....	11
Extension Methods .....	13
Reading Definitions.....	13
Reading & Writing Data .....	14
Integrating with the PnP Core .....	14
Building & Distributing the Core Framework.....	19
Appendix 1: PnP Core Code Notes .....	21
Tracking Provisioning Template Application Progress .....	23
Troubleshooting Tips .....	24
Figure 1: How Strategik build solutions.....	5
Figure 2: Core Framework solution structure.....	6
Figure 3: Strategik.Definitions project structure .....	7
Figure 4: Extract of STKContentType definition class.....	8
Figure 5: STKChoiceField class structure .....	9
Figure 6: Strategik.CoreFramework project structure (Helper classes expanded).....	10
Figure 7: Structure of STKContentTypeHelper (at MVP) .....	11
Figure 8: Provisioning a set of Content Types using the STKContentTypeHelper .....	12
Figure 9: Extension methods .....	13
Figure 10: Example read definitions call.....	14
Figure 11: Strategik.CoreFramework.PnP project structure .....	15

Figure 12: Provision (Content Type) in the STKPnPHelper .....	15
Figure 13: Extension classes in the Strategik.CoreFramework.PnP project .....	16
Figure 14: Extension method to convert a STKContentType definition to the format required by PnP .....	17
Figure 15: Extension method to convert a PnP template Content Type to a STKContentType definition .....	18
Figure 16: Core Framework - Early GitHub Repo .....	19
Figure 17: Core Framework Documentation .....	20
Figure 18: Office dev PnP Core structure (August 2015) .....	21
Figure 19: PnP Core app model extensions .....	22
Figure 20: PnP Provisioning Engine .....	22
Figure 21: Get Templates.....	23

## DOCUMENT HISTORY

Version	Author	Date	Notes
<b>1.0</b>	Dr Adrian Colquhoun	June 2015	First version
<b>1.1</b>	Dr Adrian Colquhoun	Oct 2015	Updated prior to Oct 2015 MVP GitHub release

## INTRODUCING THE CORE FRAMEWORK

### OVERVIEW

Strategik build complex information management solutions for Office 365 and SharePoint 2013. These solutions require definition in code. They require the ability to provision and de-provision Office 365 artefacts in response to business events throughout their lifecycle, the ability to analyze and update customer installation remotely and support for simple, scalable application lifecycle management.

At Strategik, we have taken the view that there is little commercial value in creating and maintaining a close source code based provisioning framework. A number of valuable community efforts already existing in the space, including the Core PnP repository and SPMeta2. Whilst both these frameworks are useful, neither currently fully satisfies the business requirements and needs of the Strategik solutions practice. As such, our approach with the Core Framework is to create a thin wrapper around those frameworks (where the functionality we require already exists) and to use best practice Client Object Model and remote API calls to create whatever functionality we require where it does not.

Thus we have created the Strategik Core Framework, our open source library for defining our solutions (in code), provisioning and helper code for SharePoint 2013 and Office 365. It is the foundation of our solution development efforts, containing much of the plumbing that we would otherwise need to create for every custom solution we build. The code is shared on GitHub (<https://github.com/strategik/CoreFramework>) and is available for others to use in the MIT License.

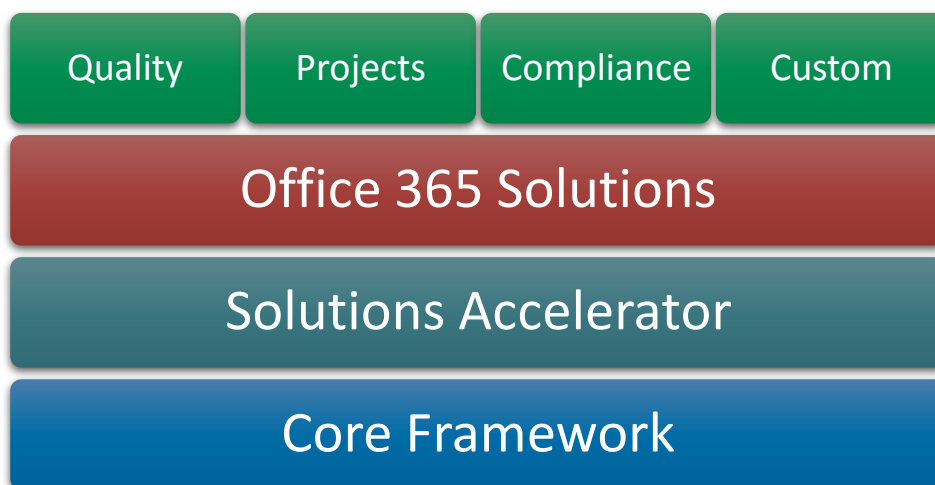


Figure 1: How Strategik build solutions

## KEY FEATURES

### OVERVIEW

The Core Framework encapsulates much of the plumbing required to create solutions on Office 365. It provides a set of simple C# classes that can be used to create definitions of the solutions and all their artefacts that need to be deployed, along with extension methods and helper that wrap up client object model and Office 365 calls.

Additional functionality is provided to read back definitions from Office 365 tenancies and to calculate deltas (the differences between a solution definition and the solution currently deployed) to that intelligent decisions on solution deployment can be made. (Note - Delta are not included in the Oct 2015 MVP release).

The core framework provides generic, schema independent methods for reading and writing SharePoint list and document library data. (Note – Only basic functionality is included in the Oct 2015 MVP release).

The core framework wraps around the latest Office Dev PnP code framework, calling that code wherever possible. A PnP extension, designed to allow the Strategik models to be deployed using the PnP provisioning engine is provided as a separate project. Additional projects are also included for unit testing. The structure of the Core Framework solution is shown below:

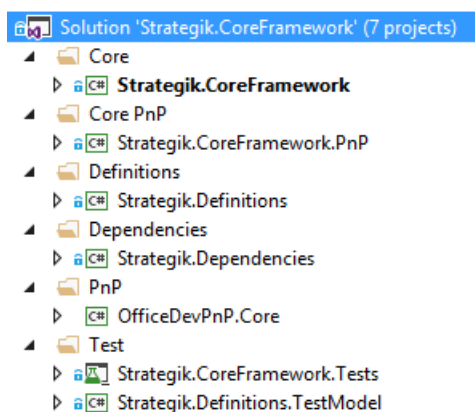


Figure 2: Core Framework solution structure

## DEFINITIONS

The Core framework provides the Strategik.Definitions project. This project contains simple .NET classes that allow us to create definitions of common SharePoint and Office 365 objects, which we can then feed to our helper classes for provisioning.

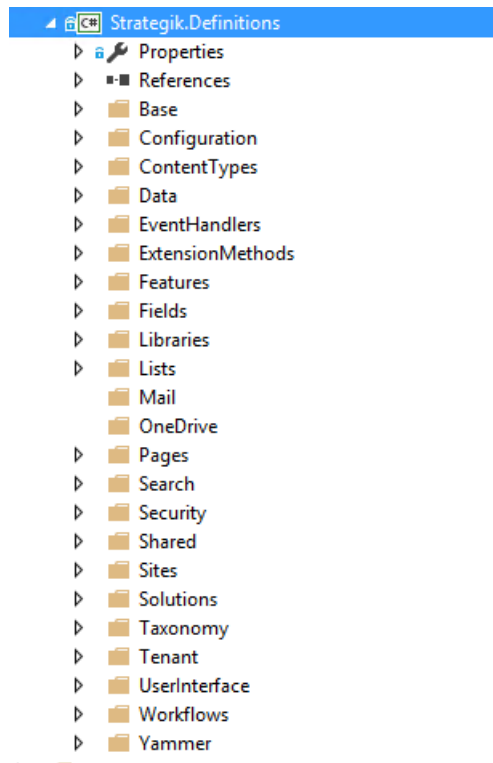


Figure 3: Strategik.Definitions project structure

Classes in this project mirror the Office 365 objects that they define, with additional properties that deal with the provisioning rules to be applied when (for example) the artifacts already exist. These definition classes are deliberately simple, with no functionality (methods) defined. Where functions are required these are added as extension methods, not to the classes directly. All definitions derive from the `STKO365Definitionbase` class, which defines common properties. Classes are prefixed `STK` to avoid naming collisions with the similarly named classes in the client object model. For illustration, an extract of the `STKContentType` definition class is shown below.

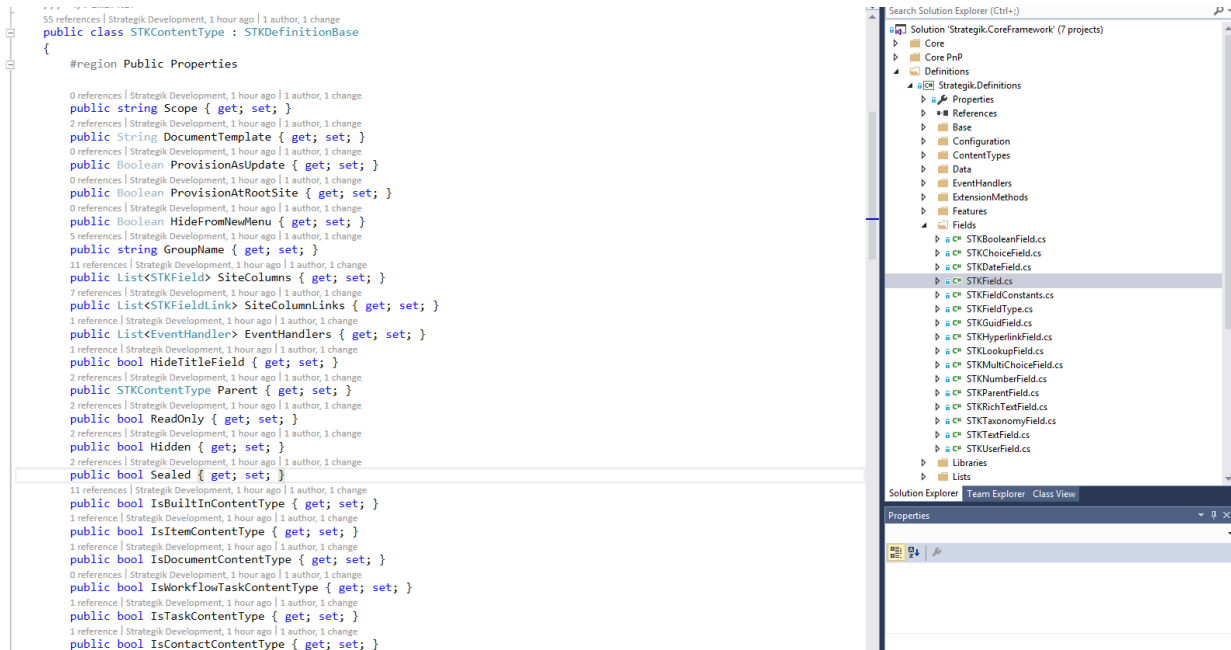


Figure 4: Extract of STKContentType definition class

Definition classes use a standard object orientated approach. For example, when defining fields (i.e. list or site columns in SharePoint), you will find a base STKField class which is then extended by the specific field we require (text, Boolean, choice etc.). The structure of the STKChoiceField class is illustrated below.



```

namespace Strategik.Definitions.Fields
{
    9 references | Strategik Development, 1 hour ago | 1 author, 1 change
    public class STKChoiceField : STKField
    {
        #region Properties

        11 references | Strategik Development, 1 hour ago | 1 author, 1 change
        public List<String> Choices { get; set; }

        4 references | Strategik Development, 1 hour ago | 1 author, 1 change
        public String Format { get; set; }

        2 references | Strategik Development, 1 hour ago | 1 author, 1 change
        public bool AllowFillInChoice { get; set; }

        #endregion Properties

        #region Constructor

        3 references | Strategik Development, 1 hour ago | 1 author, 1 change
        public STKChoiceField(...)

        0 references | Strategik Development, 1 hour ago | 1 author, 1 change
        public STKChoiceField(String id, String name, String displayName, STKFieldType type, String groupName, string staticName)

        #endregion Constructor

        #region Custom Provisioning CAML

        13 references | Strategik Development, 1 hour ago | 1 author, 1 change
        protected override void AddCustomFieldAttributes(XmlWriter xmlWriter)

        #endregion Custom Provisioning CAML
    }
}

```

Figure 5: STKChoiceField class structure

## HELPERS

The Strategik.CoreFramework project contains a set of helper classes. Helper classes are organized around the type of functionality there are intended to work with (Lists, Content Types, Taxonomy, Site Column, Search etc.). The project structure is shown below.

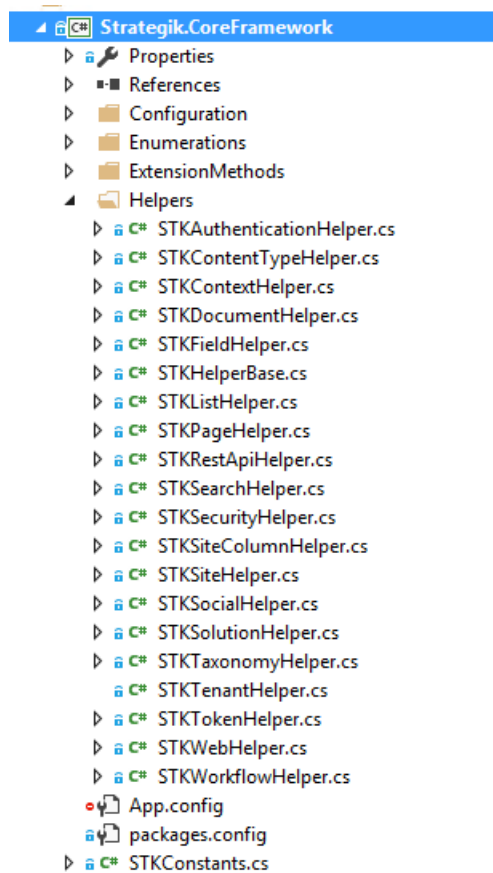


Figure 6: Strategik.CoreFramework project structure (Helper classes expanded)

Helper classes consume or produce the definition objects from the Strategik.Definitions project to either provision Office 365 artefacts or read back the structure of those artefacts that are already provisioned. Each helper class is written to operate within a specific context (site collection and site). The context is set by passing a ClientContext object to the helper's constructor. Functionality is provided by calling the appropriate helper methods. The Structure of the STKContentType helper (which is typical of all the helpers included) is how (in collapsed form) below.

```

namespace Strategik.CoreFramework.Helpers
{
    7 references | Strategik Development, 1 hour ago | 1 author, 1 change
    public class STKContentTypeHelper : STKHelperBase
    {
        [Data]

        #region Constructor

        3 references | 0/2 passing | Strategik Development, 1 hour ago | 1 author, 1 change
        public STKContentTypeHelper(ClientContext clientContext) ...

        #endregion Constructor

        #region Ensure Content Types Methods

        2 references | 0/1 passing | Strategik Development, 1 hour ago | 1 author, 1 change
        public void EnsureContentTypes(List<STKContentType> contentTypes, STKProvisioningConfiguration config = null) ...

        1 reference | Strategik Development, 1 hour ago | 1 author, 1 change
        public void EnsureContentType(STKContentType contentType, STKProvisioningConfiguration config = null) ...

        #endregion

        #region Read Content Types

        1 reference | 0/1 passing | Strategik Development, 1 hour ago | 1 author, 1 change
        public List<STKContentType> ReadContentTypes() ...

        #endregion

        Utilities
    }
}

```

Figure 7: Structure of STKContentTypeHelper (at MVP)

## PROVISIONING

Provision can be applied at a complete solution, a composite definition or a single artefact (content type, site column and so on). Provisioning operations are applied through the helper classes. The code snippet below shows how the STKContentTypeHelper can be used to ensure that List of Content Types (each defined in code) are provisioned to the current site collection. The code is taken from the unit test project included with the Core Framework (Strategik.CoreFramework.Tests)

```
public void TestEnsureContentTypes()
{
    using (ClientContext context = STKUnitTestHelper.GetTestContext())
    {
        STKContentTypeHelper helper = new STKContentTypeHelper(context);
        List<STKContentType> allContentTypes = STKTestContentTypes.GetAllContentTypes();
        helper.EnsureContentTypes(allContentTypes);
    }
}
```

Figure 8: Provisioning a set of Content Types using the STKContentTypeHelper

Within the Core Framework, we are considering implementing the following provisioning semantics in a future release:

#### **Delete If Present**

The current artefact and any children are deleted and completely replaced with the current definition. This can have profound effects on the target environment (e.g. if a list definition is deployed as Delete If Present any existing list items will be deleted and the list replaced with the current definition).

#### **Overwrite if Present:**

If the current artifact already exists it is overwritten (but not deleted). For example, if a content type definition is set to Overwrite If Present then it will be updated (existing site column links removed and replaced) but the content type itself will not be deleted. After provisioning, the content type will exactly match the definition supplied.

#### **Update If Present:**

If the current artifact already exists, its definition is updated with the current definition but none of the existing definitions will be deleted. For example, when provisioning a term set, any new terms in the definition will be created but no existing terms will be deleted. After provisioning the term set deployed may differ from the definition supplied (for example if a user already added terms in the target environment).

#### **Ignore If Present:**

If the current artifact is detected, no further provisioning is applied to this (or any child object).

## EXTENSION METHODS

The Core Framework features a number of extension methods. Where additional functionality is required (definitions, core SharePoint objects and so on) that functionality is added to extension methods wherever possible. This follows the code style that is adopted in the PnP Core itself. Extensions methods are defined at a number of points in the framework, including the definitions assembly and the PnP extension project.

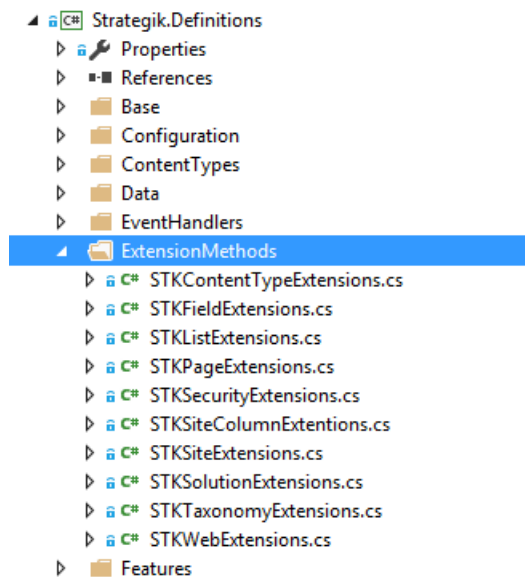


Figure 9: Extension methods

## READING DEFINITIONS

Where appropriate, the helper classes within the Core Framework contain a number of ReadDefinition() methods. These methods allow the corresponding definition objects to be generated from the custom artefacts currently deployed in the target SharePoint site collection or site. The read definitions methods generate the appropriate types definition object (STKTaskList, STKChoiceField and so on).

```
public void TestReadSiteColumns()
{
    using (ClientContext context = STKUnitTestHelper.GetTestContext())
    {
        STKSiteColumnHelper helper = new STKSiteColumnHelper(context);
        List<STKField> allSiteColumns = helper.ReadSiteColumns();
    }
}
```

Figure 10: Example read definitions call

## READING & WRITING DATA

The Core Framework provides rudimentary methods for reading and writing data into SharePoint lists and libraries. This support will be enhanced in future releases.

## INTEGRATING WITH THE PNP CORE

Wherever possible, the functionality provided in the Core Framework calls into corresponding functionality (if available) in the PnP Core. This reduces the amount of code that we need to write and allows us to take full advantage of the community efforts being advanced elsewhere. Integration between the Core framework and PnP Core is provided by the Strategik.CoreFramework.PnP project.

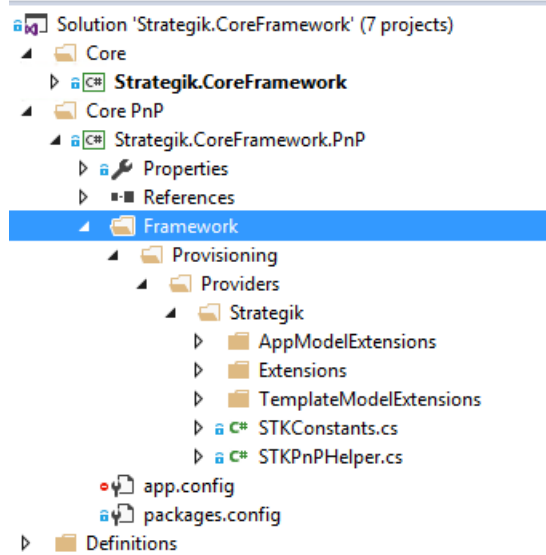


Figure 11: Strategik.CoreFramework.PnP project structure

Within this project, the STKPnPHelper class provide the provisioning and read methods which are called by the respective helpers in the Core Framework. The provisioning method for a content type is show (for illustration) below:

```
public void Provision(STKContentType contentType, STKProvisioningConfiguration config = null)
{
    if (contentType == null) throw new ArgumentNullException("contentType");
    if (config == null) config = new STKProvisioningConfiguration();

    // Check we have a valid definition
    contentType.Validate();

    // Provision any new site columns defined in the content type
    Provision(contentType.SiteColumns);

    // Generate the PnP template
    ProvisioningTemplate template = new ProvisioningTemplate();
    template.ContentTypes.Add(contentType.GeneratePnPTemplate());

    // Provision the content type
    _web.ApplyProvisioningTemplate(template);
}
```

Figure 12: Provision (Content Type) in the STKPnPHelper

Within this project a number of extension classes are provided. Within these classes, methods are included to convert from the Strategik definitions classes to the PnP Template (used in the Core PnP provisioning engine) and from a populated PnP provisioning template back into the corresponding code definitions.

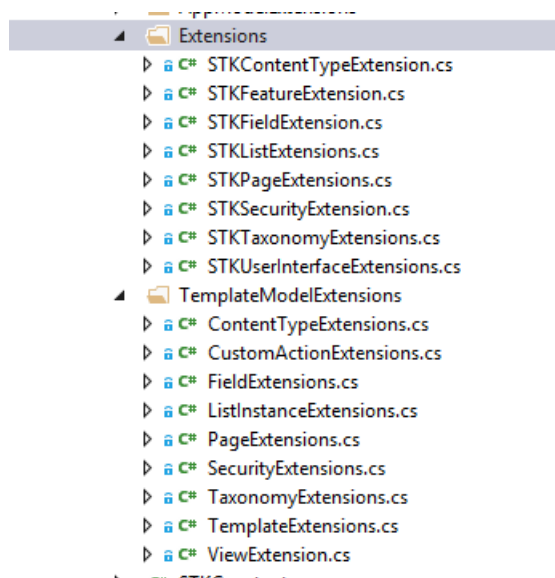


Figure 13: Extension classes in the Strategik.CoreFramework.PnP project

The extension method to generate a PnP Content Type template object is shown below:



0 references | Strategik Development, 3 hours ago | 1 author, 1 change

`public static partial class STKContentTypeExtensions`

{

1 reference | Strategik Development, 3 hours ago | 1 author, 1 change

`public static ContentType GeneratePnPTemplate(this STKContentType contentType)`

{

 `ContentType contentTypeTemplate = new ContentType()`

{

 `Description = contentType.Description,  
 DocumentTemplate = contentType.DocumentTemplate,  
 Group = contentType.GroupName,  
 Hidden = contentType.Hidden,  
 Id = contentType.SharePointContentTypeId,  
 Name = contentType.Name,  
 ReadOnly = contentType.ReadOnly,  
 Sealed = contentType.Sealed,`

};

 `foreach (STKField siteColumn in contentType.SiteColumns)`

{

 `contentTypeTemplate.FieldRefs.Add(siteColumn.GeneratePnPFieldRefTemplate());`

}

 `foreach (STKFieldLink siteColumnLink in contentType.SiteColumnLinks)`

{

 `contentTypeTemplate.FieldRefs.Add(siteColumnLink.GeneratePnPTemplate());`

}

 `return contentTypeTemplate;`

}

Figure 14: Extension method to convert a STKContentType definition to the format required by PnP

The corresponding code to convert back from the template to the Strategik definition is:

0 references | Strategik Development, 3 hours ago | 1 author, 1 change

`public static partial class ContentTypeExtensions`

{

`#region Generate strategik definition from corresponding PnP template object`

2 references | Strategik Development, 3 hours ago | 1 author, 1 change

`public static STKContentType GenerateStrategikDefinition(this ContentType contentType)`

{

`STKContentType stkContentType = new STKContentType()`

{

 `SharePointContentTypeId = contentType.Id,  
 Description = contentType.Description,  
 DocumentTemplate = contentType.DocumentTemplate,  
 GroupName = contentType.Group,  
 Hidden = contentType.Hidden,  
 Name = contentType.Name,  
 ReadOnly = contentType.ReadOnly,  
 Sealed = contentType.Sealed`

};

`foreach (FieldRef fieldRef in contentType.FieldRefs)`

{

 `STKFieldLink stkFieldFieldLink = new STKFieldLink()`

{

 `Name = fieldRef.Name,  
 DisplayName = fieldRef.DisplayName,  
 IsHidden = fieldRef.Hidden,  
 IsRequired = fieldRef.Required,  
 SiteColumnId = fieldRef.Id`

};

 `stkContentType.SiteColumnLinks.Add(stkFieldFieldLink);`

}

`return stkContentType;`

}

Figure 15: Extension method to convert a PnP template Content Type to a STKContentType definition

## BUILDING & DISTRIBUTING THE CORE FRAMEWORK

The Core framework is developed and distributed using GitHub (<https://github.com/strategik/CoreFramework>). The latest stable package (when released) is available via Nuget.

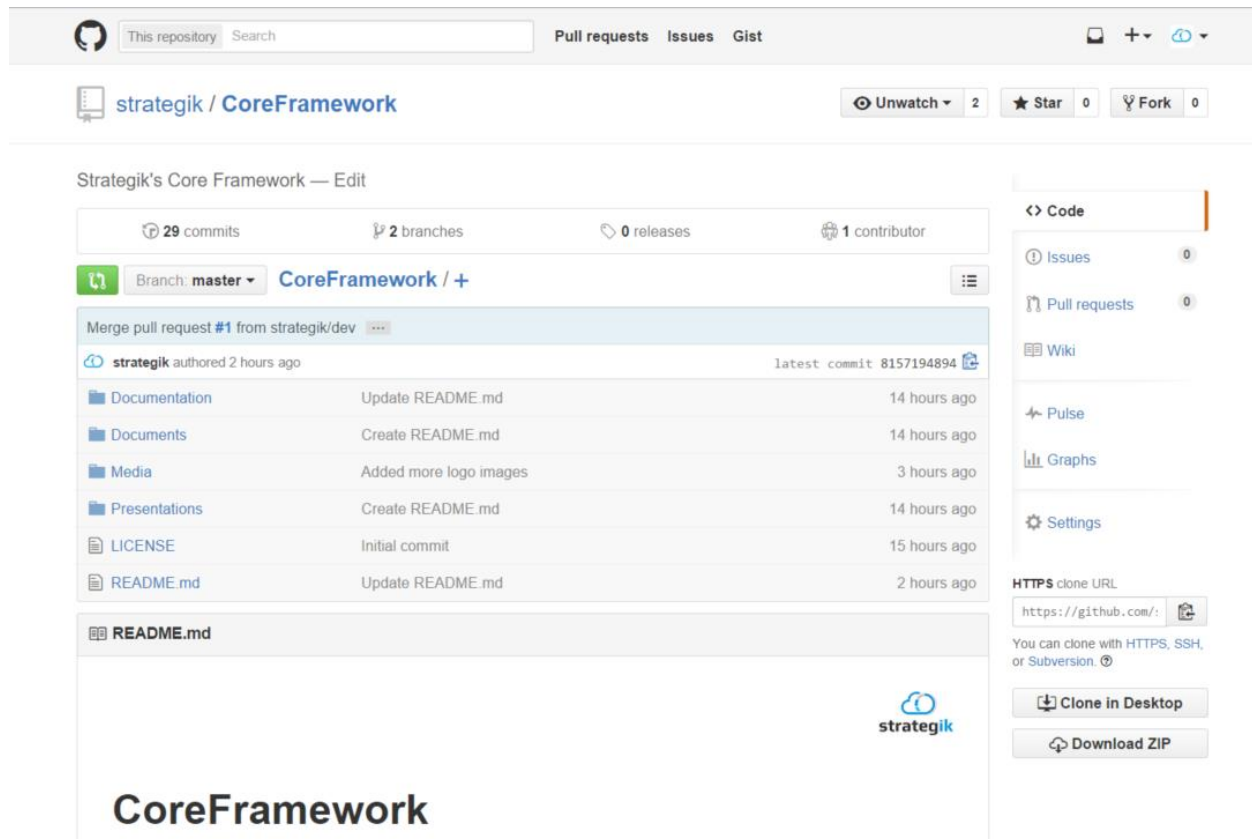




Figure 16: Core Framework - Early GitHub Repo

 README.md



You can clone with [HTTPS](#), [SSH](#), or [Subversion](#).

Clone in Desktop

Download ZIP

## CoreFramework

Strategik build complex information management solutions for SharePoint 2013 & Office 365. These solutions require definition in code. They require the ability to provision and de-provision Office 365 artefacts in response to business events throughout their lifecycle, the ability to analyze and update customer installation remotely and support for simple, scalable application lifecycle management.

At Strategik, we have taken the view that there is little commercial value in creating and maintaining a close source code based provisioning framework. A number of valuable community efforts already existing in the space, including the [Office Dev Patterns and Practices](#) repository and [SPMeta2](#) project. Whilst both these frameworks are useful, neither currently fully satisfies the business requirements and needs of the Strategik solutions practice. As such, our approach with the Core Framework is to create a thin wrapper around those frameworks (where the functionality we require already exists) and to use best practice Client Object Model and remote API calls to create whatever functionality we require where it does not.

Thus we have created the Strategik Core Framework, our open source library for defining our solutions (in code), provisioning and helper code for SharePoint 2013 and Office 365. It is the foundation of our solution development efforts, containing much of the plumbing that we would otherwise need to create for every custom solution we build. It is available for others to use under the MIT License.

Quality

Projects

Compliance

Custom

Office 365 Solutions

Solutions Accelerator

Core Framework

The Core Framework encapsulates much of the plumbing required to create solutions on Office 365. It provides a set of simple C# classes that can be used to create definitions of the solutions and all their artefacts that need to be deployed, along with extension methods and helper that wrap up client object model and Office 365 calls.

The core framework wraps around the latest Office Dev PnP code framework, calling that code wherever possible. A PnP extension, designed to allow the Strategik models to be deployed using the PnP provisioning engine is provided.

Figure 17: Core Framework Documentation

## APPENDIX 1: PNP CORE CODE NOTES

This section records notes about the PnP Core that we discover as we develop against it.

The PnP core project (August 15) is now a reasonably stable repository (low rate of change, introduction of new features). The structure of the project is shown below. For simplicity and convenience, the Strategik Core Framework uses the same structure where appropriate to do so.

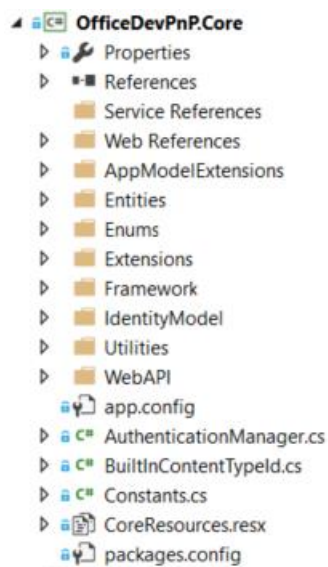


Figure 18: Office dev PnP Core structure (August 2015)

The App model extensions provides various extension methods (helper functions) to the SharePoint client object model.

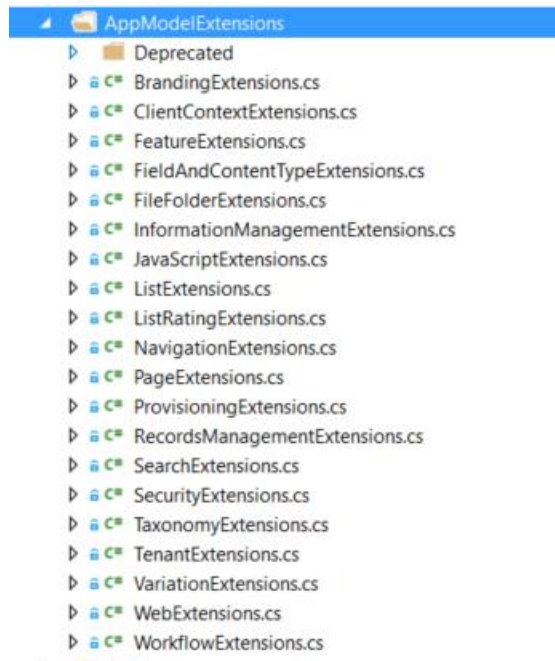


Figure 19: PnP Core app model extensions

Most of these extension methods provide extra functionality to the web object – the classes are organized by the general function which they perform.

The provisioning engine code is located under the Framework → Provisioning folder structure.

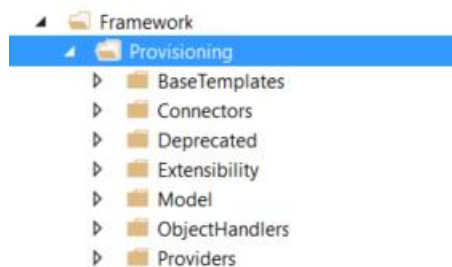


Figure 20: PnP Provisioning Engine

Reading the template (from an existing site) starts in `WebExtension::GetProvisioningTemplate()`

```
public static ProvisioningTemplate GetProvisioningTemplate(this Web web)
{
    ProvisioningTemplateCreationInformation creationInfo = new ProvisioningTemplateCreationInformation(web);
    // Load the base template which will be used for the comparison work
    creationInfo.BaseTemplate = web.GetBaseTemplate();

    return new SiteToTemplateConversion().GetRemoteTemplate(web, creationInfo);
}
```

Figure 21: Get Templates

This method starts by loading pre-created (and embedded) templates for the web concerned – this is used later to remove the out of the box components from the template delivered. This is passed in the creationInfo object to the GetRemoteTemplate() method of the SiteToTemplateConversion class where the grunt work of template generation begins.

#### TRACKING PROVISIONING TEMPLATE APPLICATION PROGRESS

The GetRemoteTemplates() and ApplyRemoteTemplates() methods both accept a ProvisioningTemplateCreationInformation object as an optional second parameter. This Object in turn contains two delegates that can be wired up to receive progress notifications and provisioning messages from the engine respectively.

## TROUBLESHOOTING TIPS

If you encounter a Duplicate Content Type Id (on premise) – make sure that SP1 is applied and latest CU's (CSOM did not support setting content type ids until after SP1).