Sitecore Helix Documentation

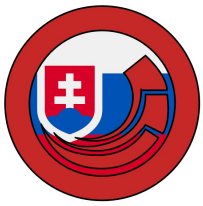
Peter Prochazka

Table of Contents

Sitecore Helix Documentation

Compiled from official Sitecore Helix Documentation last updated on 21st August 2017.

Compiled by:



Peter Prochazka (@chorpo / tothecore.sk)

19th October 2018

More Sitecore guidelines and Sitecore related topics can be found on my blog tothecore.sk.

You can find them also directly in my github repositories.

1. Introduction

There are some good *overall design principles* that should be used when designing any software project. One of the key principles is to be very strict about the dependencies between software modules. This makes the modules easier to code, test and maintain. The principles apply to any software project in any language so they are equally applicable to Java as they are to .NET and Sitecore.

The development guidelines and recommended practices in this documentation describe the application of the *overall design principles* applied to a Sitecore project. It says how modules should be structured and how individual sites should use these modules. It is a set of recommendations for making your Sitecore project as easy to create, test, extend and maintain as possible.

Helix is the overall design principles and conventions for Sitecore development described in this documentation.

1.1. What is Helix and Habitat?

Helix is a set of overall design principles and conventions for Sitecore development.

Habitat is a real Sitecore project implemented on the Sitecore Experience Platform using Helix. It is an example that allows developers to see how Helix is applied and lets developers experience a project based on these principles. It also provides an excellent basis on which to develop additional modules and extend existing ones. In addition, it saves the developer from having to create these modules from scratch. Throughout this document is a series of examples of how Habitat implements the described principles and conventions.

Habitat Example

Throughout the Helix documentation you will find various examples from Habitat.

You can find the running Habitat project on http://habitat.sitecore.net and the source code on http://github.com/sitecore/habitat

1.1.1. Fuelling fast paced Sitecore development

Still not sure? Let's try an example with racing cars.

When you design a racing car, there are some good things to have. For example, the shape of the car affects wind resistance so you choose the right shape to minimise drag. The engine efficiency affects the amount of fuel you need and thus weight so you design efficiently and lightly to maximise acceleration. To achieve the fastest car, you apply the *overall design principles*.

If you are a Formula One racing team, you have to apply these *overall design principles* within the bounds of the Formula One rules for car

length, weight, size of engine, spoilers, etc. This is like Helix because you have taken some *overall design principles* and applied them to a particular type of racing car – in this case a Formula One racing car. The *overall design principles* are good for any type of racing car in any race but only the *overall design principles* used with Formula One are correct for a Formula One race.

Habitat is like the Formula One car itself. It is an example of an F1 Car (*Habitat*) you can race around a track that follows *overall design principles* for Formula One (Helix). The difference between Formula One and Sitecore is that Formula One does not provide the racing teams with an example car to give all the racing teams a head start, whereas Sitecore does.

1.1.2. Why be interested at all in Helix or Habitat?

Helix provide a set of guidelines for your Sitecore projects. The *Habitat* example provides you with a pre-built and tested set of common modules that you can use as an inspiration to your project. Both improve the efficiency of your projects, reduce costs and time to market. As more and more people and organisations adopt the Helix conventions and principles, it will become a Sitecore standard. This means that people who are familiar with the conventions or the Habitat example will be able to work more easily on other convention-based projects with minimal training. It will be easier for Sitecore Product Support to understand projects built using the conventions, enabling them to resolve issues more quickly. Sitecore will test its software using the conventions so any compatible project that has been implemented for a customer will be more reliable. And since Sitecore will test its software using the conventions, Sitecore will be able to provide better guidance on how to update and upgrade existing Sitecore projects when new versions and new products are released.

Still need a car example? A race engineer has been working for an F1 team and decides to transfer to another F1 team. The race engineer is

already familiar with Formula One (Helix). She recognises just about everything on the car (*Habitat*). There are a few minor differences for which she will need some training but those small differences do not stop her opening her bag of wrenches and getting to work right away.

1.2. Reading this documentation

Before you start venturing into a complete understanding of Helix – and particularly the reasoning behind it – it is a good idea to have a reasonable experience in developing in ASP.NET and in Sitecore. Although the Habitat example site works well as an end-to-end example of a Sitecore implementation – and as such is a good supplement to the Sitecore developer training – the added intricacies of the conventions can add to an increased learning curve of Sitecore.

However, if you are new to Sitecore and are determined to understand more about Habitat and Helix, please understand that although Sitecore is not generally conceived as hard to get started with, there are an incredible number of opportunities and possibilities to make your new journey with Sitecore both fun, interesting and sometimes challenging. Treat Helix and Habitat as just another one of these great opportunities.

Helix and this documentation do not define conventions or principles for all aspects of ASP.NET or Sitecore, but rather focus on the macro architecture conventions in Sitecore development. This means that you find that many of the traditional aspects of software development such as object-oriented architecture and ASP.NET MVC conventions are not mentioned in this document. In other words, despite the conventions and recommended practices in this document Helix still gives you great freedom in your choice of tools and general development practices.

Although the overall Helix conventions and principles can be applied to ASP.NET Web Forms, this document assumes the use of ASP.NET MVC with Sitecore. If you are currently working in ASP.NET Web Forms, the adoption of Helix might be an appropriate excuse to switch to ASP.NET MVC, as it is more future-proof and generally recommended for Sitecore development.

Although Helix is a set of recommended principles and conventions from Sitecore itself, it is not a set of rules. Development teams, businesses and requirements are different. Although there are great benefits to be gained from aligning to a defined set of conventions, there is also a need for pragmatism. The advice is to generally read all development conventions, patterns and principles – including Helix – with a critical mindset and apply them in the context of your own business, team and solution.

1.3. Definitions

Module
A Module is a conceptual grouping of assets which relates to a business requirement. The conventions, processes and tools described in this document relate to managing all these assets in a module-centric – and thereby business centric – way. For example, when the company asks that their Sitecore solution contains website search, all assets, business logic and configuration relating to search belongs to the Search module.

Solution
Technically often refers to Visual Studio solution, but conceptually, it can also refer to an *implementation*. For example, Michael is a developer at his company. Most of his time is spent working in the Visual Studio solution that contains the code that powers his company's Sitecore solution.

Project
In technical terms, *project* often refers to Visual Studio project, but conceptually can also to the process of implementing the business requirements into an *implementation*. For example, Michael is a developer at his company. Most of his time is spent working in one of the five Visual Studio projects that contain the code that powers his company's Sitecore solution. He is starting on a new implementation, which is a project that will result in new features being added to his company's website.

Serialization
The process of writing data in the Sitecore databases to disk so that it can maintained in version control and packaged into deployments across environments.

Assets
Technically an Asset refers to a digital asset such as images, but can

also mean the actual output from any process or task in your entire application lifecycle: code, files, visual design, data, content, configuration changes, deployment packages etc.

Dictionary
A collection of named text snippets which can be translated across languages and used in the UX, for example on websites, Sitecore tools, e-mails, etc.

Tenant
A product owner of one or more sites in a Sitecore implementation. Sitecore allows multiple tenants to share a single *implementation*, which allows certain resources to be shared (such as templates and digital assets), while allowing other resources (such as sites and other business entities) to be defined and managed independently (see Multi-site and multi-tenant).

Site
A collection of content and output with a common overall business objective, and sharing a common set of assets. A site can output content to any channel, not necessarily as a website to the web channel. In Sitecore, technically a site is a context under which content is output, i.e. which assets the business logic can access.

Website
A website is a *site* that can output content to the web channel. See Site.

Implementation
An implementation, or customer implementation, is the total number of modules, features and functionalities developed and deployed to solve the customer business problem. Also often referred to as the *solution*.

2. Patterns, Principles and Conventions

Helix describes the overall architecture of your Sitecore solution and thus communicates some guidelines and conventions which should be durable and flexible enough to be applied to any Sitecore project or business. The architecture pattern described by Helix is often referred to as Component-based Architecture or Modular Architecture.

Modular architecture provides you with a framework to optimize and increase productivity by describing how to isolate domain logic to make the whole solution or implementation more manageable. Modular architecture is therefore in its foundation a way of making sure that the resulting solution is flexible enough for whatever change might be coming. There are many great benefits to modular architecture such as reusability and rapid development, but the core motivators behind the conventions are simplicity, flexibility and extensibility in the implementation.

Keep in mind though, that the principles behind modular architecture will not ensure you remain within the conventions or confines of said principles. This is defined by the methodology on which you apply the architecture and even the specific tools you use to build the final solution.

Habitat Example

The Habitat example implementation is not meant to dictate the specific methods or tools to use in your Sitecore project but should rather be an example of how the architectural design pattern, namely modular architecture, can be implemented including a methodology and a set of tools to support it.

You should always consciously select the tools and methods which fit the scenario, your development team and your business.

2.1. Architecture Principles

The architecture principles and conventions defined in Helix focus largely on macro architecture, i.e. how the complete solution is put together for maximum productivity, quality and longevity. This is not to say that lower level architecture and principles – class design, code structure, naming conventions etc. – are not important, but these principles are often more focused on general developer practices and less related to Sitecore.

There are three main topics important to Helix and modular architecture:

Dependencies
which describe how feature and functionality in the solution relate to each other.

Layers
which control the direction of dependencies and thereby assure a manageable solution.

Modules
which define the isolation of features and functionality leading to greater discoverability and simplicity in the development process.

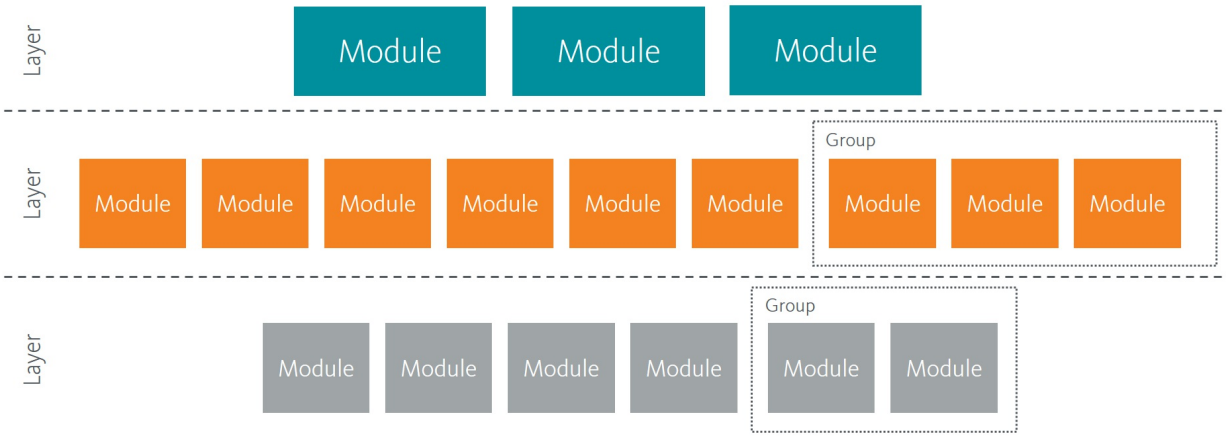


Figure: Logical architecture of Helix

2.1.1. Dependencies

Controlling dependencies throughout a solution is critical to software development in any shape or form. This is both important in the micro architecture – how you break down functionality into methods, classes, class inheritance etc. – but equally so in macro architecture – how you define the overall features of the solution and how these features are coupled.

One of the fundamental principles of object-oriented programming is High Cohesion and Low Coupling. High cohesion relies on breaking the solution down into the right parts with logic that belong together – in Helix referred to as modules – and low coupling relies on keeping the number of dependencies between the different parts down to the absolute minimum.

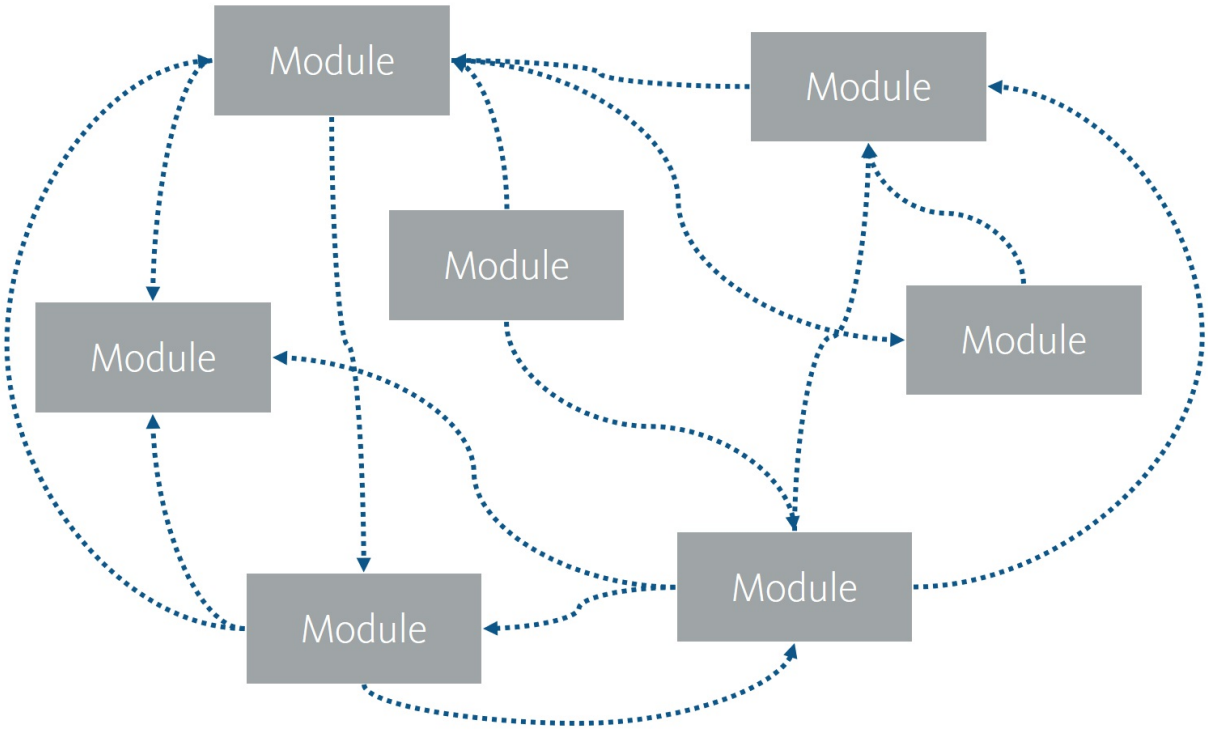


Figure: Uncontrolled Dependencies, High Coupling

A solution where dependencies are not controlled and with high

coupling between the various parts (maybe even to a degree where there are no tangible parts of the solution) very quickly becomes unmanageable and general stability is affected because changes cannot be made without affecting many parts of the solution. This decrease in productivity resulting from the lack of architectural focus is often referred to as technical debt.

With an increase in functionality, the interconnection between modules becomes so high that productivity slowly grinds to a halt. The effort is spent on maintaining the relationships between features, testing and stabilising the solution as opposed to developing new functionality.

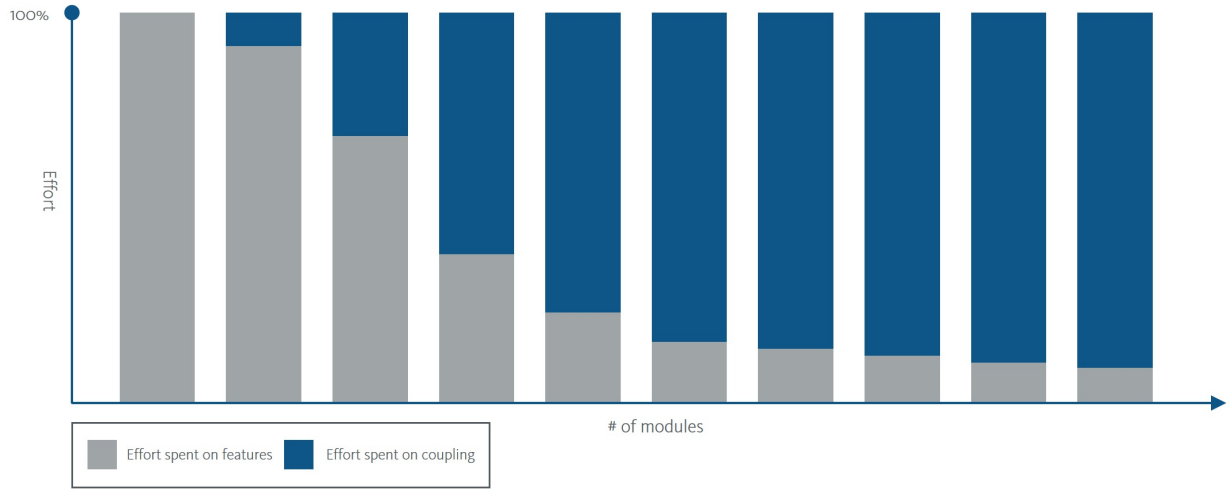


Figure: Effort wasted on High Coupling

If a more structured approach to coupling is taken, as described in Helix, the number of dependencies between features is reduced dramatically. And by reducing the number of dependencies and also making them apparent and obvious, the developers will know exactly what effects any change will have on the wider solution, greatly reducing the effort spent on stability and testing. Also, by isolating features, with defined interfaces and clear dependencies, the internal workings of the individual modules become less of an impediment, as developers can focus exclusively on the business feature they are addressing, thus greatly increasing flexibility of the solution and productivity of the team.

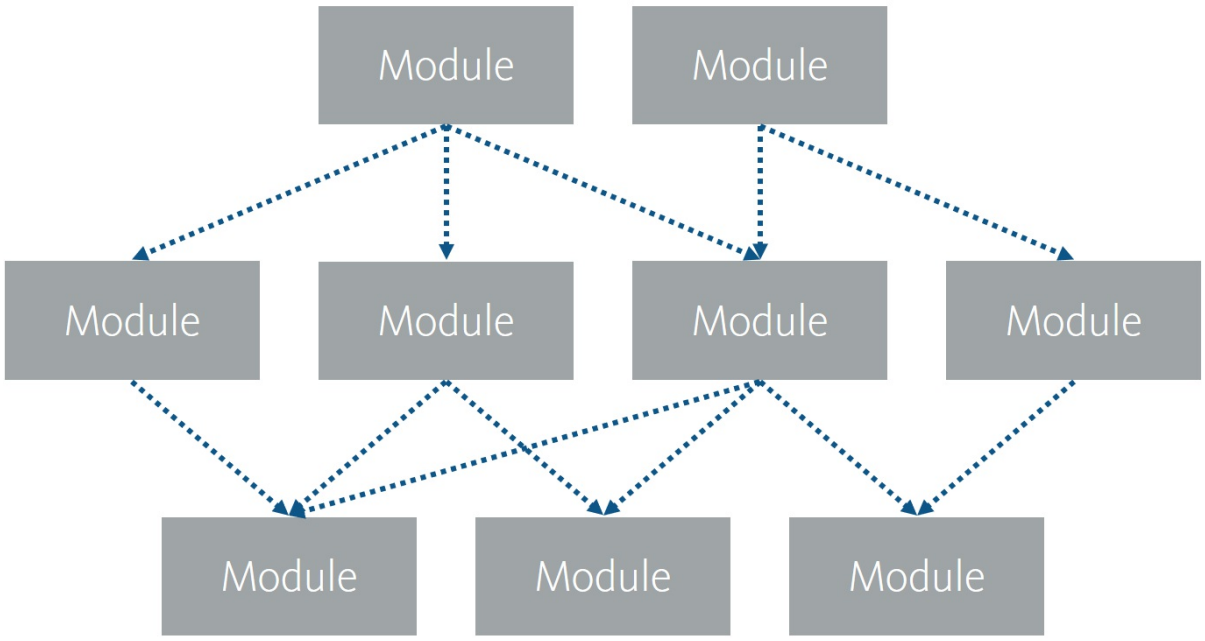


Figure: Controlled Dependencies, Low Coupling

2.1.1.1. Types of dependencies

In software, dependencies can either be explicit or implicit. Examples of explicit dependencies are the keyword `using` in C#, and a reference in one assembly to another. Examples of implicit dependencies are a class string in the HTML mark-up, references to Sitecore fields by name or reliance on specific technology behaviours in one module without explicitly referencing the module or exposing this behaviour in the interface of the module.

It is very important to stress that conceptually, as well as practically, all dependencies between modules count. In a Sitecore context this includes not only references between C# classes and .NET assemblies, but also references from code to Sitecore templates and fields, references between templates, references from templates to renderings, references from HTML mark-up to CSS and so on. Therefore, when working in a modular architecture environment with Sitecore, make sure you constantly keep an eye out for loose coupling or implicit dependencies, and actively ensure dependencies are as

explicit as possible.

2.1.2. Layers

The layer concept in Helix supports the architecture by making the dependency flow completely clear everywhere in the solution, in Sitecore, in Visual Studio and even in the file system. Furthermore, the layers provide a structure that is extremely suitable for creating and maintaining solutions of any size and steers both new and experienced developers to producing more maintenance-friendly and clean code.

Note that the layers in Modular Architecture are not equivalent to the layers seen in 3/n-tiers architecture even though they bear resemblance in terms of dependency direction.

Even though layers are a conceptual construct in the architecture, layers are physically described in the implementation by folders in the filesystem, Visual Studio and Sitecore, along with namespaces in code and layers defines in which direction modules can depend on other modules.

Layers helps control the direction of dependencies – the importance of which is described by the Stable Dependencies Principle or SDP, which is one of the cornerstone principles in Modular Architecture:

Note

Stable Dependency Principle

The dependencies between packages should be in the direction of the stability of the packages. A package should only depend upon packages that are more stable than it is.

http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

This principle tells us that a module should only depend on a module

that is more stable than itself. Code stability is a way of expressing how likely it is that the code in a module and its interfaces will change. Stability is important since if you depend on a module that changes, then these changes could affect all the dependent modules as well.

If dependencies are not controlled, then you can end up in a situation where a change will unintentionally affect seemingly unrelated parts of the solution – in effect, an unstable codebase - and an unstable code-base will make your solution hard or impossible to maintain over time.

The Sitecore Architecture Conventions defines three layers: Project, Feature and Foundation. Each layer have a very clearly defined purpose. In order to structure your Sitecore implementation properly, it is important to understand the principles behind each of these.

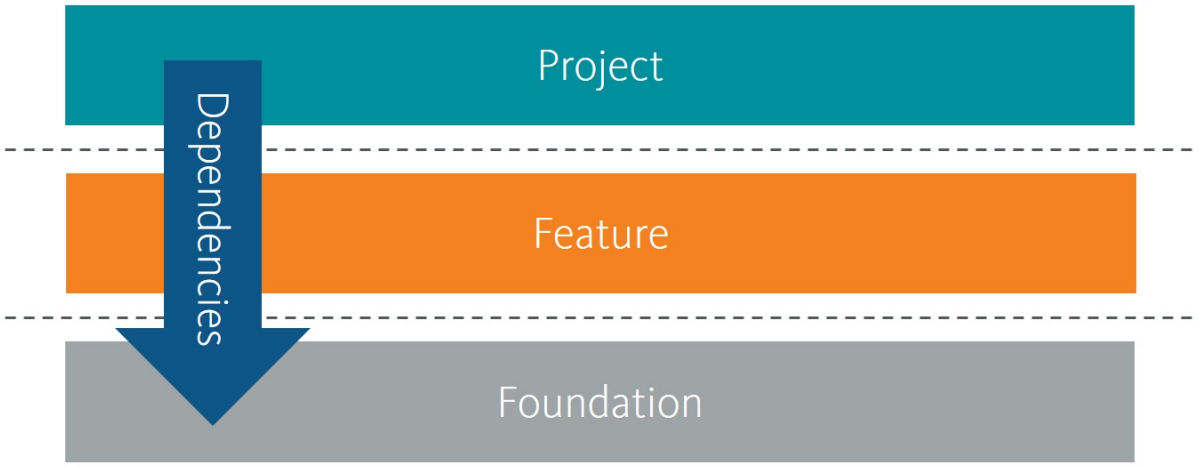


Figure: Layers in the Sitecore Architecture Conventions

Practically speaking, the three layers are defined in Visual Studio as Solution folders, in the file system and directories and in Sitecore as corresponding content type folders.

These are the typical places where layer folders are defined in Sitecore:

Master:/sitecore/system/Settings/*[Project|Feature|Foundation]*

Master:/sitecore/templates/*[Project|Feature|Foundation]*
Master:/sitecore/templates/branches/*[Project|Feature|Foundation]*
Master:/sitecore/layout/renderings/*[Project|Feature|Foundation]*
Master:/sitecore/layout/layouts/*[Project|Feature|Foundation]*
Master:/sitecore/layout/placeholder settings/*[Project|Feature|Foundation]*
Master:/sitecore/layout/models/*[Project|Feature|Foundation]*
Core:/sitecore/templates/*[Project|Feature|Foundation]*

Habitat Example

In Habitat the Sitecore root folders for the layers – as described above - are managed by the Sitecore.Foundation.Serialization project, as this project is the foundation for Sitecore content serialization in the solution. (See Managing items in development)

In Visual Studio, the layers are defined as solution folders:

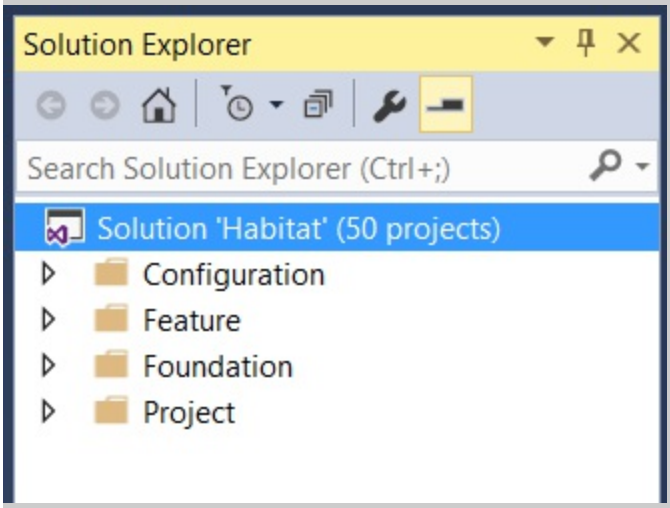


Figure: Layer Solution Folders in Visual Studio

On disk, the layers are defined as folders under /src:

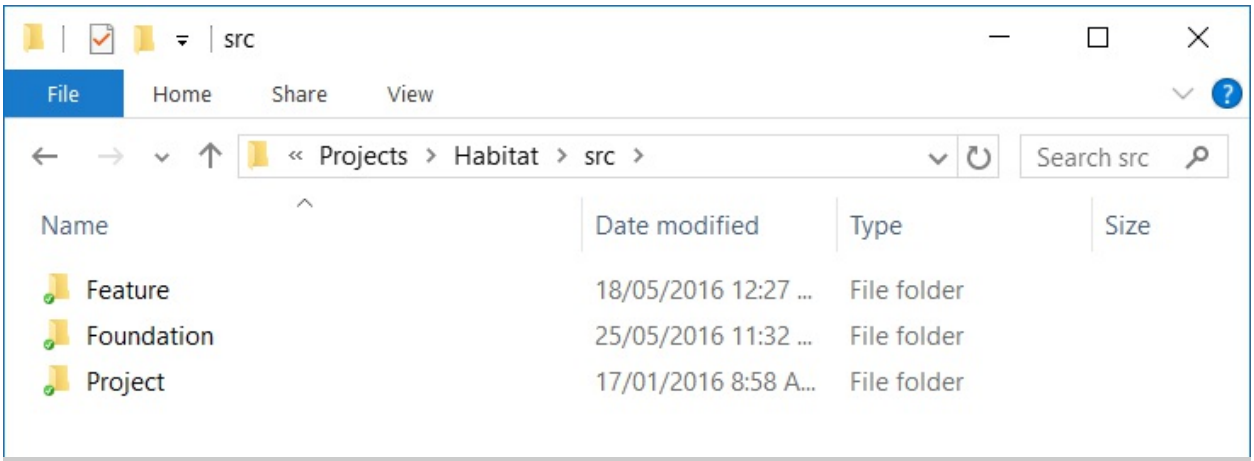


Figure: Layer folders under /src on disk

Example of layer folders in Sitecore under Templates and Layouts:

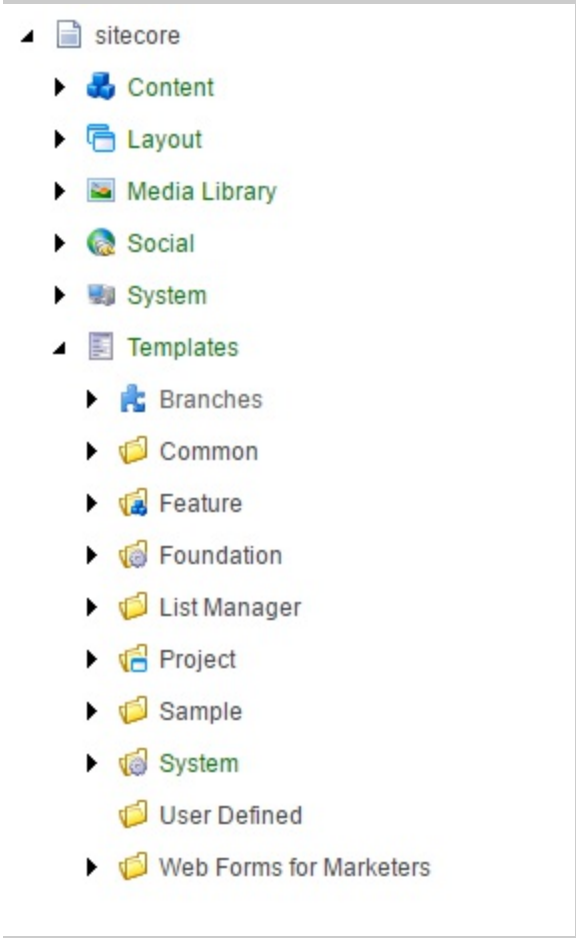


Figure: Layer folders under /sitecore/templates in Sitecore

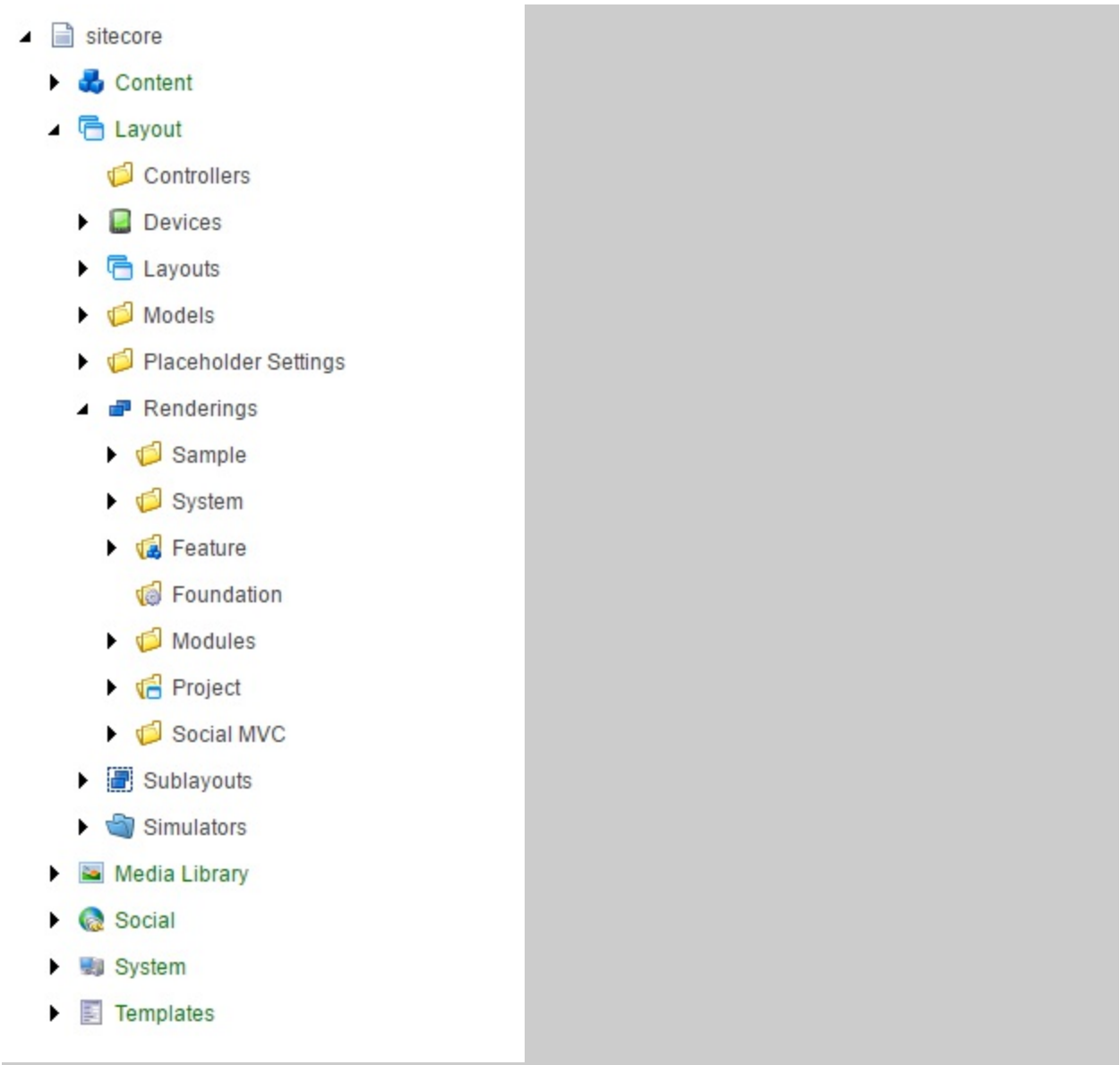


Figure: Layer folders under /sitecore/layout/renderings in Sitecore

2.1.2.1. Project Layer

The Project layer provides the context of the solution. This means the actual cohesive website or channel output from the implementation, such as the page types, layout and graphical design. It is on this layer that all the features of the solution are stitched together into a cohesive solution that fits the requirements.

Each time there is a change in a feature, a new one is added or one is

removed, then this layer will typically change. The layer also brings together the concrete graphical design of the solution, which means that in terms of the *Stable Dependencies Principle*, the modules in this layer are *unstable*. The most unstable modules in your solution and thus, remembering the Stable Dependency Principle, should have the fewest dependencies on it.

The Project layer is typically small and contains few modules, often determined by the number of tenants in the solution and their needs (see Multi-site and multi-tenant).

Typically, in a single tenant solution there will only be a single module, namely the specific website or requirements that fits the needs of the tenant, and this will contain little or no pre-compiled code but instead consist of mark-up, styling, layout and templates of the item types in Sitecore which the editors can create (see Template types).

Habitat Example

In Habitat there are two modules in the Project layer: *Habitat* and *Common*. Habitat represents the actual Habitat website or channel output, and therefore connects the features and page layouts in a way that fits the requirements for this website. The Common module paves the way for a multi-tenant implementation by defining some of the shared templates and settings between tenants.

Project layer modules can also be used to expose one feature's functionality to another without having to directly make one Feature module dependant on another, for example through the inversion of control or pipeline patterns. However, be careful not to implement actual feature-specific business logic in the Project layer in this process.

2.1.2.2. Feature Layer

The *Feature* layer contains concrete features of the solution as understood by the business owners and editors of the solution, for example news, articles, promotions, website search, etc.

The features are expressed as seen in the business domain of the solution and not by technology, which means that the responsibility of a Feature layer module is defined by the intent of the module as seen by a business user and not by the underlying technology. Therefore the module's responsibility and naming should never be decided by specific technologies but rather by the module's business value or business responsibility.

Each Feature layer module has to strictly conform to the Common Closure Principle.

Note

Common Closure Principle

Classes that change together are packaged together.

http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

This principle ensures that changes in one feature do not cause changes anywhere else, and that features can be added, modified and removed without impacting other features. For example, in a Sitecore context, it is important that all Sitecore items – such as the interface templates and rendering items – are managed, versioned and packaged with the views and code files of the feature. This can be done by serialization (see Managing items in development). Likewise, changes to configuration files (web.config or Sitecore .config include files) must be managed as part of the feature module (see Configuration and settings).

A strict awareness of dependencies within the Feature layer is very important. One Feature module must *never* depend on another

Feature module as this certainly makes you lose many of the benefits that that Modular Architecture provides, such as the overall flexibility and reliability of the solution. This principle can sometimes be challenging as functionality in some features often rely on data from other features and you will have to rely on architectural patterns to get around this. For example, website search will rely on data from other modules as part of the indexing and search results rendering. To get around this a typical approach would be to add the concept of indexing and rendering search results to a foundation level module (see Layers) which the Search feature module then utilises. Other modules can then offer their content to search by plug into the indexing and rendering functionality in the Foundation module – through for example an inversion of control pattern.

Note that although several modules in the Feature layer can be grouped together semantically (see Modules) this only suggests a conceptual coherence between modules – not in any way a technical dependency.

2.1.2.3. Foundation Layer

The lowest level layer in Helix is the Foundation layer, which as the name suggests forms the foundation of your solution. When a change occurs in one of these modules it can impact many other modules in the solution. This mean that these modules should be the most stable in your solution in term of the *Stable Dependencies Principle*.

Conceptually, it is helpful to think of all the frameworks and software you rely on in your solution as foundation modules. This includes the Sitecore platform, .NET and other technology frameworks such as Bootstrap, Foundation, jQuery etc. In the context of your implementation, these are typically very stable modules but when they do change, it often requires a more rigorous testing process and potentially a lot of changes to your Feature and Project layer modules. By controlling dependencies even to these frameworks, you can

greatly decrease the time needed on technology upgrades and increase the stability of the solution.

Habitat Example

In the Habitat example, the Sitecore.Foundation.Theming module implements most of the CSS stylesheets for the Habitat website. This might be seen as a very Project layer specific functionality but on close inspection, you will notice that the CSS of the module merely pulls in, wraps and extends the standard Bootstrap framework, and thus exposes an implementation specific design framework for all Feature modules to use. Any website or page specific CSS additions can be added in the Project layer modules – just as you would if you are styling on top of standard Bootstrap, Foundation or other frontend frameworks.

CSS is the single most common cause for implicit dependencies between modules, so be sure to have a strategy for how to deal with the graphical design implementation in your Helix compliant solution (see Visual Design and Theming).

Typically, modules in the Foundation layer are either business-logic specific extensions on the technology frameworks used in the implementation, or shared functionality between feature modules that is abstracted out into separate frameworks.

Typically, modules in the Foundation layer are conceptually abstract and do not contain presentation in the form of renderings or views - as these are to be considered concrete. Some framework modules might still contain mark-up in code though, examples being precompiled web-controls and html helper functions, but in order to control dependencies, any Feature or Project specific knowledge should be passed as parameters from the depending module.

Habitat Example

The Sitecore.Foundation.Indexing module in Habitat allows all Feature modules, and their content types, to participate in the search functionality of the solution. This means that a new Feature layer module can be exposed through the search pages of the websites by simply implementing the interface or configuration defined in the Foundation layer module – and without the Sitecore.Feature.Search module knowing anything about the new module or its content.

Unlike the Feature layer, there is no strict convention on dependencies between modules in the Foundation layer. This means that one Foundation layer module can depend on another Foundation layer module in the solution – as long as they rely on the basic principles on component architecture such as the Acyclic Dependencies Principle and the Stable Abstractions Principle:

Note

Acyclic Dependencies Principle

The dependency graph of packages must have no cycles.

http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

Note

Stable Abstractions Principle

Abstractness increases with stability.

http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

2.1.3. Modules

The concept of modules in Helix is derived from the concept of components in Component-based Architecture, which is described in the book Agile Software Development by Robert C. Martin. However, in the Sitecore context the word “component” might be confusing as it often refers to an element of a page – a rendering – and therefore a more generally accepted term *module* is used.

Keep in mind that in Helix, modules are business-centric. This means that they should relate to business objectives and group together multiple technology entities that refer to this objective. This principle goes against many traditional software conventions - such as the ones dictated by MVC (models, controllers and views) or even Sitecore (templates, layouts, settings) - that define grouping based on their type, rather than their business objective.

For this reason, the breakdown and naming of modules can be one of the most challenging parts of adopting Helix. Developers are often caught by the type-centric nature of many development tools or methodologies and forget about the business- or feature-centric nature of modular architecture. Be careful not to fall in this trap and always keep the Common Closure Principle in mind.

Note

Common Closure Principle

Classes that change together are packaged together.

http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

Never be too afraid or cautious of having modules with very little logic or modules with just a single technology. There might be management issues with having a large number of modules, but these challenges

can often be overcome by breaking the implementation into multiple Visual Studio solutions or with DevOps and automation. The *Single Responsibility Principle* should always have higher priority than any management or tools related issues – if you deviate from this, it will start to devalue your clean architecture and potentially lead to the technical debt you are trying to avoid.

The name of a module should reflect the business requirement or the use case of the feature – never the technology or implementation – and should always be in the domain language. (See Domain language).

Habitat Example

These are examples of modules in the Habitat example site:

Feature/Accounts
This feature handles everything involving user accounts: login, registration, xDB integration, user profiles, etc.
Feature/Navigation
All elements of navigation on the website are handled by this module. This includes: primary and secondary menus, breadcrumbs and link lists.
Feature/Social
The Social feature enables Twitter feeds on the pages as well as allows the editor to add Facebook metadata to the website pages.
Foundation/Indexing
This Foundation layer module provides an abstraction layer on the search functionality of Sitecore and allows features both to search within the data they provide and to integrate into the site-wide search.

2.1.3.1. Groups

Modules can be conceptually grouped together for better maintainability, readability or structure, for example commerce related features and foundation modules can be grouped together to

distinguish them from the standard website features. However, this kind of grouping does not override any of the other conventions. Nor does it introduce new architectural layers to do things such as enable references across feature modules. This kind of grouping exists solely for readability or discoverability purposes.

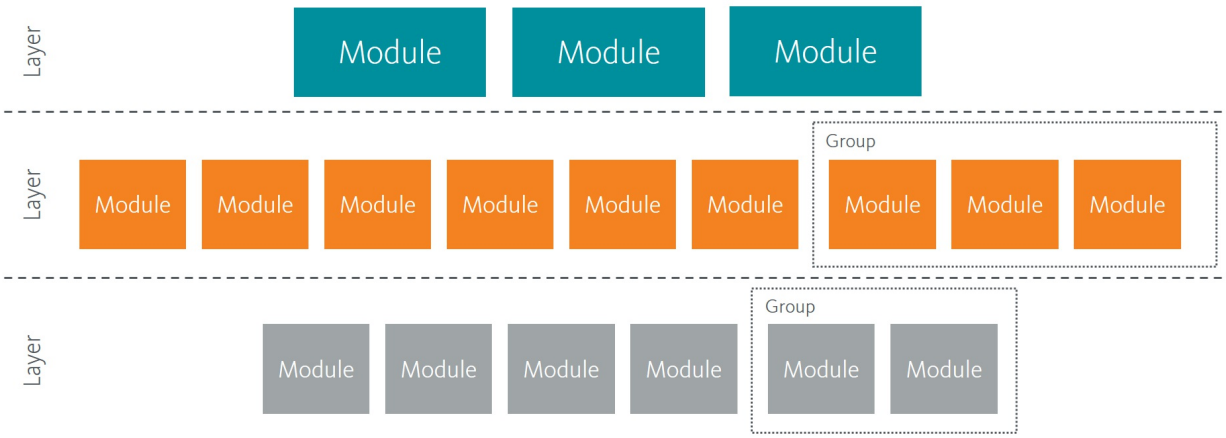


Figure: Modules, Layers and Groups

2.1.4. Domain language

Terminology and naming is one of the hardest things to get right in software development – and one of the biggest sources of confusion and inefficiency.

Remember that the true users of the code you are writing are never the developers or technical project members. The true users are the business users, editors, administrators or visitors. Therefore, establishing a common domain language between the developers and business users should be a top priority in any project. Likewise, once a domain language has been established, it is imperative that the terminology be ubiquitous. What is even worse and more confusing than using the wrong terminology is using multiple terminologies at once.

Habitat Example

In Habitat *Teaser* refers to a rendering which can be used across different page types to highlight another type of content, for example *News Teaser* (highlighting and linking to a news article) or *Person Teaser* (relating a person to the content and linking to the person's page or contact details). This concept is often known as or referred to as *Spots*, *Promotions* or just *Promos*. Choosing the term which the end users are familiar with can make the learning curve flatter and drive an easier adoption.

Also be cautious that terminology can be ambiguous between domains, for example the marketing and technology domain, and a specific technical tool can define a business domain term in another way than the common or individual company understanding. For example, the Sitecore Experience Platform uses the terms campaign and template in very particular ways – and is often in conflict with the common understanding in the marketing domain.

A close attention to terminology always increases productivity.

2.2. Visual Studio

A single Sitecore implementation can end up consisting of a large number of modules and this can lead to technical management issues.

Good rules of thumb are that your architecture should always have higher priority than any tools or technology related issues and only with sincere thought should you deviate from your conventions. Deviations or exceptions will most often lead to technical debt and instability or lack of flexibility. Management and development problems, such as slow development environments or complex deployment procedure etc. – especially with the pressure and stress of performance and deliveries - will lead to deviations from architectural principles and therefore to technical debt.

In large scale implementations, it is therefore often helpful to break down your implementation into multiple Visual Studio Solutions – each with a logical grouping, for example independent Foundation modules, cross-project or reusable modules or feature groupings such as Basic Website or Commerce features.

In this context, though, it is important to stress that each architectural module, for example the Navigation Feature module, most often will consists of a number of different technologies and tools used for the feature and can therefore be represented by different Visual Studio projects of different types (for example the main code assembly, unit tests, item serialization projects etc.) – but which should always be managed together in one Visual Studio solution. Avoid grouping Visual Studio project together by technology or type.

2.2.1. Implementation structure

A Visual Studio (VS) solution is the manifested grouping of one or more architectural *modules*, and can represent many different types of groupings. One VS solution could represent an entire Sitecore implementation, with all Project, Foundation and Feature layer modules managed together, while another Sitecore implementation could span multiple VS solutions, with one VS solution grouping together the feature modules for commerce, another VS solution housing the regular website features and with the Project layer modules in additional VS solutions per tenant.

How a customer implementation is structured between Visual Studio solutions is entirely up the requirements, complexity and logical architecture of the specific customer implementation. Be aware that a Visual Studio Solution should logically group together conceptual modules – not VS projects – and therefore all Visual Studio projects belonging to a logical module should be managed together in one VS solution.

Habitat Example

Internally in Sitecore Habitat is used as the foundation for many Sitecore demos, each focused on demoing specific business scenarios, such lead nurturing, campaign management and portals, and how these are handled by the Sitecore products range, such as the Sitecore Experience Platform, Sitecore Print Experience Manager or Sitecore Commerce. The Habitat demo framework is therefore maintained within its own Visual Studio solution (and version control repository) and each demo scenario – with any specific scenario features - is maintained in its own solution.

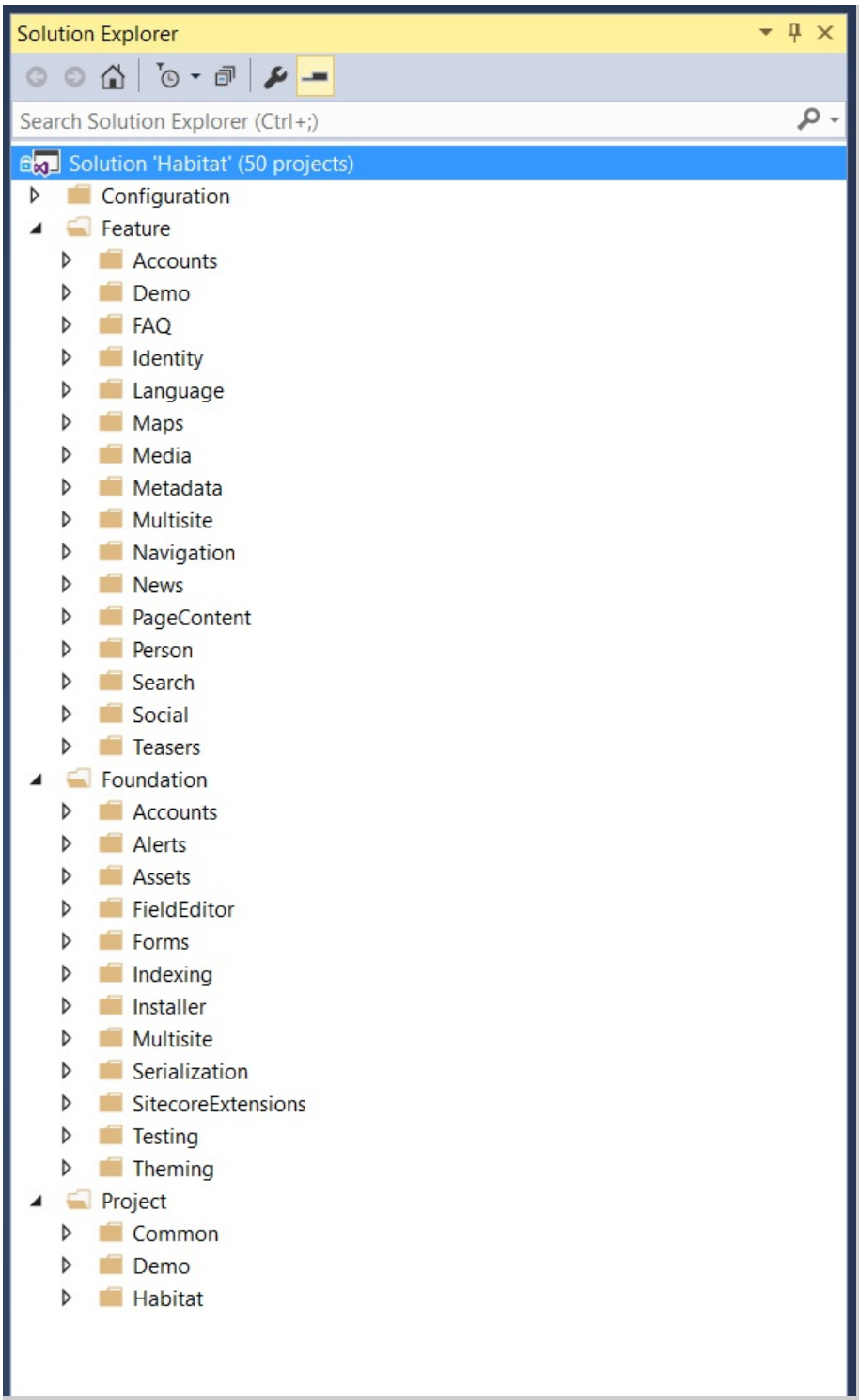


Figure: Habitat Visual Studio Solution

2.2.2. Solution structure

For maximum discoverability, the structure of the Visual Studio solution must represent the layer and module structure, i.e. have solution folders for Project, Feature and Foundation layers and separate solution folders for each module. If there are any additional grouping of modules, these can have their own solution folders.

The VS solution can have other solution folders (beyond Project, Feature and Foundation) in the root of the solution, to increase discoverability for specific files. However, these folders will never represent new layers and must not contain any module projects.

Habitat Example

Figure: Habitat solution with feature and foundation modules

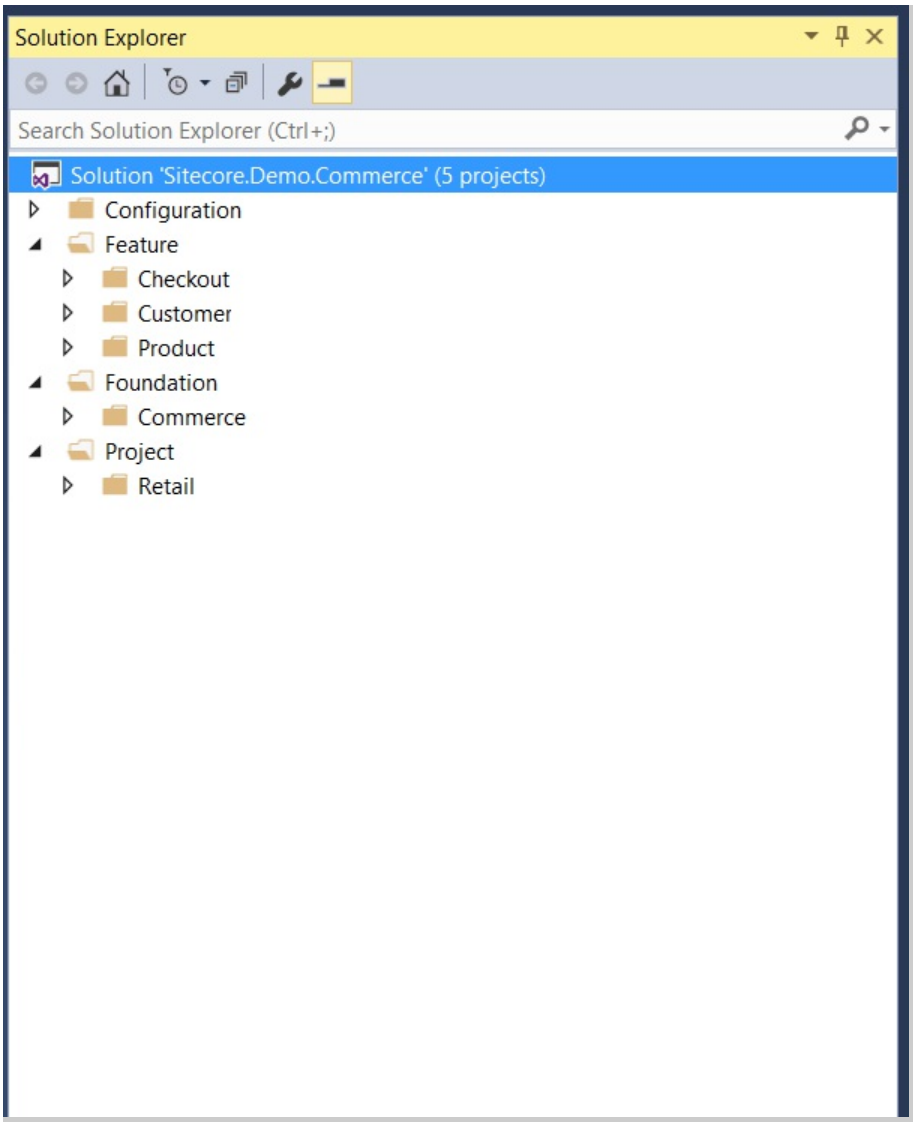


Figure: Demo solution with specific feature and foundation modules

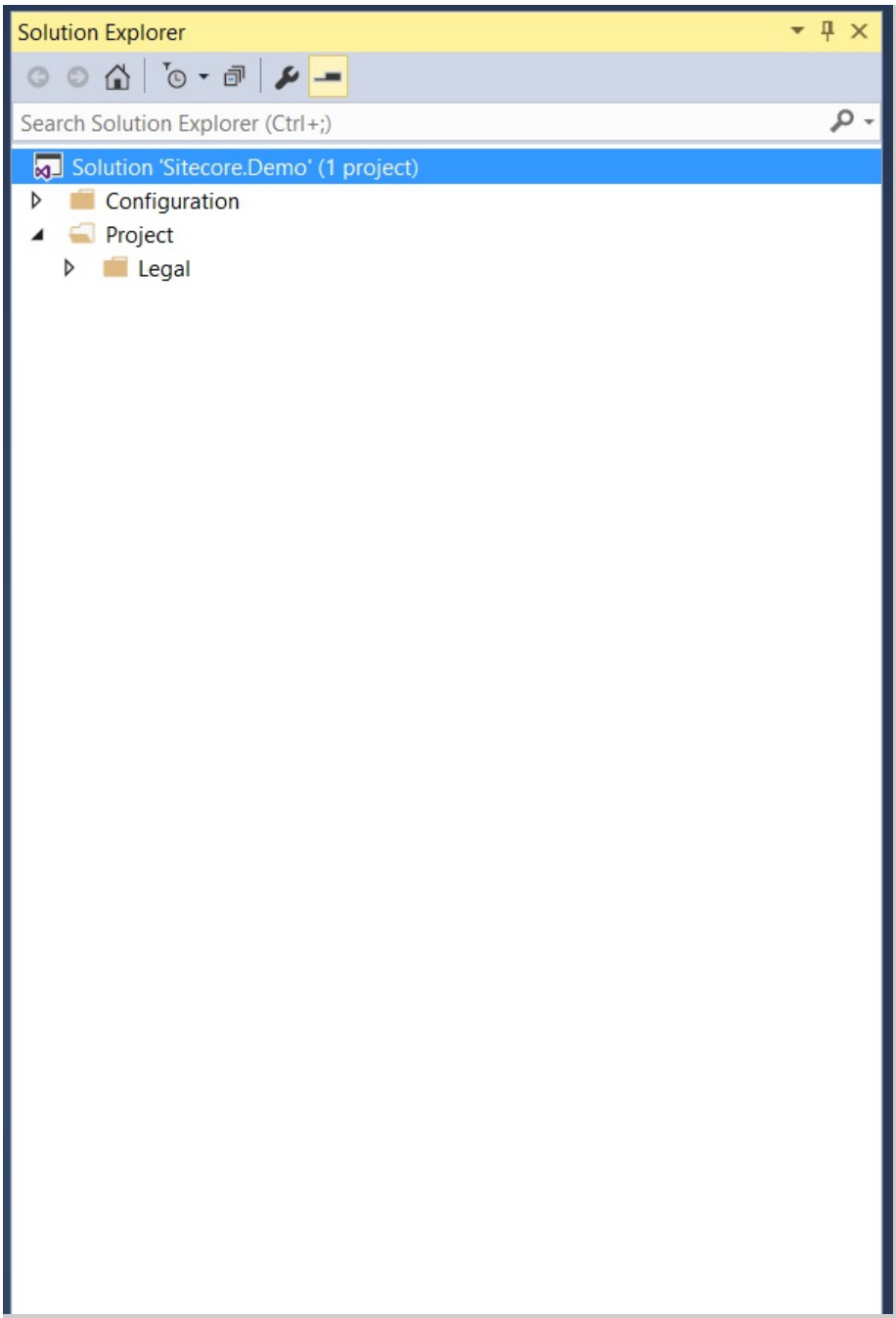


Figure: Simple demo solution without feature or foundation modules

2.2.3. Projects

A Visual Studio solution can host a number of different project types, such as Web Application projects, unit test projects, Team Development for Sitecore (TDS) projects, behavioural testing projects, Xamarin projects, etc., but modules are always grouped by their logical connection to a module – and never by type.

Projects are grouped together in a solution by the layer and module to which they belong.

Habitat Example

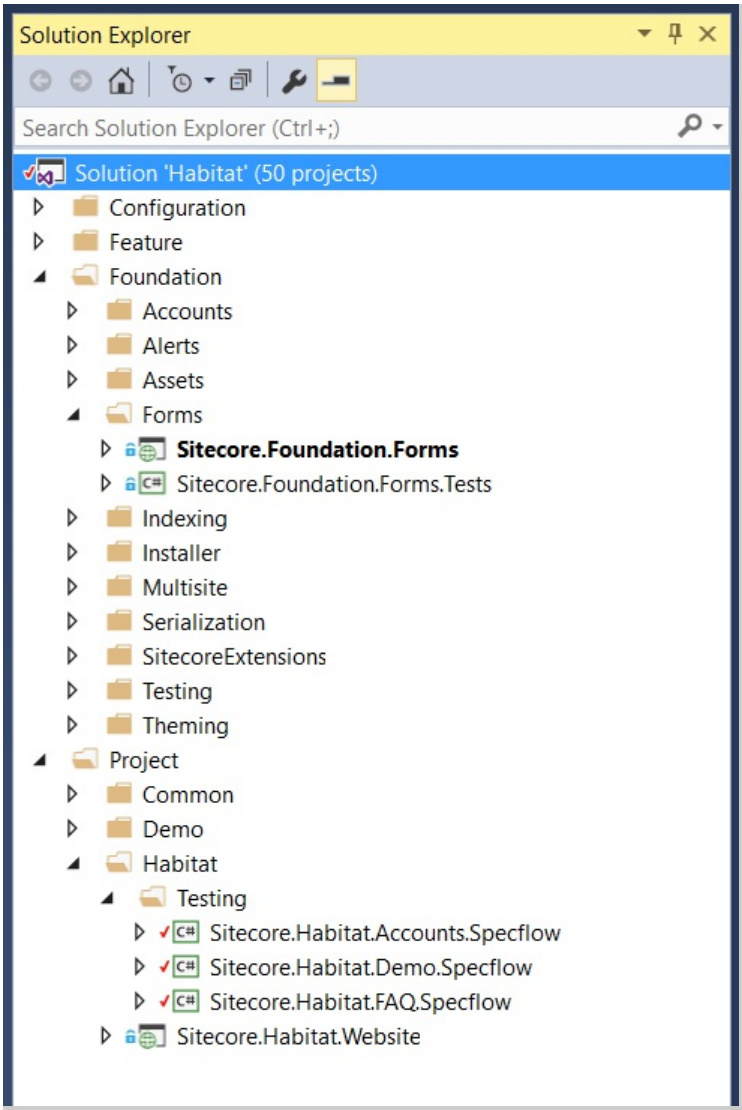


Figure: Project grouping. Please note that even though the SpecFlow (see Integration, Acceptance or other automated testing methods) projects might test the individual features, they are grouped with the Sitecore.Habitat project because they depend on the entire running website, and therefore the Sitecore.Habitat module.

A project, and assembly, should be named in a namespace-like fashion with:

The overall customer, partner or solution name
The layer (optional for project layer modules)
The logical module grouping (optionally)
The module name

The logical function of the project
For example, the following VS project contains the unit tests for the commerce orders feature module:

```
Sitecore.Feature.Commerce.Orders.Tests
```

2.3. File and Disk Structure

A single customer implementation can consist of multiple solutions that are all managed as free standing solutions with references between them.

2.3.1. Solution structure

A single solution has a fixed folder structure on disk:

```
/                                   // Solution root folder, contains th
    /src                            // The root of the solution source (
        /[Feature|Foundation|Project] // Layer folder (ONLY *Project* or *
            /[Module Group]         // Optional logical grouping of modu
                /[Module Name]      // Module folder
    /[…]                            // Other global solution folders, fo
```

2.3.2. Module structure

Each module has the following structure:

```
/…
    /[Module Name]     // Module root folder, named after the module (with
        /code          // Houses the main code for the module, for example
        /serialization // Contains the serialized data from Sitecore, for
        /tests         // Unit or other test types for the module
```

Habitat Example

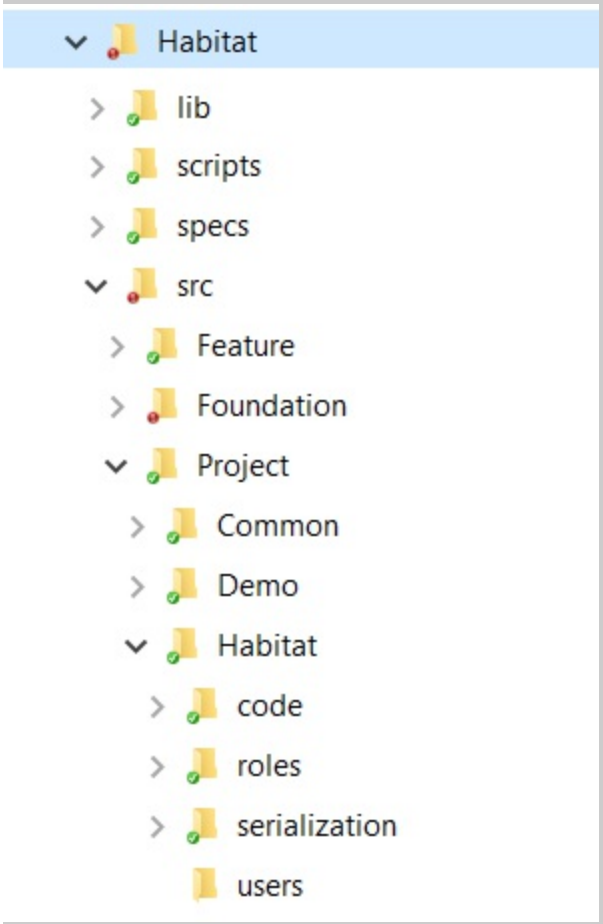


Figure: Solution and module structure on disk

2.4. Managing Sitecore Items

Just as files and Visual Studio projects are a part of our Sitecore implementation, so are Sitecore items.

2.4.1. Item types

The items in your Sitecore databases can be split, for management and governance purposes, into two main categories: *Content Items* and *Definition Items*. The categorization determines the process in the application lifecycle that “owns” the creation and management of the items, as well as in which direction the items flow: from development, through to test, through to production – or the opposite.

Although it is not always 100% possible in practice, the better you can clearly split the Sitecore items in your implementation into these categories, the easier it will be to manage your implementation through development, integration and deployment - as well as new features and upgrades.

2.4.1.1. Definition items

Definition items are items that typically define the configuration or structure of the implementation or which contain metadata for assets in the solutions. These items are owned, i.e. created and managed, in the development environment and moved as part of versioned deployments from development to test to production.

Think for example of a *View Rendering* in Sitecore, which consists of a Razor view file (.cshtml) and an item somewhere under `/sitecore/layout/renderings`. Both these pieces together make up the View Rendering in Sitecore, and they should be managed and versioned together in the development process. If the filename of the view file changes, the filename in the Sitecore item needs to change – and these two pieces need to be deployed in a versioned and consistent process through from development to test to production. The item itself should never be changed by editors in production (just as the .cshtml file should never be changed in production), as the next deployment might overwrite the production change. The View Rendering item in Sitecore is a therefore a Definition item.

These item types are typically Definition items:

Layout and Rendering items
Template and Field items
Placeholder setting items
Custom field types
Lookup items for settings
All items in the Core database

Habitat Example

Note how in the Habitat example sites, all Feature and Foundation modules almost exclusively contain Definition Items. This means that when developing or extending modules developers can know that the items are managed in the development environment and deployed along with their business logic as a part of new versions.

The Project layer modules on the other hand consist of a majority of Content items, which are managed by editors.

2.4.1.2. Content items

Content Items are items which are managed by the editors on the website and contain content that is output on the sites or channels of the website, or that are part of shaping the specific user experience of the pages. Content Items are owned by the production environment, i.e. the editors and administrators.

An example of a Content Item is a site home page item. Although this item is often created very early in the initial development process and deployed into production on the first deploy, the item itself is owned by the production environment and should never be overwritten by an item coming from the development or test environments. On the other hand, it could be very useful to get a snapshot of the Home Page from production back into the test or even development environments for

realistic testing purposes.

Even if Content Items are owned by the production environment, sometimes the business logic will have to know about the specifics of these items, for example their location or type, or the development process might need to change specific data in them – for example, if business logic introduces new settings or fields in a template that require initial values in the content of the sites. Therefore, Content Items can be split into two sub-categories: items that are created in development and deployed once into production, and items that are created and managed in production.

2.4.2. Managing items in development

Since Definition Items and some Content Items in Sitecore are created and managed in the development environment and, given the Common Closure Principle (*“what changes together should live together”*), we need not only to make it obvious which items in Sitecore belong to which modules, but also to manage and version the items alongside the code in these modules.

2.4.2.1. Associating items with modules

Now, for connecting items with modules in Sitecore, we use a basic convention-based approach, where Definition Items are placed in folders according to which the layers and module they belong to. This applies to the items that belong to the typical configuration areas in Sitecore such as:

/sitecore/templates/[*Project*|*Feature*|*Foundation*]/[*Module*]
/sitecore/system/Settings/[*Project*|*Feature*|*Foundation*]/[*Module*]
/sitecore/layout/renderings/[*Project*|*Feature*|*Foundation*]/[*Module*]
/sitecore/layout/layouts/[*Project*|*Feature*|*Foundation*]/[*Module*]
/sitecore/layout/placeholder settings/[*Project*|*Feature*|*Foundation*]/[*Module*]
/sitecore/layout/models/[*Project*|*Feature*|*Foundation*]/[*Module*]

Habitat Example

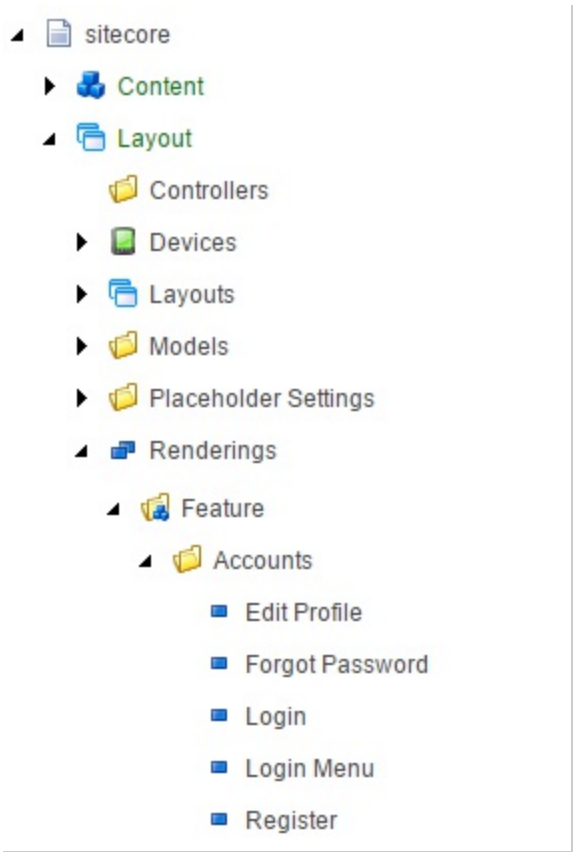


Figure: Rendering items in the Feature/Accounts module

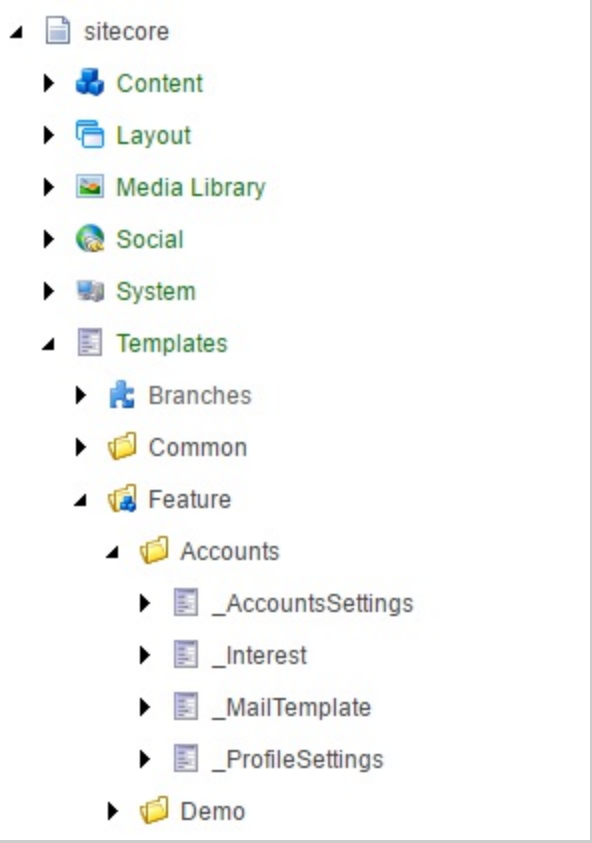


Figure: Template items in the Feature/Accounts module

2.4.2.2. Versioning items in modules

Managing the Definition Items along with the business logic that uses them poses a technical challenge, as the items are physically stored in the Sitecore SQL databases while the business logic in code files are stored on disk. For this challenge, the Sitecore concept of serialization comes in handy.

Serialization allows items in the databases (typically in the Master and Core database) to be written to disk in a text-based format and subsequently restored into a database.

The Helix conventions define that serialized items should be versioned as part of the owning module, next to the code and project files. Place the serialized items on disk in a subfolder beneath the module:

/src/[*Foundation*|*Feature*|*Project*]/[*Module*]/serialization

Sitecore supports serialization through the developer toolbar in the Content Editor, but this mechanism will by default serialize items to a single location on disk – and it therefore not directly compliant with the Helix modular architecture. Serialization is also supported through the package generation tools.

Both these approaches are very manual in nature, and can be inefficient to use with the Helix methodology. It is therefore recommended to look into the third party tools available for item serialization, for example Team Development for Sitecore or Unicorn.

Habitat Example

The default repository for the Habitat example site uses Unicorn for serialization. The Foundation/Serialization module imports the required NuGet packages and configures the module for the other modules in the implementation. Each module then subsequently configures Unicorn to serialize the relevant items into the /serialization subfolder for the module. Please refer to the Unicorn documentation for more information on using Unicorn.

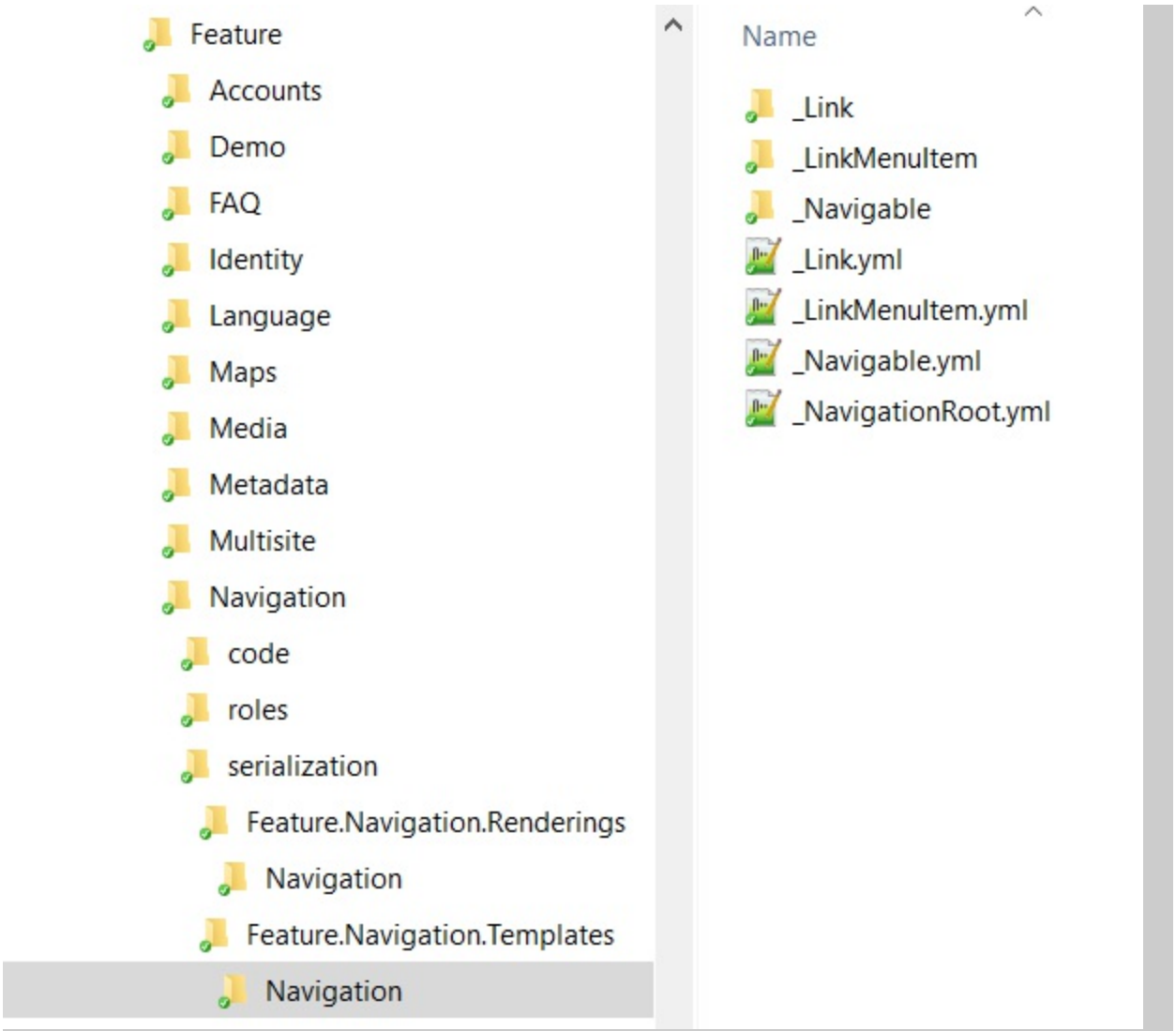


Figure: Serialized items for the Feature/Navigation module

The following shows the Unicorn configuration for the Feature/Navigation module of Habitat

```
<configuration xmlns:patch="http://www.sitecore.net/xmlconfig/">
<sitecore>
    <unicorn>
    <configurations>
        <configuration name="Feature.Navigation" description="Feature Navi
        <targetDataStore physicalRootPath="$(sourceFolder)\feature\navigat
        <predicate type="Unicorn.Predicates.SerializationPresetPredicate,
            <include name="Feature.Navigation.Templates" database="master"
            <include name="Feature.Navigation.Renderings" database="master
        </predicate>
```

```
        <roleDataStore type="Unicorn.Roles.Data.ReverseHierarchyRoleDataSt
        <rolePredicate type="Unicorn.Roles.RolePredicates.ConfigurationRol
            <include domain="modules" pattern="^Feature Navigation .*$" />
        </rolePredicate>
        </configuration>
    </configurations>
    </unicorn>
</sitecore>
</configuration>
```

Habitat Example

Hedgehog Development, the company behind Team Development for Sitecore (TDS), maintains a separate branch for TDS and Habitat. Please refer to this branch, TDS documentation and support for more information on using Helix and Habitat with TDS. https://github.com/HedgehogDevelopment/Habitat/tree/TDS.

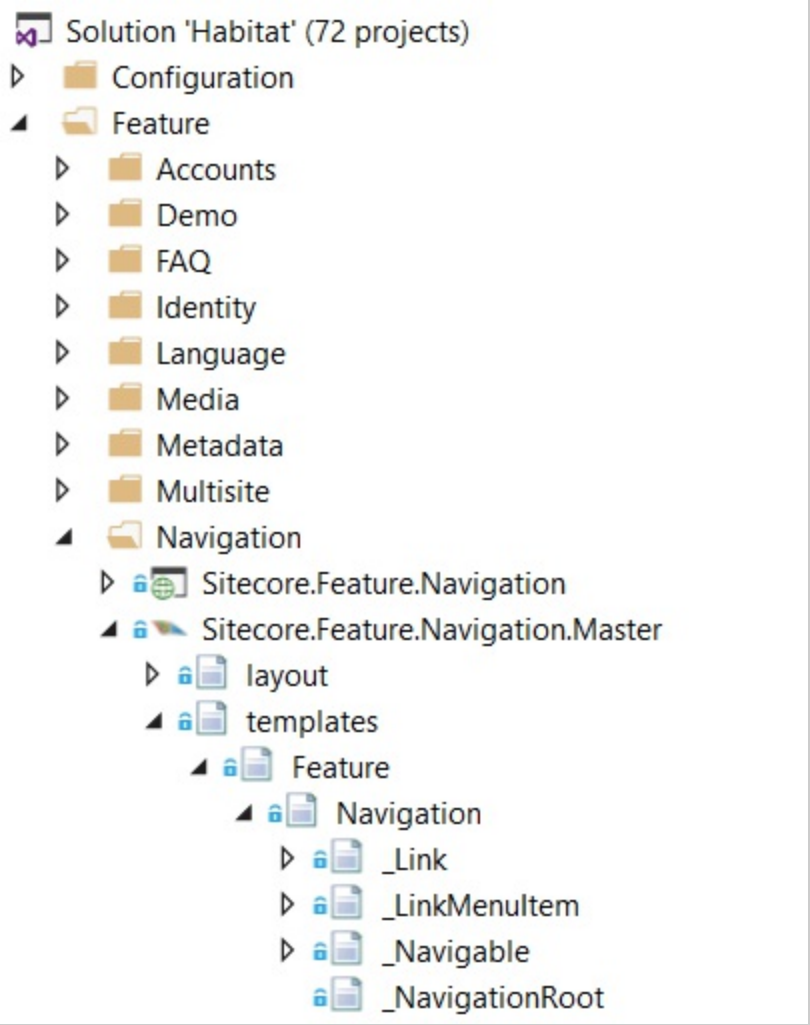


Figure: The Feature/Navigation module using Team Development for Sitecore

2.4.3. Deploying items

Even though the details of integrating and deploying your implementation can be very specific to the business requirements, there are general recommendations to integrating and deploying your items through development to production environments (see Deployment).

Just as it is recommended to version and manage items and business logic together in your development process, it is also highly recommended to deploy business logic and items together in your deployment process.

Given the split between Definition and Content Items, the deployment process can continuously deploy all Definition Items as part of a new version of your implementation. Having this strict approach will help assure that the ownership of Definition Items lies in the development process and that no occasional changes are made in production or test environments.

Note that the modularity of Helix is in no way meant as a general modular approach to deploying and running Sitecore in production. Even if you version and maintain modules separately in the development process, your entire implementation should be integrated and deployed in a single versioned process (See Build and integration).

Habitat Example

Given the nature of the Habitat example site, all Content and Definition Items are versioned and managed in the development process and integrated and deployed as a single package. This is not possible in a real life scenario where content items are managed in production.

Tools such as Team Development for Sitecore can help automate and secure the deployment of all Sitecore items – even Content items – through the environments.

2.5. Templates

Items and fields in Sitecore are used for a number of purposes, the most obvious being to hold content which is presented on the channels and structuring the information architecture for the sites. But items can also be used for other purposes such as configuring the business logic in features or structure the content in with an organisational focus to make maintenance easier. The use of items - and thereby the template on which they are based – helps determines the governance model around them.

Fundamentally it is important that the data structures – i.e. templates and fields - needed by modules are owned by the module itself, as this makes it possible to separate the different modules and the implementation and the logic in them. On the other hand, templates in Sitecore are also the integration point of the pages and sites in Sitecore, bringing together content and presentation into pages and the business information architecture.

The template inheritance technique in Sitecore is instrumental in allowing this separation of concerns in modules: To allow page types to derive from the features required and to allow individual renderings to reference items deriving from the features needed by the renderings. A correct template structure allows this separation while still allowing the content structure and management of the content to be designed with the focus of the owning organisation or specific use case, i.e. with focus on organisation structure, security, languages, etc.

2.5.1. Structure

Given the type-centric nature of Sitecore, templates are all maintained under /sitecore/templates in the Sitecore content tree, but implementation specific templates are maintained as part of a module. Specifically, in the following structure:

/sitecore/templates/[Project|Feature|Foundation]/[*module*]

where module is the short name of the module, i.e. without prefixes.

Habitat Example

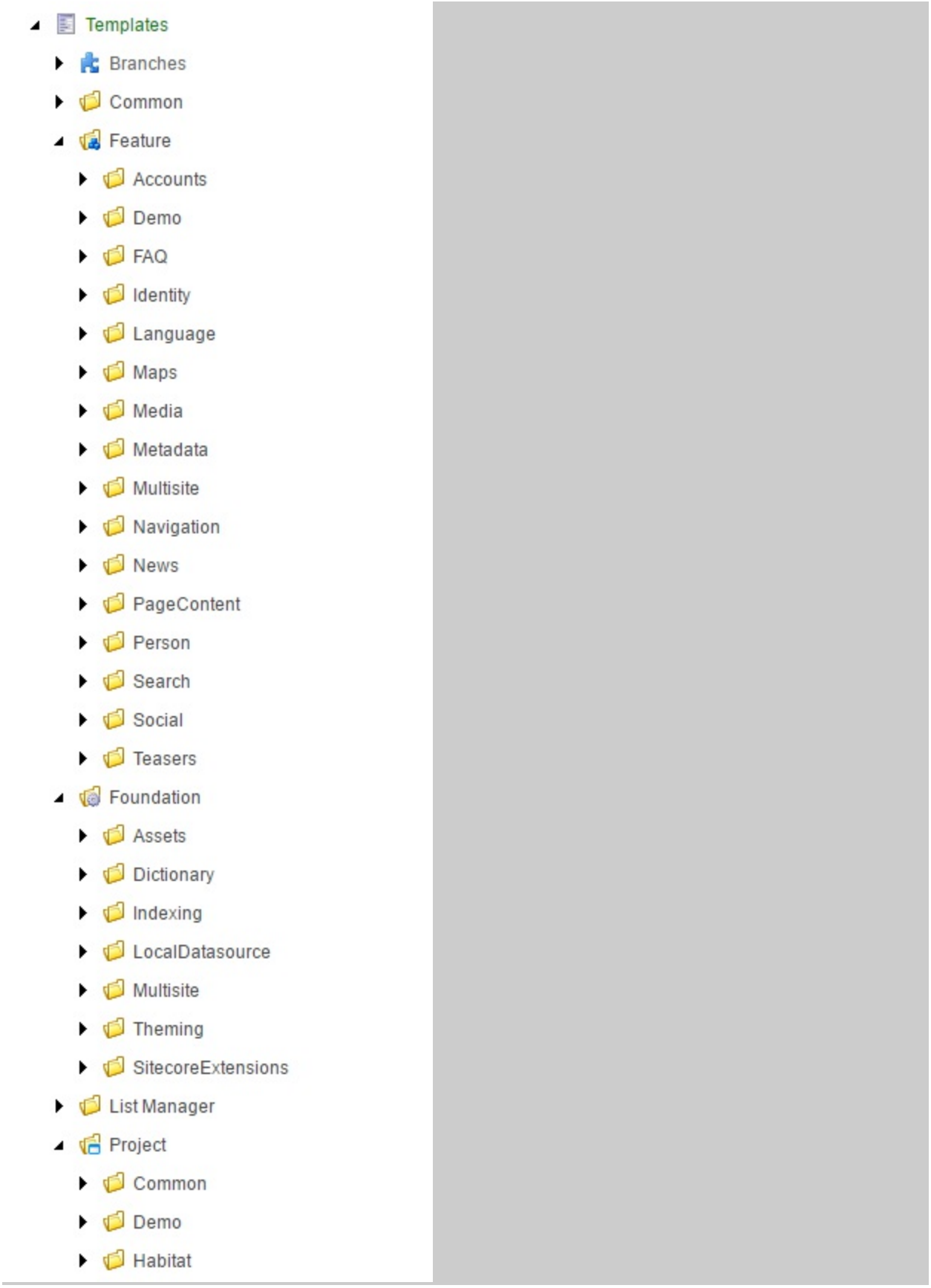

Templates
Branches
Common
Feature
Accounts
Demo
FAQ
Identity
Language
Maps
Media
Metadata
Multisite
Navigation
News
PageContent
Person
Search
Social
Teasers
Foundation
Assets
Dictionary
Indexing
LocalDatasource
Multisite
Theming
SitecoreExtensions
List Manager
Project
Common
Demo
Habitat

Figure: Module specific template folders in Habitat

Since the root layer folders (Project/Feature/Foundation) are not a native part of Sitecore, these folders are created by the Foundation/Serialization module.

2.5.2. Inheritance

In Helix, think of template inheritance as interfaces on an item which exposes a certain set of data to the business logic. This makes it possible for a multiple modules and business logic to share items and content, but simple look at the interface of the item which they understand.

The architecture does not have the concept of a single common base template across all templates – which is a practice that is commonly discouraged as it will often lead to bloated items with unnecessary fields.

Any business logic or views which uses content on an item should therefore only reference the base template of the item – by using a *is* operator approach (for example by using the Item.Template.InheritsFrom) – which allows for much less coupled feature modules and a freer content architecture.

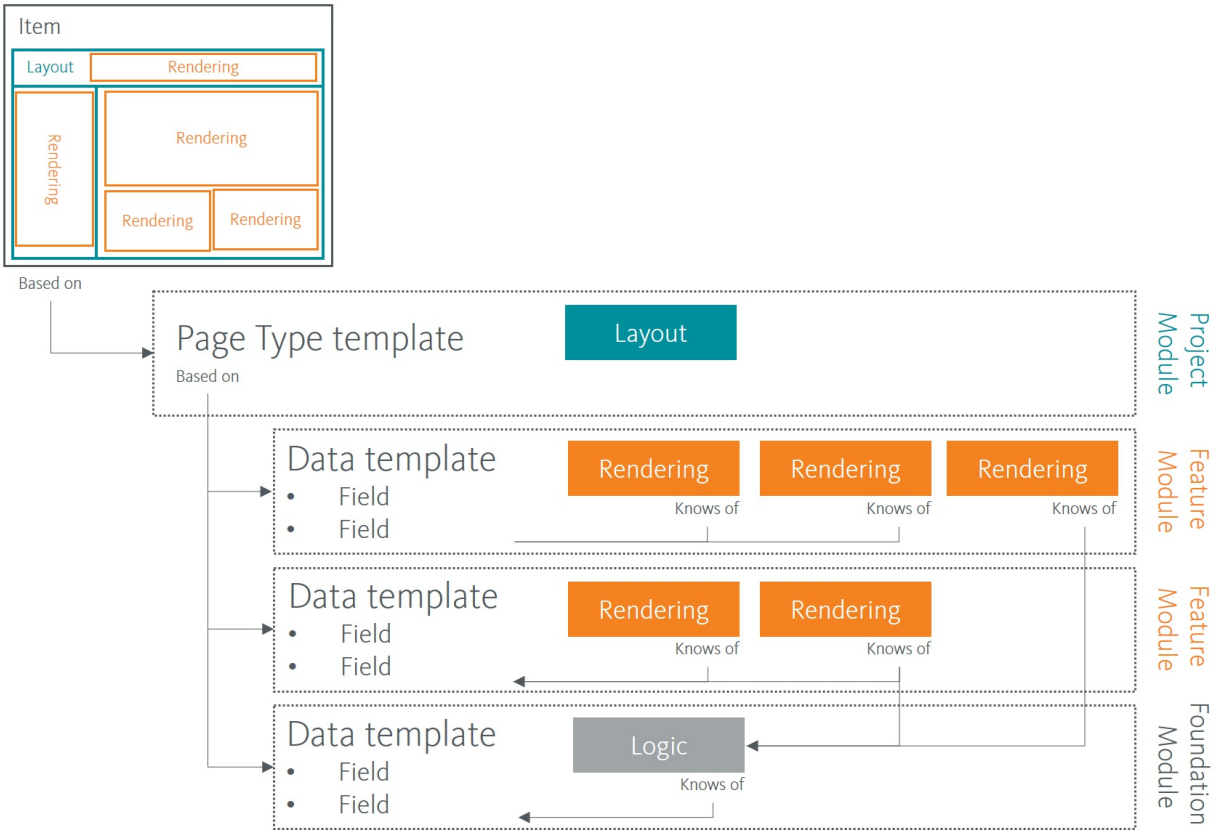


Figure: Template inheritance

This figure shows how the architecture allows business and presentation logic, i.e. renderings, and other logic to recognize the templates and fields they require while at the same time allowing pages and page types to be designed with the specific presentation and organisational governance model in mind. In other words, renderings and business logic can be written with sole focus on the domain of the specific feature and oblivious to the other features or logic in the implementation.

2.5.3. Template types

Now, to separate the purposes of the templates in the solution, we need to define a number of different logical template types. These templates types are not represented as such in Sitecore but rather differentiated by the naming, where they are defined and how they are used in the system.

Interface template
Defines an interface for solution logic to work against, for example by defining the fields that are used by a module's logic or by simply being a template in the template inheritance hierarchy of an item. Can also be referred to as a base template.

Page type template
Makes up the pages of the website. Derives from interface templates and has a presentation layout.

Datasource template
Items from this template are referenced by renderings as a datasource. Derives from interface templates but has no associated layout.

Settings template
Used for lookup items for fields or to hold fields to configure the business logic in modules.

Folder templates
These templates are used for the folder items that make up the scaffolding of the content structure.

Template Type	Can have a page layout?	Exists in which layers
Interface template	No	Feature or Foundation
Page type template	**Yes**	Project

Datasource template	No	Project
Settings template	No	Feature or Foundation
Folder templates	No	All

2.5.3.1. Interface templates

Interface Templates are maintained in the Feature and Foundation modules and form the base of the content for the solution. They are never instantiated but are solely used as base templates for Page Type Templates or Data Source templates. The primary reason for this is that page items require a layout definition, and a layout definition inherently contains references to renderings in other features, which means that templates defined in one feature will start referencing other features, thus breaking the architectural principles, creating dependencies and ultimately leading to higher coupling and less flexibility and stability in the solution (see Dependencies).

Interface templates are prefixed with an underscore (_) to signify that they are interface templates and cannot be instantiated as items.

Interface templates can have an associated Standard Values item – given that the module in which they are created wants to set universal standards for the fields. Be aware that this standard value is universal and will be applied across all inheriting templates, sites and tenants.

Habitat Example

Examples of Interface Templates in Habitat are present in almost all Feature modules, for example in the Sitecore.Feature.Navigation module where the _Navigable template makes a page part of the navigation structure of the site.

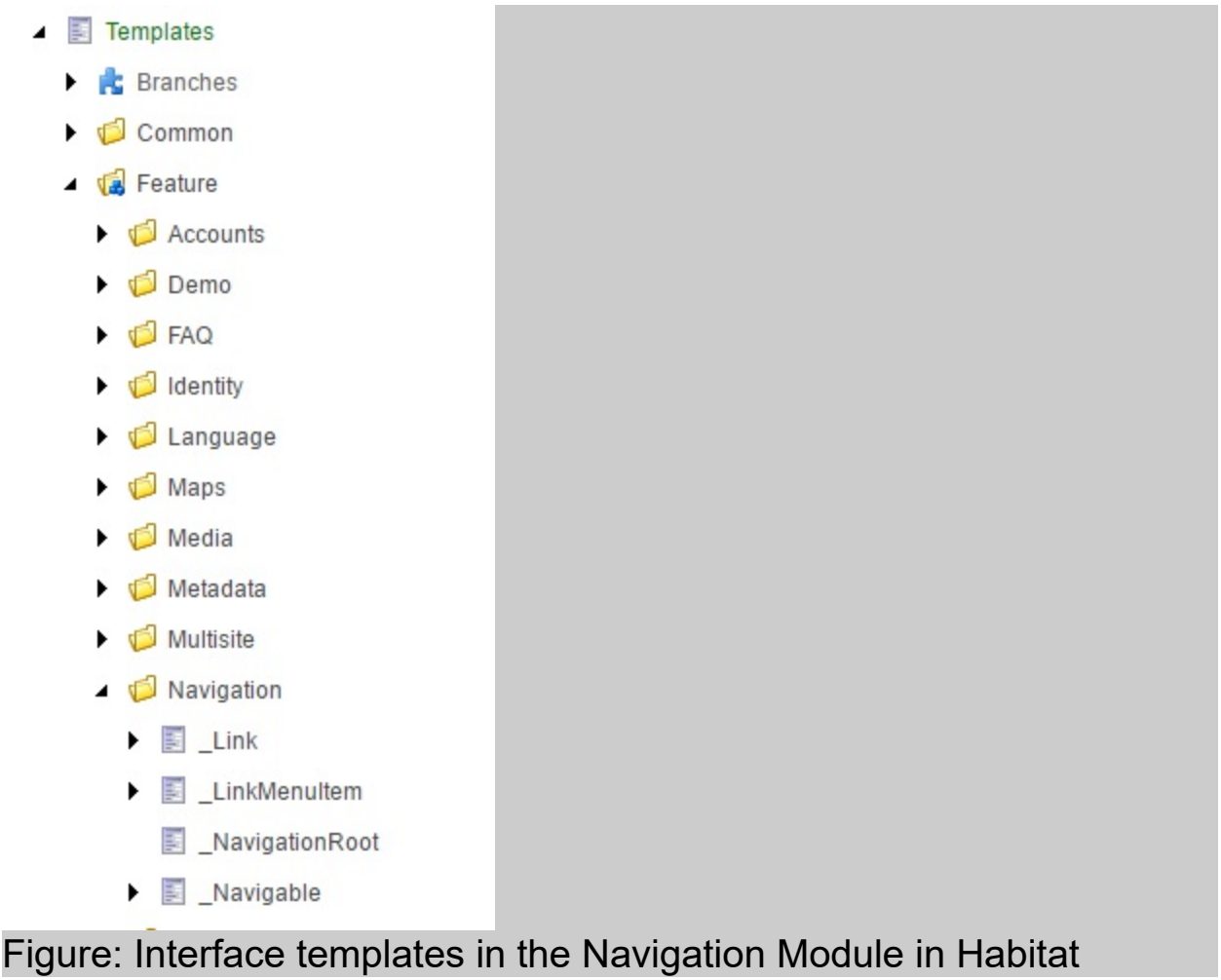


Figure: Interface templates in the Navigation Module in Habitat

2.5.3.1.1. Fields

The content fields for business and presentation logic all live in Interface Templates – never in Page Type templates or Datasource Templates. Therefore, these content fields live with the logic that use them (see References from code).

Since fields and templates should always be referred to by ID and not by name (see References from code), try to be as descriptive and editor friendly as possible when choosing a field name. Even though the item used to define a field has a Title field that can be used to specify an editor-friendly name for the field, it is common that this configuration option is not used. This means that the actual fieldname is presented to the editor. Therefore, avoid CamelCasing or

AbbevInFldNames as this is not considered editor friendly – in other words, always prioritize the editor experience over technical considerations.

See more about fields and language support in Language and culture support.

2.5.3.2. Page Type Templates

Because of the flexible nature of the page layout in Sitecore, it can be challenging to determine how many actual page types a site will require. Theoretically a site could consist of a single page type, but with enough flexibility in the layout and datasources to cater for any type of page or content. This will greatly increase flexibility for the site editor, but would also require more work when adding simple content pages such as a news article. Alternately you could also configure a very elaborate set of page types with all necessary permutations of content and layouts, thereby making it easy for editors to create and edit new pages and perhaps also increasing consistency in the overall look and feel of the website.

Although there is no fixed result, the best option often lies somewhere in between these two examples. A good rule of thumb is to create page types based on the maturity and workflow of the editors, the information architecture of the websites, as well as the main entities of content. Page Type Templates are mainly used to assert consistency in the pages, to facilitate management, and to force stringent information architecture by defining insert options.

For example, it might be beneficial to define page types for the home pages, sections, campaign landing pages etc. of the website in order to define and maintain the overall hierarchy of the website. The leaf nodes of the website will most likely be Article or common Content Pages, or some of the more specialized News, Event, Product, etc. pages.

Furthermore, by defining very specialized pages, such as login, registration, forgot password etc., as actual page types as opposed to generic service pages, it can make it easier to maintain a consistent look and feel for these pages with the other page types.

The architecture of Helix makes this kind of flexibility possible by allowing you to define your page type and datasource structure independently of the features, and the primary instrument in this flexibility in template inheritance.

Interface templates and template inheritance in Sitecore can be compared to multi-class inheritance object-oriented programming, such as C++. In this type of programming, classes define the data they need and the business logic to manipulate or present it. Another class can then derive from multiple classes to inherit multiple capabilities from them.

In the same manner, a Page Type Template in Sitecore can derive from multiple Interface Templates to inherit their data and use the renderings (business logic) in its layout.

Page Type templates are only ever present in the Project layer, as these are the integration points for the functionality in feature and foundation modules. Page Type Templates are therefore maintained in a common folder for a Project, equivalent to a site type. Each page in a site of the given Project type are instances of a Page Type template. This is actually very handy as all page types of a site are maintained in a single location, which can make it easier to manage site-wide changes to all page types.

Page Type templates typically have standard values that set up the page type for all sites of the given project.

Page Type templates typically never have fields since there is never any feature-specific business logic in the project layer that can leverage these fields. These templates will get their content fields from

the Interface templates from which they derive.

Habitat Example

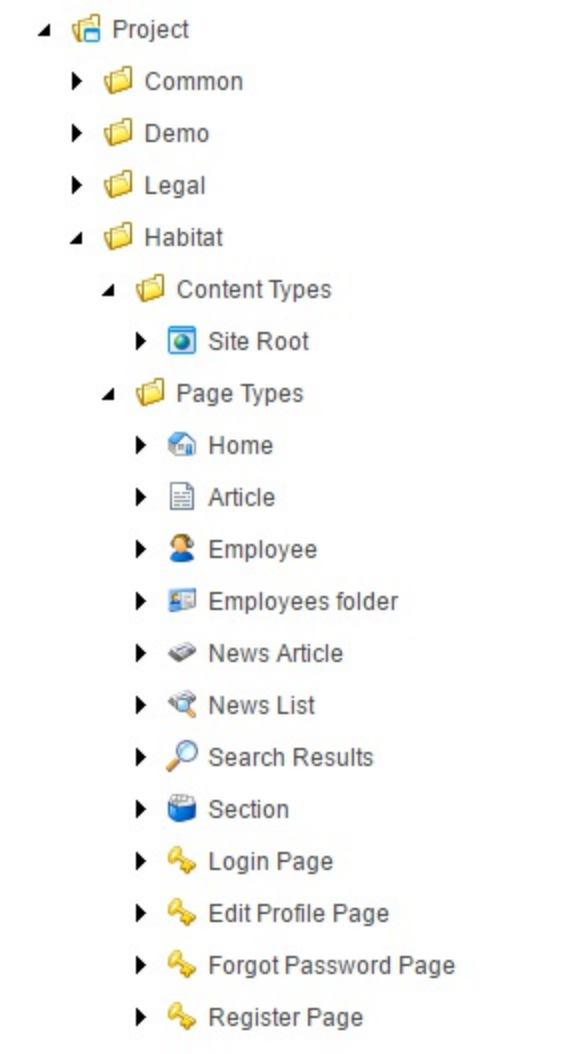


Figure: Page Type Templates of the Habitat website

2.5.3.3. Datasource template

Datasource templates are similar to Page Type templates in that they derive from Interface templates for their content. Datasource templates however do not have any renderings and are therefore used for items that are not part of the page or navigation structure of the website.

Like Page Type Templates, Datasource templates live only in the Project layer and typically do not have fields themselves.

Habitat Example

Because of the multi-site/multi-tenant nature of the Habitat project, the Datasource templates in Habitat are maintained in the Common Project layer module. This allows multiple project layer modules and sites (such as the Habitat site) to use these templates. This does not however stop a Project layer module from overriding one of the Common Datasource Templates and adding more or another functionality to it.

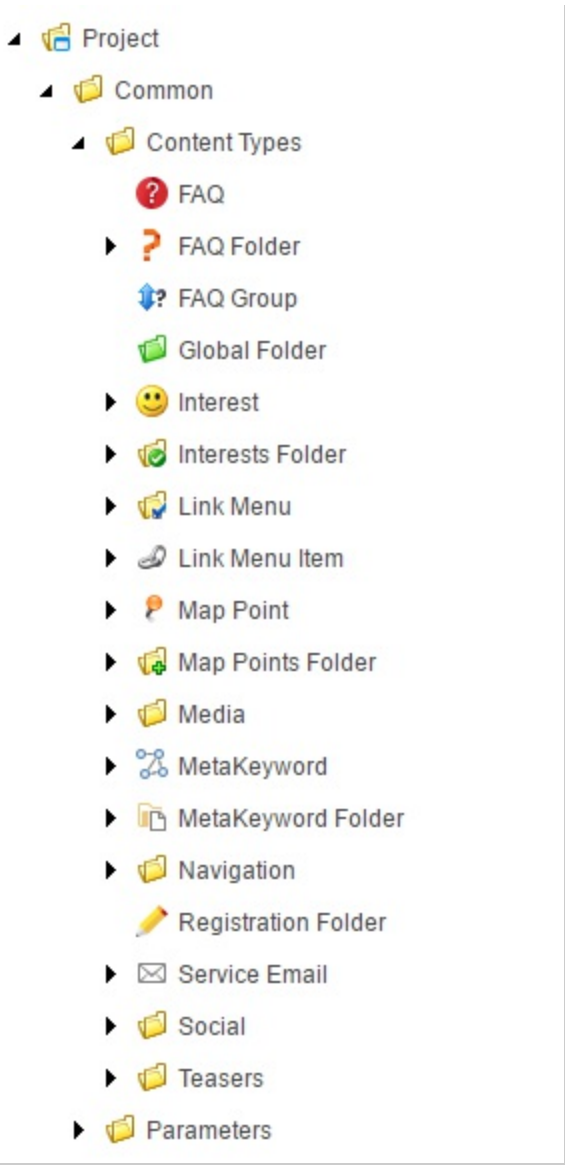


Figure: Datasource Templates in Habitat managed in the Project.Common module

Given the loose coupling and inheritance structure of the templates, renderings typically are unaware of whether the context item they are rendering is a Datasource or Page Type template. This allows the functionality of feature modules to be used in a wide variety of ways across Project layer modules.

2.5.3.4. Settings templates

Settings templates can be managed in all business logic modules, for example Foundation and Feature layer modules, and are for any configuration settings (global or site-specific) needed by the module. Unlike Interface Templates, these templates are often not used by Project Layer modules as base templates, but are instantiated in the content tree (or under /sitecore/system) directly from the template defined in the module.

Depending on how dynamic the configuration needs to be, settings can be single items with predefined template with fields for the configuration data – or it can be an item structure where each item under a setting root holds the configuration settings, like key/value pairs.

To avoid coupling and to encourage greater flexibility, avoid reusing Settings templates across modules. Each module should define its own Settings templates and settings structure, even if two modules use the same technique for settings (for example Key/Value pairs).

Good Helix practice is to store global implementation-wide settings under /sitecore/system/settings/*[Layer]*, as some settings can be confusing for the low maturity editor personas - but always carefully consider the maturity of the user managing the settings before deciding. For example, /sitecore/system is not generally available to the average, low maturity, editor and thus settings which are managed by this persona should be placed as close to the content as possible, for example under /sitecore/content/settings.

If there are settings that are tenant or site specific, they should be stored either under a site or tenant related item under /sitecore/system/settings or directly under the tenant or site in the content tree.

It is also considered good practice to allow low maturity editors to use the Experience Editor to manage the entire experience – including settings. Consider implementing feature specific Experience Editor

extensions to allow for this.

Avoid storing environment specific settings in Sitecore. In order to move Sitecore content items freely between environments (for example on deployment to production or when testing with product data) these settings should reside in for example .config files. If there is a need for administrators to manage these types of settings, settings in Sitecore can point to different .config file settings for example <connectionStrings> or <sitecore><settings>.

Habitat example

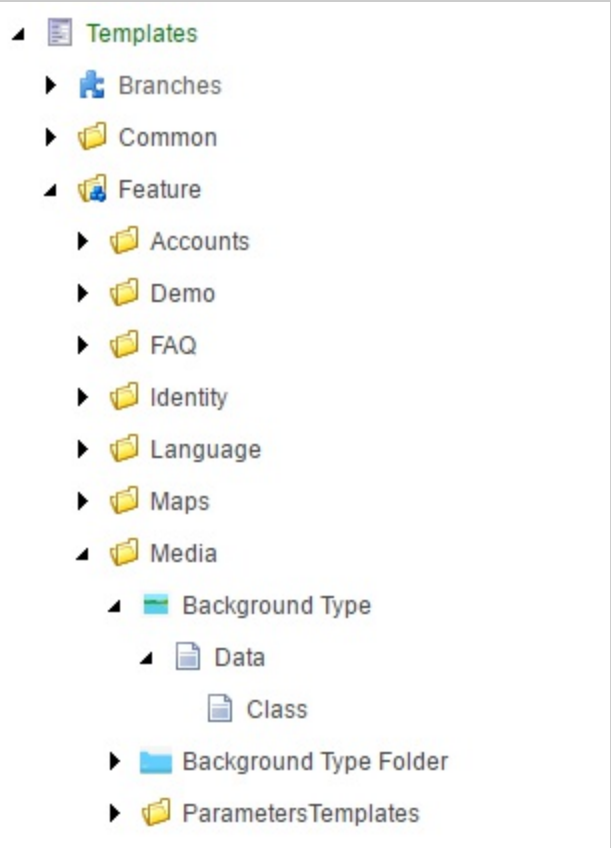


Figure: The background type Settings Templates for the Media Feature module in Habitat

2.5.3.5. Folder templates

Folder templates are the templates that make up the content structure

outside the actual website structure, for example in datasource repositories, settings, etc.

Avoid using the Folder template provided with Sitecore (/sitecore/templates/common/folder). Instead, have each module define its own folder templates. This will allow greater flexibility for example in insert options and the content structure and provide better user friendliness, for example, in icons.

Habitat example

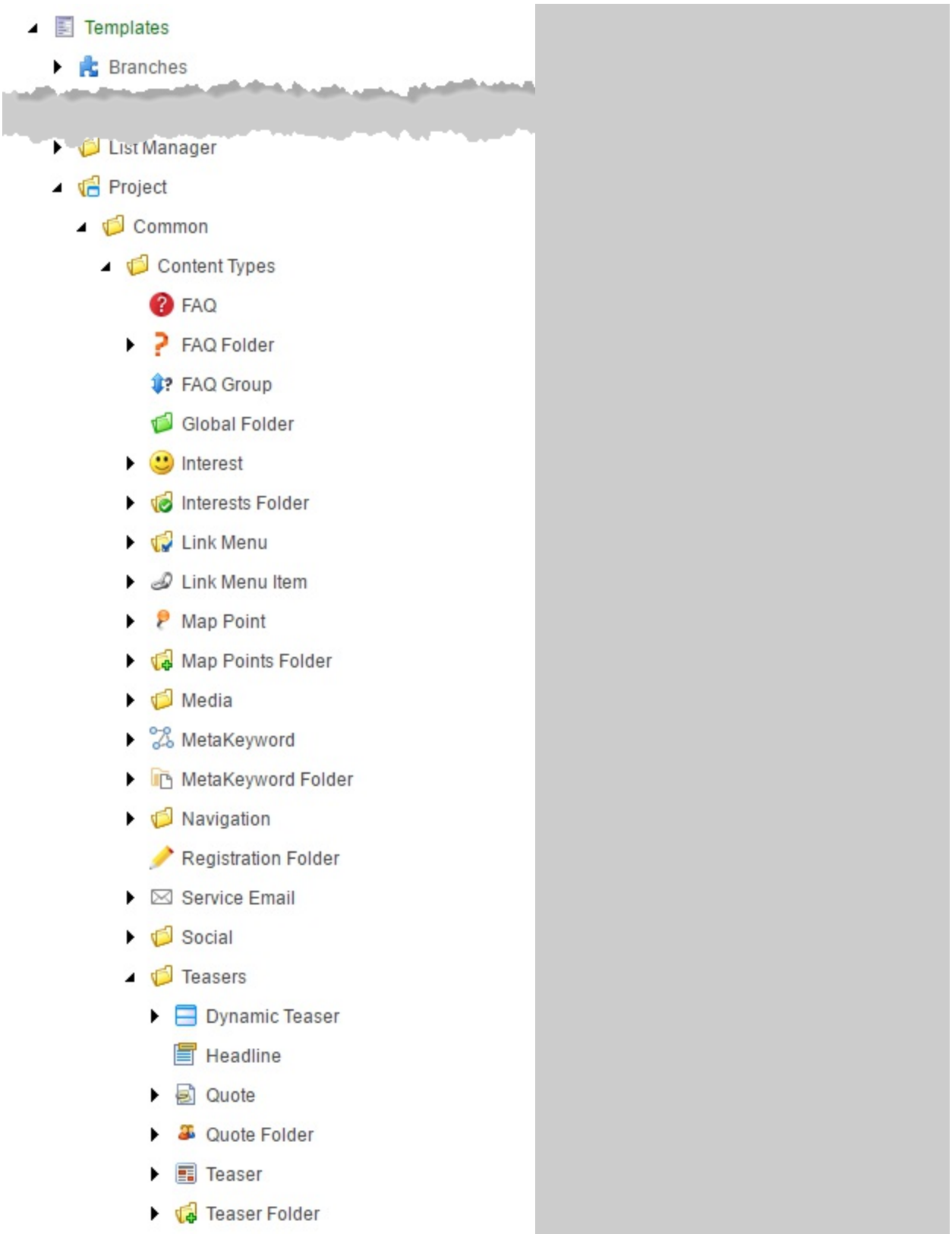


Figure: The datasource folder templates defined in the Common project in Habitat

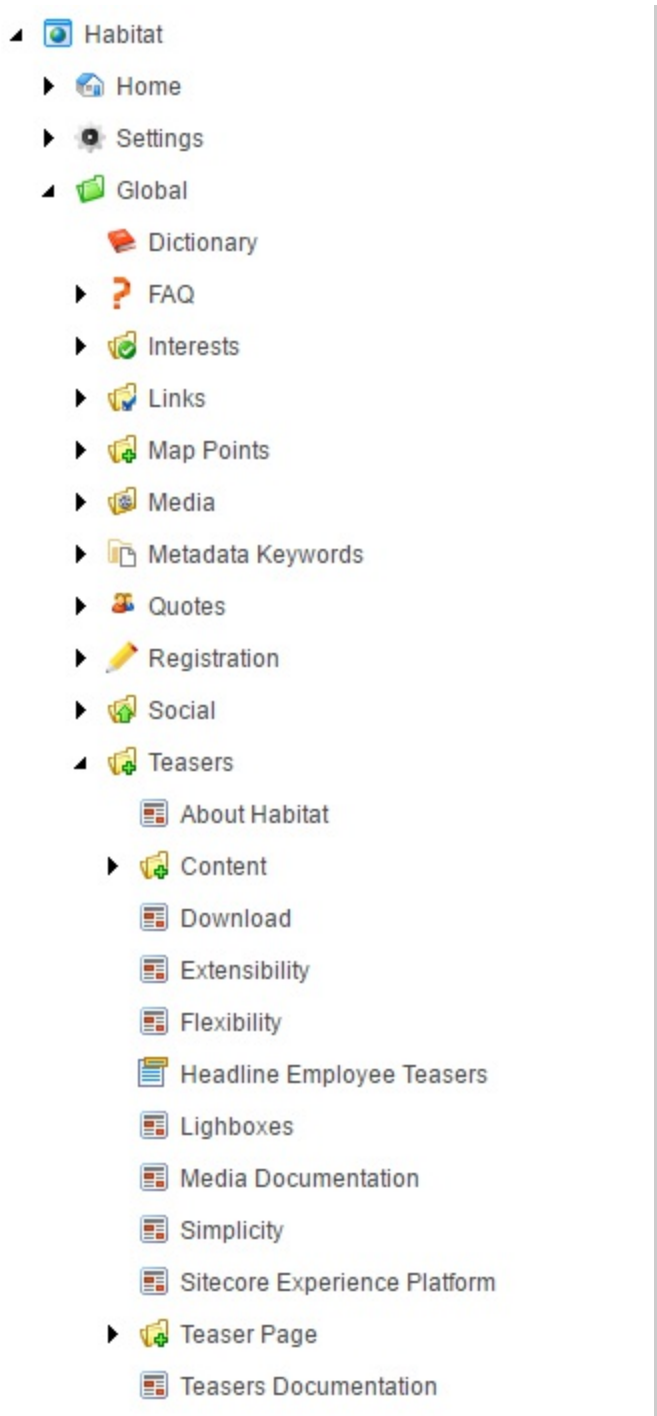


Figure: The Habitat datasource repository using different folder templates

2.5.3.6. Rendering parameters templates

In Sitecore, templates used for rendering parameters must derive from

the Standard Rendering Parameters (as opposed to the Standard Template).

In Helix rendering parameters templates should be prefixed with ParametersTemplate_ to distinguish them from the other template types.

2.5.4. References from code

In Sitecore solutions, references to templates and fields are among the most frequent sources of rogue dependencies across modules (see Dependencies). Therefore, maintaining tight control over these dependencies is important in a Sitecore project.

Templates and fields should always be referenced by ID and never by name. This makes it easier to have editor-friendly names or to change field names if needed. Furthermore, hardcoded GUIDs tend to stick out in code and views and therefore makes it easier to detect and avoid implicit dependencies to templates and fields.

Define constants for a module's templates in a single struct named Templates. This struct is located in the root namespace for the module. This makes it easy to explicitly reference a template in the business logic or views and makes it easier to discover references to a template or field. The conventions define this as structs to clearly signal the Templates type's unique function as a constants holder only.

Each Templates struct should have a nested struct for each template which each contains an ID member and a nested struct, Fields, which contains all fields in the template.

Habitat Example

The following shows the Templates struct for Sitecore.Feature.Navigation:

```
namespace Sitecore.Feature.Navigation
{
    using Sitecore.Data;

    public struct Templates
    {
```

```
        public struct NavigationRoot
        {
            public static readonly ID ID = new ID("{F9F4FC05-98D0-4C62-860
        }

        public struct Navigable
        {
            public static readonly ID ID = new ID("{A1CBA309-D22B-46D5-80F

            public struct Fields
            {
                public static readonly ID ShowInNavigation = new ID("{5585
                public static readonly ID NavigationTitle = new ID("{1B483
            }
        }

        public struct Link
        {
            public static readonly ID ID = new ID("{A16B74E9-01B8-439C-B44

            public struct Fields
            {
                public static readonly ID Link = new ID("{FE71C30E-F07D-40
            }
        }

        public struct LinkMenuItem
        {
            public static readonly ID ID = new ID("{18BAF6B0-E0D6-4CCE-918

            public struct Fields
            {
                public static readonly ID Icon = new ID("{2C24649E-4460-41
                public static readonly ID DividerBefore = new ID("{4231CD6
            }
        }
    }
}
```

A templates class should never define constants for templates that are not created in the module itself. If a module needs to reference a template or field in another module, it should reference the Templates struct in that module.

The practice of referencing different fields across modules by their shared name – an equivalence to *duck typing* – is discouraged.

Note

Duck Typing

“If it walks like a duck and it quacks like a duck, then it is a duck”

https://en.wikipedia.org/wiki/Duck_typing

An often seen Sitecore example of this is to define that any field named “Title” is used as the header of the page. The problem of this is that “Title” might have different connotations for different content types. For an Article, the title is typical headline of the text – and can be used as the header for the page, but for a person, the Title is typically their job title and might not make sense as the header of the page. If modules need to share data, consider using design patterns such as providers or pipelines to allow one feature to inject content into another feature.

Since template inheritance in Sitecore can be compared to class inheritance object-oriented programming, it is important that, when querying an item’s template, you not use the *equals* operator but rather an *is* operator.

Habitat Example

The *is* operator is not native to Sitecore items and templates. Sitecore Habitat introduces the IsDerived extension method for the Item class. This method is located in the Sitecore.Foundation.SitecoreExtensions.Extensions.ItemExtensions class.

2.6. Page layout

The dynamic layout engine of Sitecore is one of most significant features of the system. When designing the user experience or information architecture of a Sitecore powered website, this should be kept in mind.

Many of the unique tools and features in the Sitecore platform, such as the Experience Editor with built in personalization and MV testing capabilities, rely on the dynamic layout engine and the definition of Layouts, Renderings and Placeholders.

The balance between flexibility – allowing the editors to dynamically construct the pages by placing feature renderings in placeholders – and consistency – having a coherent visual design and user experience across pages – needs to be found in all implementations. Swing too far in one direction and the user experience may be hurt, swing too far in the other direction and the business might end up with a rigid and inflexible solution that frustrates users, or lessens the return of investment by not supporting the full capabilities of the Sitecore platform.

2.6.1. Layouts and sub layouts

Sitecore Layouts and Sub-layouts are in Helix primarily used to structure the pages with the outer HTML mark-up.

Avoid statically binding renderings in sub-layouts, but rather bind all renderings to layouts via layout definitions and placeholders. Static binding will make the page and solution structure less flexible and introduces multiple maintenance methodologies. Although you might end up with longer lists of renderings in your layout definitions, the centralised Page Type templates (see Template types) and single layout management methodology will prove more maintainable and flexible in the long run.

Habitat Example

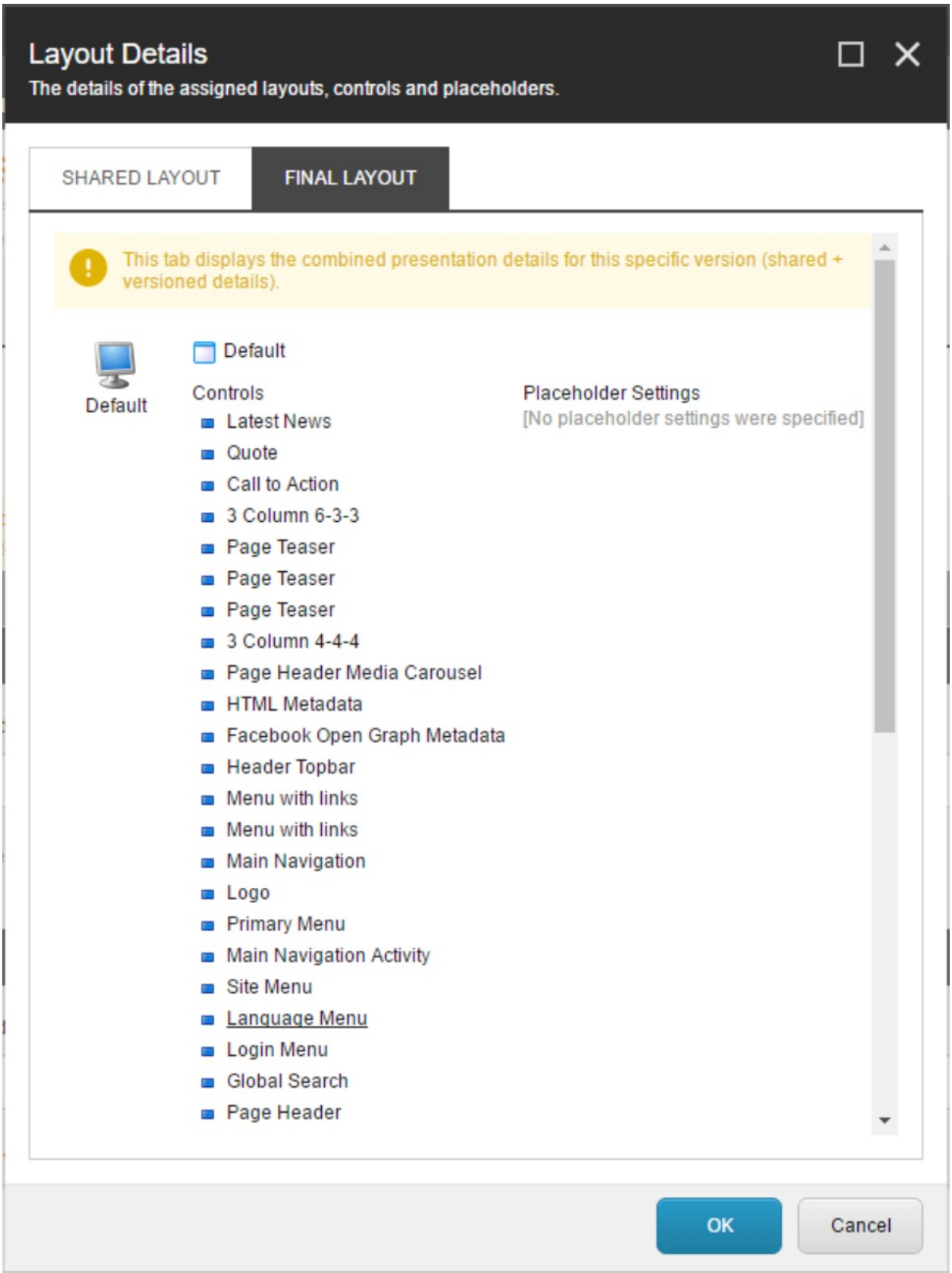


Figure: Habitat home page layout definition

As layouts and sub-layouts (in MVC defined as View Renderings) typically control the overall page design and therefore contain very site or project specific mark-up, they belong in Project layer modules.

Reduce the number of layouts and sub-layouts by dynamically assembling the pages in layout definitions and controlling assets such as scripts and CSS on pages using asset management techniques (see Visual Design and Theming). In some scenarios, for example when implementing adaptive design using the Devices in Sitecore, multiple layouts can be required, but in most scenarios a single layout per site type or project module is sufficient.

Having elastic page layouts, i.e. page layouts that at editing time can vary in structure, can reduce the need for sub-layouts quite considerably and should be considered for all projects. Elastic layouts are typically achieved through the use of dynamic placeholders, i.e. allowing the number of placeholders of a page to vary and thereby allowing an editor to construct page variations at will. Note that the Sitecore Experience Platform does not provide support for dynamic placeholders out of the box, but there are multiple variations available as open source or through the Sitecore marketplace, for example https://marketplace.sitecore.net/en/Modules/I/Integrated_Dynamic_Pla

Habitat Example

The Sitecore Habitat example site contains only a single layout for all page types – defined in the Habitat Project layer module - and few sub-layouts needed for consistent headers and footers across pages – defined in the Common Project layer module.

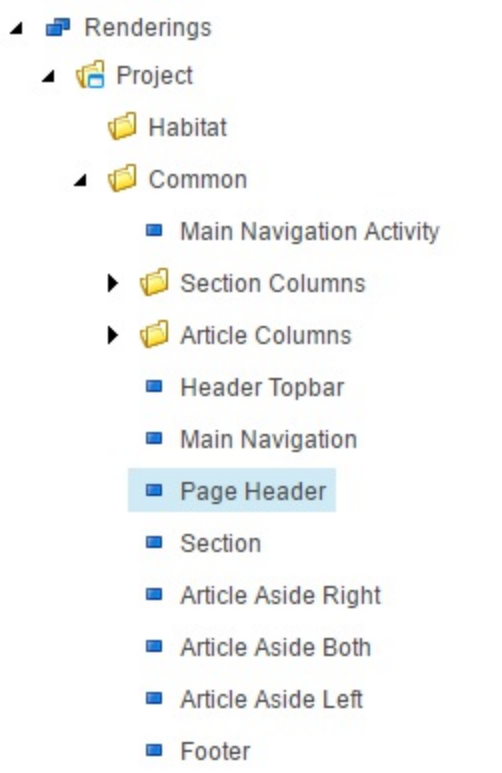


Figure: Habitat sub-layouts defined as View Renderings in the Project/Common module

```
@using Sitecore.Foundation.Assets
@using Sitecore.Foundation.Assets.Models
@using Sitecore.Foundation.SitecoreExtensions.Extensions
@using Sitecore.Mvc.Analytics.Extensions
@inherits WebViewPage
@{
    Layout = null;
}
<!DOCTYPE html>
<!--[if IE 9]><html lang="en" class="ie9 no-js"><![endif]-->
<!--[if !IE]><!-->
<html lang="@Sitecore.Context.Language.CultureInfo.TwoLetterISOLanguageNam
<!--<![endif]-->
    <head>
        <meta charset="utf-8" />
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta http-equiv="Content-type" content="text/html; charset=utf-8"
        <meta content="width=device-width, initial-scale=1.0" name="viewpo
        @if (!Sitecore.Context.PageMode.IsExperienceEditor)
        {
            @Html.Sitecore().Placeholder("head")
```

```
        }
        <!-- Latest compiled and minified JavaScript -->
        @RenderAssetsService.Current.RenderScript(ScriptLocation.Head)
        @RenderAssetsService.Current.RenderStyles()
        @Html.Sitecore().VisitorIdentification()
    </head>
    <body class="header-static @(Sitecore.Context.PageMode.IsNormal ? "" :
        <div id="main-container">
            <header class="header-static">
                @Html.Sitecore().Placeholder("header-top")

                @Html.Sitecore().Placeholder("navbar")
            </header>
            <main role="main">
                @Html.Sitecore().Placeholder("page-layout")
            </main>
            <footer>
                @Html.Sitecore().Placeholder("footer")
            </footer>

            @Html.Sitecore().Placeholder("page-sidebar")
        </div>
        @RenderAssetsService.Current.RenderScript(ScriptLocation.Body)
    </body>
</html>
```

The Habitat example site uses the DynamicPlaceholders.Mvc package available through NuGet (https://www.nuget.org/packages/DynamicPlaceholders.Mvc/) to allow elastic page layouts.

2.6.2. Renderings

Renderings, whether they are Controller or View Renderings typically belong to the *Feature* layer modules as they are connected to business features in the solution.

Only sub-layout style renderings, i.e. view renderings with no business logic but purely mark-up structure and placeholders, should occur in the *Project* layer. Any controller renderings in project modules are typically an example of business logic creeping into the project layer, which should always be avoided.

In rare occasions renderings can occur in the *Foundation* layer, although this should generally be avoided. Examples could be a rendering which purely output metadata on the page or is only executing business logic.

2.6.2.1. Design for simplicity

When designing your business-centric feature modules, keep in mind that you are designing and implementing for maintainability and simplicity, not for reusability. This means that you should keep focus on simplicity in the razor views and avoid advanced configuration options. In other words, if multiple variations of a view are required, prioritize the simplicity of creating another view with alternate markup as opposed to coming up with an elaborate mechanism for a generic view generating different sets of markups.

For discoverability and simplicity, the naming of view files and the rendering items in Sitecore should align as much as possible. This said, the name of the Rendering item in Sitecore should be as editor-friendly as possible – even if it differs from the actual filename. Generally, throughout Sitecore you avoid using CamelCasingInNames or AbbevInNames as it not considered editor friendly – in other words, in Sitecore always favour the editor experience over the technical

aspects.

Partial views should be prefixed with underscore (_) to separate them from the views references through Sitecore items or controllers.

Place view rendering files in subfolders under /views named after the module to which they belong, for example /views/navigation/menu.cshtml for the Feature layer Navigation module. In the case of Controller Renderings, follow the standard ASP.NET MVC conventions for working with razor views and Areas if these apply to your project.

2.6.3. Datasource settings

Renderings are defined in the feature layer, but they reference elements in the project layer, such as Page Type Templates, Datasource Templates and locations within the content tree. These references can result in dependencies that go the wrong way, or dependencies that cross features.

When dealing with datasource locations and datasource templates, Sitecore offers limited help in overcoming these dependencies – and therefore, to preserve the stringency of the architecture, you will have to introduce custom logic to overcome this limitation, typically by introducing a Foundation module (See Layers).

2.6.3.1. Datasource Template

The Datasource Template field in Editing Options section of the rendering's item has two functions: it controls which types of items are allowed as datasource items for the rendering, and it designates the template used for new items created through the datasource dialog.

The Datasource Template field supports template inheritance. This means that you can select an item based on the specified template, or an item based on a template that derives from the specified template. This is very useful since a rendering in a Feature layer module most often relies on an Interface Template – which should be defined in the same Feature layer module (See Template types). The problem is, however, that Sitecore will use the specified template as the template for new items created through the Select the Associated Content dialog. This means that even though Sitecore supports it, you should not specify an Interface Template in the Datasource Template field, as you will end up with pages or datasource items based on this template – which ends up violating the architecture and create dependencies (see Dependencies).

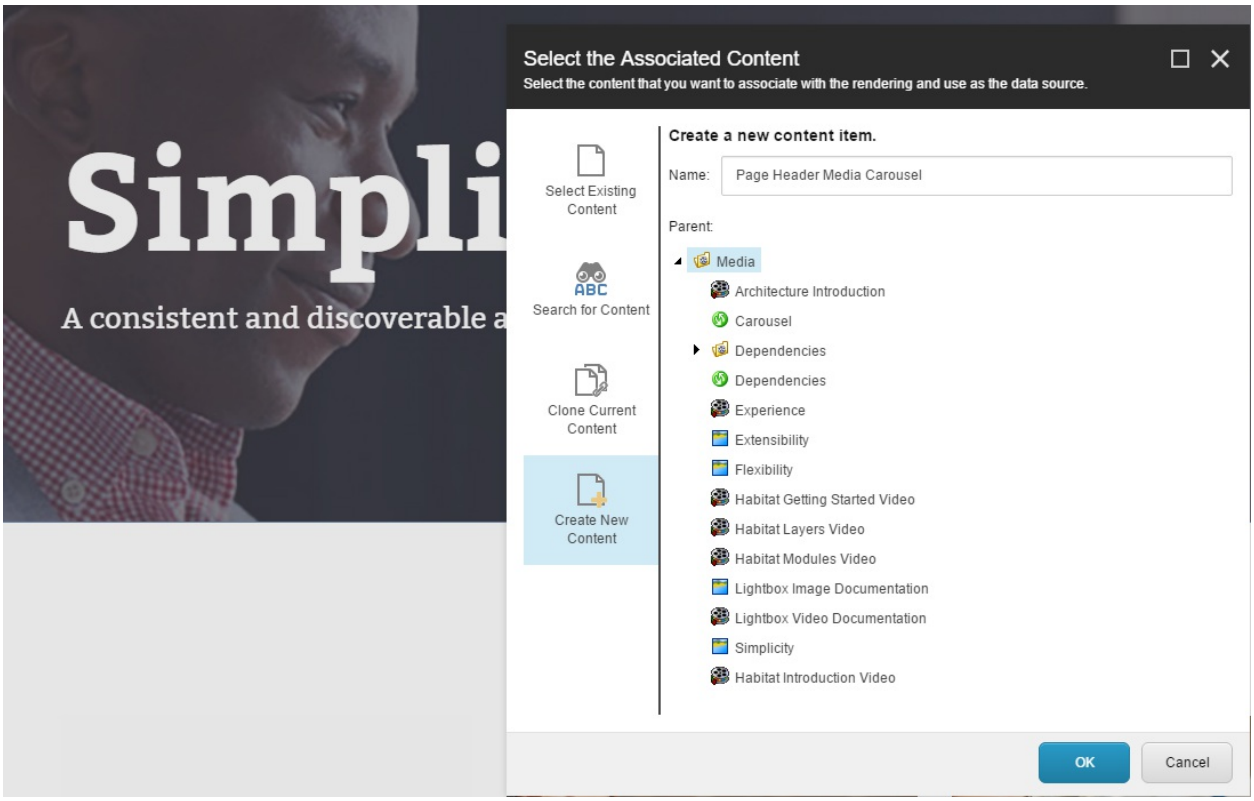


Figure: Creating new content in the datasource dialog

Unless you plan to disable the opportunity to create new content in the Associated Content dialog or simply ignore the architectural principles and specify a Page Type or Datasource Template (see Template types) in the Datasource Template field, there is no easy solution for this conundrum. However, Sitecore does allow you to hook in to a pipeline to resolve the datasource template in context and thus be smarter around this. See the Habitat Example below.

2.6.3.2. Datasource Locations

The Datasource Location field in the Editor Options section of the rendering item allows you to specify one or more content locations from where the editors can select content for the rendering. More locations can be specified using a piped *query:* syntax.

The immediate challenge is that locations from which the renderings draw their content are often defined in the Project layer – by the sites

themselves – and thus specifying the Datasource Location of the rendering will create a dependency from a feature rendering to a Project layer module. Furthermore, the challenges with specifying datasource locations can vary according to the complexity of your content architecture, for example if you are building a multi-site or even a multi-tenant solution where a rendering will have different locations depending of the context in which it is used.

A simple solution to this challenge is for the features themselves to own the datasource location itself. For example, a feature Teasers module could point to /sitecore/content/teasers as the datasource location for all its renderings, thereby forcing all sites to have a shared teasers repository. This will work perfectly fine for a simple single site/single tenant solution. But in other scenarios this might be challenged by the content governance model. Habitat provides an example.

Habitat Example

The Habitat example site is a true multi-tenant and multi-site solution. The Habitat project layer module, as well as our internally built product demonstrations, represent tenants (and sites) that define their own page types and datasource locations (see Multi-site and multi-tenant).

In order to accommodate this scenario, the datasource location and template resolution have been extended in the Habitat project. This means that it is also possible to define datasource templates and locations for each site, in addition to on the rendering itself. This is done through an extension of the getRenderingDatasource pipeline and the addition of a site: prefix to the Datasource Location field. In short: if the Datasource Location field contains a site: prefix, the pipeline extension will attempt to lookup the datasource location and template in a site specific list. This functionality is implemented in the Foundation.Multisite module.

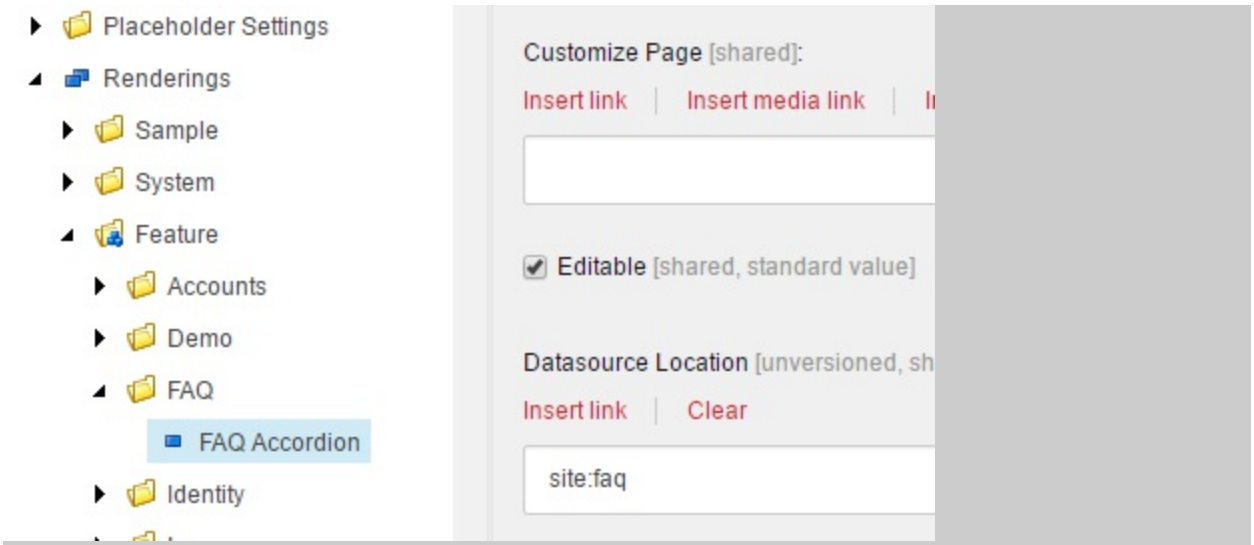


Figure: Site specific datasource location for a FAQ rendering

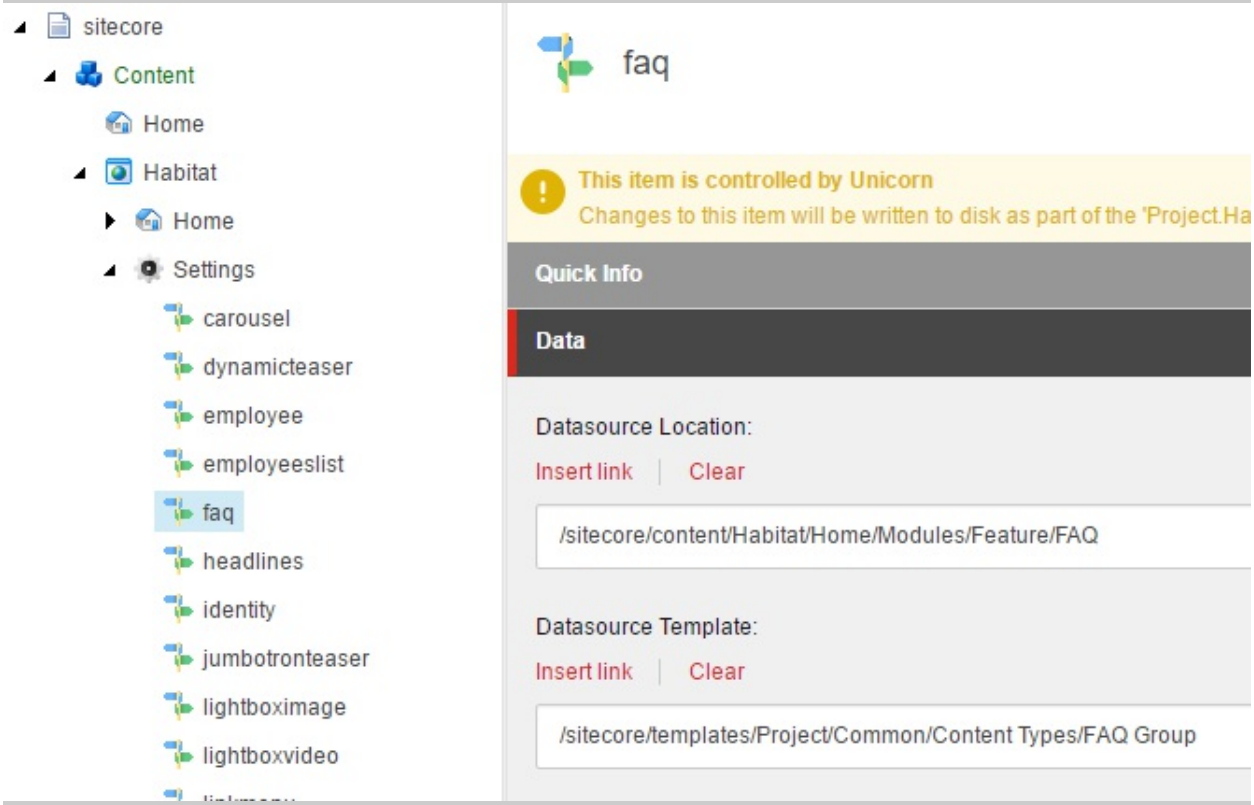


Figure: The FAQ site specific datasource settings for Habitat

2.6.4. Rendering parameters

Rendering parameter templates are managed in the same module as the code that uses them, either in the feature module where the rendering and controller logic resides or in a foundation level module. In the latter case a rendering parameters template in a feature module can derive from one or more rendering parameters templates from foundation modules and include all parameters in the rendering. If the rendering parameters template reference any setting items, make sure that the items along with their templates reside in the same module (or in a lower layer module) as the parameters template itself.

Please consider carefully what configuration of the rendering belongs in the datasource or on the context page, as opposed to the rendering parameters. Rendering parameters are generally considered less editor friendly than content fields and are often restricted to administrators, but it is possible through rendering parameters to preconfigure a number of compatible renderings – which makes the rendering parameters both accessible and very user friendly.

Consider using Field Editor buttons for managing those hidden fields in the Experience Editor (http://www.nonlinearcreations.com/Digital/how-we-think/articles/2016/03/Sitecore-8-and-8-1-How-to-add-a-Field-Editor-Button-to-a-component-in-Experience-Editor-Mode.aspx).

Habitat Example

Habitat has many examples of Rendering Parameters applied throughout the solution, for example, many renderings use the *ParametersTemplate_HasBackground* template to render different background colours and images for the rendering.

Habitat also has an example of a general Field Editor button (implemented in the Foundation.FieldEditor module) which makes it

possible to edit all fields in a datasource item from the Experience Editor.

2.6.5. Compatible renderings

Compatibility between different renderings are typically defined by business rules in the solution, such as whether they can share the same datasource and whether they can be placed in the same placeholder.

In Helix, compatible renderings are renderings which have identical datasources (or no datasources) and placed in the same module. These conventions ensure that there will be no unwanted dependencies between feature modules from Compatible Renderings fields.

2.6.6. Placeholders

Placeholders in Sitecore allow the dynamic assembly of page layouts either by allowing the editors to design specific pages, or by allowing administrators to create predefined variations of layouts on Page Type templates (see Template types). Most often, placeholders are contained in the layouts and sub-layouts in the Project layer modules. In rare cases renderings in feature or foundation modules can contain placeholders, for example when dealing with elastic or composite page components such as tabs, accordions, carousels etc.

The actual conventions for placeholders and placeholder definitions are very project specific, as they affect the dynamic page layouts (which renderings can go where on the page) and the visual design of the pages (how will renderings change appearance based on where they are placed).

Keep in mind that placeholders are not only used by editors to dynamically design pages, but also allow developers to create variations of pages merely by reconfiguring a Page Type template. Therefore you should favor the dynamic constructing the page layout with many placeholders and renderings, but ensure that you keep a consistent visual design and coherent user experience by controlling the page layout editing with security or by marking placeholders as non-editable. This practice will greatly increase the flexibility and simplicity of creating new page types and layouts as it reduces the need for development.

Habitat Example

Even though the header of the Habitat example site is relatively static across the page types, it still contains around 10 placeholders - most of them marked as non-editable and thus only configurable by a developer. This makes it possible to reconfigure the header and reuse the renderings in the header for other site variations in a multi-site

scenario or page variations such as campaign landing pages or commerce check-out pages.

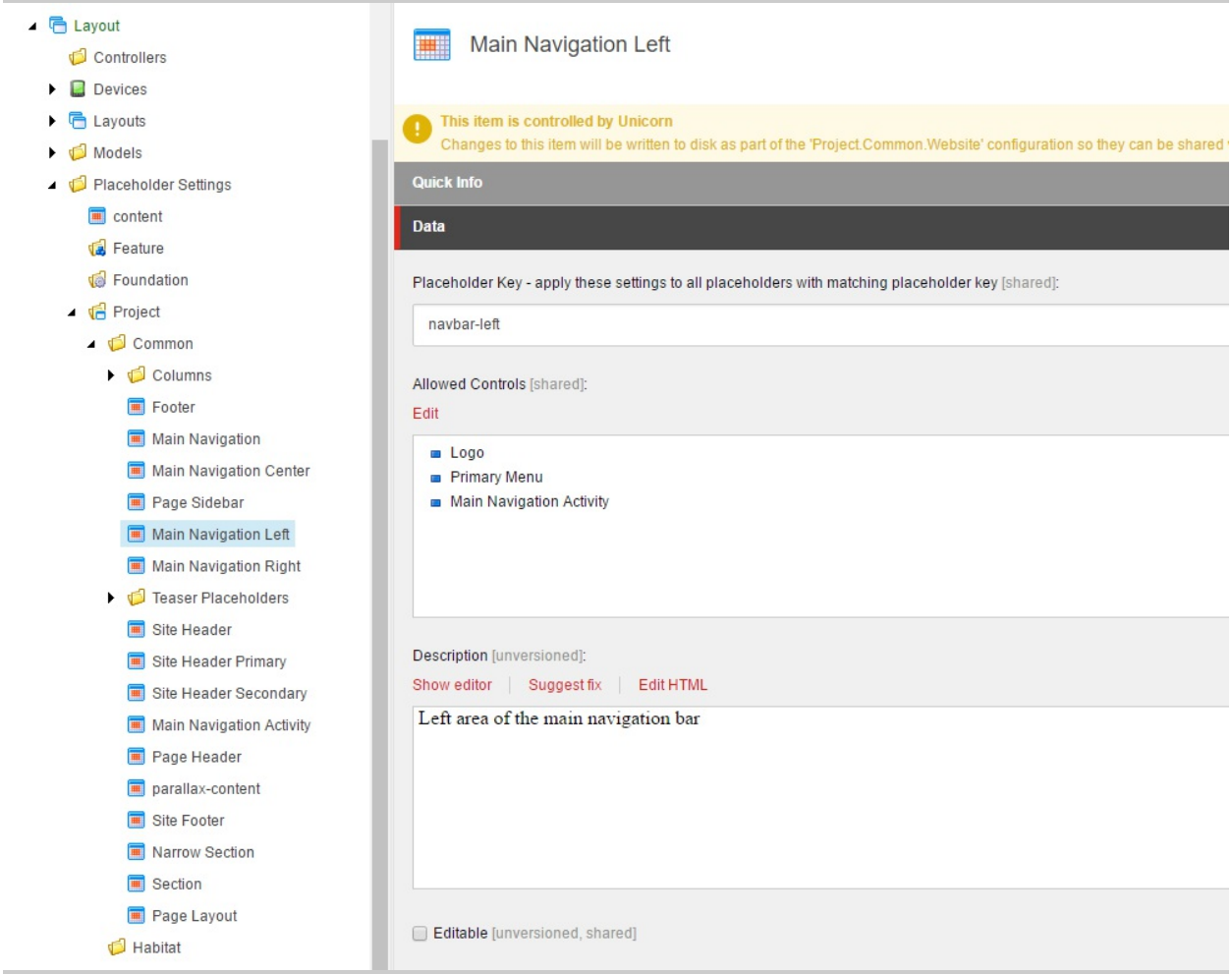


Figure: Example of a header placeholder from Habitat

Define Placeholder definitions in the Project layer modules, as they will need to reference the renderings from the feature modules. This also allows multi-tenant scenarios where each tenant defines its own unique mark-up and site structure.

2.7. Configuration and settings

Configuration and Settings describe the overall configuration of the Sitecore system, implementation or configuration of specific features or functionalities. These can take many forms, such as .config files or data in Sitecore, and can be defined by the installed software such as Sitecore or 3rd party modules or by the business logic in the solution modules.

2.7.1. Configuration strategy

It is very important to have a strategy for configuration and settings management, as wrong or missing configuration can not only have a negative impact on for example the performance and stability of the running solution, but poor configuration management can also slow down the productivity of the development team, as aspects such as deployment and upgrades can be unnecessarily complicated.

We differentiate *Configuration* from *Settings* in that *Settings* are aimed at the user roles (editor, administrator) and *Configuration* is aimed at the developer or IT system administrator roles.

This differentiation makes it significant where and how Settings and Configurations are managed in the Application Lifecycle. For example, you should be cautious of having Configurations defined as Sitecore content, as this can make it harder for developers and IT administrators to manage during deployment and upgrades, and can make it difficult to manage environment or server specific values such as connection strings. Take particular notice of the scopes of the configuration as defined in the following.

Each Configuration or Setting – whether predefined by for example Sitecore or defined by a module in your implementation - will have a *Definition Scope* and a *Value Scope*:

Definition Scope
describes on which level the configuration is defined, i.e. in what scope of effect a change of value will have.

Value Scope
defines in which circumstance values changes.

The following table describes the possible combinations of scope, and how configurations and settings map to those.

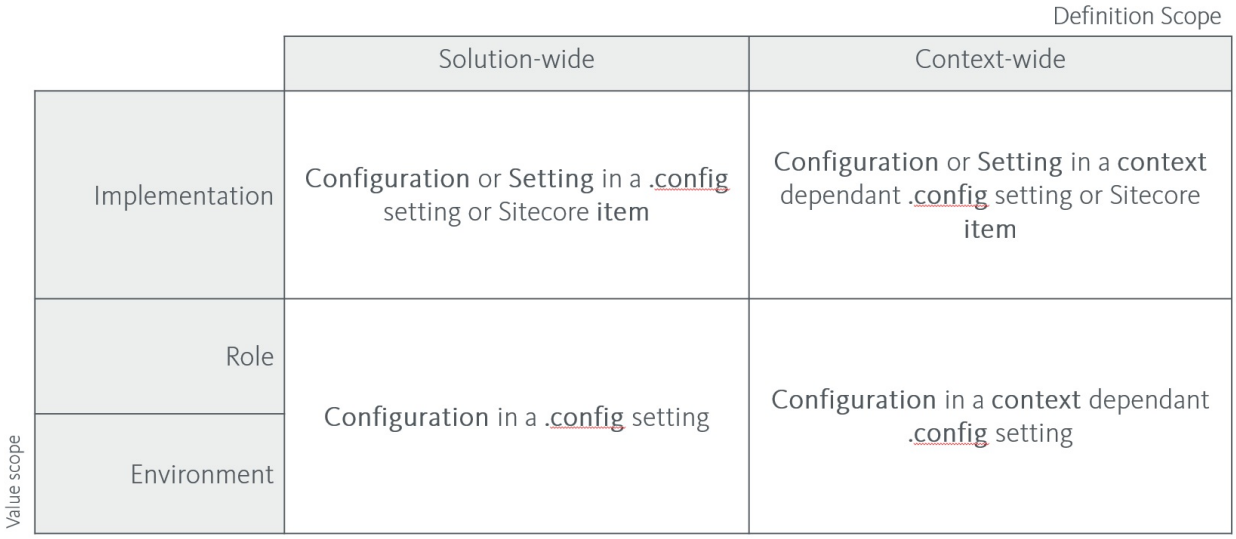


Figure: Scope of Configuration and Settings

Avoid having *Settings* defined in .config files as this counters the purpose of Sitecore users being able to change them. Please note that – although confusing – the <settings> section in the Sitecore .config files is considered *Configuration*.

2.7.2. Definition Scope

Just like code, templates, views etc., configurations and settings defined in your implementation relate to a specific module and should therefore be managed and versioned with that module.

Project layer modules should never define new settings or configurations, as it indicates business logic in the project layer – which should always be avoided.

2.7.2.1. Solution-wide

Solution-wide settings and configurations cover the entire implementation. In other words, they are defined once within the solution and their values will affect the entire implementation. Examples of solution-wide configurations are all the <settings> defined under <sitecore> in the web.config. Examples of solution-wide settings in Sitecore are the installed languages and aliases, which are defined under /sitecore/system.

Solution-wide Configurations should be defined in a .config file, whereas Solution-wide Settings should be defined under /sitecore/system/settings or similar.

Settings defined for a single Project, Feature or Foundation layer module should be defined in their corresponding module folder under /sitecore/system/settings/[layer]/[module].

Be cautious when creating solution-wide configurations in your feature and foundation modules, as you might be restricting the flexibility of the implementation. It is generally good practise to make features as context-aware as possible, for example in terms of support for multi-site, multi-tenant or multi-language. For example, by moving a configurable path from a Solution-wide configuration under <sitecore> <settings> to the Sitecore <site> node in the web.config, one can help

make a feature site context specific - or by changing the value of an Unversioned setting field, one can make a feature context language specific.

2.7.2.2. Context-wide

Context-wide covers, as the name implies, a given context. This means that a configuration or setting can exist in multiple places, for example per tenant, per site, per language, per database, content hierarchical, by taxonomy etc. The context is, in other words, defined by the business logic in the module that defines the configuration or setting. Examples of context configurations are domain names on sites, search indexes, security domains, user profiles etc.

Sitecore defines a number of contexts to which you can associate your own configuration or settings, for example the site definitions in the web.config.

Habitat Example

The Foundation/Dictionary module in the Habitat example site allows dictionary content to be defined on a site context. This is done by defining a dictionary path on the `<site>` definition in the web.config.

```
private Item GetDictionaryRoot(SiteContext site)
{
    var dictionaryPath = site.Properties["dictionaryPath"];
    if (dictionaryPath == null)
        throw new ConfigurationErrorsException("No dictionaryPath was spec
    var rootItem = site.Database.GetItem(dictionaryPath);
    if (rootItem == null)
        throw new ConfigurationErrorsException("The root item specified in
    return rootItem;
}
```

Context-wide settings are often created in the content section as part

of a hierarchical content tree, for example site-context settings are defined on the site root item or on an item under the site root. This allows easy access to the settings for Sitecore users.

2.7.3. Value Scope

2.7.3.1. Environment scoped values

Environment scoped values are configurations or settings whose values change depending on the environment on which the implementation is running. Environment scoped values are typically associated with solution-wide configurations, such as connection strings, but can also be context-wide – for example by allowing sites to specify the connection string to use.

Environment-scoped values should be managed in .config files as they are directly affected by the application lifecycle and deployments – which is typically managed by IT.

In cases where environment scoped values are defined as Settings that Sitecore users can manage, for example in cases where different sites or content trees have different connection strings, let these settings point to values defined in a .config file. This will make it possible to change the values in Sitecore, while allowing deployments and the IT application lifecycle to manage the environment's specific values.

Do not manage environment scoped values as part of the development process, but rather as part of your deployment procedure (see Deployment).

2.7.3.2. Role scoped values

Sitecore implementations can extend over multiple servers in a single environment. This includes content management, content delivery, email delivery, processing and more. Some configurations have different values, for example disabling features or functionalities, on specific server roles. An example of role scoped values is the configuration in the

App_Config/Include/Sitecore.Publishing.DedicatedInstance.config.exar file, which configures a server as a dedicated publishing server.

Role scoped values can be managed in the development process, for example by managing role specific Sitecore .config include files or role specific web.config files as part of the build process. Alternately the specific values can be managed as part of the deployment process.

Role scoped values are typically restricted to IT/developer managed configurations, as Settings are typically managed in Sitecore items – whose values are shared across server roles.

2.7.3.3. Implementation scoped values

This is by far the most typical value scope, as it is a value which, when set, affects the entire implementation. Most configurations in the Sitecore .config include files that are implementation scoped and user managed Settings.

Implementation scoped values for Settings are typically initially defined in the development process, but are managed as part of the content management process in the production environment. As implementation scoped settings values are therefore “owned” by the Sitecore users in the production environment – and it should be able to bring these back through the environments as part of the content - for testing purposes, it is good practice to manage these as part of the Sitecore content.

Always manage implementation scoped values in configurations in the development environment and deploy them through the usual deployment procedure. Avoid at all cost making implementation scoped changes in specific environments as this will differentiate environments and can severely cripple a consistent testing effort.

In Helix, implementation scoped values should always be associated

with the module which requires them. For example, if a Foundation module requires the standard Sitecore configuration setting to change, the module itself should manage this change – by including an App_Config/Include file in the module.

2.7.4. Managing .config files

Managing .config files for even a simple Sitecore project can, over time, become a very complex and time consuming task, and if done wrongly it can be a source of many problems such as performance, consistent testing, instability and waste of resources.

The key to long-lasting good configuration management is isolating not only the implementation specific changes and additions from the standard Sitecore and .NET configuration, but also keeping any changes and additions to configuration together with the business logic which needs it. The first makes it possible to easily identify changes – making upgrades much easier – and the latter makes it easy to identify the reason for changes – making issue resolution and implementation changes stress-free.

Changes or additions to configuration files are most often associated with specific features, and in Helix, this means that these configuration changes belong in the modules that need them.

2.7.4.1. Sitecore include files

Changes to any configuration under the <sitecore> root in the web.config should always be done through app_config/include files.

Habitat Example

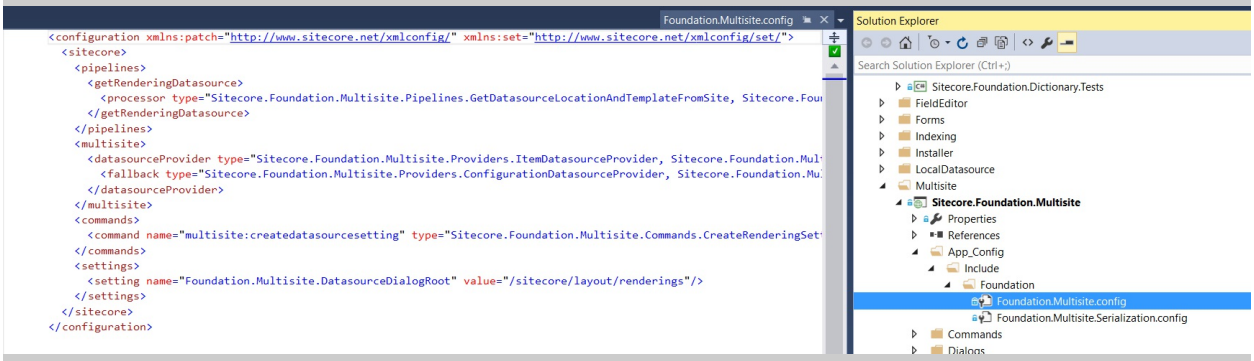


Figure: Foundation/Multisite specific configuration changes

Place the configuration changes for a given module in the modules layer subfolder under /include and name the configuration file with [Layer].[Module].config.

Sitecore merges include files alphabetically, and if a certain configuration file needs to be included last in the web.config merge with in a layer, prefix the file with z., for example z.Foundation.Indexing.config. If the file needs to be run last of all include files, place the file in a subfolder under /include called zzz. Refer to the Sitecore documentation for more information on config patching.

2.7.4.2. Other .config files

Changes to .config files outside the <sitecore> element in the web.config, such as the .NET or IIS parts of the web.config or other .config files such as domains.config, will require another strategy. Keep in mind the Helix convention: keeping the configuration change together with the business logic which requires it.

Sitecore does not provide an out-of-the-box approach for this challenge, and other tools such as the Visual Studio web.config transformations or SlowCheetah works in a file-based approach – which is not directly compliant with Helix, as all changes to a single file are not necessarily associated with a single module.

The MS Build xml transforms can be used to apply feature centric configuration changes to files, but it will require custom integration into the MS Build system. Please refer to the MS Build documentation on the TransformXml task.

Habitat Example

Habitat does use the MS Build XmlTransform task to make multiple file transformation across features. This is done as part of the gulp

build system supplied with the Habitat example site.

The functionality allows each module to have one or more .transform files placed in the same sub folder as the target .config file and with the same name. For example, the /App_Config/Security/domains.config.transform will apply a transformation to the domains.config file as part of the build. The syntax of the .transform file follows the MS Build web.config transformation syntax (See https://msdn.microsoft.com/en-us/library/dd465326(v=vs.110).aspx).

All .transform files are picked up in the 04-Apply-Xml-Transform gulp task in the /gulpfile.js and applied one by one in a separate MS Build command defined in the /applytransform.targets file.

At the time of writing, Habitat has four config transformations across the web.config and domains.config files.

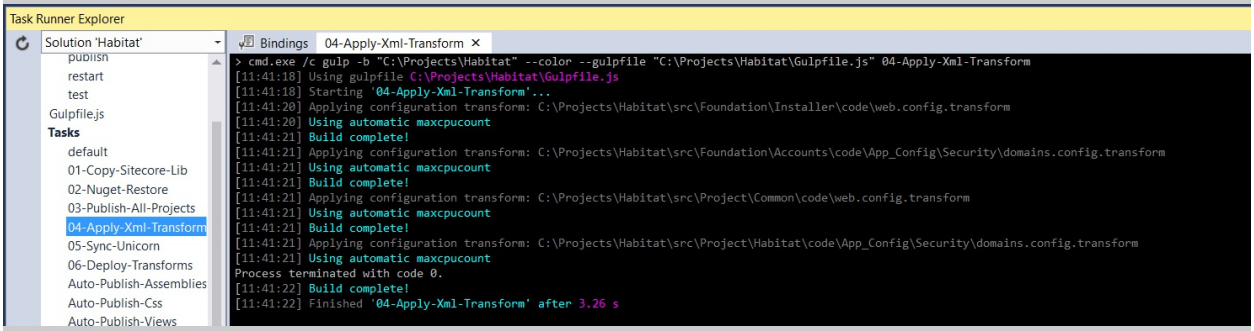


Figure: Output from the Habitat xml transform task

2.8. Multi-site and multi-tenant

One of the exciting parts of building Sitecore implementations is the flexibility it offers to solve pretty much any business requirement. There is a great range of choice when it comes to putting together the content and information architecture to suit the business. This includes the ability for a single Sitecore implementation to host multiple tenants, sites and channels.

Even though Sitecore does not natively support a Tenant context, the separation between multi-site and multi-tenant is important, as multi-tenancy introduces a whole new level of governance which needs to be considered both in the architecture and in the managing organisation.

2.8.1. Tenants

Tenants are groupings in the business organisation that need freedom and autonomy to define and manage their own sites, channels and data, while at the same time desire or require the ability to share with other tenants. For example, this could be multiple departments in an organisation with separate websites, but a willingness to collectively define and share the features available on these sites in order to reduce the cost of development and maintenance.

Multi-tenancy in Sitecore is primarily about creating a governance model and technical architecture that support different requirements and decision models.

Helix suits this scenario perfectly, as it defines a common feature and foundation layer in the implementation, but allows multiple project layer modules – one per tenant – that bring these features together. Tenants can even define and implement new business specific features, new modules, which are only used in their project layer module – and thereby the architecture allows one tenant to adapt its solutions or to change its business objectives without disrupting other tenants.

Note that although many Sitecore implementations have multiple sites, most are single tenant, where a single organisational entity can make decisions on the business direction of the overall implementation.

In any multi-tenancy scenario, it is very important to completely understand and highlight the restrictions in autonomy, as well as the benefits of the shared implementation, and subsequently define the governance model for these overlaps between tenants. Although critical for the success of the implementation, this is largely an organisational task that lies outside the realm of the technology.

In a scenario where one implementation hosts two or more tenants

with no common content or desired sharing of features – in other words completely autonomous tenants – it is highly recommended to split into multiple implementations, with separate governance models.

2.8.1.1. Defining a tenant in Helix

Although Sitecore offers a great platform to support multi-tenancy in the software, there is no predefined tenant context in Sitecore. This means that the borders between what is shared between tenants and what can be defined autonomously is largely for you and the business to define. This includes content hierarchies, content repositories, media, workflows and more.

It is recommended that each tenant in a Helix compliant implementation has its own project layer module, i.e. that each tenant has the freedom to define its own set of datasource and page type templates (see Template types) as well as its own overall page layouts and sub layouts. If multiple tenants share templates or layouts, it is recommended to have a shared project layer module to host these entities. This gives the flexibility for a tenant to use shared as well as specific page types and layouts in their site or sites.

Habitat Example

In Habitat the Project/Common module contains primarily Datasource templates and placeholder settings that can be used across multiple tenants, i.e. project layer modules.

Content sharing or isolation between tenants comes largely down to content hierarchy and security. By defining organisational roles, users can belong to different tenants and content can be shared or separated. Because of the powerful, feature-rich and fine grained security model in Sitecore, pretty much any business requirements can be met. See more on security in Helix in Security and workflows.

It is often not relevant for business logic to understand the tenant context under which the business logic is running. Tenant related locations such as datasource locations, datasource templates etc. can be configured in Sitecore and there is often no use for an actual tenant context API. If this is needed, it is recommended to create it.

Habitat Example

The Habitat example site defines one project layer module – effectively one tenant. Although the current implementation only defines a single site under the tenant - which lives under /sitecore/content/Habitat - all features support a multi-site scenario. In Habitat, the tenant or site is defined to have its own shared content repositories for features such as teasers, dictionary, FAQ's etc. – which means this content could be shared across sites but not necessarily across to other tenants.

The Habitat tenant defines isolated tenant specific data for the following areas:

Page Type and Datasource Templates
Page Layouts and Sub-layouts
Personas and profiling
Goals and outcomes
Campaigns and engagement plans
Social network accounts
Forms
Security domain

Please note that some 3rd party modules and features in Sitecore do not fully support multi-tenancy or multi-site, and thus are limited in ways they can be isolated across tenants or sites. An example is that, by default, the Sitecore Analytics only allows analytics to be broken down by site and thus not by tenant. Please refer to the product documentation for the modules you are interested in using in your implementation.

2.8.2. Sites

The connotation of *a Site* varies, but in Sitecore a Site is a concrete context to which different properties can be attributed. Hence a multi-site implementation is an implementation where content can exist in multiple different contexts (for example different channel types, domains, websites etc.). Despite the association with a website, a site can output content and define contexts for a range of channels, web, mail, print etc. and different features and modules (such as for example EXM – Email Experience Manager – and FXM - Federated Experience Manager) can attribute different properties to the site context.

In this perspective, Feature or Foundation layer modules should always be implemented in a multi-site setting, i.e. the business logic in modules has to be context aware. Furthermore, this means that not only can the Feature layer or Foundation layer modules in Sitecore utilise the site definitions in Sitecore to determine context, they can also extend the Sitecore site context with their own properties.

Habitat Example

The Foundation/Dictionary module extends the site context in Sitecore with the ability to configure a site specific dictionary location. This is done by adding an attribute to the Site definition and reading this through the Site context API.

Note also how the Sitecore WFFM module uses the same technique to set the root folder for forms for the site.

```
<sites>
    <site name="habitat" patch:before="site[@name='demo']"
        database="web"
        virtualFolder="/"
        physicalFolder="/"
        rootPath="/sitecore/content/habitat"
```

```
        startItem="/Home"
        dictionaryPath="/sitecore/content/habitat/global/dictionary"
        dictionaryAutoCreate="false"
        domain="extranet"
        allowDebug="true"
        cacheHtml="true"
        htmlCacheSize="50MB"
        registryCacheSize="0"
        viewStateCacheSize="0"
        xslCacheSize="25MB"
        filteredItemsCacheSize="10MB"
        enablePreview="true"
        enableWebEdit="true"
        enableDebugger="true"
        disableClientData="false"
        cacheRenderingParameters="true"
        renderingParametersCacheSize="10MB"
        formsRoot="{4BC8A78C-44A7-46EB-8126-040D3F12CAA0}"
        enableItemLanguageFallback="true" />
</sites>
```

Most standard modules and functionality rely on the Site definition in Sitecore for context, and it is therefore recommended to use the standard Sitecore API to determine site context in business logic – as opposed to implementing new logic to define the site context. If any additional properties or extended logic are needed on the site context, it is recommended to provide these to the feature modules through a foundation layer module.

Habitat Example

The Habitat example site implements the Foundation/Multisite module that defines multi-site and multi-tenancy specific functionality for the feature modules. The most important feature in the Foundation/Multisite module is the ability for renderings to define site specific datasources. This means that each site can maintain a list of datasource locations for different renderings – which means that for example teasers does not have to be shared between all tenants in a solution, but only between sites within a tenant.

By default, sites are defined in the configuration files, in the <sitecore> <sites> section of the web.config, and the site context is resolved as part of the request for each page or context. It is however possible to add site definitions configured by other means, for example the FXM (Federated Experience Manager) feature adds site definitions for each domain matcher defined under /sitecore/system/Marketing Control Panel/FXM, to allow requests from 3rd party sites to be isolated by domain. There are also 3rd party modules which allow you to define new site definitions using Sitecore items (see for example the Multiple Sites Manager in Sitecore Marketplace https://marketplace.sitecore.net/Modules/M/Multiple_Sites_Manager.as

When adding properties to the site definitions and context, consider the separation between configuration and settings (see Configuration and settings). In a standard configuration scenario, site definition additions should be added to the <site> definition in the configuration file. To add editor managed settings to the site, modules should define templates which can be added as base templates to the site root item for the project layer module, or add settings items inside the site hierarchy. Regardless of which of the two approaches you choose in your implementation, be consistent in how you define context specific settings. Also, be conscious that advanced settings might be confusing for the average editor, so consider limiting visibility of and access to these settings to administrative users.

Habitat Example

The Habitat example site generally uses the base templates approach, for example, the Feature/Accounts module defines an AccountsSettings template which is assigned to the Site Root template in the Project/Habitat module.

The module uses the site hierarchy to find the settings items for the module.

```
public virtual Item GetAccountsSettingsItem(Item contextItem)
{
    Item item = null;

    if (contextItem != null)
    {
        item = contextItem.GetAncestorOrSelfOfTemplate(Templates.AccountsS
    }
    item = item ?? Context.Site.GetContextItem(Templates.AccountsSettings.

    return item;
}
```

Keep in mind that parts of the site definition configuration or settings can be environment specific, and should the managed with an environments specific perspective (see Configuration and settings).

2.9. Language and culture support

How your Sitecore implementation handles languages is largely dependent on the business requirements, but it is recommended that you include support for multiple languages or cultures in your modules. This not only adds the possibility for adding more languages or cultures over time, but also encourages good practices such as not hard coding any content and letting an editor manage and edit all text on the website.

2.9.1. Enabling multi-language support

Language support includes having all content on your sites and channels provided through Sitecore, including main page content, related content, headings, labels, metadata etc.

Remember to set up site metadata to reflect the context language and culture, and that all number and date formats follow the format of the context culture.

Consider Right-To-Left language support both in the content and visual design implementation.

It is generally recommended not to mark fields as shared or unversioned unless carefully thought through in a language and culture context. Consider that some fields, such as image fields have metadata which should be translated and list fields can have different sort orders depending on language.

Languages are defined on a global implementation-scope level in Sitecore and thus all sites and tenants will have the same languages installed. In a multi-tenant and multi-site scenario (see Multi-site and multi-tenant) consider adding business logic to make languages and cultures configurable by site and tenant.

Habitat Example

Habitat relies on the standard Sitecore language support, but includes a Feature layer module which allows individual sites to define the languages they support. This will determine the languages shown to the site visitor in the language selection dropdown.

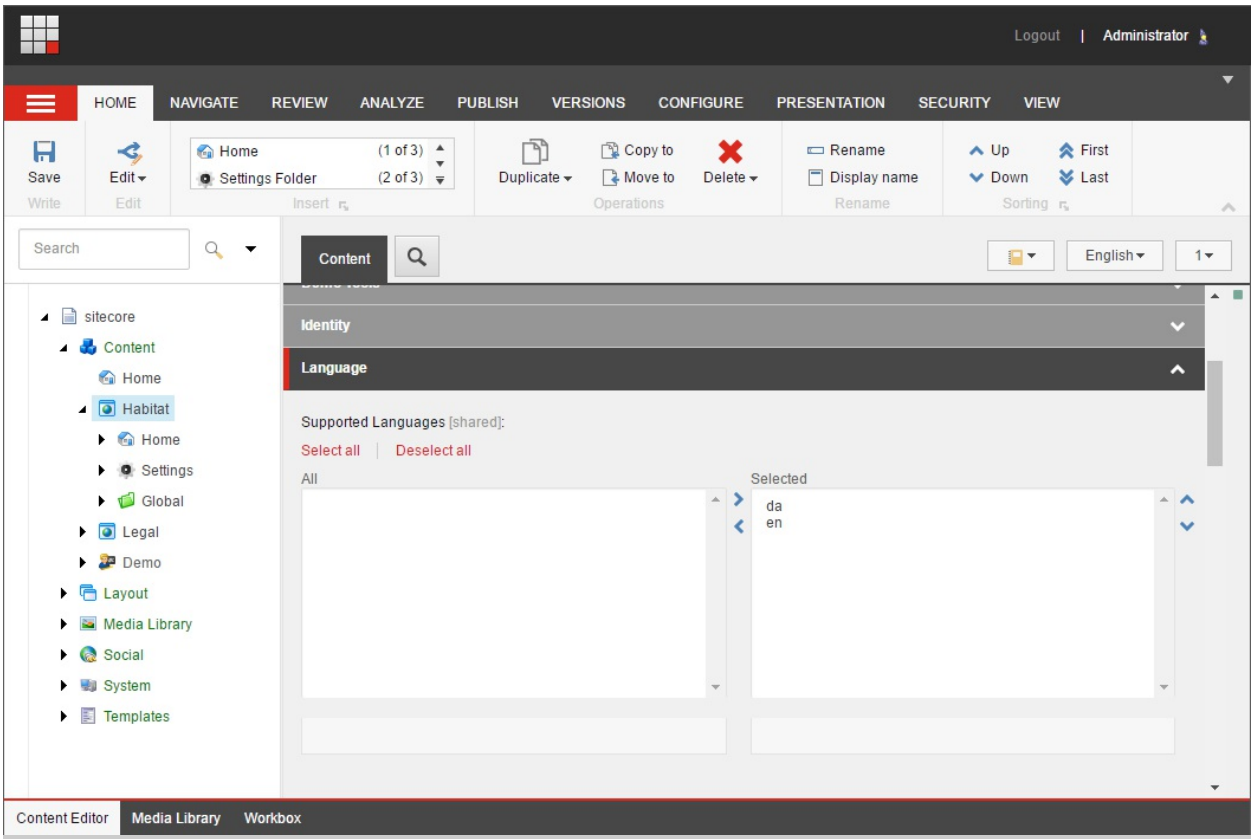


Figure: Language feature in Habitat

2.9.2. Dictionary

When defining the content architecture of your implementation, some pieces of presented content on the pages will not necessarily be part of the main or related content, i.e. the context item or datasource items, this could include labels for business forms, buttons or similar. As it is recommended that all content on the pages can be edited through Sitecore – for flexibility as well as language support reasons – there needs to be a way of managing this content.

There are a few different mechanisms for managing these types of content, but the most traditional way is through a Dictionary – which is also supported in the Sitecore platform through the Sitecore.Globalization namespace. Although the default Sitecore implementation is somewhat rudimentary and does not for example include Experience Editor or hierarchical support, it is often enough for many use cases. The Sitecore Dictionary does have support for a context specific (multi-site, multi-tenant or modular architecture) through the use of dictionary domains, however for advanced use cases it is recommended to re-implement the Dictionary concept within your solution.

Avoid reusing dictionary texts across modules, features or views as it can limit the flexibility for the editors. Just because two labels have the same value at the time of implementation does not mean that an editor does not want to change them independently at some later time.

Dictionary entries are used in the presentation and business logic of the modules and are therefore owned by the feature modules themselves. In order to make itself available to all features, any custom Dictionary API, functionality or principles herein should be a foundation level module in the Sitecore implementation.

To increase modularity and discoverability in the dictionary, entries should be referenced in a hierarchical structure. For example, their

structure [*module*]/[*view*]/[*text*] would be represented as *Accounts/Login/Remember Me*. Furthermore, dictionaries should be available on a site or tenant level to allow labels to be managed individually across sites.

Habitat Example

The Habitat example site includes a custom Dictionary foundation layer module which enables site specific dictionaries, default values and development support, enables Experience Editor and hierarchical support.

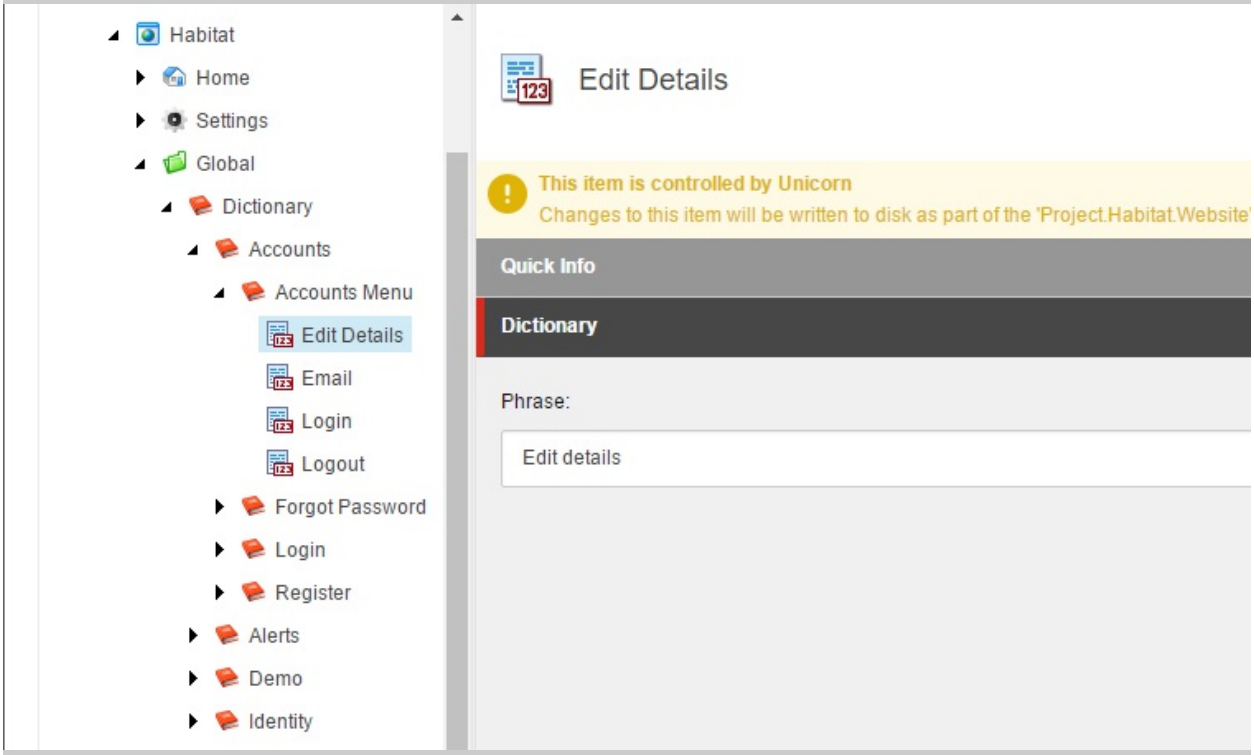


Figure: Hierarchical dictionary in Habitat

2.10. Security and workflows

Security, i.e. what access editors have to features, pages, content, languages, workflows, fields etc. can be set extremely granularly in Sitecore.

How granularly you will need to set workflows or security in your application is entirely up to the business requirements, but there are some aspects to workflows and rights to be cautious of if you want to preserve the references between layers, the decoupling between modules and any multi-tenant requirements.

2.10.1. Rights management

Most aspects of rights and access are defined in the content area of the sites and therefore in the Project layer modules or directly in the production content itself.

These types of rights and roles could be called Organisational Rights or Organisational Roles, as they typically will define the organisation or groups to which the users belong, and thus the hierarchical content they can access – and which rights they have to it.

But there are aspects of security that reach into the feature and foundation modules – and which therefore needs to be addressed in the modular context of Helix. This is particularly true for individual fields, as these are defined in Interface Templates in the feature and foundation layer modules. It is also true for configuration settings, and even specific tools and editor extensions within Sitecore that are contained within the feature modules.

These types of rights and roles are called Functional Rights or Roles, as they define which types of functional access the user is given inside for the hierarchy that he or she can access.

For example,: If a user has the functional role of News Editor and he has the Organisational role to edit Site A, this user will be able to edit news on Site A. If the same user is given the Organisational role to read Site B, the user will not necessarily be able to edit news on Site B, as he – although he has the functional right to edit news - will not have the organisational right to edit content on Site B.

Functional Rights should be defined on Functional Roles owned by the Feature or Foundation layer modules which defines them. Certain Functional Roles and Rights will be managed on the project layer, for example which Page Types an editor can create, which languages or which workflow states.

This type of functional and organisational combination makes it possible to define functional roles and rights on the features which defines them, while staying independent of how the content is structured.

Habitat Example

The Habitat example site defines functional roles for the Feature and Foundation modules that define access to fields and configuration on the module. It also defines organisational Roles for the Habitat project.

Role
habitat\Project Habitat Content Author
habitat\Project Habitat Modules Admin
modules\Feature Accounts Admin
modules\Feature Demo Admin
modules\Feature FAQ Admin
modules\Feature Identity Admin
modules\Feature Language Admin
modules\Feature Maps Admin
modules\Feature Media Admin
modules\Feature Metadata Admin
modules\Feature Multisite Admin
modules\Feature Navigation Admin
modules\Feature News Admin
modules\Feature PageContent Admin
modules\Feature Person Admin

Figure: Example of Habitat feature roles

Feature roles are prefixed with the layer and module name followed by a descriptive name for the role, for example sitecore/Foundation

Accounts Admin or sitecore/Feature Maps Editor.

It is highly discouraged to assign specific rights to individual users inside Sitecore. Always set rights to domain roles and assign these roles to the individual users.

The fact that a user in one domain can be a member of roles from another domain can be very useful to separate rights in multi-tenant solutions or between modules.

2.10.2. Domains

Sitecore ships with two standard domains, sitecore and extranet, which in most cases are sufficient to accommodate the business requirements of Sitecore implementations.

In advanced multi-tenant solutions, it can be required to create multiple security domains to set up isolated roles and rights for each tenant – either for the website or for the editors in Sitecore - prohibiting access for the users of one tenant to another tenant's content – or even providing access across tenants or websites in some cases.

For example, in a case where two tenants, A & B, have two different projects with their own page types and content trees. This case could have two domains *ProjectA* and *ProjectB*:

An editor in the *ProjectA* domain, *ProjectA/User*, could be granted access to his own organisations content by enrolling him in the *ProjectA/Editor* role.
Likewise, *ProjectB/User* in the *ProjectB* domain could be member of the *ProjectB/Editor*.
By enrolling *ProjectA/User* in the *ProjectB/Editor* role, the editor – despite belonging to the tenant A domain – would gain rights to the *ProjectB* content.

If there is a need for more granular rights on a feature level (as described in Rights management) it could be beneficial to add an additional domain on which modules across the layers can register feature level roles. Using the previous example:

The *ProjectA/Editor* role could be a member or *Features/Accounts* Admin and *Features/News* Admin, not only giving *ProjectA/User* access to the ProjectA website and content but also to administer the feature configuration for the *News* and *Accounts* feature.

Habitat Example

Because of its multi-tenant and multi-site nature, Habitat defines a domain for the Habitat Project layer. To exemplify the configuration of rights on a feature level, Habitat also defines a Modules domain where Feature layer modules define roles which grants access to the functionality.

2.10.3. Workflows

Workflows in Sitecore reflect how editors in an organisation – or tenant - work with the content. In a multi-tenant scenario, two autonomous tenants (see Multi-site and multi-tenant) with different project layer modules, different content architectures, page types, and sites will often need different workflows assigned. Therefore, the actual Sitecore workflow definitions belong in the Project layer modules.

To cater for flexibility across sites and projects, workflows are always assigned to Page Type or Datasource templates (see Template types). Therefore, workflows are managed in a Project layer module.

Any custom actions or other business logic extensions to the Sitecore workflow engine will belong to a foundation or feature layer module.

2.11. Working with code

2.11.1. Code formatting

Formatting of code is deep in the tools layer, but can often be a great source of discussion and frustration that can lead to decreased productivity – and therefore important for establishing actual business value. Focus on a common team standard, and then look at the tooling available to make transitions for team members. As with most conventions and standards consistency is essential. Less friction will ease adoption.

Habitat Example

Habitat uses two spaces for indentation along with a whole series of conventions, for example for brackets and syntax. Code indentation is specified in the .editorconfig file in the solution root. The editorconfig extension for Visual Studio will read the file when the solution is loaded and configure the editor for this formatting.

Habitat also uses ReSharper for easy code formatting and has an associated .DotSettings file that defines the chosen conventions.

2.12. Visual Design and Theming

The user experience, in particular the visual design in an implementation, is the part of an implementation likely to change the most frequently. Often this area involves separate technologies, separate qualifications from Sitecore and ASP.NET and sometimes even separate development teams.

Therefore, this area often needs special attention in the architectural phase – emphasized by the specific business requirements of the greater implementation, for example multi-tenancy, multi-site, marketing maturity and so on.

2.12.1. Front-end technologies

Avoid looking at front-end technologies, such as JavaScript, CSS, Sass, Less etc., as a monolithic technology maintained in a single module, but rather as another technical part of the business domain modules in Helix. In other words, front-end technologies and assets such as JavaScript and CSS are technologies and are related to features and modules as other technologies.

Technologies with specific purposes, such as, bootstrap, jQuery, sass, bower, grunt, gulp etc. can be introduced and managed in separate modules. However, if a technology is used across multiple modules – for example in a multi-tenant or multi-site implementation – it can be useful to introduce it in a foundation layer module to make it available to all modules.

Habitat Example

The front-end technologies used in the Habitat example site are introduced in the Foundation/Theming module and can therefore be shared across to all foundation, feature and project layer modules.

The main front-end technologies introduced in Habitat are Bootstrap, jQuery, Bower and Sass. Furthermore, it uses the Visual Studio Web Compiler to bundle and minify JavaScript and CSS.

2.12.2. HTML mark-up

In a web solution, the relationship between mark-up and theming is one of the biggest sources of dependencies. If there is no strategy for managing HTML mark-up in a decoupled manner, the whole implementation can easily become very monolithic, and flexibility and productivity become stifled.

Most renderings, and therefore the HTML mark-up, in a Helix implementation will reside in the feature layer. On the other side, the business requirements are fulfilled in the project layer, by combining the features into the user experience. Therefore, in a multi-tenant or multi-site scenario, the challenge is that features need to address multiple business requirements – or simply different visual designs.

In other words, in a multi-tenant scenario some assets will always be shared between tenants (see Multi-site and multi-tenant) and when following the Helix conventions, the mark-up is very often one of these assets.

In a single tenant scenario, this is not such a big issue, as features can be built with a single business in mind. Features can be changed and adapted as the requirements changes.

A recommended approach to the shared mark-up challenge is to align your front-end technologies with the Helix principles and split the implementation of the Visual Design across the layers. In your foundation layer, you place the principal front-end framework or frameworks on which you are basing the implementation. These are the frameworks that define how to structure the HTML mark-up in the features. This includes technologies like Bootstrap, Foundation, 960 Grid System, and jQuery. In reality, the foundation layer module will likely contain a multitude of frameworks and utilities – perhaps extended with custom mark-up - which together form the conceptual base that feature developers need to follow. It can even be a custom

defined mark-up strategy that is clearly known to feature developer.

The point is that by defining a mark-up strategy in the foundation layer, the feature modules can follow this strategy and know that they are adhering to the implementation conventions.

Habitat Example

The module Foundation/Theming establishes the mark-up framework for the Habitat example site. The module uses bower to bring in a number of external frameworks based on Bootstrap and jQuery.

```
{
    "name": "sitecorehabitattheme",
    "private": true,
    "dependencies": {
        "bootstrap-sass": "~3.3.5",
        "bootstrap-block-grid": "~1.1.2",
        "modernizr": "~2.8.3",
        "hover": "Hover#~2.0.2",
        "animate.css": "~3.5.0",
        "wow": "wowjs#~1.1.2",
        "shufflejs": "~3.1.1",
        "responsive-bootstrap-toolkit": "responsive-toolkit#~2.5.1",
        "OwlCarousel": "~1.3.2",
        "jquery": "~2.2.0",
        "font-awesome": "4.4.0",
        "ekko-lightbox": "~4.0.1",
        "ace-builds": "~1.2.2",
        "imagesloaded": "~4.1.0"
    },
    "overrides": {
        "bootstrap-sass": {
        "main": [
            "assets/stylesheets/_bootstrap.scss",
            "assets/fonts/bootstrap/*",
            "assets/javascripts/bootstrap.js"
        ]
        },
        "font-awesome": {
        "main": [
            "fonts/*"
        ]
```

```
        },
        "OwlCarousel": {
        "main": [
            "owl-carousel/owl.carousel.min.js",
            "owl-carousel/owl.carousel.css"
        ]
        }
    },
    "devDependencies": {},
    "resolutions": {
        "jquery": "~2"
    }
}
```

2.12.3. CSS and Theming

The visual design – CSS and script – can be implemented based on the mark-up frameworks and strategy – either in the foundation module as part of a general theming approach or individually in the project layer modules. This allows more individual design per tenant.

Habitat Example

The theming in the Habitat example site is primarily based on Bootstrap. The module Foundation/Theming implements the base CSS theme based on an Atomic Design approach. It references no feature specific mark-up, but clearly follows the mark-up defined in the frameworks.

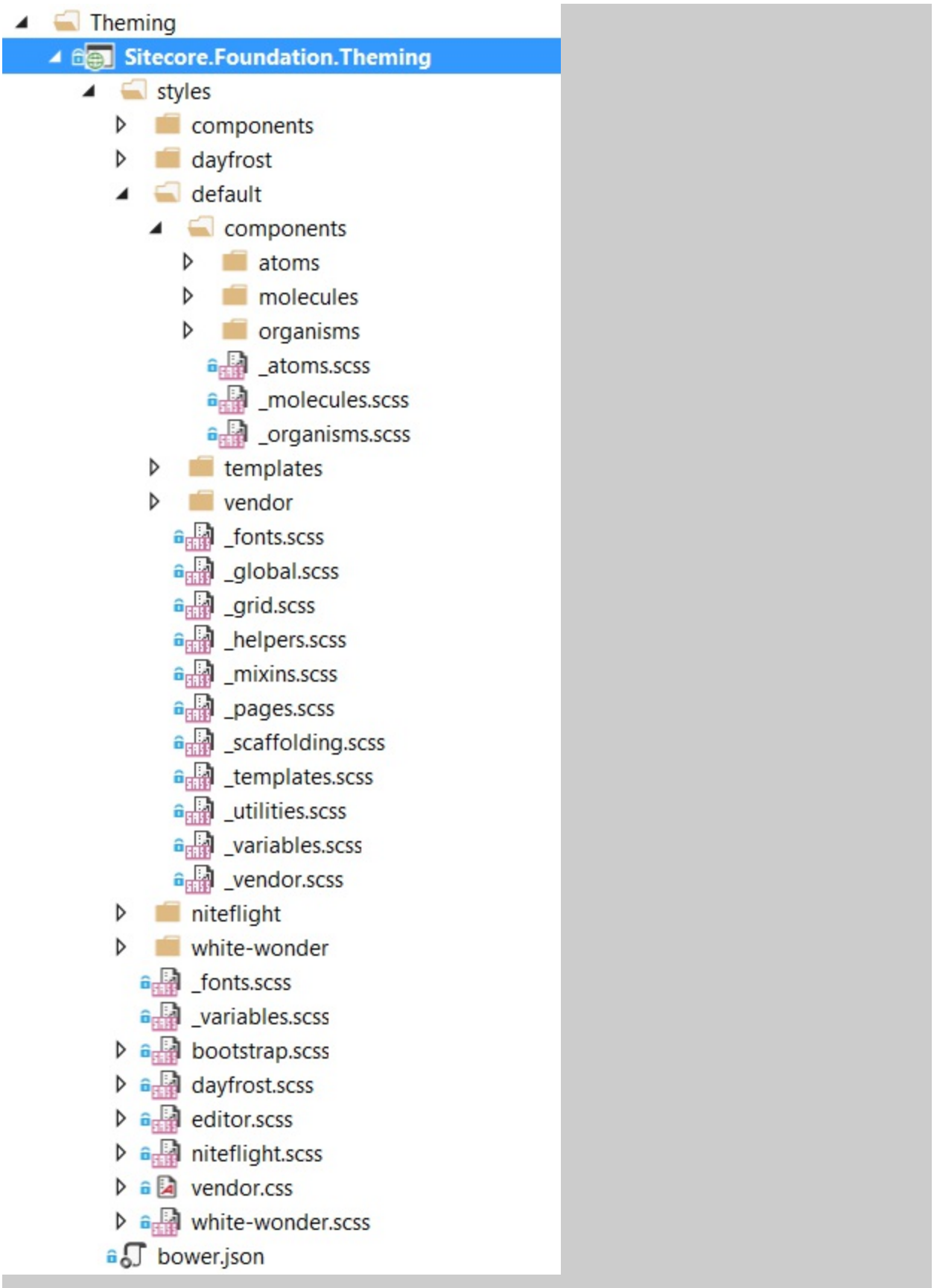

Theming
Sitecore.Foundation.Theming
styles
components
dayfrost
default
components
atoms
molecules
organisms
_atoms.scss
_molecules.scss
_organisms.scss
templates
vendor
_fonts.scss
_global.scss
_grid.scss
_helpers.scss
_mixins.scss
_pages.scss
_scaffolding.scss
_templates.scss
_utilities.scss
_variables.scss
_vendor.scss
niteflight
white-wonder
_fonts.scss
_variables.scss
bootstrap.scss
dayfrost.scss
editor.scss
niteflight.scss
vendor.css
white-wonder.scss
bower.json

Figure: Atomic Design Sass implementation in Habitat

The Helix approach will often require a close collaboration between front-end development and Sitecore/.NET development as it requires a common understanding of the architecture and reasoning behind it. A good relationship and understanding between the different competencies in the team is always a thing to strive for.

Avoid at all cost outputting HTML mark-up in code or business logic – try as much as possible to stick to a single technology – for example razor views – for mark-up, as this will facilitate management and increase flexibility immensely.

2.12.4. Scripting

Although front-end scripting technologies such as JavaScript are often associated with visual design and CSS, they are starting to serve more purposes in website implementations. And even though Sitecore is predominantly ASP.NET and C# driven, JavaScript and its associated frameworks are taking an increasingly important role in the actual business logic of the application.

Scripting technologies are typically related to the different layers, depending on what purpose they serve:

Framework technologies (such as jQuery, Angular.js, Backbone.js, Require.js and so on) are typically Foundation layer modules that provide a standardized API or framework for feature modules to leverage. These should therefore be introduced into the implementation in a Foundation layer module.
Functionality-specific scripts, i.e. presentation or business logic which enables parts of a feature to function belongs in the Feature layer. For example, scripts that enable menus to open or close, results to lazy-load on scrolling of the page and so on, belong in the module that has the related content or views.
The last group is presentation-related scripting, which relates to the way a page or parts of a page presents itself. This includes scripts that relate to the holistic visual design of the page and its CSS, and scripts that often require knowledge of the DOM of the page in order to function. These types of scripts belong with the page or site specific design implementation in the Project layer (see Scripting)

In order to support this type of separation, the implementation will need a mechanism for features to register scripts to be loaded – either solution-wide or on the pages where they are used. Since Sitecore does not natively provide such a mechanism, you will need to choose the approach and provide this in the Foundation layer. This can be accomplished by using either a standard framework, such as

require.js, or custom logic implemented in a Foundation layer module.

Habitat Example

The Habitat example site provides an Assets module in the Foundation layer that allows modules to register script or styling files through .config files, which is subsequently loaded on the website. Files and assets can also be bound to specific renderings and dynamically loaded on pages which use those renderings.

Although the Habitat example shows a highly pluggable and Sitecore-centric approach to assets, it does not include aspects such as minification and other optimization techniques. These can however be integrated using standard technologies such as require.js.

3. DevOps and development lifecycle management

The development lifecycle consists of a number of different phases. The length of each phase depends on the development methodology you use (Waterfall, Agile, etc.). These phases can include Analysis, Design, Specification, Planning, Development, Testing, Deployment and more.

Depending on the complexity of your specific Sitecore implementation – or general Sitecore governance model – the need for automation of all or some of processes in the Application Lifecycle phases can vary greatly. Take, for example, the differences between a small in-house development team independently working solely and consistently on a single Sitecore implementation, a Sitecore implementation partner trying to consistently develop and maintain numerous Sitecore implementations and an enterprise product owner trying to govern a Sitecore implementation with multiple implementation partners and design agencies working together. In other words, in some governance models, it would be acceptable to have documented and manually executed processes, where as in others it could be catastrophic to be unable to repeat processes consistently.

Therefore, DevOps can be as much a challenge of identifying and prioritizing the important processes to automate as it is the task of actually implementing the automation.

In the scope of this document we will only look into DevOps processes in specific phases of the application lifecycle:

Development
in which the features or modules are being built

Build
or *Integration* in which the implementation is put together as a single

testable package

Testing
in which the features or integrated package are tested against the specifications

Deployment
in which the package is ultimately deployed onto the production environment

Please note that this is in no way an in-depth look at the DevOps or a governance model for these phases, but rather is an inspection of some of the Sitecore related processes and activities in these phases.

3.1. Development

Development is typically the process in which the team will spend the most time, and therefore an obvious target of optimisation and automation. Two parts of the development process in particular should be the focus, as they can be instrumental in productivity and consistency: the initial setup and continuous development.

3.1.1. Setting up a development environment

Optimising the task of setting up a local development environment is not just about on-boarding new developers or resetting a developer's local environment – although these tasks can be important to optimise. It is also – and sometimes more importantly – about being able to consistently and quickly set up a given version of the complete clean implementation for testing and troubleshooting.

Typically, after the initial build phase – where there often is an abundance of resources – comes the support or extension phase. In this phase, where there are often fewer resources and maybe even less experience with the implementation and processes, there is often a need for working on multiple branches at the same time, testing on production-like environments, the need to establish troubleshooting environments, etc. and so the need to quickly spin up a specific version of the local development environment increases.

Setting up a local developer machine with a running environment can be more or less complex given business requirements and dependencies, but the following is an example of the tasks that could be involved and example of what each might contain:

Check out
Get the required branch or version from the version control.

For example: This is typically a manual clone from git or similar of the version which needs to be running.

Dependencies
Import and configure external frameworks and dependencies

For example: Execute package managers such as NuGet or npm or run custom scripts to restore required dependencies

Build
Compile the modules

For example: Run compilers such as MSBuild and other tools, such as CSS/JavaScript processors.

Baseline
Set up a running website/environment based on a clean Sitecore or a defined baseline

For example: Install a local Sitecore setup, for example through SIM, PowerShell or similar. This may involve restoring backup databases with production-like test data. This may also involve restoring a virtual machine with the full running environment.

Publish
Publish the modules to the baseline website

For example: Deploy the compiled assets and files to the running website - for example through Microsoft WebDeploy, PowerShell or similar.

Configure
Configure the baseline for the modules

For example: Add implementation specific configuration – to, for example, web.config or other files – using MSBuild, PowerShell or similar.

Deserialize
Restore the Sitecore items

For example: Deserialize or install implementation or version specific items into the running website.

It is recommended that you automate these steps as far as possible.

This will increase productivity and quality in both the short and especially the long run.

Habitat Example

Habitat includes an example of an almost fully automated local environment configuration using a combination of various tools. The overall build system uses Gulp, which can be triggered through the Visual Studio Task Runner.

Please note that the Habitat example site has all content versioned through the version control system – unlike a production environment in which content is managed in production – and can therefore use a clean Sitecore installation as the baseline.

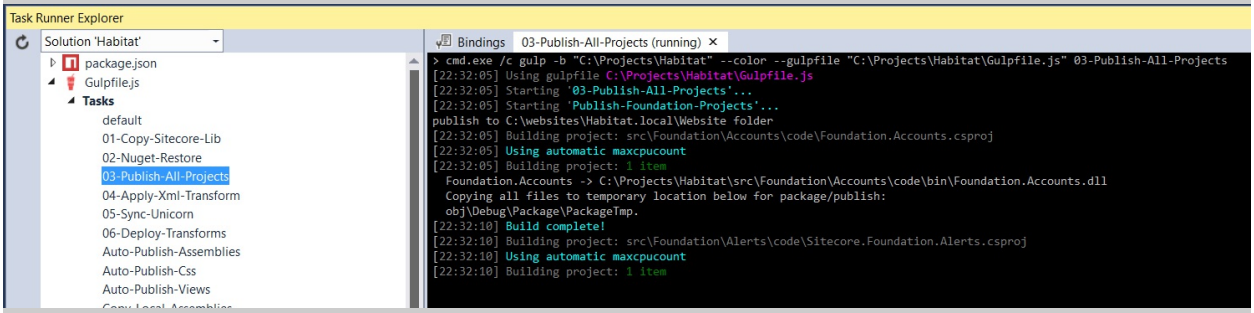


Figure: Running the Publish task through the Visual Studio Task Runner

3.1.2. Local deployment

One of the most frequently asked questions when working with Sitecore is whether to develop locally inside the web root or outside. Although the general recommendation is to work outside the web root, a general qualification is needed.

There are two major aspects which should lead to choosing one or the other:

Firstly, it is very important to be able to separate the implementation specific files and changes from the standard frameworks and files. The importance of this separation cannot be overstated, as it will help in many aspects in the long run, such as general development, troubleshooting, upgrades and more. This separation is easier if the implementation specific files are physically separated from the files in the standard frameworks etc. On the other hand, tools and processes might cater for this separation, even when working inside the web root.

Generally, if tasks that are repeated frequently become taxing, they will be circumvented – and this will lead to inconsistencies in the processes and ultimately inconsistencies in quality. Therefore, secondly, the ease of development is important. Working outside the web root will lead to an additional step in the development process – deploying the change to the running environment – and given the frequency of this task it is imperative that this is easy. Tools and automated scripts can help this to become almost transparent for the developer.

Habitat Example

The build system in Habitat includes a number of tasks that will watch files of a given type and deploy these automatically to the local instance. This includes changes to assemblies after a build, changes

to .cshtml view files and changes to .css files. Habitat also includes tasks for deploying all files of a given type.

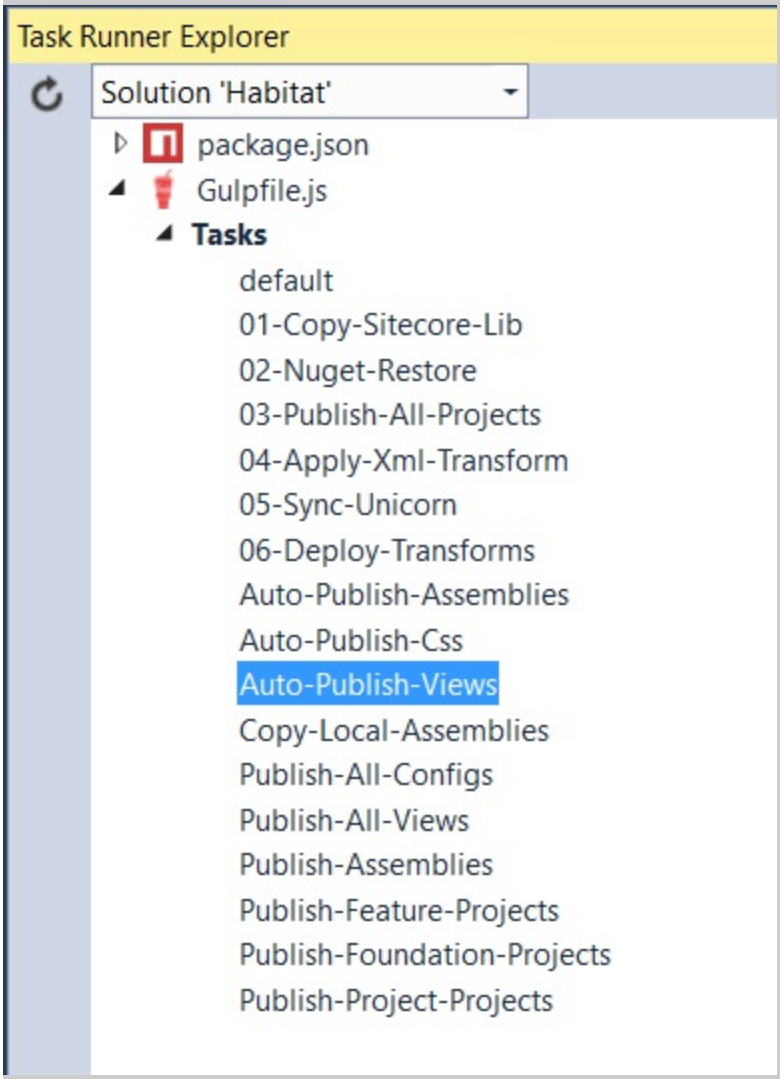


Figure: All gulp tasks as available through the Visual Studio Task Runner

3.1.3. Version Control

Throughout the development phase, the single source of truth should be your version control system. In order to integrate and build versions of your implementation across environments in a consistent and automated fashion, you should strive to keep everything that your toolset needs in the version control system. This does not in any way mean that all files, frameworks and data need to be versioned, but rather that your version control repository should have enough information to restore a given version of your implementation.

This ambition assures that you consistently organise and version files, frameworks and data together that belong together, so the individual parts have less of a chance of getting out of sync with others.

The parts you should look to store together in your version control system of choice include, but are not limited to: source code, tests scripts, configuration files, Sitecore definition items, and build scripts package manifests.

Some types of data can be harder to version control than others – for example Sitecore items – (see Managing Sitecore Items). Finding tools to manage all types of data through version control should be a priority as it will greatly diminish the risk of manual errors and inconsistent deployments.

3.1.3.1. Versioning external requirements

Keep in mind that your version control is not an integration environment. In order to improve your maintenance and to better upgrade and control the various parts of your solution, limit your version control system to only contain the implementation additions and changes. As much as possible, avoid adding standard frameworks and standard files to version control. This includes the implementation web.config and standard Sitecore config files (see

Managing .config files).

There are a number of package managers, for example NuGet, Bower, node/npm, that are useful to help install and integrate external frameworks for different purposes. These all largely use the same approach. They require a manifest file that points to the required versioned packages. This manifest can be versioned alongside your implementation specific files to allow your system to pull in the packages from external repositories.

Habitat Example

Habitat uses NuGet packages for .NET dependencies and Bower for front-end technologies. The build system in Habitat is based on gulp and node.js, and therefore uses the node.js Package Manager (npm) for build script dependencies.

Some external dependencies, for example Sitecore itself, cannot immediately be integrated via a standard package manager such as NuGet – or might require custom configuration to be integrated correctly. For this purpose, consider building custom scripts that automate the integration or configuration of the dependencies. Remember to version these scripts (either in the version control system itself or through a package manager manifest) as part of your implementation.

Habitat Example

Build scripts in Habitat are versioned as part of the implementation, and have a simple task for pulling in Sitecore dependent assemblies for the build.

Note that there are approaches to adding the Sitecore assemblies to a local NuGet repository. One option is to use the Sitecore Instance Manager (https://marketplace.sitecore.net/en/Modules/Sitecore_Instance_Manag

3.1.3.2. Environment specific settings

Generally, be careful about storing environment specifics in version control, as version control tools are typically used exclusively used in the development process, and not during deployment or system configuration. Having a too rigid process for deploying environment specific changes (for example connection strings for new servers etc.) might lead to changes directly in production environments – circumventing processes altogether. Therefore, pay close attention to who “owns” the environments and where these environment specific settings then go.

For example: Consider an implementation team that consists of a development team, a QA team and an IT admin team, . The development team owns the individual development machines, CI environment and even the QA environment, whereas the IT admin team owns the pre-production/staging test servers and the production environment. If an emergency arises and the IT team needs to switch a server (SQL, CDN, etc.), they must ask the development team to make the change. This creates a bottleneck. Therefore, the settings specific to the production environment , such as those that live in the web.config, should not be maintained through the development team tools, but rather through the IT admin tools.

3.2. Build and integration

3.2.1. Building your solution

Building your implementation is typically part of three processes:

Local builds
which typically happen in Visual Studio and assisted with scripts or tools for deployment onto the local IIS (See Setting up a development environment).

Continuous Integration (CI)
which happens automatically on a build or CI server when someone commits a change to the version control.

Deployment
or *integration*, which is when the implementation is combined into a distributable package for deployment through the environments.

The continuous integration and deployment builds are very similar as they most often deal with the combined implementation. This is opposed to the local builds, where a developer might be working on a single module or modules logically grouped together in a single Visual Studio solution (See Visual Studio).

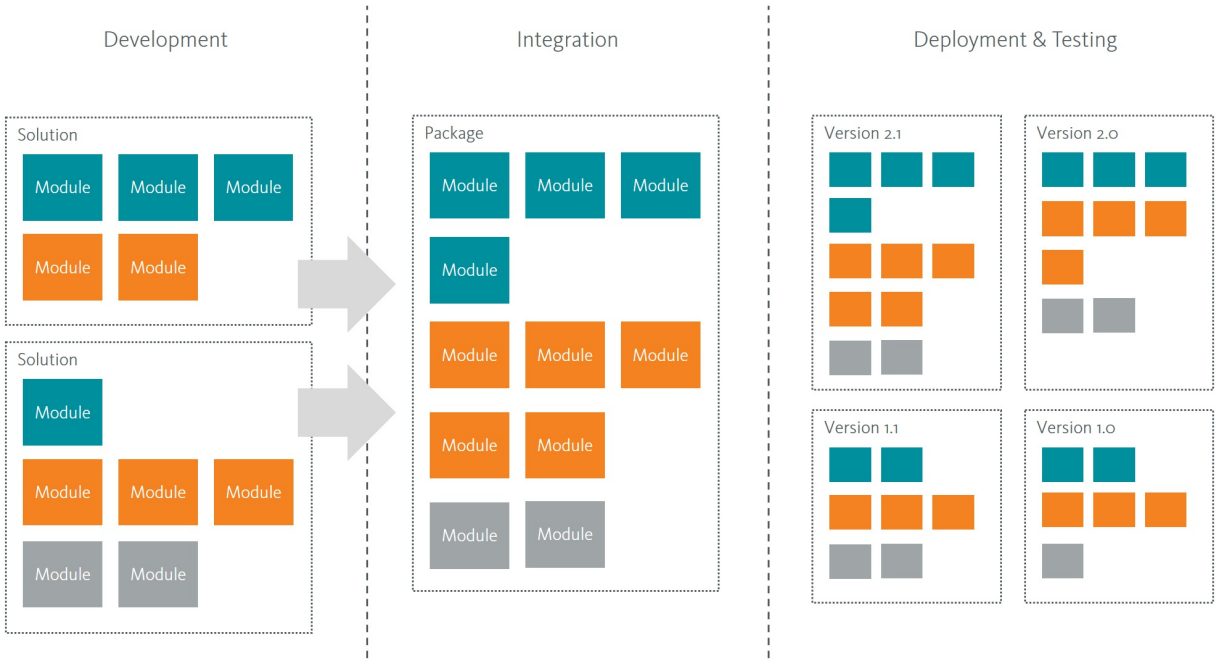


Figure: Development, integration and deployment process

Modular architecture is really only focused on the development process, not the deployment. Do not confuse Helix with a modular approach to building or running Sitecore environment. In the integration process (continuous or during deployment builds) the modules should be combined into a versioned and distributable package that can be tested and deployed consistently through the environments (see Deployment).

It is recommended to run deployment builds – and continuous integration builds – on a separate build server, using a dedicated build system such as Microsoft Team Foundation Server, Visual Studio Team Services, TeamCity, Jenkins, Bamboo, etc. This assures a consistent build and integration process, which is important for quality in the deliveries and agility in development and support.

3.2.2. Integration

Integration is the point where the implementation is combined into a versioned solution. It is recommended to include as much as possible in the packages generated in order to allow the deployment process to be as simple as possible. The packages should include your implementation specific files, standard Sitecore and other framework files, configuration files, definition items in the Sitecore databases (see Item types) and any scripts to run during the deployment process. By covering as much as possible in this process result in a more simplified deployment process by detaching the deployment process from your development tools and repositories. Ideally only the configuration of the specific environment and roles should happen during the deployment process.

Depending on your tools stack – version control, configuration management, item serialization etc. – various tools can help you with the integration of your modules and solutions into a distributable package. The format of this package will depend on the deployment strategy and tools you choose. For example, Team Development for Sitecore covers the complete application lifecycle from development through to integration, and supports generation of Sitecore Update and NuGet packages that can be automatically installed with for example Octopus Deploy.

Habitat Example

The internal build process used for the Habitat example site runs continuous integration and deployment through a TeamCity build server. Please note that these tools are publicly available on GitHub.

The CI process is triggered by each change in the GitHub version control system and runs separate builds for each branch and pull request. The process executes the following tasks

Restore NuGet packages and other external dependencies
Build using MSBuild
Run Unit and SpecFlow Tests
No packages or assets are created from the CI build.
The deployment build is triggered nightly on changes to the master branch – or can be triggered manually to create official releases or special branch releases. The process runs the following tasks:

Restore NuGet packages and other external dependencies
Build using MSBuild
Run Unit tests
Setup a clean Sitecore on the build server
Publish solution to the Sitecore instance using the Gulp build scripts
Create Sitecore package by executing the Sitecore Package Generator
Publish Package to GitHub as a release
The Sitecore package distributed via the GitHub releases page (https://github.com/Sitecore/Habitat/releases).

Internally in Sitecore, official releases of Habitat and our demo sites that are built on Habitat are distributed as Sitecore Instance Manager (SIM) packages, which include database backups and analytics data. This allows deployment to local machines to happen in a one click process through the SIM user interface.

These SIM packages are created on the build server too using PowerShell and the Sitecore Instance Manager command-line interface (https://github.com/sitecore/sitecore-instance-manager).

3.3. Testing

3.3.1. Managing Tests

All tests, independent of testing methodology or technology, should be located in the /tests folder under the corresponding module. If there are multiple testing methodologies for one module, for example unit testing and integration testing, the module can contain sub-folders.

Habitat Example

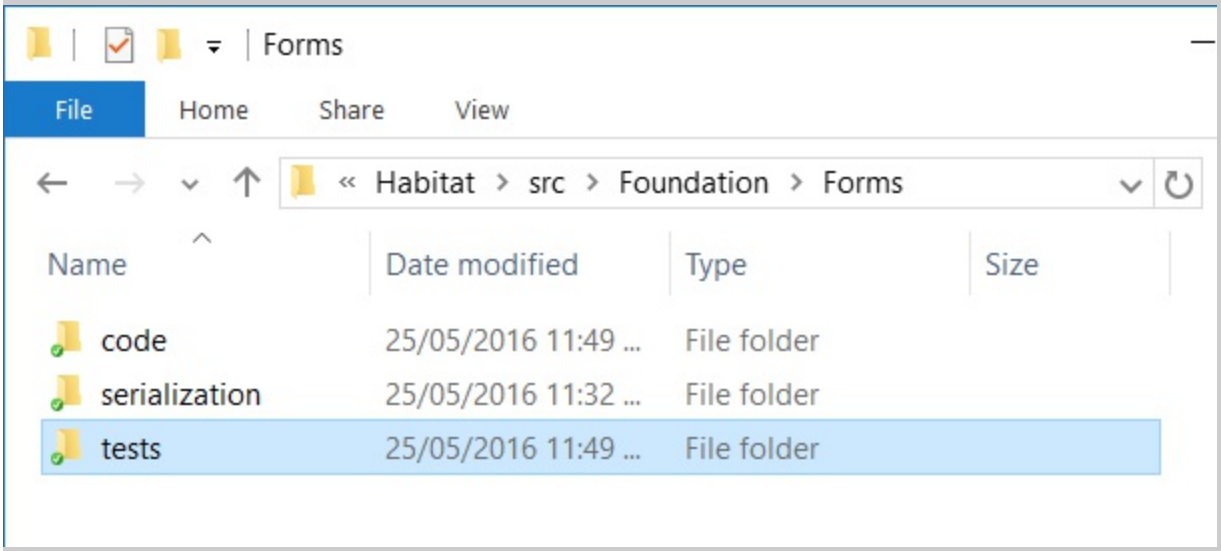


Figure: Tests folder under the Forms module in Habitat

3.3.2. Unit tests

There is a fair number of unit testing approaches and frameworks for .NET and Sitecore, but as with other technologies and methodologies, Helix does not dictate which to choose. We do emphasise the value of adding unit tests to your business logic, and highly encourage the general practice.

On disk, unit test projects should be contained in the /tests folder below the module folder. In Visual Studio the unit test project should be in the same Visual Studio solution as the corresponding module projects and contained with the module solution folder.

Habitat Example

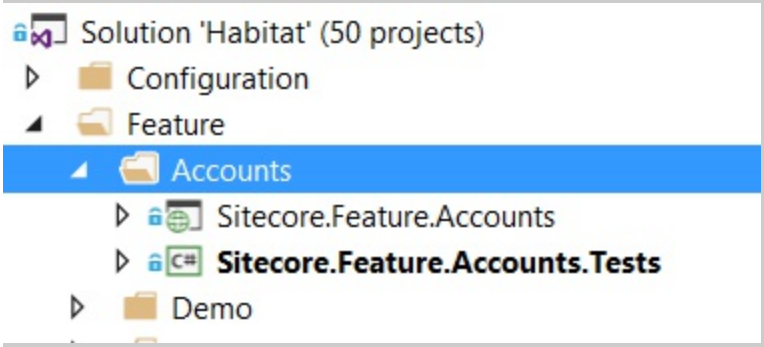


Figure: Unit Test Project in Visual Studio

Habitat Example

Unit Tests in Habitat are developed as Arrange-Act-Assert and uses the following modules:

xUnit (https://xunit.github.io) framework as the main unit testing framework
NSubstitute (http://nsubstitute.github.io/) for mocking objects
AutoFixture (https://github.com/AutoFixture/AutoFixture) for "Arranging" the unit tests
FluentAssertions (http://www.fluentassertions.com/) for the assertion

syntax
FakeDb (https://github.com/sergeyshushlyapin/Sitecore.FakeDb) for faking Sitecore objects and services
Example of a Habitat unit test for Sitecore.Feature.Language.LanguageRepository:

```
public class LanguageRepositoryTests
{
    [Theory]
    [AutoDbData]
    public void GetActive_ShouldReturnLanguageModelForContextLanguage(Db d
    {
        var contextItem = db.GetItem(item.ID);
        Context.Item = contextItem;
        var activeLanguage = LanguageRepository.GetActive();
        activeLanguage.TwoLetterCode.Should().BeEquivalentTo(Context.Langu
    }
}
```

3.3.3. Integration, Acceptance or other automated testing methods

When you are doing automated testing – and especially for automated browser testing – you are often running the tests on an assembled solution and therefore the tests might be dependent on the combined pages of the website. Even though your tests might just focus on the features or functionality in a single module, they depend on the functionality of integration mapping in other modules or layers. Therefore, the tests belong in the Project layer as opposed to together with the module in the Feature or Foundation layer.

Code and other files related to tests – independent of whether they are unit tests or other testing methodologies – belong in the /tests folder under the respective module.

A single implementation can consist of multiple Visual Studio solutions. It might be beneficial to keep automated testing projects (apart from the unit tests) in a separate solution than the code.

Habitat Example

Habitat has a whole range of acceptance tests built with SpecFlow. These are generally structured to test the functionality within a single module. But since they rely on the integrated Habitat website to run, they are all located in the *Habitat* project layer module.

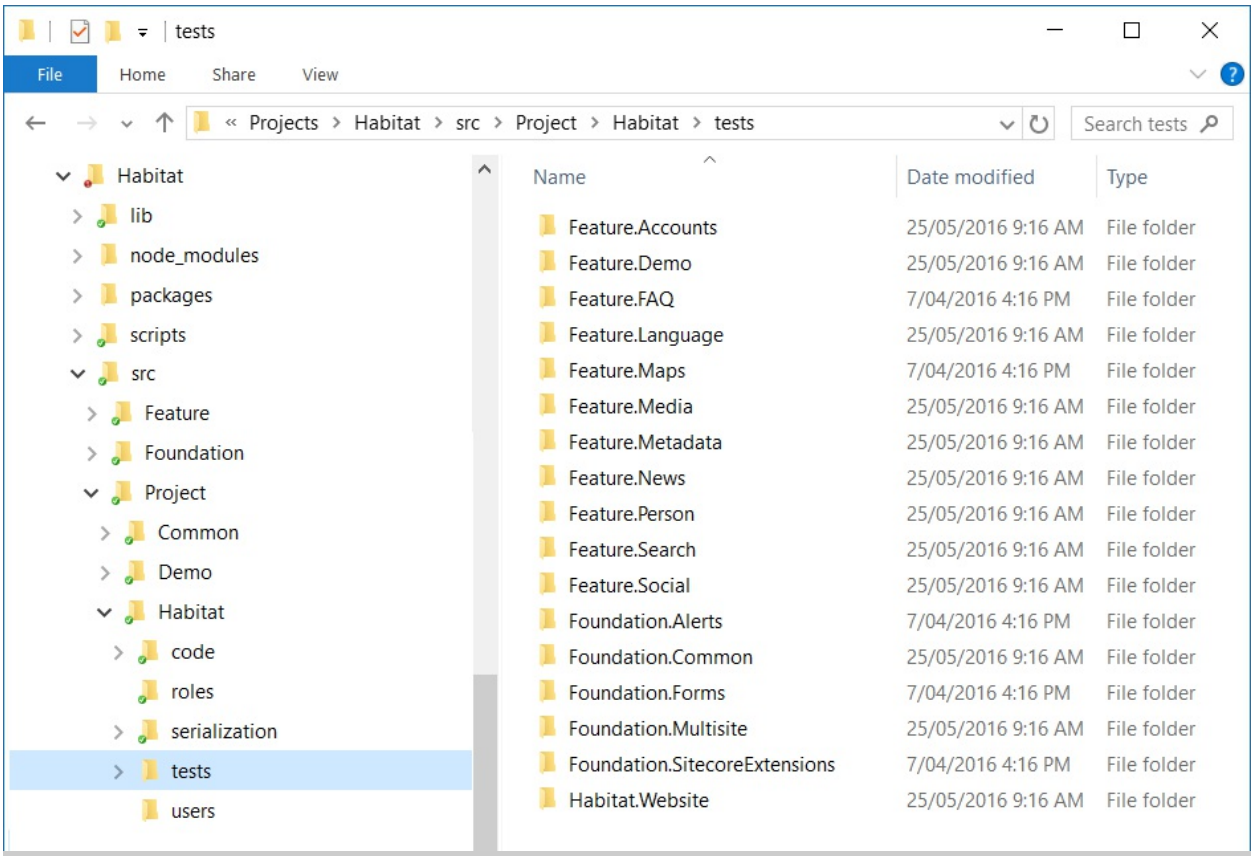


Figure: SpecFlow Projects under the Habitat Project Module

3.4. Deployment

3.4.1. Deployment strategy

A successful deployment strategy is highly connected with a good versioning strategy. To consistently run a successful develop, test, deploy cycle, it is important to be able to consistently version the full application stack you are running on each environment and to be able to recreate this across environments.

The modular architecture - as exposed through Helix - is not to be confused with a modular deployment model where a base version of the system is deployed and modules are installed on top of it – and where the product lifecycle of the modules, underlying system and websites on top of it are independent.

In Helix – as with development and Sitecore generally – it is highly recommended to follow a strict develop, test and deploy lifecycle for the whole application. This means that although development of the individual modules might have different development speeds, the assembly and versioning of the application stack happen at a particular point in the application lifecycle, before the deployment onto the environments. In other words, it is discouraged to do partial deploys of modules or features from development to production. Deployments should happen on an application-level scale, which fits with the availability requirements of the business.

Managing the full application stack - which includes vendor systems such as SQL Server and Sitecore XP, modules for those systems, and the underlying operating system - can be handled in a variety of ways. The best option often depends on the level of automation of the deployment process. In most cases this is done through documentation. It can also be handled by manifests that are read by automation systems and that then automatically pull in the right underlying systems to create the application stack in the build process. This methodology is done by popular package systems such as Node/NPM, NuGet, Bower and even the Sitecore Instance

Manager. This methodology can also be employed as part of your custom build process to build more complete deployment packages.

Habitat Example

Habitat leverages a number of package tools including NuGet, Node/npm and Bower to pull versioned packages into the build process.

The internal continuous integration and build server for the Habitat project – which is hooked up to the public GitHub repository – is set up to create versioned Sitecore packages on nightly builds or releases. This means that the full Habitat application with all modules can be installed as a single package onto any environment – and thereby it is possible to consistently recreate a given version.

Furthermore, the server fully automated reinstalls the Sitecore IIS website on an internal test server as part of the automated build. This includes reinstalling a blank Sitecore XP and required modules, and installing the Habitat content and application package on top. Even though this server is not publicly available as an example, parts of the build scripts, specifically the Gulp scripts, for the CI process are in the Habitat GitHub repository.

Tools such as Microsoft Team Foundation Server or Octopus Deploy can help immensely with automating of the build and deployment process and thereby in securing consistent versioning and deployment across environments.

3.4.2. What to deploy and to where

Some data and configuration will sit outside the development and build phases of the application lifecycle. Therefore, your process should cater for this environment specific data or configuration (see Value Scope).

Some data and configuration is owned by the development process (such as C# code, views and the template structure) while others are owned by the production environment (such as the content items). It is important that your deployment process takes this into consideration and maps this carefully. This is especially important for items in Sitecore and configuration in .config changes –both of which can be managed in multiple places. Also, this is important in an automation process, as you do not want to overwrite production content or configuration by accident or be unable to test on a representative production-like dataset.

The detailed ownership and direction of flow depends on the business logic and requirements of the solution. For example, although templates are most often managed in development and should at all times be deployed from development to production, in some rare cases Project layer templates, such as Page Type templates and Datasource templates, can be managed partly in production to allow for dynamic fields and content.

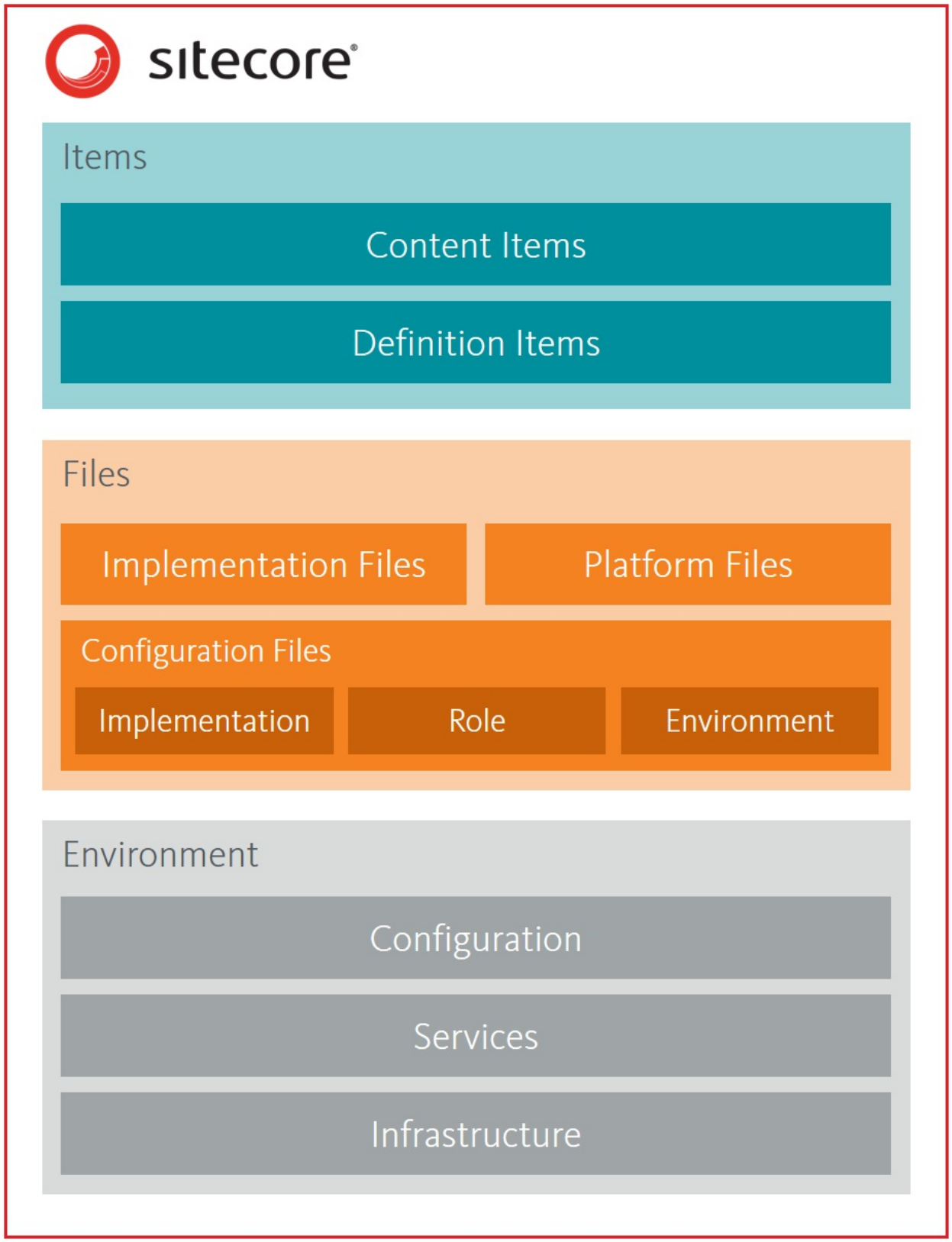


Figure: Parts of a deployed Sitecore instance

The diagram above breaks down the overall parts of a typical deployed Sitecore instances, and highlights some of the data types and configurations to consider. The following describes the diagram in more details. Ownership in the table means that a change in the given environment always takes precedence – and therefore will overwrite changes in other environments.

Items

What is it: Items in the Sitecore databases (core and master)

Content Items

What is it:	The content that is displayed on a website or other digital channel, and settings that affect the behaviour of a website or another digital channel. This content can be edited by content authors.
Owned by:	Production
Direction:	Typically moved from production to other environments for testing. Some items might initially be created in development and deployed to production in an install-once process.

Definition Items

What is it:	Sitecore data items that configure the implementation and that have a direct relationship with the presentation and business logic in the code, for example templates, fields, layouts, placeholders etc.
Owned by:	Development
Direction:	Installed as part of application deployments from development to QA and ultimately production.

Files

What is it: Files on disk on the servers

Implementation Files

What is it:	The implementation specific files, for example assemblies, views, CSS and JavaScript files.
Owned by:	Development

Direction:	Installed as part of application deployments from development to QA and ultimately production.

Platform Files

What is it:	Vendor specific files, i.e. files that are installed as part of a standard module
Owned by:	Development
Direction:	Installed as part of application deployments from development to QA and ultimately production or as part of the initial configuration of the environment.

Configuration files

What is it: .config or other files that configure the system.

Implementation

What is it:	Configuration that sets up the functionality, but that is application wide
Owned by:	Development
Direction:	Installed as part of application deployments from development to QA and ultimately production.

Role

What is it:	Configuration that sets up the instance as a particular Sitecore instance role, for example a delivery, management or xDB processing server.
Owned by:	Deployment
Direction:	Set up by the deployment process as part of the configuration of the installation. Any individual role configuration files can be managed in Development as part of the implementation.

Environment

What is it:	Configuration file changes relating to the specific running server or specific environment, for example connection strings, server names, domains etc.
Owned by:	Deployment
Direction:	Set up by the deployment and managed in the specific

environments.

Environment

What is it:	The infrastructure needed for running the Sitecore and the application.

Configuration

What is it:	Server or environment specific configurations such as network, DNS, hosts file changes, machine.config etc.
Owned by:	Deployment
Direction:	Set up as part of the initial deployment process. Can be automated but often is not.

Services

What is it:	Related services running in the environment or on the instance server, for example operating systems, IIS, SQL servers, Windows Services etc.
Owned by:	Deployment
Direction:	Set up as part of the initial deployment process. Can be automated but often is not.

Infrastructure

What is it:	The underlying server, virtual or physical,
Owned by:	Deployment
Direction:	Set up as part of the initial deployment process. Can be automated but often is not.

Once you have mapped the ownership and direction of data in your implementation, avoid making changes that violate this, for example by submitting code or configuration changes directly to test or product environments and circumventing the QA and development procedures. Violating this mapping is highly discouraged and should be avoided at all cost.

Consider every type of deployment onto the environments, including initial deployment, vendor upgrades and minor or major application updates, when designing your deployment process. Remember to

take into account the availability of the running solution.