Orchard Dojo Library

The Orchard Dojo Library is a portable package of Orchard goodies. It supplements Orchard Dojo's trainings and tutorials. These are also part of the best practices and guidelines we use at Lombiq.

- Topics:
  - Orchard Link Collection: useful Orchard-related links
  - Software Development Guidelines: various development guidelines we find useful
  - Orchard FAQ: some frequently asked questions about Orchard usage and development
  - Orchard Training Guidelines: used for Orchard trainings
  - Development Utilities: utilities that make Orchard development faster and easier
  - Orchard Wiki: pieces of Orchard-related knowledge, organized into articles, wiki-style
  - Examples: examples of how to do something in Orchard
- Contribution Guidelines
- License

You can download the whole Library, file issues or fork it from its repository. Also you can download the Library's textual content as one big concatenated document in HTML.

Orchard link collection

Official sites

- Orchard homepage
  - Documentation
  - Translations (and for modules in the Gallery)
- Orchard GitHub project homepage
  - Issue tracker
  - Releases
- Orchard Codeplex project homepage: currently Orchard moves away from Codeplex, only the discussion board remains
- Orchard Gallery
- Orchard continuous integration server (click "Login as a Guest")

Community sites

- China
- France
- India
- Iran
- Magyarország

Blogs

All blogs from the Orchard community are automatically scraped by Orchard Blogs.

Technologies behind Orchard

- ASP.NET MVC
- Autofac for dependency injection
- C#
- jQuery as the javascript framework and jQuery UI for UI enhancements
- Nhibernate for object-relational mapping

Miscellaneous

- Community Meetings recordings and meeting room
- DotNest, the Orchard SaaS
- Orchard Beginner
- Orchard Cheatsheet
- Orchard Dojo
- Orchard Harvest conference website
- Orchard Marketplace
- Orchard Prime
- Orchard Pros
- .NET Foundation, the foundation supporting Orchard
- Show Orchard
- Stackoverflow Orchard questions
- Try Orchard! where you can test drive already installed Orchard demo sites

Software development guidelines

If something's not specified, general C# guidelines apply: C# Coding Conventions and General Naming Conventions. Also see Orchard Code Conventions. The talk How To Design A Good API And Why It Matters is an evergreen as well as the .NET Framework Design Guidelines Digest.

- Best practices
- Code styling
- Naming conventions
- Rules of thumb for refactoring
- Inline documentation guidelines
- Code review guidelines
- Development environment advices
- Orchard performance optimization guidelines
- Updating your Orchard source
- Creating a new repository of the full Orchard source for your project

Coding best practices

General principles to keep in mind

- Don't Repeat Yourself
- Loose coupling
- Composition over inheritance
- Single responsibility principle and separation of concerns

Topics

- C# best practices
- Orchard best practices
- JavaScript best practices
- CSS best practices
- Source control best practices

C# best practices

When returning a collection, always return an empty collection if there are no elements, but never null. When accepting a collection as a method argument, however, always check for null.

```
IEnumerable<int> MyMethod(IEnumerable<int> collection)
{
    // Check for null and handle it somehow
    if (collection == null) throw new ArgumentNullException("collection");
    if (nothingToReturn) return Enumerable.Empty<int>();
    else return normally;
}
```

Keep interfaces as short as possible so it's relatively simple to provide alternative implementation for them (even when doing unit testing).

If a method would serve just as a shortcut for multiple method calls on the same interface, use extension methods. Whether or not to use an extension method should be decided on a case-by case basis as future-aware as possible: only use extension methods if the shortcut is (almost) trivial and add the method to the interface if the optimal solution is more likely to depend on the specific implementation.

```
// Good example: the shortcut is simple
public interface IService
{
    void Register(int id);
}

public static class ServiceExtensions
{
    void Register(this IService service, DbEntity entity)
    {
        service.Register(entity.Id);
    }
}

// Extensions are also useful if you want to provide default arguments for methods and want to do it with overloads
public interface IService
{
    IEnumerable<DbEntity> GetItems(int maxCount);
}

public static class ServiceExtensions
{
    IEnumerable<DbEntity> GetItems(this IService service)
    {
        // This extension provides a default value for the GetItems() method call
        service.GetItems(15);
    }
}


// Bad example: GetMany() results in many Get() calls. The implementation of GetMany() is something that the implementation of IService is likely to decide on better.
public interface IService
{
    object Get(int id);
}

public static class ServiceExtensions
{
    IEnumerable<object> GetMany(this IService service, IEnumerable<int> ids)
    {
        return ids.Select(id => service.Get(id));
    }
}
```

For the extension class use the naming convention of [interface name without the leading I]Extensions as above and put them in the same namespace with the interface (so consumers seeing the interface will likely be able to see the extensions without adding another using statement).

Try to keep the maximal number of arguments on a method to 3.

Almost always return an interface type and return the most generic one making sense for the typical consuming code.

```
public interface IService
{
    // When in doubt, use IEnumerable<> for collections
    IEnumerable<int> GetItems();

    // If you need List's certain features like mutability or the ability to access items by index commonly in the consuming code return an IList<>
    IList<int> GetItemsList();
}
```

Never use view models in a service interface: services and views have nothing to do with each other.

Use the "empty pattern" where you want to provide a default object.

```
public class MyClass
{
    // Default will return this single instance, initialized with its default constructor
    private static readonly MyClass _default = new MyClass();
    public static MyClass Default { get { return _default; } }
}
```

This is used by .NET's String class (String.Empty) and also by Orchard's QueryHints class (QueryHints.Empty).

When checking if an `IEnumerable<T>` is empty always use `enumerable.Any()` instead of `enumerable.Count() == 0`.

When writing "async void", think twice. Unless written for event handlers async void should be avoided at least because exceptions in such methods can tear down the whole application. See this SO post. If you write such methods always surround it with a try-catch that catches the base Exception so no exception can escape.

If you insists on using short variable names then use `ex` for exceptions and `e` for event handler arguments.

When your class implements multiple interfaces with a lot of methods it's best to explicitly implement them. This way it's immediately visible which method corresponds to which interface.

Using initialization methods on your classes like `Init()` is a sign of bad design most of the time as this requires the user to remember to call it before anything can be done. Consider refactoring the class to require necessary data through the constructor (probably even using a static factory) or by computing initialization data on the first demand, lazily.

Service classes should be stateless, i.e. their methods should give the same output for the same input.

When referencing another project from the same solution always add a project reference, not an assembly reference.

Orchard best practices

Always do part shape-related heavy work in shape factories inside drivers: this way if the shape is not displayed (i.e. not specified in or hidden from Placement.info) no work will be done.

```
protected override DriverResult Display(MyPart part, string displayType, dynamic shapeHelper)
{
    return ContentShape("Parts_My",
        () =>
        {
            // This delegate will only run if the shape is actually displayed.
            var heavy = /* Some heavy work */;
            return shapeHelper.Parts_My(Heavy: heavy);
        });
}
```

When writing a theme if something is achievable by only CSS, then use only CSS and avoid having shape template overrides with minimal modifications. If you absolutely have to create shape overrides then try to override the most specific shape possible: e.g. if you need to override the markup of blogposts' date shown then override just Common.Metadata (the shape responsible for showing the date) and not the whole Content shape.

If a template uses a static resource (stylesheet or script) always include/require it there even if the template is part of a bigger layout where those resources are already referenced. This makes it easier to keep track of dependent resources and is not prone to errors caused by changes outside the specific template.

For improving client-side performance by preventing blocking script loads always include scripts in the foot if they're not required immediately on page load. Also consider using the async attribute on scripts (by setting it with SetAttribute() at the time of inclusion) if the order in which they're executed is indifferent.

```
@{
    // This script will be downloaded asynchronously, without blocking the page loading, but you can't count on it being available at any point in other scripts (so if you have de
    Script.Include("my-async-script").SetAttribute("async", "async");

    // This script will be downloaded synchronously but since it's in the foot it won't block the page load and the user will be able to see the full page sooner.
    Script.Include("my-script.js").AtFoot();

    // Use such usings to run script blocks depending on foot scripts
    using (Script.Foot())
    {
        <script type="text/javascript">
            // Use footscripts here
        </script>
    }
}
```

When you have multiple features in a single module always make the sub-features depend on the main feature for clarity. This will prevent confusion if you want to place some common functionality in the main feature. It should also be the requirement anyway: sub-features are in that module because they have something in common with the main feature.

Always set the Build Action of non-code files (like Placement.info) to Content (under the file's properties in Visual Studio) if they are included in the project (and don't have Content set by default, what they mostly have). Otherwise MSBuild will fail when building Orchard. See a related blogpost.

Although not mandatory, it's good practice to route all your admin controller to under /Admin in a similar way how controllers named AdminController are routed by default. This makes it easier to set up rules for the admin area if one needs it.

Texts presented to the user should always be in form of LocalizedStrings (aka T()). When you want to display dynamic data in the string, it should always have its parameters supplied to it. Never concatenate localized strings with other values as this prevents complete localization. E.g. if you want to display the number of elements use a printf-like pattern:

`T("Number of elements: {0}", Model.Count)`

See the relevant documentation.

It's nice to have a consistent ordering for dependencies in module manifest files. A good way is to begin with third-party features, then list built-in ones (Orchard.*), both in alphabetical order.

When writing a recurring scheduled task (i.e scheduled tasks that re-schedule themselves) then add the re-scheduling as early as possible to the task handler's Process() method. This lowers the chance of an error causing the task not to be re-scheduled.

When doing I/O-bound work, always use async APIs if available (e.g. web requests, file writes). Using async I/O greatly increases the throughput of the server by not blocking threads to wait for I/O completion.

When you add a client-side plugin to your module or theme (like a jQuery plugin that uses various JS and CSS files, has a readme, etc.) then it's best to keep the folder structure of the plugin intact and copy it to the extension's Content folder (a folder simply serving static files with the same Web.config as Styles and Scripts folders) under its own subfolder. This way maintaining and upgrading the plugin will be easier, not to mention that developers will be able to see all of its files at once.

You'll be able to still include static resources from such a folder through the ResourceManager, you'll just have to use relative paths, e.g. like this in a resource manifest:

```
manifest.DefineScript("MyScript").SetUrl("~/Themes/MyTheme/Content/Plugin/script.js"...
```

The same goes for `Script/Style.Include()` calls from view templates.

When writing Migrations it's best to consolidate the latest schema in the Create method and only make UpdateFromX() run for existing installations.

```
public class Migrations : DataMigrationImpl
{
    public int Create()
    {
        SchemaBuilder.CreateTable(typeof(PersonRecord).Name,
            table => table
                .Column<int>("Id", column => column.PrimaryKey().Identity())
                .Column<string>("Name")
                // The Bio column was added later, so it's added in UpdateFrom1() for existing installations.
                // It's also added here for new installations.
                .Column<string>("Bio", column => column.Unlimited())
            );

        // UpdateFrom1() won't run for new installations, they will have the Bio column added by default.
        return 2;
    }

    public int UpdateFrom1()
    {
        // Adding the Bio column for old installations.
        SchemaBuilder.AlterTable(typeof(PersonRecord).Name,
            table =>
                table.AddColumn<string>("Bio", column => column.Unlimited())
            );

        return 2;
    }
}
```

Never do any non-trivial work (i.e. pretty much anything apart from variable assignments) in the constructors of injectable types. The dependency injection framework can instantiate your type any time, as the tree of dependencies can result in hundreds of instantiations happening when a type is resolved. Thus any work done in a constructor can possibly have a negative performance effect in seemingly unrelated cases.

If you want to produce a value for a field that won't change during the lifetime of the object then do this by lazily producing that value when its first accessed (e.g. with `Lazy<T>`) .

When you want to store the ID of a content item always use the `ContentItem.Id` property, never the Id of a content part (if you have a reference to a part you can access the content item ID simply through `part.ContentItem.Id`). This is because a content part can have a different ID (e.g. due to versioning) than the content item it is attached to.

When you want to access a form field from JavaScript that was built with a statically typed Html helper for a view model property (like with `Html.HiddenFor()`) then never hard-code the field element's ID into your script: such generated IDs can change with the underlying implementation and by changing the editor prefix. Instead, populate such IDs from your templates, e.g. by passing the output of `Html.FieldIdFor()` to the script.

When creating a new controller action don't forget to set the page title somewhere, best from the main view template of the action. I.e.:

```
<h1>@Html.TitleForPage(T("My Page"))</h1>
```

Or if you just want to set the content of the `<title>` tag directly (like it is necessary on admin pages, where the title is already displayed):

```
@{
    Layout.Title = T("My Page");
}
```

Note that generally it's bad practice to set the title from content part shape templates: those are meant to be a fragment of the layout so they shouldn't set the title directly; the title is to be set by a higher level component that actually knows what the whole page is about.

About displaying validation info in templates:

If you want to display the validation errors corresponding to a specific field, which is generally a good practice, then you can display it like this:

```
@Html.ValidationMessageFor(m => m.MyField)
```

Most of the time it's good practice to also, or instead display a validation summary on the top of the page, but close to the form:

```
@Html.ValidationSummary()
```

Never display a validation summary from a content part editor for the same reason as not to set the page title (see above).

When creating ad-hoc shapes then (unless the shapes are very generic) prefix the shapes' names with the module's name (e.g. `My_Company_My_Module_My_Shape`). Shape names are global identifiers, so if they're only interesting for your module you have to use an appropriate name.

Remember authorization! When letting the user fetch content items by ID or otherwise in any way remember that a malicious user might try to trick your code into fetching content not intended to be shown. As a rule of thumb you should always authorize the user's access (through the `IAuthorizer` service when in a controller if you also want to display authorization messages; otherwise through `IAuthorizationService`) to a content item object.

Never check the "Own" content permissions (like `DeleteOwnContent`) directly, just the generic ones (e.g. `DeleteContent`) as the former ones are handled internally by the latter ones.

When you have no choice but catching the base `Exception` then use exception fatality check.

Checklist to go through when finishing a new module or theme:

- Is the manifest properly filled? No "Description for the module" and similar defaults remain?
- Are there no empty default folders, e.g. a Styles folder with just one Web.config?

JavaScript best practices

Prefix jQuery objects with the dollar sign ($) so they can be distinguished from other objects.

```
var $header = $("#header");
```

Instead of using the $ variable directly use a wrapper to inject the jQuery object and only use the dollar sign in the local scope.

```
// The dollar sign will be used only inside the anonymous function here.
(function ($) {
    // Notice the shorthand document.ready function. Always wrap your jQuery DOM-manipulation code in the document.ready!
    $(function() {
        alert("Document ready!");
    });
})(jQuery);
```

Try to avoid adding variables to the global scope. A handy way of exposing globals is to namespace them under jQuery as demonstrated with the following example:

```
(function ($) {
    $.extend(true, {
        myModule: {
            // Such deep nesting is not always necessary, the method could be on this level directly
            myClass: { // More of a "class" than a real class of course
                myMethod: function () {
                    alert("myMethod called!");
                }
            }
        }
    });

    // You can use the above like this:
    $.myModule.myClass.myMethod();
})(jQuery);
```

When you want to access resources under a given URL of the current web application (like fetching data from a web API endpoint) never hard-code the URL into yours scripts. URLs can change and may depend on the environment (a trivial example being the usage of ApplicationPath that e.g. could prefix URL's during local development but can be empty in the production environment).

Instead inject such information into your scripts from templates.

CSS best practices

Use a language that eases CSS development and compiles into CSS like LESS or SASS. It's really worth trying! (And there's good tooling support.)

When something is possible to style in a straightforward way without the usage of images by only using CSS (even e.g. by using font icons), then do it from CSS.

Try to avoid HTML markup that serves just to enable some kind of styling.

For HTML classes and IDs use dashed names e.g. `this-is-a-class`. In Orchard modules you may prefix these with the module name.

Source control best practices

The following advices apply to the Mercurial source control system and assume the usage of the TortoiseHg client. See the Mercurial Kick Start for a more holistic tutorial. Also there's a nice tutorial regarding TortoiseHg and Codeplex.

Committing

- Try to only include changes corresponding to a single task in a commit.
- Push your commits often if you're working in a team.
- Use descriptive commit messages. If a commit corresponds to an issue tracker ticket start the message with the ticket number.
- Don't store assets (i.e. files generated from the source) in source control if possible, e.g. the `bin` and `obj` folder should be excluded.
- When you rename a file tell Mercurial that you've renamed the file (you can use, there's not a removed and an added one. This makes possible to maintain file history. You can also let TortoiseHg automatically detect renames.
- If you use an issue tracker then have issues open for every non-trivial task. Then, having issues created, prefix commit messages with the issue ID, e.g. "#11: Fixing title" or somehow reference issues. If you don't use an issue tracker or commit something not related to an issue but your code base is large and consists of multiple distinct sections then prefix your messages with the section name, e.g. "Media Management: Fixed image upload". You can even use the two technique at the same time.
- Discrete changes should go into discrete changesets. Try to avoid having multiple logical changes in the same commit as this will make it difficult to back out of certain changes if necessary.
- Commit often if you have finished something. If you have something working, or a section done, commit it. It will make much easier to track changes instead of having one big changeset. If you don't want to clutter everybody else's code base with half-ready changes then commit them to a feature branch. If your code is not affecting anybody else's work (e.g. it's not something that every other developer also uses or that will be included in the next release, or in case of Orchard it's a module that's not yet enabled anywhere, or you work in your own feature branch) then the only criterion a commit must meet is that it should compile properly. This is to encourage frequent commits. If your code change affects somebody else's work too then of course you should take more care and test it.
- If your repository uses subrepos: when you did some change to a subrepo then always commit the subrepo change into the main repo too in the end. The point is that everybody pulling the main repo will also get the current version of all subrepos too. This doesn't mean that you have to make a main repo commit for each subrepo commits (commits in the subrepo can very well be more frequent), but once you're done with a batch of work, commit that to the main repo too. Because of the same reason never push to the subrepo alone but rather push the main repo (what, the subrepo changes being committed to it, will also push the subrepo).

Branching

- If you use branches for developing features prefix the branch names with something and use a pattern like "feature/[branch name]" to make those branches distinguishable from other branches.
- Try to avoid merging branches with themselves. If you committed to a branch locally but meanwhile somebody else did the same first, after pulling do the following: instead of merging the two changesets rebase your changeset on top of the remote head. (See the Rebase documentation and the TortoiseHg Workbench documentation.) **Be careful when using subrepos!** If you already committed a subrepo to its top repo then you can't rebase the subrepo as it would cause incosistency in the top repo. In such cases simply do a merge in the subrepo if there is a more recent head changeset.
- When doing work in a temporal branch and you want to merge the changes back to another branch and close the branch do it in the following order: close then merge and NOT merge then close as this will result in an unnecessary dangling head. See this SO post for more details. Always close a branch if you don't want to use it any more.
- When doing work in a branch separated from the main line of development merge frequently from the main branch. Frequent, small merges are easier and less error-prone than big merges and also you'll be able to integrate your changes with the work done by others.

- See the excellent "A Guide to Branching in Mercurial".

Code styling

C# styling

If there length of the parameter list for a method is too long to read conveniently in terms of line length (due to the 3-argument rule this should rarely happen for methods but constructors with dependency injection) break it into multiple lines parameter by parameter.

```
public class MyClass
{
    public MyClass(
        IDependency1 dependency1,
        IDependency2 dependency2,
        IDependency3 dependency3)
    {
        // ...
    }
}
```

Prefix private variables with an underscore (_).

Keep logical blocks of codes separated by multiple line breaks, forming logical “islands”. This makes the code more readable.

```
// Notice the double line breaks between fields/properties and the constructor as well as between the constructor, public and private methods.
// Properties are separated by a blank line from fields.
public class MyClass
{
    private string _myField = "field";

    public int MyProperty { get; set; }


    public MyClass()
    {
    }


    public void MyMethod1()
    {
    }

    public void MyMethod2()
    {
    }


    private void MyPrivateMethod()
    {
    }
}
```

If you have multiple types (e.g. an interface and a class) defined in the same file, similarly divide them with two line breaks.

Have a standard ordering of members depending on their visibility and whether they're instance- or class-level, etc.

```
//  Notice the order: static, private, protected, public, const, constructor, public, protected, private, static, inner classes
public class MyClass
{
    // Static fields first
    private static string _myStaticField = "field";

    // Private fields
    private string _myField = "field";

    protected string _myProtected = "field";

    // Properties next
    public int MyProperty { get; set; }

    // Constants just before the constructor
    public const string MyConst = "const";


    // Then the constructor(s)
    public MyClass()
    {
    }


    // Public methods
    public void MyMethod()
    {
    }


    // Protected methods
    protected void MyProtected()
    {
    }


    // Private methods
    private void MyPrivateMethod()
    {
    }


    // Static methods
    private static void MyStaticMethod()
    {
    }


    // Inner classes
    public class MyInnerClass
    {
    }
}
```

If an expression is short, omit line breaks when applicable to keep the code compact (as long as readability is not hurt), e.g.:

```
public class MyClass
{
    private int _myField;
    public int MyProperty { get { return _myField; } }
}
```

CSS styling

Structure your stylesheet's content logically under titles. Use the following comment formats for different levels of titles:

```
/* First-level title
***************************************************************/

// Second-level title
// -------------------------
```

```
/* Third-level title */
```

Use line breaks to space out blocks of code.

Naming conventions

- Use the suffix “Base” for abstract classes.
- Suffix part classes with "Part", records with "Record" and records of parts with "PartRecord".
- Name js and css files with the segments of their names delimited by dashes, e.g. my-style.css. It's also advised to prefix names of such resources with the module name, since these names are global, e.g. my-module-my-style.css.
- Resource names declared in resource manifests are also global so prefix them with the module's name e.g. "MyModule.MyStyle".

Rules of thumb for refactoring

Consider refactoring in these cases:

- When a class’s net length is above 300 lines
- If the number of injected dependencies (services) exceeds 5
- If the number of arguments for a method exceeds 3
- It adds invaluable safety if you have unit tests for the code being refactored. If you don't have unit tests for a piece of code, before heavy refactoring is probably the good time to create them.
- Try not to over-engineer things. A typical and simple to detect sign of an over-complicated system is if you have classes that are almost exclusively proxying calls to other classes.

Renaming a project

You should do the following steps to rename an existing .NET project (including an Orchard module or theme).

1. Make a backup or commit to source control before attempting the rename.
2. Rename the project from inside Visual Studio. This will change the project's name in a lot of manifest files.
3. Search and replace the project's name in all files of the project or even of the solution (if you project's name is not a unique text be careful). This will rename all namespaces too.
4. Search and replace the project's name in the project file (.csproj file). This will rename the project's default root namespace and its resulting assembly's name.
5. Rename the project's folder (if it has one) to match the project's names. You'll have to re-add the project file under its new location to the solution as well as to other projects' references (if any).

Inline documentation guidelines

- Don’t overdo documentation as it can do more harm than use when going out of date, what it tends to do.
- Always document complex pieces of logic by briefly explaining what the code does and why.
- Always document unusual solutions, hacks or workarounds and explain why they are necessary.
- It’s advised to document interfaces, best with usage samples. This is especially true for services: always document services, as these are commonly used by other developers too.
- Never use comments for mental notes (like "TODO"). Such notes should go into more appropriate places like an issue tracker, some common documentation or something else.
- Documentation should be as close to what it documents as possible to avoid going out of date.
- It's good to document what the aim of a type (mostly class or interface) is. This is to be able to quickly understand what a type does without having to understand its code.
- Write documentation as you write code: use correct grammar and punctuation (remember that comments are sentences), apply to style conventions.

- Constants referenced in C# XML comments like `true` should be wrapped into a code block, i.e.:

  /// <returns>Returns <c>true</c> on success, <c>false</c> otherwise.</returns>

Code review guidelines

Doing static code reviews is a great way to improve code quality and share knowledge in a team.

- Read this study about code reviews at Cisco; lot of good tips there. Also, this article is an ideal and simple checklist that can be used for code review.
- It's best to use a tool for code review where you can give comment for specific lines in the code. Bitbucket and GitHub both offer code line commenting for commits and pull requests. On Bitbucket you can "approve" commits and you can use this feauture during code reviews: when you reviewed a commit and found everything all right, then just approve it; otherwise, make comments on the code lines or the whole commit. Then later when all of your comments were addressed, you can approve that commit too. This way approval marks always show which commits shouldn't be dealt with any more.
- When doing code review, not just look at the code lines but also try to understand the whole component you review pieces of. This way you can also give you opinion about higher-level architectural decisions.
- Pay attention to pinpoint all kinds of issues: e.g. architectural, logical, styling, maintainability issues all count.
- Don't just look at what's there but also think about what's missing (e.g. validation, error handling, access control...).
- If you see repeated issues with the code (e.g. the same mistake repeated all over the code base) then don't add the same comment multiple times. Either reference the other locations in a single comment (e.g. "This is also repeated in the other controllers here.") or if you're reviewing a bigger piece of code then collect such issues in a shared document (or a comment for the whole codebase if you tool allows it) and just reference them (e.g. "See CodeReview.docx #3").
- During the core review you can find issues of high importance, such as critical security problems. Apart from adding these to the affected piece of code also separately collect them to make them apparent.
- Some tips on using Bitbucket for code reviews:
  - When reviewing individual commits that are not part of a pull request then mark every reviewed commit as Approved. Alternatively if you found issues with it comment on it. So a commit should be either Approved or have commits.
  - Once every comment of yours was addressed you can mark a commit as Approved too; this way you'll see which commit needs your attention.
  - When reviewing a pull request you can review the whole delta instead of individual commits. Most of the time it's better to review the whole delta.
  - When a pull request is done, set it as Approved and also merge it into the upstream branch. You can do this from Bitbucket too; then make sure to make Bitbucket also close the branch with the merge.

Development environment advices

Some advices on how to set up your development environment for Orchard Development.

Software to install

Below you can find pieces of software that you should install for the best Orchard developer experience.

- Visual Studio 2017, 2019 or later with the following plug-ins:
  - Web Essentials for better client-side development tooling (specifically Web Essentials 2017 and Web Essentials 2019).
  - Web Compiler to be able to easily compile client-side assets (e.g. LESS to CSS).
  - TestDriven.Net or NUnit Test Adapter for running unit tests (beware with NUnit Test Adapter that you have to build the solution while Test Explorer is open to get tests discovered; then use "Group By -> Project" to see better).
  - Productivity Power Tools 2017/2019 mostly for the feature of being able to clean-up unneeded using statements and for re-opening files just closed.
  - Attach To All The Things for quickly attaching the debugger to an IIS (Express) and other processes or ReAttach to quickly re-attach the debugger to previous debug targets. Debug Attach Manager helps with the same thing: If you select to attach a process for a given app then it'll remember it and select the suitable process the next time automatically, even after a VS restart (especially handy for .NET Core apps with their *dotnet.exe* processes).
  - SQL Server Compact & SQLite Toolbox for browsing an SQL CE database.
  - Code Maid for various goodies, including progress indicator for builds.
  - Lombiq Orchard Visual Studio Extension with various Orchard-related features.
  - Visual Studio Spell Checker (VS2017 and Later) for avoiding typos in comments and other texts.
  - Markdown Editor to be able to edit Markdown files with a preview window (which can also be used to easily browse Markdown-formatted documentation files locally).
  - File Nesting is very handy when adding files to a project with dependencies between them (e.g. scss files with corresponding css, min.css and css.map files).
  - Visual Studio IntelliCode to give you some personalized IntelliSense suggestions as well as generate an .editorconfig file based on your project's/solution's coding style.
- Web Platform Installer for installing any necessary local developer tool or SDK. Install WebMatrix for simply browsing local SQL CE databases if you don't use the SQL Server Compact & SQLite Toolbox VS extension linked above.
- Local IIS and SQL Server (as well as SQL Server Management Studio) set up as per the following article: "How-to: running Orchard locally through IIS using SQL Server"
- An up-to-date browser with developer-aiding tools:
  - Chrome with the JavaScript Errors Notifier extension to get notified of client-side errors easily.
  - Firefox with the Web Developer and HttpRequester (or REST Easy) extensions.
- Fiddler for inspecting any HTTP traffic.

Make sure to always run Visual Studio as an administrator!

Visual Studio tips

- Use code snippets; try out the ones in this package too that specifically aid Orchard development.
- Use keyboard shortcuts. The most useful ones are:
  - F5: start with debugging
  - Ctrl+F5: start without debugging
  - F6: build
  - F10: run to the next line, when debugging
  - F11: step into the code on the current line (e.g. if the current line is a method call this will forward you to the body of the method), when debugging

    - Ctrl+F10: run to the cursor, when debugging
    - Ctrl+.: opens the Smart Tag
    - Ctrl+Shift+Enter: adds a new line below the current line
    - F9: places a breakpoint on the current line
    - Ctrl+K, Ctrl+D: format document
    - Ctrl+K, Ctrl+C: comment selection
    - Ctrl+K, Ctrl+U: uncomment selection
    - Ctrl+,: opens the Navigate To windows (i.e. search for types or members)
    - Ctrl+D, Ctrl+E or Ctrl+Alt+E: opens the Exceptions configuration page. Tick Common Language Runtime Exceptions - Thrown to see all exceptions, even if they're caught.
    - Ctrl+Q to access the Quick Launch bar
    - Shift+Del: delete line
    - You may want to set up Ctrl+W for closing the current file for the File.Close command and Ctrl+Shift+T for Edit.UndoClose (only available if PowerCommands is installed).
- You may want to always run VS as an administrator. This will simplify debugging web apps running in IIS since you can only attach a debugger to the IIS worker process if VS is run as an administrator.

Mercurial and TortoiseHg tips

- Interact with Mercurial through the TortoiseHg Workbench. You can add a cloned repository to the Workbench by opening it from the repository folder: right click on the folder and select the Workbench.
- Use groups in the Repository Registry to group your repositories.
- See the mercurial.ini in file in this package that you can use to initialize your Mercurial instance quickly with some useful settings. You can open your mercurial.ini file quickly from inside TortoiseHg by going to File/Settings/Edit File.
- .hgignore files (drop into the repositories' folders and rename to .hgignore):
    - For Orchard modules and themes
    - For a complete Orchard solution

Orchard performance optimization guidelines

Orchard performance checklist

When optimizing an Orchard site's performance (or just putting it into production) check these points for the most obvious ways for a boost.

- Debug is set to false in Web.config
- Compiled in Release mode and deployed preferably with the Precompiled build target
- Make sure you read Optimizing IIS Performance
- Output caching (starting with the built-in OutputCache module), 2nd level caching (e.g. Syscache) is used when possible
- Combinator is installed for bundling and minifying static resources to enhance client-side performance
- Cookie-less domain or CDN is used for static resources (including Media files)
- It’s possible to opt out of sessions completely in Orchard (if you don’t use it anywhere else) which is told to give massive performance improvements. Session state can be switched off for whole modules with the “SessionState: disabled” option in the Module.txt. Beware though that TempData, widely used in Orchard (for notifications and also elsewhere like in Comments) depends on sessions. You could try non-session TempData providers to overcome this (e.g. with a cookie-based one).
- When doing I/O tasks, use async APIs. Not blocking threads to wait for an I/O task to complete increases the throughput of the server.
- If your application is under heavy load memory usage will inherently increase. If you have more than 4GB of memory and you're on a 64b machine don't be afraid to use a 64b application pool: this will also increase memory usage but the site will be able to use more than what is available for it from the 32b address space. You can set up an application pool to run a 64b worker process simply by setting "Enable 32-Bit Applications" to False (under advanced settings of the AppPool).
- Disable IIS Logging or too chatty Failed Request Tracing.
- Set DB IsolationLevel.ReadUncommitted if you know what are you doing because it can give a huge performance boost.
- In menus use Custom Links to display links to single content items instead of Content Menu Items. Custom Links store the URL while Content Menu Items need to do multiple database queries to fetch the URL, thus they're significantly slower (and the convencience of usage doesn't justify this most of the times).

Detecting performance bottlenecks

Mini Profiler is an easy to use Orchard module for pinpointing (mostly DB-related) bottlenecks quickly, even on a production machine.

Updating your Orchard instance by copying the latest source

This list serves as a guideline how to update your Orchard source if you maintain a copy of the full source (see a description for this).

1. Clone or pull the latest source from the main Orchard repository and checkout to the changeset you want to update your instance to.
2. Archive a snapshot of the repo at the specific changeset.
3. Remove the lib folder in your own solution folder. Outdated libs can cause nasty errors.
4. Copy the source over to your own solution folder, without the .gitignore file (unless you use it in your own repo of course).
5. These only apply if you're using a solution file other than the default Orchard.sln (if you're using Orchard.sln just merge that file:
    - Merge Orchard.proj (it references the solution file).
    - Add any new modules to your solution and remove deleted ones (including the removal of the modules' folders).
6. Merge Orchard.Web.csproj. You may have some custom files included there.
7. Merge Orchard.Web/Web.config if you have modified anything in the original Web.config (better to use custom config files for different build targets) or the targeted Web.configs (Release and Debug).
8. Rebuild the solution to check for any build errors.
9. Run the site to test if everything is working as intended.

Creating a new repository of the full Orchard source for your project

If from the possible ways of source controlling an Orchard solution you've chosen to keep the full Orchard source in your project's repository then these are the steps to follow when creating the solution:

1. Copy over the full Orchard source into your repository.
2. Copy Orchard.sln and rename the copy to the name of your application. Having a copy of the solution file will make upgrades more complicated but it will help to distinguish between the different Orchard solutions you may work on.
3. Rename any references to "Orchard.sln" to your own solution file in the Orchard.proj file in the root.
4. If you're using IIS Express to run the app then it's best to change the default Project Url: right click Orchard.Web/Properties/Web. Using a different app path instead of the default "OrchardLocal" is enough to differentiate between different solutions.
5. Depending on your preference for text file line endings (CRLF - Windows style or LF - Linux style) and the source control system to use (Git or Mercurial; if you use something else then no need to do anything) need you to remove some config files: if you want to store files with the LF line ending in your repo then nothing to do. Otherwise if you use Mercurial remove the .hgeol file, if you use Git remove the .gitattributes file.

For doing upgrades see this other article.

Orchard FAQ

How to configure how many characters are displayed for the Body summary?

You'll need to override the Parts.Body.Summary shape template in your theme or in a module. See Bertrand Le Roy's tutorial on the topic.

Where to place modules and themes?

- Physically modules are located under src/Orchard.Web/Modules and Themes under src/Orchard.Web/Themes. You should create a corresponding folder for modules and themes (this is only necessary if you install them by copying their source from somewhere, like cloning their repository; it's not necessary if you install them from the Gallery) and the folder should be named the same as the module/theme's **ID**. This ID name of the module/theme's project file (e.g. "Orchard.Alias"). If the theme doesn't have a project file then its folder can be named anything: now the name of the folder becomes the ID.
- In the solution, although not mandatory, it's advised that you place your modules in the "Modules" solution folder (or in your own, custom module folder) and Themes in the "Themes" folder.

How to run and debug Orchard locally?

The easiest way to run Orchard is through Visual Studio's built-in Cassini devserver. Just open the Orchard solution and hit Ctrl+F5 (starting without debugging: much faster than starting with debugging with F5).

After the site is started you can attach the debugger to the devserver: Debug/Attach to Process/select WebDev. It's useful to enable breaking when an exception is thrown, even if it's swallowed somewhere: Debug/Exceptions/tick Thrown at CLR Exceptions.

See more tips on setting up your dev environment under the Development Guidelines.

Where are the log files?

If you experience issues in a production environment the best way to start investigating the issue is by taking a look at the log files. These are under App*Data/Logs in the deployed site's folder or under src/Orchard.Web/App*Data/Logs if you're running from the full source code locally.

How can I test SSL locally?

If you're running your Orchard instance through the auto-configured IIS Express (i.e. by hitting Ctrl + F5) you can access the site through SSL by changing the protocol to https and using the port 44300. This is useful if you want to test e.g. if you've configured the Secure Sockets Layer module correctly.

Orchard training guidelines

The following guidelines serve as a base for Orchard trainings and you're welcome to hold your own Orchard training using these guidelines.

Training methodologies

For methodologies for various forms of Orchard training see training methodologies.

Prerequisites for the participants

Enumeration of technologies and paradigms used in Orchard: C#, MVC (ASP.NET MVC), C# LINQ, C# lambda expressions, dependency injection, inversion of control container, loose coupling, object-relational mapping (NHibernate), composition over inheritance, single responsibility principle, separation of concerns, .NET dynamic, Razor syntax.

- Mandatory: basic usage of Visual Studio, basic knowledge of C#, basic knowledge of client-side web development (HTML, CSS, JS)
- Strongly advised: basic understanding of ASP.NET MVC and LINQ
- Advised: understanding inversion of control containers and dependency injection, usage of Razor

Technical requirements

The following tools are needed for an Orchard training:

- Lab computers or participants' computers:
  - Software listed under "Software to install" in "Development environment advices".
  - Administrative account to install other components if necessary and to avoid permission issues when running Orchard
- Trainer's PC having all of the above and connected to a projector for demonstrations
- A whiteboard or something similar

Topics

The topics are each divided into individual modules. These modules can, but don't necessarily have dependency on each other.

- Core concepts and basic usage
- Theme development
- Module development and Orchard APIs
- Extended APIs
- Web API
- Deployment and optimization
- Team training (for development teams)

Training methodologies

Following are methodologies for various forms of Orchard training.

- Remember that it's a good thing to have regular breaks about every 45 minutes.
- Keep the time between explaining something new and demonstrating it short; i.e. if you explain something, show it.
- After a session (lesson, tutorial video...) do a quick recap of what was covered: click through what was explained and briefly mention again if there is something to emphasize.

University course

Class work and examination

Lesson structure

Every lesson begins with a short warm-up task incorporating the topics of the previous lesson.

Lessons generally have repeated cycles of the following form:

1. Presentation: the course leader explains the current topic (10-15 minutes), if the content is practical (like doing some dashboard work) students follow individually
2. Group work
   - After hearing the presentation about how to solve a certain problem students are encouraged to try out the new techniques for given tasks in form of a group work. The task either
     - consists of the topic demonstrated before
     - or is something slightly new that can be derived from the demonstration or learned by reading a short documentation. The former one is a good choice if there is enough time, the latter one is efficient if there's only limited time available.
   - Groups of 2-3-4 try to solve a problem while the course leader is helping their work and is available for questions
3. Evaluation of the group work: discussing common questions and issues

Examination

The course has no special examination, instead students should create and finish and Orchard-based web application project.

- Students form teams of arbitrary size (it's also possible to work alone)
- Every team's task is to create a real-world Orchard-based web application
- There are no limitations or obligations to that, although the groups are encouraged to develop one custom module that uses existing Orchard features in a creative way but doesn't replicate any built-in functionality
- One student gives a presentation of the project after completing it with a maximal length of 5 minutes. The presentation should be a live demonstration of the web application. After the presentations the course leader has very brief code review sections with all of the other students were the students present a section of their software, demonstrating their understanding.

Schedule

Additional is the time needed for student presentations (depends on the number of attendees) since the course's final lesson consists of student presentations and code reviews.

Intensive course

Since intensive courses should be tailored to the participants' needs the following points are just outlines and tips. Time constraint is also a factor that determines how in-depth the training can be, how many demonstrations can be carried out and how big is the part of the API that's only shown.

- Live demonstration of the usage of Orchard APIs
- Live demonstration of some inner workings with the debugger (e.g. demonstrating how the tree of shapes is built up)
- Showing aspects of the Orchard API without running them just so participants can get to know what piece of API to look for when they want to achieve something (e.g. IStorageProvider is a good candidate: it's easy to use but one needs to know about it).
- Few hour-long hackathon with a certain aim (e.g. to develop a module that's needed by the participants)
- Code review: participants write some code on their own, then the trainer checks them and comments on them line by line with code review tools. Common mistakes can be discussed with the whole group.

Core concepts and basic usage (training topic)

- Introduction
- Basic site management
- Intermediate content management
- Advanced content management
- Expert content management
- Customization features
- Basic maintenance

Introduction

- Orchard ecosystem as under the Orchard link collection
- Showcasing complex/notable Orchard apps:
  - The websites on Show Orchard
  - DotNest for creating hosted Orchard sites
  - Special apps: Associativy, Orchard Application Host
- Architectural overview
  - ASP.NET -> ASP.NET MVC -> Orchard
  - Contains open-source projects (eg. NHibernate, jQuery)
  - Modularized architecture (modules, themes)
- How to get my own Orchard instance?
  - Official Orchard releases
  - Azure
  - DotNest (also from Azure)
- Installing Orchard (running the setup): choosing a database engine (and its implications), recipes
- Basic site settings
- Demo: installing and basic settings

Time requirement: 1h 0m

Dependencies: none

Parent topic: Core concepts and basic usage

Basic site management

- Content model and content management:
  - Content types, parts and fields
  - Creating and editing content items and content types
  - Versioning: drafts (and Draftable metadata) and published versions
  - Demo: basic content types (Blog, BlogPost and Tags, Page with Layouts)
  - Demo: creating a content type
  - Demo: editing and versioning a content item
  - Exercise: creating a content type with fields and parts and creating items based on instructions
- Comments: listing comments and Comments settings
- Themes and modules:
  - Enabling/disabling features and themes
  - Installing new modules and themes
  - Dependencies
- User management and roles, permissions
  - Users: allowing new registrations, user list, settings
  - Roles: editing existing roles, creating new roles, permissions (content type and content item)
  - Demo: users and roles admin pages
  - Exercise: creating a role with selected permissions; enabling registration, registering a new user and assigning it to the newly created role

Time requirement: 2h 15m

Dependencies: none

Parent topic: Core concepts and basic usage

Intermediate content management

- Taxonomies
- Widgets
  - Layers and layer rules, zones (theme anatomy)
  - When to use Pages (Layouts) and when widgets
  - Demo: creating a new layer and an HTML Widget
  - Exercise: creating NotTheHomepage layer and a HTML Widget in the AsideSecond zone in that layer
- Navigation
  - Creating and editing a menu
    - Adding and editing custom links and content item links
    - Editing link hierarchy
  - Navigation Widget and breadcrumbs
  - Exercise: creating a menu with all kinds of items
- Search and indexing
  - Enabling Search engine and an Indexing service; what is an indexing service (eg. Lucene)
  - Enabling a content type for indexing (content type editor)
  - Selecting content types to be indexed and rebuilding index
  - Adding a Search widget to the layout
- Media management
  - Managing and editing media with Media Library and Image Editor
  - Media Processing and overview of Media Profiles

Time requirement: 2h 0m

Dependencies: Basic site management

Parent topic: Core concepts and basic usage

Advanced content management

- Projections
  - Queries: example for filtering, ordering and layouts
  - Projection Widget, Projection Page
  - Query Link (Navigation)
- Dynamic Forms, Tokens and Workflows
  - An overview of Workflows features
  - Overview of tokens
  - Using antispam features
  - Demo: creating a workflow that displays a greeting for users upon logon, displaying their name with tokens
  - Demo: creating a contact form using tokens, handling form posts with a workflow
- Importing and exporting
  - Using the Import/Export module
  - The importance of having an identity-providing part (AutoroutePart, IdentityPart)

Time requirement: 2h 30m

Dependencies: Intermediate content management

Parent topic: Core concepts and basic usage

Expert content management

- Content item access control:
  - Content item permissions
  - Content type-level permissons with Securable metadata
- Other content type metadata: Listable, Creatable, Stereotypes
- In-depth content customization with Layouts
  - Master Layouts
  - Building a grid
- Auditing content with Audit Trail

Time requirement: 1h 45m

Dependencies: Advanced content management

Parent topic: Core concepts and basic usage

Customization features

- Templates and the Shape Layout Element
- Multi-tenancy
- Command line
  - The "help" command
  - Running multiple commands in a batch (from file, with @filename)
  - Selecting the tenant to run the command on with the `/tenant:` switch.
- Writing recipes:
  - Installing and enabling themes/features
  - Command line commands
  - Adding exported content types and content items
- Optional: localization
  - Installing a .po package with Vandelay.TranslationManager
  - Setting up locales and creating localized content items
- Optional:
  - Overriding the current theme's behaviour with the Theme Override module
  - Using a custom theme on DotNest: Media Theme

Time requirement: 2h 0m

Dependencies: Intermediate content management

Parent topic: Core concepts and basic usage

Basic maintenance

- Troubleshooting:
  - Enabling the display of all thrown exception
  - Understanding log files (see sample) and viewing them with the Error Log Viewer module
  - Contacting module authors and filing reproducible, meaningful bug reports
- Useful 3rd-party modules
  - Orchard Forums by Nicholas Mayne
  - SEO by Onestop/Lombiq
  - Error Log Viewer
  - Module Profiles
  - External Pages
  - Shoutbox

Time requirement: 0h 30m

Dependencies: Basic site management

Parent topic: Core concepts and basic usage

Theme development (training topic)

- Getting started with theme development
- Advanced theme development

Getting started with theme development

- Structure (i.e. "Anatomy of a theme"):
  - Explaining Theme.txt: BaseTheme and Zones
  - Theme.png
- Command line scaffolding
- Shapes:
  - Notion of shapes, tree of shapes
  - The notion and use of the Layout shape (and the Document shape), checking it out with the debugger
  - Using Shape Tracing
  - Shape templates and important view variables (e.g. WorkContext, Layout, Model), T-strings
  - Alternates and overrides (templates, stylesheets, scripts); Url Alternates and Widget Alternates
  - Writing editor and display shape templates, explaining Model object usage
  - Ad-hoc shapes
- Static resources: styles/scripts (how to include/require them) and resource manifests

Time requirement: 2h 0m

Dependencies: none

Parent topic: Theme development

Advanced theme development

- Placement.info
  - Placement: shape name, zone, weight
  - Matching (DisplayType, ContentType, Path)
  - Placement editor on Admin UI (only for reordering)
- Approaches to building a new theme:
  - Top-down: how to convert an existing site build to an Orchard theme
  - Bottom-up: building on top of an existing theme (e.g. TheThemeMachine, Pretty Good Base Theme), using it as the base theme, how to create a new theme from a given design suite
- View engines (Razor, ASPX, PHP)

Time requirement: 1h 0m

Dependencies: Getting started in theme development

Parent topic: Theme development

Module development and Orchard APIs (training topic)

The below topics are the core of what an Orchard developer should know. Other, less important items are listed under Extended APIs.

- Getting started with module development
- Basic techniques in module development
- Developing custom content
- Intermediate techniques in module development
- Advanced techniques in module development
- Complementary topics of module development

Addendum

Most of this comes from Sipke Schoorstra's Orchard Harvest session content (APIs, content part development).

We've created a demo module for the purpose of teaching all the topics here with well explained examples. See the Orchard Training Demo module.

The Orchard Cheat Sheet by Sébastien Ros is a great small API reference.

Getting started in module development

- Orchard structural overview:
    - Contents of the Orchard folder (libraries, source, App_Data, module/theme folders...)
    - Solution overview
- Module structure:
    - Module.txt: features and dependencies
    - Scripts, Styles, Views folder
    - Other folders
- Command line scaffolding
- Integrating with the current theme
- Dependency injection and basic services
    - Notifier
    - Localizer
    - Logger
    - Work Context
    - Creating a dependency: difference between `IDependency`, `ITransientDependency` and `ISingletonDependency`
    - Ways of injection:
        - Single dependency
        - `IEnumerable<TDependency>`
        - Lazy injection: `Work<T>` and `Lazy<T>`

Time requirement: 1h 30m

Dependencies: none

Parent topic: Module development and APIs

Basic techniques in module development

- Data storage:
    - Records and repositories, record migrations
    - Content manager:
        - Content querying, updating and removal
        - Joins and `QueryHints` for optimization
    - Abstracted file storage with `IStorageProvider`
- Exception handling:
    - OrchardException
    - `IsFatal()`
- Ad-hoc shape creation
- `OrchardFeature` attribute

Time requirement: 3h 0m

Dependencies: Getting started in module development

Parent topic: Module development and APIs

Developing custom content

- Content part development
    - Parts and part records, versioning, `LazyField`
    - Content type migrations
    - Drivers: display and edit methods, export/import
    - Editor and display shapes
    - Handlers and filters
    - Placement
- Content field development
- Exercise
    - Module scaffolding
    - SpaceShip content type
    - Title part for name
    - `AutoroutePart` for url
    - `SpaceShipPart` content part
        - Captain
        - Class
        - Number of crew
    - `MediaPickerField` (needs the Fields feature) for an image: for this also a shape template override (hint: Shape Tracing) with a stylesheet include (e.g. the image should be displayed with rounded corners)

Time requirement: 3h 0m

Dependencies: Basic techniques in module development

Parent topic: Module development and APIs

Intermediate techniques in module development

- Custom routes
- Navigation providers: implementing an admin menu (with corresponding admin controller)
- Resource manifest
- Filters: result and action filters, `FilterProvider`
- Event bus and event handlers; creating a "filter" extension point with prioritized injected dependencies (like `IHtmlFilter`)
- Permissions and authorization
- Background tasks

Time requirement: 3h 0m

Dependencies: Basic techniques in module development

Parent topic: Module development and APIs

Advanced techniques in module development

- Ways of storing settings
    - Site settings
    - Content type settings (e.g. Indexing)
    - Content part settings: part-level and type-level settings
- Caching:
    - `ICacheManager` for instance-level caching
    - `ICacheService` for farm-wide caching
- Recipes inside modules

Time requirement: 1h 30m

Dependencies: Intermediate techniques in module development

Parent topic: Module development and APIs

Complementary topics in module development

- Optional: writing unit tests
    - Mocking and using stubs for services (including existing stubs for Orchard services)
    - Database-enabled tests
    - Running unit tests with TestDriven.Net
- Releasing to the Gallery

Time requirement: 1h 0m

Dependencies: Basic techniques in module development

Parent topic: Module development and APIs

Extended APIs (training topic)

The knowledge of the below APIs is not necessary for all developers as these are needed only for more special tasks.

- Extended APIs I.
- Extended APIs II.
- Extended APIs III.

There is no need to go into details about all providers, evaluating a few provider types used in Projector and a Token provider is enough.

Extended APIs I.

- Common Orchard provider model
- Form provider
- Projector providers:
    - Filter provider
    - Sort criteria provider
    - Layout provider

Time requirement: 1h 30m

Dependencies: Intermediate techniques in module development

Parent topic: Extended APIs

Extended APIs II.

- Token providers
- Workflows (`IActivity`) providers:
    - Events
    - Tasks
- Creating a custom Layouts (Dynamic Forms) Element
- Overriding Orchard services with `OrchardSuppressDependency`

Time requirement: 3h

Dependencies: Intermediate techniques in module development

Parent topic: Extended APIs

Extended APIs III.

- Creating an admin theme
- Advanced shapes:
    - Shape events
    - Shape table providers
    - Shape methods

Time requirement: 1h 30m

Dependencies: Intermediate techniques in module development

Parent topic: Extended APIs

Web API (training topic)

- Creating simple API controllers
- Defining custom API routes

Time requirement: 1h 0m

Dependencies: Intermediate techniques in module development

Deployment and optimization (training topic)

- Optimization
- Manual deployment
- Automated deployment

Optimization

- Installing performance-tuning modules:
    - Enterprise-level caching with Orchard.Caching
    - Output caching with the Output Cache module
    - Resource bundling and minification with Combinator
    - Second level NHibernate caching with SysCache
- Profiling with Orchard MVC Mini Profiler

Time requirement: 0h 45m

Dependencies: Basic techniques in module development

Parent topic: Deployment and optimization

Manual deployment

- Choosing the right build target
- Web.config transformation and settings
- Deploying to Azure App Services
    - Creating a web app in Azure
    - Deploying from Visual Studio using a publish profile
- Deploying to an Azure Virtual Machine
    - Creating an VM
    - Setting up IIS on the VM
    - Deploying Orchard from Visual Studio

Time requirement: 1h 45m

Dependencies: Getting started in module development

Parent topic: Deployment and optimization

Automated deployment

- Installing TeamCity
- Setting up a build configuration in TC

    - Installing necessary tools (e.g. Mercurial)
    - Adding a Version Control System root
    - Adding build steps to the process
        - Pull the source code and build the solution
        - Deploy to IIS
    - Dependencies, triggers, environment variables

Time requirement: 2h 0m

Dependencies: Manual deployment

Parent topic: Deployment and optimization

Team training (training topic)

- How to write code?
    - Agree on and follow common guidelines and conventions
    - Importance of documentation
- How to share code and other application data?
    - Version control best practices:
        - Branching strategy
        - 3rd party modules and themes in subrepositories
        - Main repo: cloned Orchard repo or copied source (see "Ways of source controlling an Orchard solution")
        - Proper exclusion filters
    - Share dev database snapshot or dev recipe
    - Share differential recipes for code changes
- How to communicate?
    - Issue tracking best practices
    - Knowledge management best practices
- Optionally: continuous integration practices and the Hosting Suite

Time requirement: 1h 30m

Dependencies: none

Development utilities

Contained here are various utilities aiding daily development.

- Visual Studio Code Snippets
- Orchard Test Contents

Visual Studio code snippets

To effectively use this collection of VS snippets just point the Snippets Manager to where you cloned or downloaded this folder. To do this go under Tools/Code Snippets Manager/select the C# language/Add and Add the whole folder.

Note that since C# snippets (unlike VB ones) don't support adding namespaces no matter how we wanted this otherwise you'll need to always add namespaces yourself.

Snippets follow Orchard naming conventions.

We've taken care to place the end terminating symbol to a place where one most possibly wants to follow up with coding. Thus if you escape snippets by hitting enter the cursor will be placed where you most likely want to write next.

These snippets are constantly used at Lombiq and updated according to our experiences.

General snippets

- controller: empty ASP.NET MVC controller class
- ctorinject: constructor with an injected dependency and a corresponding private readonly field
- privr: private readonly field
- propv: virtual auto-implemented property

Orchard snippets

Orchard snippets are prefixed with an "o" for distinction.

- oadminmenu: admin menu (INavigationProvider) skeleton
- oalterpartdefinition: ContentDefinitionManager.AlterPartDefinition() skeleton
- oaltertypedefinition: ContentDefinitionManager.AlterTypeDefinition() skeleton
- obackgroundtask: background task (IBackgroundTask) skeleton
- ocontrollerfull: Controller class with common Orchard services (IOrchardServices, Localizer, ILogger)
- ocreatecontentpartrecordtable: a SchemaBuilder.CreateTable() shortcut for part records, to be used in migrations
- ocreatetable: a simple SchemaBuilder.CreateTable() skeleton for migrations
- odriver: empty ContentPartDriver skeleton
- odriverfull: a full driver, complete with Display, Editor and Exporting/Importing methods
- ofeature: OrchardFeature attribute
- ofielddriver: empty ContentFieldDriver skeleton
- ohandler: empty ContentHandler skeleton
- ohandlerstorage: ContentHandler with StorageFilter
- ojqueryblock: usable in Razor templates, this snippet adds a Scipt.Foot/Head() javascript block, with a jQuery anonymous function inside it
- olazyfield: LazyField skeleton
- olazyfieldloader: skeleton for having setting the loader of a LazyField in a handler
- omigrations: basic migration class
- omigrationsusings: common namespace using declarations for migrations; wouldn't be needed if C# snippets could import namespaces
- opart: content part without a record
- opartandrecord: content part with a corresponding part record
- opermissions: permission provider skeleton
- oproppart: property for a content part for proxying a property from the underlying record
- opropstoreretrieve: property for a content part without a record for storing and retrieving data from the InfosetPart
- oproprecordstoreretrieve: property for a content part with a corresponding record for storing and retrieving data from the InfosetPart while also storing it in the record simultaneously
- oresourcemanifest: resource manifest skeleton
- oroutes: route provider skeleton
- oroutesfull: route provider pre-filled with a route descriptor
- oscheduledtask: scheduled task (IScheduledTaskHandler) skeleton
- oscheduledtaskrenewing: renewing scheduled task (IScheduledTaskHandler) skeleton (requires the module Piedone.HelpfulLibraries)
- otests: unit test class skeleton
- otestsusings: common namespace using declarations for unit tests

Surround with snippets: unfortunately it's a bit more difficult for "surround with" snippets; if we'd just include them among the other snippets it would cause the Snippet Picker to open when you try to surround some piece of code, requiring to click through the hierarchy to select a snippet. This is not very convenient so such snippets are in a separate folder, in SurroundWithVisualStudioSnippets. You have to install these snippets manually by importing them into the Visual C# category through the Snippet Manager (this way you can avoid the Snippet Picker).

- otryfatal: surrounds the block with a try-catch where the catch includes an exception fatality check.

Orchard test contents

The Orchard export file contain some contents that you can use to test your theme and site layout with. You can download the export file here.

These export files include a varied main menu and a foot menu, items of the most used content types and widgets and typography tests. You can log in with the credentials admin/password.

Orchard Wiki

This Wiki aims to complete the documentation by organizing Orchard-related technical terms into articles. It aims to mainly provide short explanations and links for further exploring each topic.

Please note that there are corresponding Visual Studio code snippets and detailed development guides (Orchard Training Demo module) in the Orchard Dojo Library related to these topics.

Admin menu

To integrate your module with Admin UI, you need to add navigation items to it to make your features easily accessible to users. For this purpose you have to create a class that implements the `INavigationProvider` interface. If you only have one class of such functionality, the convention is that the file and the class are named AdminMenu and the file is placed in the project root. If you have multiple of these classes, you may want to keep you project root clean and place these in a solution folder called AdminMenu.

The structure of an Admin menu can be the following:

- you can add any number of items to the Admin menu
- these items can have any number of childrens
- these children items can also have children items displayed as tabs (LocalNav)

For more customization, you can define whether an item should point to the same action as its first child item with the LinkToFirstChild function.

For more information please see the Visual Studio code snippet related to AdminMenu and a definite guide on how to create one in the Orchard Training Demo module, which also describes how to add icons to your navigation items.

Background task

The `IBackgroundTask` interface is one of the most simple interfaces you can find in Orchard (aside from the empty marker ones). The only method contained in this interface you need to implement in your class is the Sweep method (without input parameters). Orchard will execute the Sweep method of every IBackgroundTask implementation every 1 minute, so it enables you to simply define some logic that will run periodically. It is useful for recurring tasks and running huge tasks in smaller batches (just like the Orchard.Indexing module does with updating the indices).

For more information please see the Visual Studio code snippet related to background tasks and a definite guide on how to create one in the Orchard Training Demo module.

Content item

Content items are instances of content types, just like objects are instances of classes. Content items are the heart and soul of Orchard: they store all the data you need to handle and display.

Content items are always versioned every time you modify any data on them and are never hard-deleted to keep the integrity of your database. From code they can be interacted with through the Content Manager service.

Content field

Content fields are bits of information that can store basic data, like the common .NET classes, e.g. strings, integers and dates.

Content fields can be attached to a content part in any quantity you like (either by code - in a migration or on the Admin UI), just make sure every field has a unique name. That means that unlike with parts, you can have a specific type of field attached multiple times.

In case you are attaching a content field to a content type on the Admin UI, the fields are attached to a ghost-content part with the same as the content type (if the content type doesn't already have a content part with the same name).

Differences between content parts and content fields:

- Parts can (but not necessarily should) correspond to database tables, i.e. a part can store its data in a table.
- A field's data is stored along the content item's `ContentItemVersionRecord` in a serialized XML format (and is thus performing worse because of serialization but also better because of not having to join in or lazily load other tables; concrete performance difference depends on the usage).
- Because of the storage difference parts' properties can be directly queried and used for filtering or ordering while fields can't be queried. The Projector module overcomes this by creating indices for fields.
- While a part can be attached to a content type once, a field can be attached multiple times.
- Parts are attached to a content type while technically fields are attached to a part (when you attach fields to a content type from the admin UI in reality an invisible part, having the same name as the type, will be created).

Content Manager service

The Content Manager is the important service in Orchard that you can use to interact with content items from code.

When developing a module you can use the Content Manager to create, update, retrieve and remove (soft delete) content items, fetch different versions of the same content item as well as import/export items, fetch items' metadata or build display and editor shapes. You can find a detailed example of how to use the Content Manager in the Training Demo module.

Content part

A content part is a set of separate functionalities and data that can be applied to a content type by attaching the content part to a content type. Content parts may not store any corresponding data (in this case, they only add functionality by using existing data) in the database or they can even load data from an external data source (like a webservice). Content fields can be attached to content parts (for the differences between parts and fields, see the fields article).

If your content part stores data in the database, usually it's mapped to a record class (the corresponding content part record), which is an actual representation of the data you are storing in the database. In most cases, content parts derive from `ContentPart<TRecord>` (where the type parameter is your content part record class), but if the part isn't storing anything in a corresponding record then it can just derive from `ContentPart`.

Content type

A content type is a blueprint of how content items of that type look like: it defines the set of content parts that make up the content types.

A content type only consists of content parts, even if it seems that you can directly attach content fields to a type from the admin UI. When you attach fields from the admin UI in reality an invisible part is created that has the same name as the content type (e.g. a Page part is created for the Page content type) and the fields get attached to that.

Examples of some basic content types: Page or Blog Post (both containing e.g. Title Part and Body Part).

Core

We mean several things as "core" so initially this may be confusing:

- Core modules: these are the modules that reside in Orchard's Core project (Orchard.Core). They are always installed and always enabled, so your modules can safely depend on them. Examples include Content (basic content management features like admin content lists) and Navigation.
- The Orchard Framework: Orchard is a powerful web development framework. This framework is contained in the Orchard.Framework class library. Being in the center of Orchard it's also often called "core".
- Non-core modules that are developed by The Orchard Team and maintained in the Orchard repository are also often dubbed "core modules" but technically they aren't. These modules are sometimes also called the

"built-in modules". Furthermore there are such modules that have their category (in their module manifest) declared as "Core": such modules behave exactly like true Core modules (like they are always enabled) but they are not part of the Orchard.Core project.

Content part driver

Drivers (more precisely: content part drivers) are pieces of code used in conjunction with content parts. They are responsible for building the editor and display as well as handling importing and exporting of a content part.

Note that the same part can have multiple drivers.

Orchard-style event handlers (aka the Orchard Event Bus)

Event handlers in Orchard work just like in any other programming environment with a very interesting addition: they enable you to create extensions points to your features without worrying about coupling and references. So let's see how they work!

Every Orchard-y event handler has an interface that derives from `IEventHandler`. In your module you can inject a single `IYourEventHandler` and now you just created the extension point for your module: if you call the methods on the injected event handler you enable other module authors to interact with your module if they implement this interface. That's okay so far, but in order to make it work, other module authors (or even you, if you want to extend the funtionalities of a feature not written by you) must have a reference on the module's project that hosts this event handler interface.

And here comes the twist: instead all you need to do to implement an event handler that is in another module not referenced by your module is to have an interface in your module with the same name also deriving from `IEventHandler` and implement that interface! Aside from that, of course, the methods you implement must also have the same name and signature (if there are types that also come from an other module or project you are not referencing, you can replace them with `dynamic`s). Orchard will find these even more loosely coupled implementations solely based on their names and try to match the methods with the original interface. How cool is that?

Further reading:

- Notes about the Orchard Event Bus in the official documentation
- "About the Orchard Event Model?" MSDN Forums topic
- "The Event Bus Pattern and IEventhandler" blogpost explaining the concept

Extension

Modules and themes are collectively called as extensions. The expression is used in the Orchard documentation and also in the source code extensively when a service has to do with modules and themes as well.

Content handler

Content handlers, or more specifically content part handlers are similiar to event handlers in programming in general: these methods' job is to handle specific events of a content part, e.g. when it's instantiated, deleted or versioned.

You can find a complete list of such events on the "Understanding content handlers" page of the official documentation.

The InfosetPart content part

`InfosetPart` is one of Orchard's built-in content parts. It's always automatically attached to every content item of every content type. It represents (and provides access to) the so called infoset of the content item.

The infoset is a simple XML document that is stored along the content item in its `ContentItemRecord` or if versioned, in the `ContentItemVersionRecord`. It can store arbitrary data and is commonly used to save content fields' data or data for content parts that needn't be queried. Since the `ContentItemRecord` and/or `ContentItemVersionRecord` is always loaded for a content item the infoset is also loaded at all times. Thus anything stored in the infoset can be retrieved quickly, without any subsequent database calls.

Importance for content parts

If a content part needs to store data in the database one of the solutions would be to use a content part record. However such records, if not specifically set for eager-loading, are lazily loaded one by one when using the content item. To overcome this performance issue `InfosetPart` can be used to store the part's data, eliminating the need for further database queries.

However data stored in the `InfosetPart` can't be simply queried (i.e. filtered or ordered) using the database engine. By storing data both in the infoset and in the record, however, one can have the best of both worlds: querying is possible using the records but for any other database interaction the records are not loaded.

Orchard contains helper methods to ease the usage of the infoset.

Bertrand Le Roy has written an extensive blogpost about the way and implications of using the `InfosetPart`.

Migrations

A migration is special class usually derived from DataMigrationImpl through which you can tell Orchard what kind content parts and content types you wish to create and store in the database, including their fields and settings. In most cases you want to place a file called Migrations to your project root, but if you have more than one of them (e.g. because you have multiple features with separate migrations) you may want to place them in a subfolder called Migrations to keep your project root clean.

Module

Orchard modules are types of extensions. They are designed to extend Orchard's functionality in any way you can imagine.

Modules can have multiple features: features can be independently switched on or off (you have to decorate classes corresponding to a specific feature with the `OrchardFeature` attribute). Actually what you can enable or disable from the admin UI are features, not modules. However, each module has at least one feature, what has the same ID as the module itself (i.e. the .csproj file's name).

Features can depend on each other (not just on features of the same module but also on features of other modules); this dependency is declared in the Module.txt file (more info about it in the Docs), the manifest of the module. You can only enable a feature if all of its dependencies are installed (and if they are not already enabled, they will be when you enable the feature depending on them).

Orchard's Processing Engine

The Processing Engine (what you can use through the injectable dependency `IProcessingEngine`) in Orchard is a service that you can use to run arbitrary code in the context of an HTTP request, but after the request is processed.

Tasks queued in the Processing Engine are synchronously executed after the ambient transaction of the request ends, from `DefaultOrchardHost.EndRequest()`. This means that such tasks can be potentially longer-running as they don't endanger causing a timeout for the request transaction, since they are each run in their own transactions. Processing Engine tasks are still part of the HTTP request though, that means that the execution time of these tasks add to the user-perceivable response time of the application and they can also cause a request timeout.

Note that since Processing Engine tasks are run after a request naturally they need a request to get executed, on an idle site such tasks won't be processed. Also such tasks are only retained for the scope of the request: if something fatal happens and the request completely fails before tasks can be executed than those tasks will be lost; and tasks queued from background tasks won't be processed at all (since there is no corresponding request).

An example of how the Processing Engine is used is in the built-in Orchard.Comments module in `CommentService`: calculating the number of comments under a given content item is being done from a Processing Engine task.

Record

Records are simple classes that represent a piece of data that is stored in the database as a row in a table. The only special thing about record classes is that their public properties (that correspond to columns in the said table) should be public.

The tables corresponding to records are created and also modified if necessary by migrations.

Orchard uses the NHibernate ORM library as the database abstraction layer.

Resource manifest

A resource manifest is a class implementing the `IResourceManifestProvider` interface though which you can declare your static resources (e.g. stylesheets and scripts) towards Orchard and give them a unique name to be able to easily use them in your templates. Choosing a unique name for each of resources is quite important to avoid name collisions, since static resources are rendered as shapes and thus can be overridden (see naming conventions).

Scheduled task

Scheduled tasks, which are classes that implement the quite simple `IScheduledTaskHandler` interface allow you to run some code at a specific time (Orchard will respect it with a 1-minute precision). The way of creating a scheduled task is to first create an `IScheduledTaskHandler` implementation and put your code inside the only method in this interface called Process. Using the IScheduledTaskManager service, you can create, list and delete scheduled tasks, so in order to register your scheduled task in Orchard, you need to create it first using the CreateTask method. Orchard uses a background task to check for scheduled tasks that should be executed: each scheduled task is stored in the database as a separate record which is deleted when the execution of the given task starts to make sure that they only run once. Please also note that scheduled tasks must have a unique name: according to the Orchard naming best practices, it is advised to prefix it with the name of your module.

For more information please see the Visual Studio code snippet related to scheduled tasks (a simple and a renewing one, the latter depends on the Piedone.HelpfulLibraries module) and a definite guide on how to create one in the Orchard Training Demo module.

Shape

Shapes are dynamic view models that are used to construct the data model behind the layout of a page in Orchard. The resulting structure is a tree (the tree of shapes) where each shape corresponds to a piece of markup in the end: shapes containing other shapes produce wrappers around other pieces of markup while the leaves of this tree correspond to simple templates. Each shape contains every information to render their corresponding template: they're the view models for their templates.

Some resources:

- Documentation on shapes
- Video tutorial on creating ad-hoc shapes
- Documentation on the usage of Shape Tracing, the tool for determining which shape is behind what you see
- Documentation on alternates: shapes can have have different renderings corresponding to them depending on various factors; those different renderings are called alternates
- Documentation on Placement.info explaining how shapes building up content items' displays and editors are ordered
- Explaining how shapes are produced in the background
- Using shapes as Html helpers
- Hooking into shape events

Theme

Orchard themes are types of extensions. They are designed to be able to change the look and feel of your Orchard website.

Themes contain CSS, JS files and graphics as well as shape templates. Most of the time themes don't contain any C# code apart from resource manifests.

Tokens

Tokens are pieces of codified text that are be dynamically substituted with other values. E.g. if you want to create a template for an e-mail that should be sent to users but you want to greet the users by their name you can use tokens to send a personalized text like this: "Dear {User.Name}!"

More information about tokens is in the documentation.

Versioning

Content items in Orchard are versioned by default: this means that if you edit a content item and publish the modifications you don't overwrite what was previously published but you create a new version - that will be the published version.

Versioning illustrated

Let's take a look at Page content items, because Page is a content type that's included in Orchard by default.

- Pages are "draftable". This means that you can create draft, i.e. not published (and thus not visible) versions of it. You can also set this option from the admin UI from the content type editor of Page.
- Pages - among others - include the Title and Body content part. These parts have the capability of being versioned. Not every content part is versionable, it's the developer's decision.

That said let's see what happens:

1. You create a new Page and save it, i.e. click the Save button. You **don't** click Publish now. This means the Page is saved, but only as a draft. Nobody can see it on the frontend. The Page only has a single version that is a *draft*. At the same time this version is the *latest* version.
2. You now publish the Page (e.g. click Publish now in the editor). This means the Page is now visible on the frontend. The Page only has a single version that is the *published* one, which is also the *latest*.
3. You now edit the Page and save it. This creates a new version of the Page: one is *published* (and visible) and one is a *draft* (not visible). So our Page now has two versions; the draft one is the *latest* (since it's newer than the published version).
4. You now edit the Page again and save it. Editing a draft won't create a new version: only editing a published version and saving it will create a new version. So our Page still has two versions, one being the draft with the content you just saved and the other one being the published version.
5. You now edit the page and instead of clicking Save you click Publish now. This saves the changes to the draft, then publishes it. This means our item still has two versions, both published (the result would be the same if we would just clicked Publish draft from the admin listings: it would made the draft version published).
6. If you now unpublish the item the *latest published* item will be pushed back to the draft state. This yields that the first version, the published one will be published and again you'll have the *latest* version as *draft*.

Soft deletes

Beware that when you remove content items no record is really deleted as Orchard operates with soft deletes: content items are only marked deleted but remain in the database. Actually what happens is that all version of the content item get unpublished and simultaneously loose their flag of being latest; i.e. in the end no version will be marked as published nor latest, thus the item won't be found when fetching the published version.

However, since the versions are still there, they can be retrieved through the Content Manager.

Work Context

`WorkContext` is one of the most important types in Orchard. It's more or less a generalization of the idea of an HttpContext. It contains a lot of (mostly Orchard-specific) contextual information like the basic site settings, the current theme or user and the HttpContext itself.

The work context is an important aspect of Orchard's dependency framework too. A `WorkContext` object lives as long as its work context scope lives, what is a dependency injection scope (`IDependency` implementations live as long as their work context lives): when such a scope is created through `IWorkContextAccessor` (what you can also use to access the current `WorkContext`) also the `WorkContext` is created. Correspondingly there are also methods on the `WorkContext` class to resolve dependencies (you can use this instead of constructor injection if you want to lazily resolve dependencies).

An HTTP request in Orchard, as well as background tasks are wrapped into an ambient work context. Since work contexts are not tied to an http context you can have multiple work contexts per request and you can have a work context independently of a request too (this happens in background tasks).

Thus such work contexts are externally managed contexts and because of this somehow have to "travel" along with their scope until the latter is terminated: in Orchard the work context is either carried in the `HttpContext` or in a thread static field (what also causes some limitations).

A work context scope is the lowest dependency scope commonly used. It also has a parent, the shell's scope: this is the shell context (or more precisely, its lifetime scope). You can access a shell's (what is most of the time equal to a tenant) context through `IOrchardHost.GetShellContext()`. Work context scopes are actually created from the shell context's lifetime scope. Furthermore this also has a parent that is the application-wide HostContainer.

Most of the time you don't have to manage the work context yourself since the ambient work context around requests and background tasks are managed for you.

`IWorkContextAccessor` is also passed into `RouteData.DataTokens`. This way the `WorkContext` (and thus, Orchard services) can be accessed from code that is not under dependency injection like HTML helpers and attributes. See Sipke's tutorial on taking advantage of this.

Orchard examples

This list helps to find where to look if you need an example of something in Orchard, so you can look at it when you need to roll out your own similar solution. This is a selection only, not a full list (e.g. many modules contain content parts).

Project links are only included if the module/theme is not bundled with the Orchard source (then the links point to the source file with the example, when applicable). The Training Demo module is not linked every time, it's under its own repository.

Modules

- Content item management:
  - Content field: Training Demo
  - Content manager usage from code: Orchard.Core.Contents (/Controllers), Training Demo
  - Content part: Training Demo
  - Content type migration: Orchard.Pages, Training Demo
  - Content type/content part settings: Orchard.Core.Common (/Settings)
  - Editor groups (with site settings): Combinator (metadata in handler, grouping in driver)
- Events: Training Demo
- File handling: Orchard.Media, Training Demo
- Permissions:

    - Static: Orchard.Comments, Training Demo
    - Dynamic: Orchard.Core.Contents (DynamicPermissions)
- Projector providers:
    - Filter: Orchard.Projections, Helpful Extensions
    - Layout: Orchard.Projections
    - Sort criteria: Orchard.Projections
- Records (low-level database access): Training Demo
- Resources (static resources: css, js files, resource manifests and resource inclusions): Orchard.jQuery, Training Demo
- Search services from code: Associativy Core (also see this service)
- Site settings: SH.GoogleAnalytics, Training Demo. With new menu item under Settings: Orchard.Comments, Orchard.Users.
- Token provider: Training Demo, Orchard.Tokens, Helpful Extensions
- Unit tests: Training Demo, Orchard.Tests.Modules.Tags
- Using the Content Picker popup for custom content item selection: Orchard Scripting Extensions
- Using symmetric encryption: External Pages (look for SetPasswordEncrypted() and GetDecodedPassword())
- Workflows Activity: Training Demo

Themes

- Theme with settings: Orchard Super Classic Theme

Orchard Dojo Library contribution guidelines

The Orchard Dojo Library is fully open source and can be edited by anyone. If you found an error or would like to improve it you're more than welcome; just submit a pull request!

The Library is stored as Markdown-formatted text files in a repository on Bitbucket. The files can be edited with any text editor but we recommend Haroopad.

- The same guidelines apply as to the Orchard documentation except that documents here not only can but are required to have a leading first-level title.
- Name files and folders with PascalCasing.
- Files named Index.md are automatically opened when requesting their folder.
- You can use relative links to link between files; paths are the same online as they are in the repository. Keep in mind that links to folders (when the Index file is opened automatically) must end with a slash (/) while links to files shouldn't.
- Add 3 line breaks after an H1, 2 before (if it's not immediately after an H1) and 1 after an H2.
- When adding inline code snippets use the `apostroph-delimited syntax`.
- When adding paths or filenames set them as emphasized (italic) like *C:\path\to\file.txt*.
- Keep in mind that Markdown should also be valid HTML, so encode HTML entities accordingly. E.g. use the encoded version of the < and > signs (see this source).

Orchard Dojo Library license