Yesterday someone asked on an Orchard discussion board about a concise explanation of Orchard module design structure. Documentation, which in my opinion is very good, unfortunately doesn’t provide such information **in one place** so I thought about writing one myself. A brief introduction to [ASP.NET MVC 3](http://www.asp.net/mvc/mvc3) is needed if you want to fully understand it so I recommend reading it first.

Basically the Orchard module is a more sophisticated form of ASP.NET MVC concept of **Area**, which is dynamically loaded by Orchard framework. Writing one involves creating a couple of different objects, which I’ll try to describe.

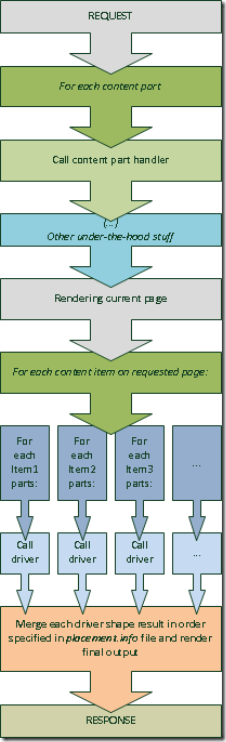
**Creating custom content part**

*This is the basic building block of content stored by Orchard. Do you remember being a kid and playing with LEGO® bricks?Uśmiech Orchard is just like that. You create content types from available content parts (eg. a body part, a route part, a some-fancy-custom-part and so on). Orchard comes with a couple of prebuilt types such as Page, Blog, BlogPost, User, even a whole Site is a content item. When you’re done creating your types, you can create instances of those types, called content items (eg. a blog post, a page, a user etc.). Simple as thatUśmiech*

Creating your own content part (eg. you want to display a Silverlight app) involves creating a few objects:

* A content part itself
* A content part record
* A handler
* A driver
* Display shapes (.cshtml Razor view files)
* Data migration

In the next part of this article I’ll briefly explain every one of those. To give you a better picture, this drawing shows (on a very abstract level) how does the Orchard rendering work:

[](http://www.szmyd.com.pl/Media/BlogPs/Windows-Live-Writer/Jumpstart-into-Orchard-module-developmen_409D/orchard-pipeline_thumb5_2.png)

**So, as I said before, we’re now going to get into the detailsUśmiech**

**A content part**

It’s and ordinary class extending either **ContentPart** or **ContentPart<T>** (where T is a corresponding record class type). You should use the former if there is nothing you want to persist in database. The latter should be used whenever you want to persist some data in the database.

**A content part record**

This object is a simple POCO entity holding your part’s data. Orchard deals with saving changes and retrieving data from database under-the-hood, so there is nothing you have to do besides declaring your record.

**A handler**

Handler is an object responsible for handling (ofc:) your content part. In other words – you define here how Orchard should deal with your part. By this I mean:

* Database persistence
* Handling content item lifetime events
* Defining which of the existing content items your part should be added to

Handlers have to extend the **ContentHandler** class.

**A driver**

Driver should be thought of as a content part’s controller (in MVC terms). It is responsible for displaying and editing your part. Drivers have to extend the **ContentPartDriver<T>** generic class, where T is your content part type. There are 3 methods you can override – one Display method and two Editor methods.

* **Display** method is called whenever your part is rendered in frontend.
* **Editor** methods – one is used for rendering edit form (GET) and second for saving the form (POST). First one get called whenever you start creating a content item (eg. a new Page) containing your part from Dashboard. The second gets called when you hit “Save” button.

Similar to controller actions – these methods have to return a shape to render your part. Shapes (.cshtml) files are returned as a dynamic object named after your .cshtml files with dots replaced with underscores and the .cshtml extension removed (eg. shape named **MyModule.MyPart.cshtml** in **/Views/Parts** folder can be accessed by **shapeHelper.Parts\_MyModule\_MyPart(…)** dynamic method – you can pass to this method any data you want to render via named parameters).

**Display shapes**

These are exactly the .cshtml Razor views – nothing more, nothing less. They are responsible only for rendering your part. By convention, shapes used to render your part to the user (and returned from Display driver method) should be stored directly in **/Views/Parts** folder. Shapes returned from Editor driver methods, used for rendering the edit form, has to be stored under **/Views/EditorTemplates/Parts** folder.

**Data migration**

This is the place, where you define the database definitions needed to persist your parts. In most cases it should be created automatically by using command line **codegen datamigration <your\_module\_name>** command. There are a few cases you may want to alter this by hand, but it is a topic for a different article.

*For the purpose of this article (and for making Orchard module devs life a bit easierPuszczam oczko) I created a Codeplex project with basic module implementation –* [*Orchard Jumpstart*](http://orchardjumpstart.codeplex.com/)*. Hope it helps you jump into Orchard module development!*

*If you would like to see a full-featured example, I’ll encourage you to take a look at a part I’ve recently written –* ***SilverlightPart****, available also on* [*Codeplex*](http://orchardsilverlight.codeplex.com/)*. Just grab the source code and play with it!Uśmiech*