1. Introduction

Moldster system is an attempt to facilitate the structuring process of UI, by decreasing the code that has to be written to create any view which increases the maintainability and readability of the code, so instead of writing ten lines to add say a text box, the label for the field the placeholder and the validation message you write one line and prepare a template for this whole thing so instead of copying and pasting the control group for each field in your system you only have to write it once and re-use it throughout your application by calling a `IHtmlHelper` extension method

When you create a certain component and that component has a script that makes it function using angular for instance, you may need to make another component just like it but missing one field or two, you shouldn't have to rewrite the whole code to create that component and make it work while the code you've written before would make it work, so you'll copy and paste but again it is bad maintainability cause if you change the behavior you would still have to go around all similar components to apply the change and if something was wrong with new code you'd have to repeat

Moldster uses the object oriented nature of typescript to make it possible for you to inherit the functionality, from a razor template you can create several components that use the same script to run while their appearance to the user being different.

Every control that you write in any view can be a customization point that can change the page behavior, after writing the control you can hide it from some components you created or disable its writing making it a display field

After your done coding `Configurator.UI` can make it possible for a non-developer user to select some of the UI functionality you created and customize every component in to fit the needs of any customer, meaning that any little application you write can actually be turned into a product that can fit different business needs of any customer

1.1. How it works

- As stated before Moldster uses html helper extension methods to make it possible to reuse any control, another advantage of having C# code inside an html template is that the Moldster system can understand the controls inside the page and then store it in the database so you would be able to hide some or make them read only
- The process of development goes as follows
    1. You create a razor template in the config.api project let's call it (`UserEdit.cshtml`)
    2. You can then create a page from that template let's call it `UserCreate`
    3. You tell Moldster to process the template you created
    4. Moldster scans the controls in that page and stores their ids under the page you created
    5. Moldster will also add a base script for that template to the UI project with the same name as the template (`UserEditBase.ts`)
    6. Now the page you created will have all the controls in the template and the typescript class for that page will inherit from a base type script you have for the template
    7. You can now select to hide some parts of the razor template using the configurator

8. The you render the page, this will extract an html template for that page after reading the configuration in the database and add it to the UI project

You can now create multiple pages from the same template they all use the same script so any changes would only have to be written once

2. Getting Started

2.1. NUGET source

The upcoming steps depends on NUGET packages that you should gave access to which you can obtain by adding our NUGET server to your Package Sources in your visual studio

Go to Tools -> NUGET Package Manager -> Package manager settings -> Packages Sources

Depending on where you are ASGA package source Url should be on

http://[DevServerIP]:8010/NUGET

2.2. Projects templates

1. Go to TFS in `$Helpers/ProjectTemplates`
2. Get latest from that folder and copy it to

   C:\Users\[username]\Documents\[vs version]\Templates\ProjectTemplates\Visual C#

3. That way when you want to create a new project you'll find `Moldster` in the template list with three project templates `Config.Api`, `Commander` and `UI`
4. First you need to create a class library for the project you are about to make, give it a one word name as an example will be using `MyProject`
5. Then you create two projects :
   a. `MyProject.Config.Api` from `Config.Api` template
   b. `MyProject.UI` from `UI` template

2.3. Initial configuration

When creating the `Config.Api` project it will come with two JSON files

A. `appsettings.json`
B. `appEnvironments.json`

2.3.1. *appsettings.json*

should look something like this

```
{
  "ConnectionStrings": {
    "Moldster": "Server=.;Database=MyProject.Config;Integrated Security=true"
  },
  "Moldster": {
    "CoreAppName": "BaseApp",
    "ConfigRoot": "{PARENT}\\MyProject.Config.Api",
    "UIRoot": "{PARENT}\\MyProject.UI",
    "LocalizationRoot": "{PARENT}\\MyProject",
    "ConfigUrl":""
  },
  "AuthenticationEncKey": "92929292929292932"
}
```

- **ConnectionStrings:Moldster :** should point to the config database for the project will get to its structure later
- **Moldster:CoreAppName :** is the folder inside the ui core folder in which the base scripts for the templates will be stored, to facilitate referencing when generating the code
- **Moldster:ConfigRoot :** should point the the config.api project path {PARENT} will be replaced with the solution folder in runtime
- **Moldster:UIRoot :** should point the UI project path {PARENT} will be replaced with the solution folder in runtime
- **Moldster:LocalizationRoot :** should point the main project path {PARENT} will be replaced with the solution folder in runtime, moldster uses resx files to store localization dictionaries for different languages, when rendering the resx files will be converted to json files readable by angular translation module, this location is where the resx files should be
- **Moldster:ConfigUrl :** deprecated but must exist
- **AuthenticationEncKey:** leave it as it is

2.3.2. *appEnvironments.json:*

Upload and database configuration will get to it later but for now you only need to fill `UserId` and `Password` for the local environment, should have permission to create database

2.4. Configurator UI

1. Get latest from $Helpers/Configurator.UI
2. Open the solution in it and publish it using “Local” publish profile
3. This should publish to

C:/Publish/Configurator.UI

4. Go to IIS Manager and add a new website “`Configurator.UI`”
5. Select the path in step three and set to port (preferably 8050)
6. Go to http://localhost:8050 on your browser and login using (admin / 963258741);
7. The select list above shows the apps the configurator can work on this comes from a JSON file in the published `Configurator.U`I folder `moldsterApps.json` for now you can use the default configuration which points to localhost:8000 which will be the URL for our `MyProject.Config.Api`

2.5. Setting up environment

1. Update NUGET packages to latest version for solution as follows
   - Right click solution
   - Manage NUGET package for solution
   - select ASGA source from sources
   - Updates tab
   - Check all and update
2. Build `MyProject.Config.Api` and run it using the run.bat file in the project
3. In Configurator Go to Development -> Builder tab
4. Click on Initialize app, Styles and fonts, TS Base scripts and Vendor Bundles
5. Go to Localization tab and click on Write Resource Files, this will add the resource files to Localization folder in the path configured as `LocalizationRoot`

2.6. Creating Configuration database

1. In the previous section the Initialize app command should have added a SQL script to your Config.Api project in

SQL/Creation.sql

2. Create a database with the same name you added to the `ConnectionStrings:Moldster` configuration in section 1.2.4
3. Run the creation script on that database to create the database

2.7. Creating a tenant

Moldster is configured for multiple tenant applications, so whatever software you build can be used as SAAS application, anyhow there has to be at least one tenant to be used for development

1. Run your `Config.Api` Project using batch
2. In configurator (http://localhost:8050) go to Tenants
3. Add Tenant, default tenant name and Code is usually ClientApp
4. The database name is the name of the database of the project, it will be created before creating the tenant
5. Later when you create other tenants you be able to duplicate the structure for that tenant and add some initialization data
6. Select environment `Local`
7. Click Save
8. Go to tenant list you should find the new tenant in the list
9. Click on details -> render
10. Go to Development -> Builder -> Bundle Config Files
11. In UI Project open the home controller set the default domain to your Tenant name
12. Return a string as your projects default title
13. Now your UI project should be ready to run

3. Integrating with an API projects

The UI project is actually an MVC application so it can be used also as an API for your application, but you can have separate projects for the API, the UI can also work with multiple API applications but for now let's see what we need to start an API project

3.1. Database design conventions

Here we propose a database first workflow, it is not mandatory to work with the frame work but we're going to explain it along with the tools and conventions we propose for better code maintainability and compatibility with build in functions in `CodeShellCore` library

3.1.1. Important columns

- It is preferred to have the primary key of all your tables an `int` or `bigint` column with the name Id as this will make to models implement the interface `IModel<T>` which can be useful
- In $/Helpers/SQL there is an SQL file that adds `AddAuditingColumns` stored to your database, whenever executed this procedure adds 4 columns (`CreatedOn, CreatedBy, UpdatedOn, UpdatedBy`) and that will also be used as an interface to automatically record the time and the logged in user that makes any change to any entity, you should run this after you finish designing your database

3.1.2. *General naming conventions*

- All table names should be plural
- All foreign key columns should follow the convention `[singlarReferencedTableName]`Id (ex: Employees table under Departments should have a DepartmentId column.. wouldn't make any technical difference for the code it is to follow a standard.

3.1.3. *Recursion*

- Recursive tables should have these two columns
  - ParentId : type should be same as your unified primary key
  - Chain : nvarchar(max)

  Also you can add a trigger for filling the chain column from

  $/Helpers/SQL/tr_RecursionTrigger.sql

  This writes the path of parent ids on every create or update in the chain column to facilitate querying by a parent
  Having these two columns will help implementing the `IRecursiveModel` interface which has built-in functionality in the library

3.1.4. *Lookups*

- Any table that will be used as lookup for select lists in the system, it would be useful if it has the column `Name` as its display property this will implement the `INamedModel<T>` interface which will also be useful

3.1.5. *Data Localization*

- If the project will have multiple language support and it is required that the user should be able to input different text for each language for each record this functionality exists using a table and a scalar function; they can be found in :

  $/Helpers/SQL/tb_Localizables.sql

  $/Helpers/SQL/fn_GetLocalized.sql

- Their usage will be explained later

3.2. Models project

- it is preferred that the model generation be done In a different solution than the current solution as it references different packages that are not required for normal development
- make sure you get latest project templates as in section 2.2
- create a new project from `Model Generator` template with the same namespace as your models project `MyProject`
- open the `Generate.bat` file

```
set connection_parameters=[your connection parameters];
..
..
set database_name=[database name]
set context_name=[desired name of the db context class]
..
```

- running the `Generate.bat` batch will generate the model from the database in Entities folder
- You should then copy the files into your models project (`MyProject`)
- By default the entity classes will inherit from an interface and base class with the same name as the project, they can be found in `BaseClasses.cs` found in the Model Generator project.
- it would be useful if you set that interface to inherit from `IModel<T>` as we explained in the Database section

3.3. Initialization

If you use the UI project as your API you don't need to do these steps.

Create a new Empty Web project in visual, currently the packages don't support .NET core 3 so you should use any version of .NET core 2

1. Remove all NUGET package reference that were added automatically to the project
2. You only need `CodeShellCore.Web`, it will come with all the needed dependencies
3. Create a class that inherits from `CodeShellCore.Web.WebShell`, you will be required to create a constructor and implement the abstract class:
   a. `defaultCulture = new CultureInfo("en")`
   b. `useLocalization = false`
4. Empty your `StartUp` class and make it inherit from `CodeShellCore.Web.ShellStartup<[the class you created in step one]>`
5. The shell class works instead of the startup class therefore you can use it to register whatever services you need by overriding the `RegisterServices` method or add your middleware by overriding `ConfigureHttp`

3.4. Resources

From API point view resources are the controllers of the API, resources are the only link between your UI application and your API application, components -as will be explained later- are mostly divided between listing components and editing component, now there's a build in functionality in the system that does the whole cycle of crud operations to any entity:

- Listing (Pagination, Search and filtering)
- Create
- Update
- Delete

To be able to do that there has to be a way to point your component to an API controller to deal with and that's where the resources entity comes in, resources are string identifiers stored in the configuration database to generate service classes in the UI that works as adapters between the UI and the API, all communication between the UI and the API should be done using these service classes

To add a new resource go to Resources in the configurator and click the add resource button, just write a unique string with the URL of the controller.

`CodeShellCore.Web` uses static pattern for routing which is "`apiAction/{controller}/{action}/{id}`" meaning that if you have a `UsersController` it will be accessible using "`apiAction/Users`" and all actions in it will be accessible using "`apiAction/Users/{action}`" without having to add any routing attributes to it.

So the resource name for `UsersController` should simply be "Users"

Leave the domain name empty for now

3.5. Controllers

- API controller classes should be able to integrate with the generated service files there, we assume here that every controller will be used to deal with a certain entity from the project's models so let's say we need to initialize complete crud operations for Users we set our `UsersController` class to inherit from `CodeShellCore.Web.Controllers.MoldsterEntityController<User,TPrime>` and user should implement `IModel<TPrime>` Where `TPrime` is the type of your primary key (`long, int, string, ..`)
- By default all controllers that inherits from `CodeShellCore.Web.Controllers.BaseApiController` are secured by a filter that filter goes through steps of authorization will be discussed later but for now to be able to access your APIs add this code to your Shell class

```
using CodeShellCore.Security;

..

..

public override void RegisterServices(IServiceCollection coll)
{
        base.RegisterServices(coll);
        coll.AddCodeShellSecurity(false);
}
```

- This will allow all requests to pass through the filter, if argument is `true` it means only logged in users will be allowed through

4. Starting development

4.1. Page Categories (Templates)

4.1.1. Razor file

Now let's create our first page category

1. Go to you Config.Api project inside /Views
2. You should add a reference to your models project in the Config.Api project, that way razor templates will be strongly typed against your models so you can feel free to change properties as you like and still be able to handle any broken references inside the views on rendering
3. In the `_ViewImports.cshtml` file add `@using` reference to your models namespaces to be visible to all views inside that folder
4. For organization purposes we usually create a folder for each resource in the views folder as a naming convention, this is not mandatory you can organize your files however you like, so let's create a folder for Users
5. Now add a new razor view to this folder with named `UserList.cshtml`
6. Assuming you have a class named `UserListDTO` we can use it as the model for the template by adding `@model UserListDTO` on top
7. It is also important to add this line to all views `@{ Layout = Html.Config().Layout; }`, this allows you to create different pages from the same template and set one to be a popup or embedded
8. Feel free to discover into the `IHtmlHelper` extension methods we will use only three for now, now paste this code inside `UserList.cshtml` file

```
<table class="table table-striped">
    <thead>
        <tr>
            @Html.HeaderCell(d => d.Name)
            @Html.HeaderCell(d => d.LogonName)
            @Html.HeaderCell(d => d.BirthDate)
            @Html.HeaderCell(d => d.Email)
            @Html.HeaderCell(d => d.Mobile)
            @Html.HeaderCell(d => d.AppName)
            @Html.HeaderCell(d => d.CreatedOn)
            <th></th>
        </tr>
    </thead>
    <tbody>
        <tr *ngFor="let model of list">
            @Html.TextCell(d => d.Name)
            @Html.TextCell(d => d.LogonName)
            @Html.TextCell(d => d.BirthDate, pipe: "date : 'dd/MM/yyyy'")
            @Html.TextCell(d => d.Email)
            @Html.TextCell(d => d.Mobile)
            @Html.TextCell(d => d.AppName)
            @Html.TextCell(d => d.CreatedOn)
            <td>
                @Html.ListModifiers()
            </td>
        </tr>
    </tbody>
</table>
```

4.1.2. Registering template

1. Make sure your Config.Api Project is running using the batch
2. Go to Configurator.UI url
3. You should first use the resources tab in the configurator and add resource called Users
4. Now go to Page Templates tab and click add new on top
5. This opens a popup with all templates in the views folder that is not registered in the database, you should find the template you created on top as It will be the latest
6. check the checkbox next to the template name, select Users for resource and List for Base Component then save
7. click refresh, you'll find users in the domain tree on the right if you click on it you should find the new template
8. now you should go to development select your tenant from above, and click process (the button with a cog on it), this should generate the base component for you and the service class you should find them in the UI project in

   Core/BaseApp/Http/UsersService.ts

   Core/BaseApp/Users/UserEditBase.ts

9. you'll find `UserEditBase.ts` to inherit from `ListComponentBase` and overriding the Service Property with the `UsersService`

10. the `UsersService` will inherit from `EntityHttpService` and override the `BaseUrl` property with the path to the Users controller
11. these two classes integrating with the `UsersController` will provide functionality to the component including pagination, search, delete and linking to create and edit components and every part of it can be customized to your needs we will discussed how later, but this is where your code will be all classes inside the `BaseApp` folder won't be affected by the generation process if already existing because they are the place where you will be writing your code

4.2. Pages

So far we haven't produced any usable functionality to our system, we just laid the base, and the page is the final product of this it is the routable or embedded component that will be used by the end user

4.2.1. Creating a page

1. Make sure your Config.Api Project is running using the batch
2. Go to Configurator.UI url
3. Go to pages and click on the add button
4. You'll be asked to fill the following fields
    a. **Tempate Path :** the razor template we registered in the previous section
    b. **Tenant Code:** because you can create a different page for each tenant
    c. **Component Domain :** this will be the folder inside which the html template and the component class will be created, you can select from existing or write it on the text box, just avoid spaces and use / as a separator not \, each folder is added to a lazily loaded angular module so you should try not to add too many pages to the same domain or that module could get too big
    d. **Component Name:** together with the domain will be the route of the page if it is routable, make sure the name is unique for the tenant, the validation for the name is only over the domain
    e. **Route Parameters :** if the page is routable it can have route parameters in the format /:(parameter name), Edit components for example should have /:id as route parameter, this will start the editing cycle in the `EditComponentBase`
    f. **Usage :** routable accessible by route, Embedded accessible by adding selector in another component or as a popup component
    g. **Apps :** the type of users that can access this component
    h. **Permission Type :**
        i. **Anonymous :** anyone can access it logged in or not
        ii. **Allow all :** accessible to all logged in users
        iii. **Resource :** linked to a permission of a specific resource
    i. **Resource Id :** valid only if routable and Permission type resource is selected this is to select which resource the page is linked to
    j. **Action Type :** if the resource is selected this is to restrict the page accessibility to a user having the permission to do that action to the specified resource, like if you have an edit user page, this should be linked to Resource : "Users" and Action Type : "Update", so a user with no Update permission cannot access this page but can access the list for

instance if linked to Action Type : “View”, this is to enable a higher level of customized permissions to your application

k. **Layout :** you can use this to create a popup component from any page by selecting the `ModalComponentLayout` from the list, all the used layouts can be found in the Configuration API in `ShellComponents/Angular/Layout`, you can add to it if you like, if left empty takes after the layout of page category, there is one for edit and one for list and more

l. **Navigation Group :** the navigation group the page will appear on (optional), navigation groups will be explained in a following section

m. **Collection :** only valid in listing pages it is the name of the collection the list will be loaded from (entity collection will be explained later

n. **Default Accessibility :** this allows you to create a read only page of any template if the default accessibility of a certain page is Read Only all the controls inside it will be read only

4.2.2. Customizing a page

1. Make sure your Config.Api Project is running using the batch
2. Go to Configurator.UI url
3. After saving the created page you should be redirected to Page customization page , you can also access this page at any time by opening Pages -> click controls icon on the page next to edit icon
4. Click on process template
5. This reads the controls on the razor template and gives you the ability to customize the controls on that specific page you just created between hiding it completely or setting it to read only

4.2.3. Navigation groups

The default navigation is on the side of the main app component, component inside UI Project

Core/BaseApp/NavigationSideBar,

by default it loads from a navigation group called ***Main***

1. Make sure your Config.Api Project is running using the batch
2. Go to Configurator.UI URL
3. Go to Navigation groups in the configurator
4. Click on the + button on the left
5. Write Main then save
6. Now click on the green button next to the newly created group
7. Select tenant
8. Click on the plus sign on top
9. Now you can select which pages to appear on this group, and order them, the select list only uses routable pages with no route parameters
10. Click save
11. Now if the user has permission to a certain page that appears in the navigation group it should appear to him on the left

4.2.4. *Render all*

1. Make sure your Config.Api Project is running using the batch
2. Go to Configurator.UI URL
3. Go to Configurator -> Development
4. Select Tenant
5. Click on the image button on top
6. Now you can run your UI application to see the final result

4.3. Publishing

4.3.1. *UI versions*

Now as we work and make our changes we want it these changes to take affect once we publish but the problem is internet browsers cache the JavaScript files if the name is unchanged and due to the fact that we use lazy loading to load the scripts it can be required of the user to do hard refresh (Ctrl + F5) on almost every screen

Therefore to force the browser to reload the scripts on every new version we include a version number every time we publish

The bundling tool gets the version number of the from the version of the UI project assembly itself so we only need to change the version of the UI project to change the version of all the published scripts

There's a tool that can allow you to update any C# project version in one click

1. First download this file

$/Helpers/Batch/ToolSet.rar

2. Extract it to any folder say `D:/Work/Batch`
3. Now right click on My PC -> Properties -> Advanced System Settings -> Environment Variables -> System variables
4. Select PATH and Edit (if it is not there create a variable with the new PATH using New...)
5. Add the folder from step 4 to your PATH environment variable
6. Inside your solution folder create a new batch file with the name versionSetter.bat
7. Past this in it

```
toolset -p MyProject 1.0.0 -d %cd%

toolset -p MyProject.UI 1.0.0 -d %cd%
```

8. Of course these project names should be replaced with your actual project names, so to change your projects versions you only need to edit this batch file and run it, of it runs successfully you should see a message like this

```
Altering version for Project MyProject                -> v1.0.0.0     SUCCESS
Altering version for Project MyProject.UI             -> v1.0.0.0     SUCCESS
```

4.3.2. Bundling Scripts

1. Make sure your Config.Api Project is running using the batch
2. Go to Configurator.UI URL
3. Go Tenants
4. Click details on the tenant you want to publish
5. If not rendered click on render
6. Select the environment you want to publish to (Local if it is the first time)
7. Click publish
8. This should bundle the TS code into JavaScript files using production AOT trans-piling with the name of the UI version, if the UI project version was not changed it will just upload the scripts without bundling, in that case you can force it to bundle by checking the + Version checkbox this will add 1 to the last portion of the version number and make it bundle again
9. If the bundling is successful it will update the tenant version in the database

4.3.3. Publishing with Web Deploy

1. On the target server create an IIS website with the name of the project (ex: MyProject.UI) and set the path to an empty folder
2. On your project right click and select

   Publish -> New Profile -> IIS, FTP, etc -> Create Profile

   a. Publish method : select Web Deploy
   b. Server : the target server IP or Host name
   c. Site name : write the name of the website you created on step 1
   d. Enter the user name and password
   e. Check Save password if you like
   f. Validate connection to make sure it's working
   g. Save
3. In the project files expand Properties -> PublishProfiles
4. Open the newly added profile
5. Add these two lines to the XML

```
<AllowUntrustedCertificate>True</AllowUntrustedCertificate>
<MSDeployUseChecksum>True</MSDeployUseChecksum>
```

6. This is to upload new files only

4.3.4. Database backup

1. Now due to this way of working the data in the configuration database we work on is actually part of the code of the application, the database is very important and if you want to return to

some older version of your app you wouldn’t be able to if the configuration database data was changed due to your work

2. As a solution to that there’s a way for you to back up the database and add it to your source control and checking it in with the rest of the code so you would be able to restore any version at any time
3. From TFS get latest from `$/Helpers/Batch`
4. You’ll find 3 batch files that backs up a database and copies the output, you’ll find instructions inside on how to fill the required parameters:
    a. `sql_backup_local.bat` : Backs up and copies on the same machine
    b. `sql_backup_net.bat` : Backs up and copies the file using network sharing
    c. `sql_backup_ftp.bat` : Backs up and copies the files using FTP
5. Create a new folder inside your solution folder and copy these files to it
6. Add a new batch file with this in it:

```
start /B /W sql_backup_local MyProject.Config ".\..\MyProject.Config.Api\SQL\Backups"
```

7. If all the parameters are correct running this batch should create a backup from the config database and place it in the Config API project
8. Make sure this backup file is added to the source control

5. TFS rules

5.1. Bugs

- Bugs have to be under a story if the bug belongs to an old story in previous iteration create a story named [(iteration name)] Fixing bus (number)
- When the bug is fixed set the status to Committed while keeping the story New
- Change the story to Committed after you publish the version so the committed bugs would be visible to the tester
- The tester should change the bug status to Done if fixed and New if not
- If all the bugs and tasks are done in the story, the tester should change its status to Done