Create Xcode Templates

Create Xcode templates to speed up our daywork and automatically create files and folders.

Davide Fin Follow
Mar 3 · 12 min read ★

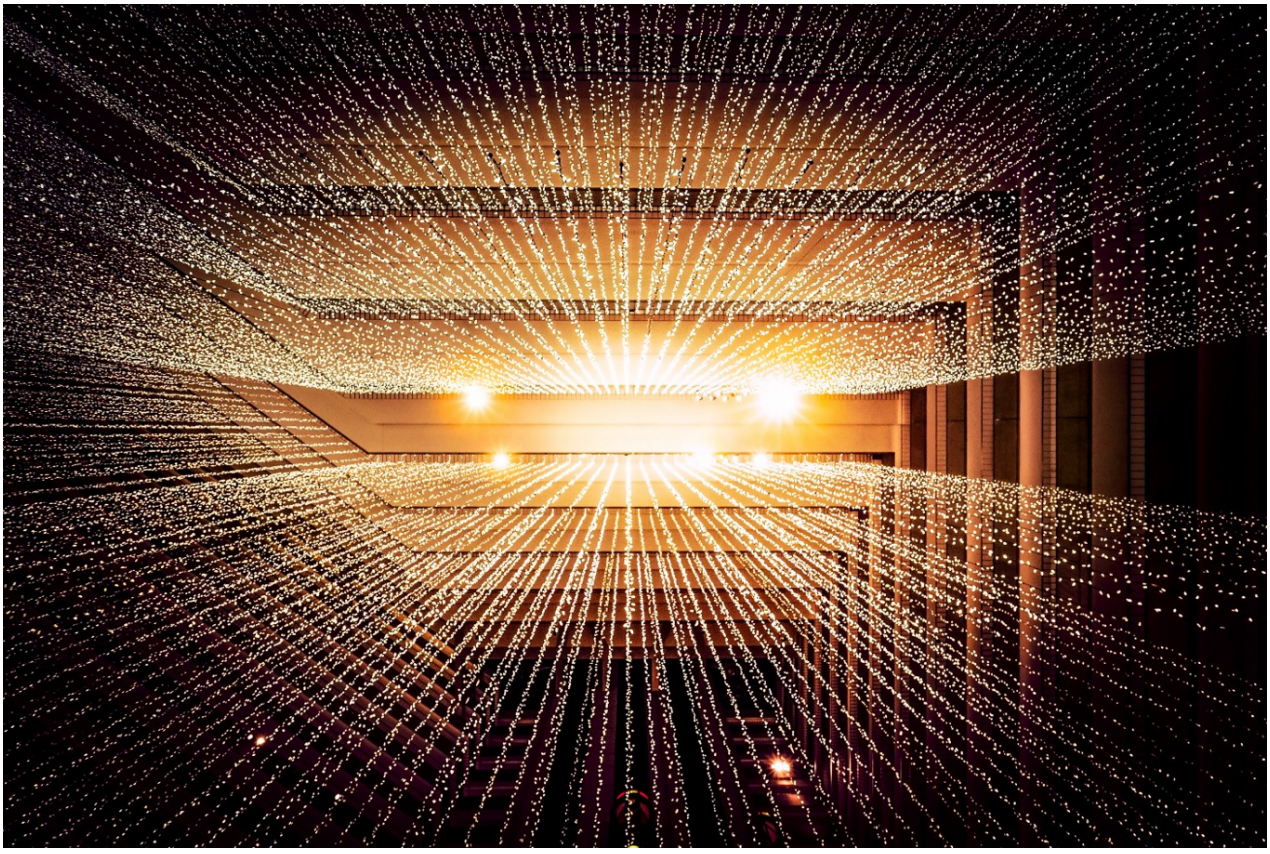


Photo by Joshua Sortino on Unsplash

With Xcode we create files and groups everyday. We commonly create files for our classes, storyboards or XIBs. We organize them into folders in order to have a logically organized project. Our preferred IDE offers a number of useful built in templates that we can use to create different types of projects or files.

When we use architectural patterns such as MVVM or VIPER, just to name a few, we find ourselves dealing with numerous files that together define each module: we create a folder for the required module, we create each files we need, probably we create subfolders and put files inside of them and we repeat this time consuming job every time we need to create new modules.

Custom templates

We can create custom templates that tailor our needs and let us speed up our work. With a custom template we can automatically create from scratch groups and files for a new module and add them to our project.

The procedure to create template for Xcode is quite easy. With a little additional work we can also implement a sort of interaction with the user to allow him to make choices.

In this tutorial I will walk you through preparing a custom template for a simple MVVM project architecture.

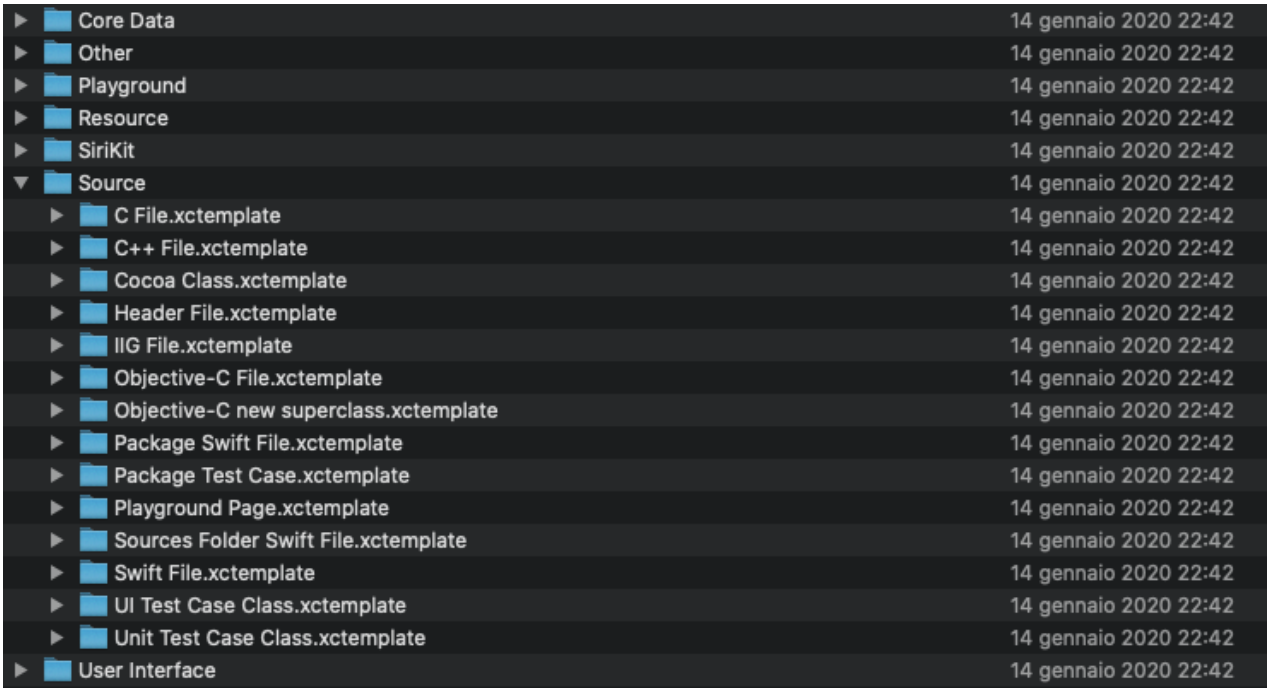
Let's start

Xcode keeps its default templates in a dedicated folder that you can find here:

/Applications/Xcode.app/Contents/Developer/Library/Xcode/Templates/File Templates

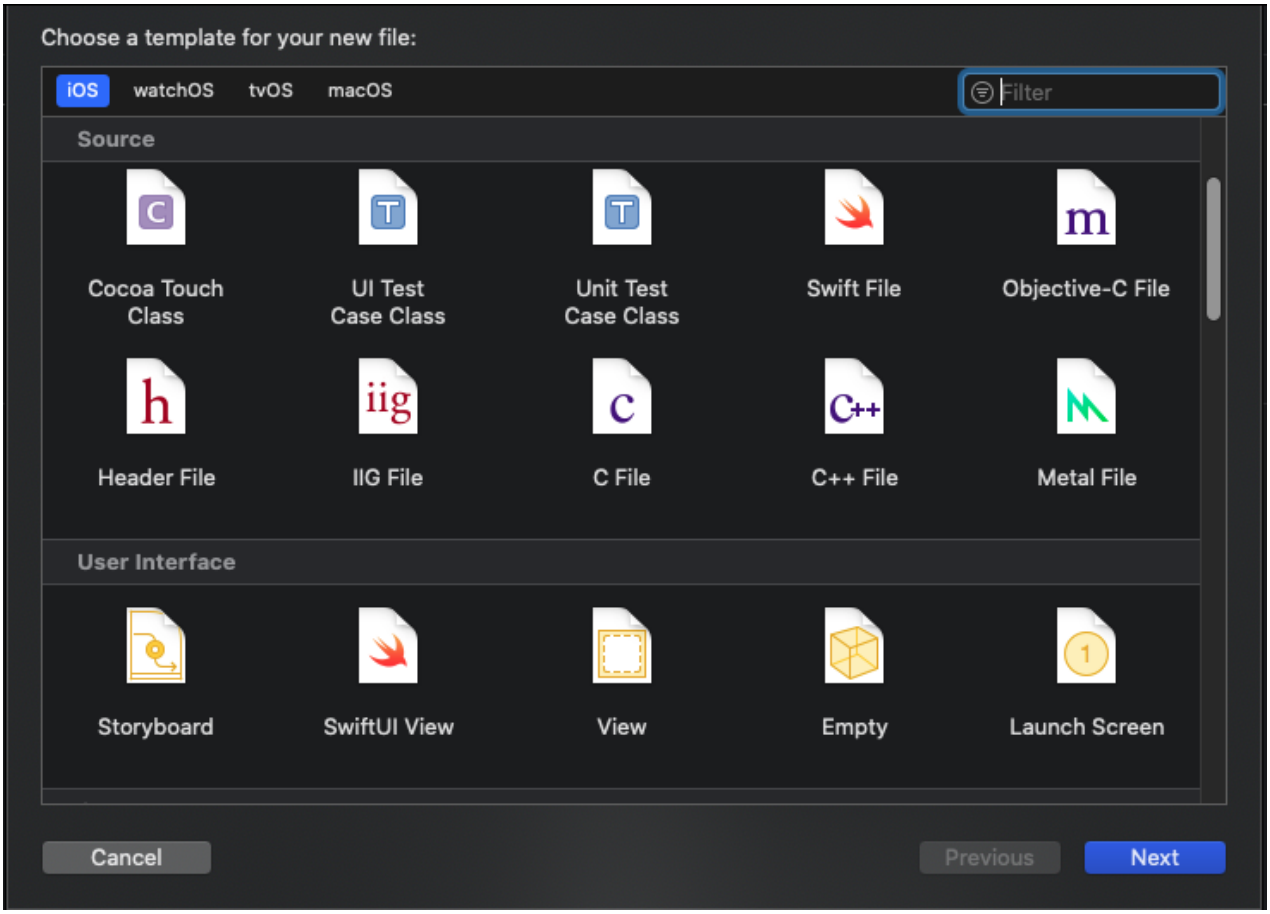
We can use **Terminal** or **Finder** to go directly to this folder. With **Finder** we can use the **Go Menu,** then **Go to folder** option and paste the path above.

Inside we find numerous subfolders and inside of them there are others that have the extension **.xctemplate** as we can see in the picture below:



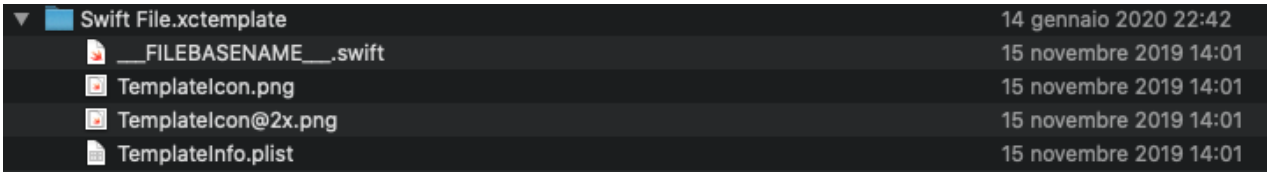
If we open an existing project with Xcode, or create a new one,

when we go to **create a new file**, this well known window is presented on the screen:



The **File Templates** folder previously opened with finder contains within it further subfolders whose names coincide with the categories shown in the Xcode window, you can find: **Source**, **User Interface**, **Core Data**, **Apple Watch** and so on.
If you open the folder **Source** with finder, you can see a certain number of files with extension **.xctemplate** inside, one for each type of file shown by Xcode in the category with the same name.

Inside **Source**, we can open the subfolder named **Swift File.xctemplate** and see a group of files:



At first look we can see a file with extension **.swift**, then we have two images and finally a **.plist** file.

You have already understood that what we are observing is the template that Xcode uses to create a simple Swift file through the wizard that provides us. If you preview one of the two images you will see that it is exactly the same icon that we have used a thousand times from the window of our IDE.

If we open the **__FILEBASENAME__.swift** file we can see this content:

```
1  //___FILEHEADER___
2
3  import Foundation
4
```

And if we effectively create a new swift file from Xcode we can see this:

```
1  //
2  //  File.swift
3  //  TestXCTemplates
4  //
5  //  Created by Davide Fin on 03/03/2020.
6  //  Copyright © 2020 Davide Fin. All rights reserved.
7  //
8
9  import Foundation
10
```

This means that, when we select a template, Xcode uses the corresponding folder, opens the **__FILEBASENAME__.swift** file that it finds inside and uses text macros to add or replace information. In this simple case it replaces the **__FILEHEADER__** text with the common text containing info about the file name, project name, author name, etc.

The file **templateInfo.plist** contains information about the template:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/D
<plist version="1.0">
<dict>
        <key>Kind</key>
        <string>Xcode.IDEFoundation.TextSubstitutionFileTemplateKind</string
        <key>Description</key>
        <string>An empty Swift file.</string>
        <key>Summary</key>
        <string>An empty Swift file</string>
        <key>SortOrder</key>
        <string>30</string>
        <key>AllowedTypes</key>
        <array>
                <string>public.swift-source</string>
        </array>
        <key>DefaultCompletionName</key>
        <string>File</string>
        <key>MainTemplateFile</key>
        <string>___FILEBASENAME___.swift</string>
</dict>
```

Original Xcode TemplateInfo.plist for Swift File Creation

It is a xml file that contains some important things:

the key **Kind** with its own particular value that is required for File Templates:
Xcode.IDEFoundation.TextSubstituionFileTempalteKind.

```
<key>Kind</key
<string>Xcode.IDEFoundation.TextSubstitutionFileTemplateKin
d</string>
```

We have some descriptive information:

```
<key>Description</key>
<string>An empty Swift file.</string>
<key>Summary</key>
<string>An empty Swift file</string>
```

The name of the swift file that will be used as template

```
<key>MainTemplateFile</key>
<string>___FILEBASENAME___.swift</string>
```

If you open different original Xcode Templates you will see that their contents may differ and contain extra information. The following list of fields enumerates the most common keys you could find, their names are quite simple to understand:

- **Kind** (type: *string*): Because we are creating a “File Template”

the value of this key is always:
`Xcode.IDEKit.TextSubstitutionFileTemplateKind`

- **Description** (type: *string*): a brief description of the template file
- **DefaultCompletionName** (type: *string*): the default name of file (without extension).
- **SortOrder** (type: *number*): used to rearrange the order in the container category.
- **Summary** (type: *string*): a brief description of the template file

You can also find these optional fields:

- **Platforms** (type: *array*): the template file will be available for all platforms by default. If we want our template only for one platform we should define something like this (`com.apple.platform.iphoneos`)
- **MainTemplateFile** (type: *string*) In case a template has more than one file, MainTemplateFile specifies which file should be opened first after template's files has been created.
- **DefaultCompletionName** (type: *string*)
- **Options** (type: *array*) determine the options what you can configure in the first step of creating a new File. We will discuss this field later.

Build a new custom template

We can build our custom template in an easy way. I would like to create my own template container and make sure that it is presented on the screen together with the others already present such as **Core Data**, **Source**, **User Interface** etc. Also, I would like to see inside of it all the templates that I am going to create.

So, I can opt to create a container folder where Xcode stores all the others containers:

/Applications/Xcode.app/Contents/Developer/Library/Xcode/Templates/File Templates

and then create inside of it a subfolder with the **.xctemplate** extension that will host all the required files and and / or subfolders.

But, this isn't the smartest place to do this job. If we had to update Xcode, the File Templates folder would be deleted and replaced with the consequence that we would lose all our templates.

Instead, we can create our specific folder here:

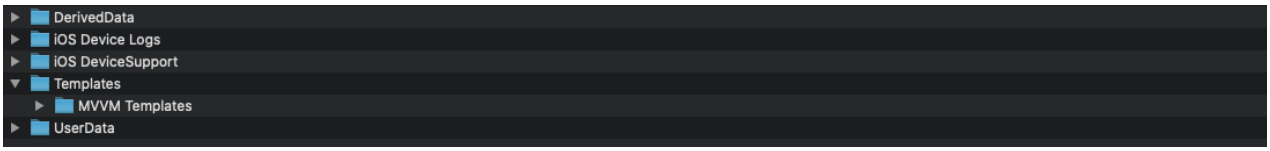
~/Library/Developer/Xcode/Templates/

Again, you can use **Finder** to reach the folder above but you will see that it does not exists. You can reach **~/Library/Developer/Xcode** but the Folder **Templates** does not

exists.

We must create it using **Finder** or **Terminal**. Once created we can add our specific container that I will name: **MVVM Templates**.

So, at the end of the operation I will have this in my file system:



I will create a new subfolder called **MVVM.xctemplate** inside my container:

The new folder is empty, so if we use Xcode and try to create a file, our container and MVVM template are not listed.

I added to the **MVVM.xctemplate** folder a set of icons:

TemplateIcon@2x.png — 96 x 96 Icon

TemplateIcon.png — 48 x 48 icon

and a **TemplateInfo.plist** file with this content:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/D
<plist version="1.0">
<dict>
        <key>Kind</key>
        <string>Xcode.IDEKit.TextSubstitutionFileTemplateKind</string>
        <key>Platforms</key>
        <array>
                <string>com.apple.platform.iphoneos</string>
        </array>
        <key>Options</key>
        <array>
                <dict>
                        <key>Identifier</key>
                        <string>productName</string>
                        <key>Required</key>
                        <true/>
                        <key>Name</key>
                        <string>Module:</string>
                        <key>Description</key>
                        <string>The name of the MVVM Module</string>
                        <key>Type</key>
                        <string>text</string>
                        <key>Default</key>
                        <string>MVVMModule</string>
                </dict>
        </array>
```

TemplateInfo.plist for my custom MVVM template

Again, the key **Kind** with its own particular value is required for File Templates:

Xcode.IDEFoundation.TextSubstituionFileTempalteKind.

By default our template will be available for all platforms. We set the value associated to the Platform key to an array with a content:(`com.apple.platform.iphoneos`)

```
<key>Platforms</key>
<array>
   <string>com.apple.platform.iphoneos</string>
</array>
```

Finally we add a key called **Options.** This key allows us to configure a sort of “Select…” option that will appear in the file creation dialog window of Xcode once we select the template. We can set the type of this option, this means that we can choose the type of control that the user can see on the screen. We can set the type in order to display a TextBox where the user can write something or we can opt for a ComboBox where the user must select a value from a list of predefined items. We added this key because we want to ask the user a string containing the name of the module. We want to use the string to name all the files automatically created by the template. In short, we want that when the user selects our template, Xcode asks him the name of the module, for example: **Login**. Then we want that the template automatically generates a list of files with classes declared inside named: **LoginViewController**, **LoginViewModel** and eventually other files and classes with **Login** prefix in their filename.

The **Options** node:

```
<key>Options</key>
 <array>
  <dict>
   <key>Identifier</key>
   <string>productName</string>
   <key>Required</key>
   <true/>
   <key>Name</key>
   <string>Module:</string>
   <key>Description</key>
   <string>The name of the MVVM Module</string>
   <key>Type</key>
   <string>text</string>
   <key>Default</key>
   <string>MVVMModule</string>
  </dict>
 </array>
```

Possible fields inside an **Options** node are:

- **Identifier** (type: *string*): used to uniquely identify options. You can use it for creating references for using in other options or template file.
- **Default** (type: *string*): selection or text.
- **Description** (type: *string*): brief description of the option. You can see the text when the mouse hovers over it.
- **Name** (type: *string*): Text is shown on the left side of the control.
- **Required** (type: *bool*): determinate is option a required. If a Required option does not have a valid value the Next button

on dialog window will be unavailable.

- **SortOrder** (type: *number*): determine the position of option. It is used to change the order in which the options are displayed, otherwise, by default, they are displayed in the order they are entered in the `Options` array.
- **Type** (type: *string*): of option. Eg `checkbox`, `text`, `static`, `combo box`, `popup`.
- **NotPersisted** (type: *bool*): determinate if the value will be saved for the next time.
- **RequiredOptions** (type: *dictionary*): can be used to enable the current option, only if certain values of another option are selected. For example, only enabling a checkbox for some values of another popup option. The key of the dictionary must be the identifier of the other option, and the value of the dictionary must be the array of subset of values from the `popup` or `combo` option that allow the current option to be enabled.
- **Values** (type: array): using for defining option of combo box.

In our Options node we added:

```
<key>Identifier</key>
<string>productName</string>
```

This **Identifier** key is a very important item that you must add

with the value '**productName**'. With this key Xcode knows that it is the base name we want to use for the files (and to populate the `FILEBASENAME` and `FILEBASENAMEASIDENTIFIER` macro).

Obviously the option that we use to prompt the user for the module name is a required one: we want do disable the next button if the user does not type a string:

```
<key>Required</key>
<true/>
```

We added a field that is shown on the left side of the control:

```
<key>Name</key>
<string>Module:</string>
```

Then we added a descriptive text:

```
<key>Description</key>
<string>The name of the MVVM Module</string>
```

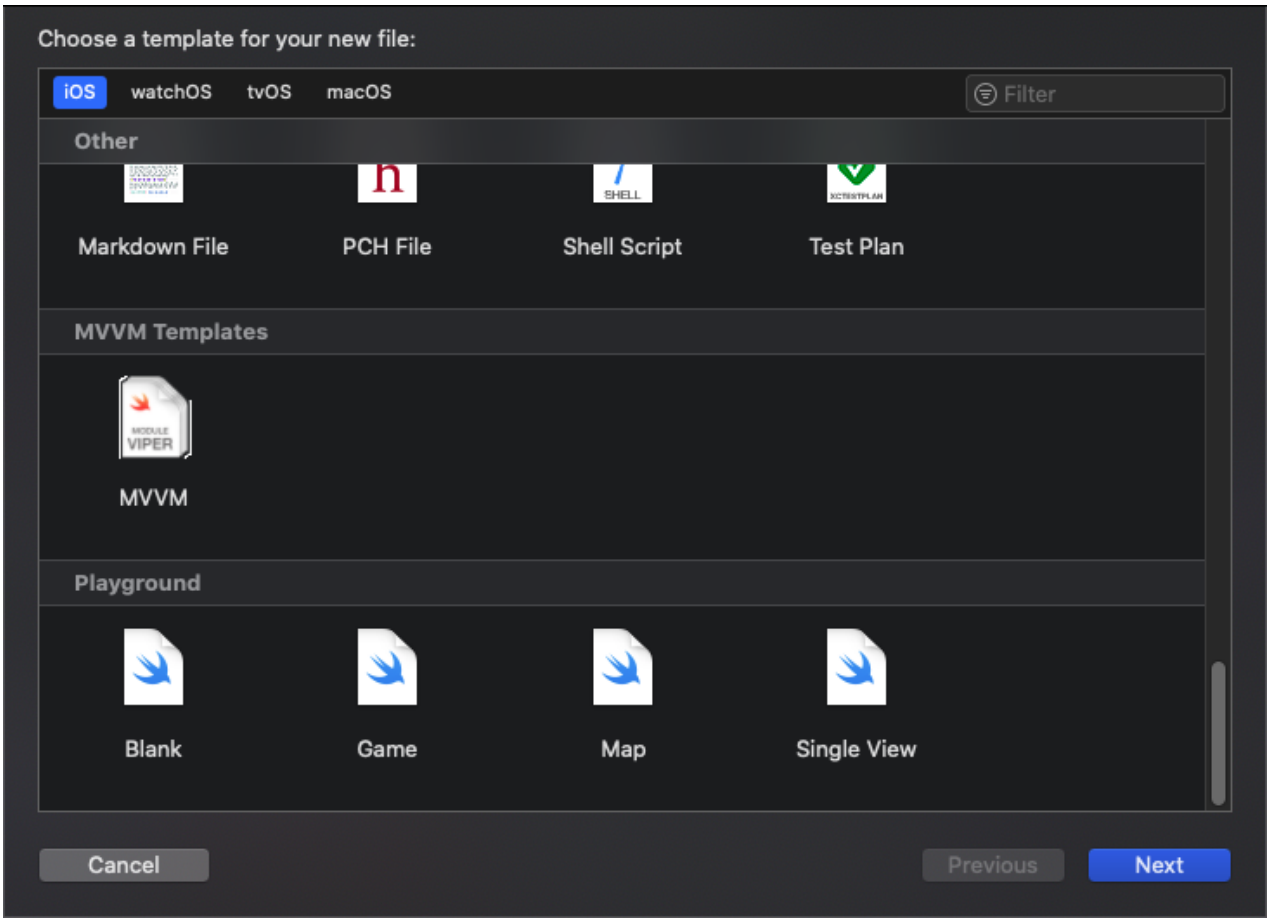
We added also the type of control, we want a textbox that the user can use to enter the name of the module:

```
<key>Type</key>
<string>text</string>
```

Finally:

```
<key>Default</key>
<string>MVVMModule</string>
```

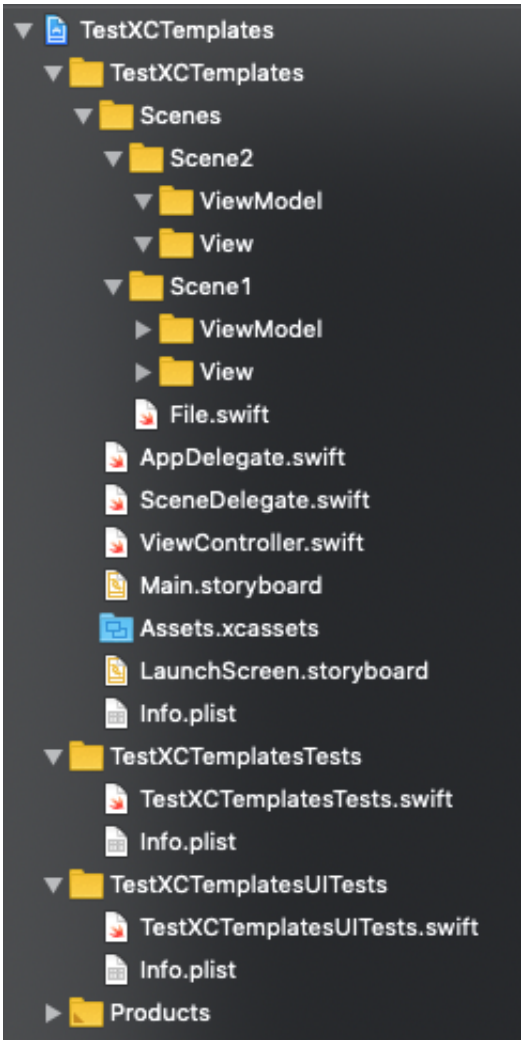
Now, all the configuration is ready. If we did everything right, Xcode should now list our container in the window for creating new files, along with the template we have created:



Xcode: file creation window with our custom category and MVVM template.

If we try to create a file using our new template you will see that nothing is created. This is because inside our **MVVM.xctemplate** folder there are no subfolders and / or files to be used as template for generating the module. It is time to create what we need.

With MVVM I like to organize my files and folders in this way:



My modules are organized in folders and subfolders. See the picture above. I have two modules name respectively **Scene1** and **Scene2**. Inside they both have subfolders for **View** and **ViewModel**. Eventually I can have additional subfolders

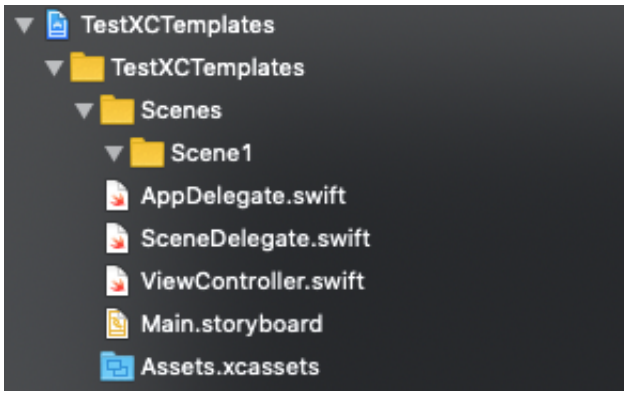
containing specialized files that participate in the functioning of the module.

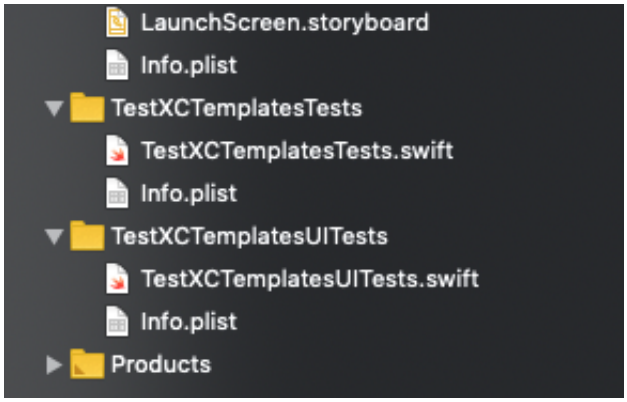
I want our new MVVM template to create the same subfolder structure and add the necessary files to them.

Go back to **Finder** in the folder that contains the contents of the **MVVM.xctemplate** and add 2 subfolders named respectively **View** and **ViewModel**.

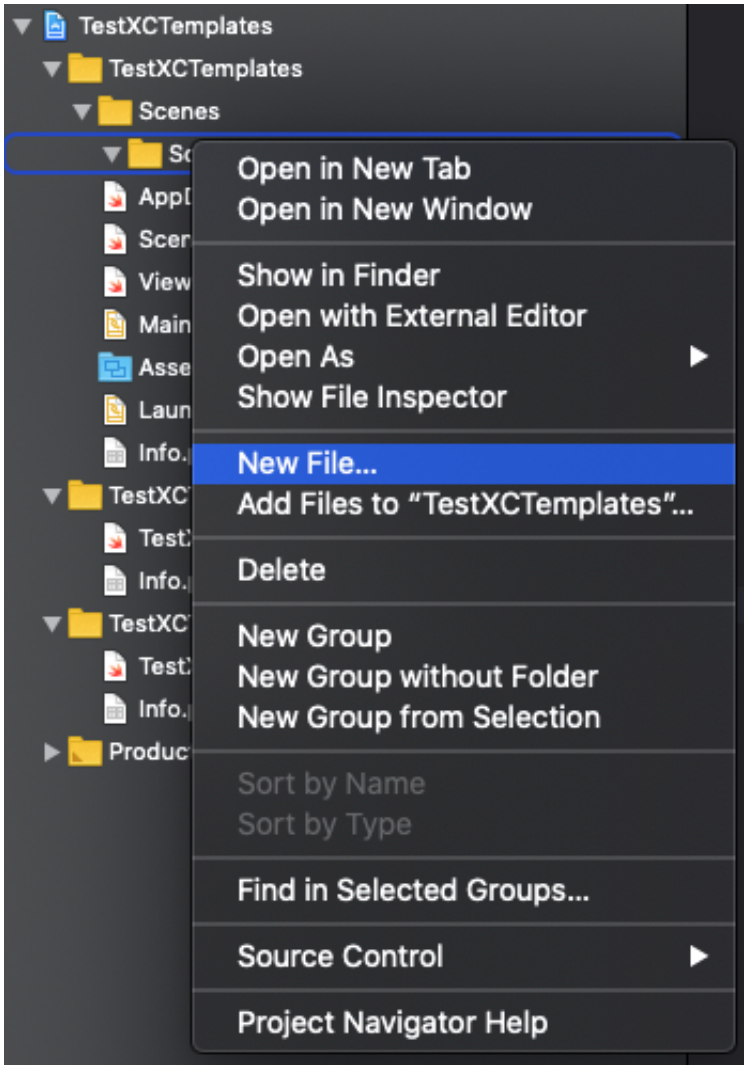
Now, go back to Xcode. Create a new **iOS Single View App project**, do the conventional setup (name of the project, company name, etc. etc.).

When you are done, from Xcode create a group called **Scenes** or **Modules** in the project. Inside the new group create another one: give it the name **Scene1**:





Now right click over the **Scene1** group and in the contextual menu select the option "**New File…**":



Select the new MVVM template:

Type a name for the module, for example **Scene**:

Finally confirm the destination folder that will host the generated content:

Xcode will create the folders we put in our **MVVM.xctemplate** item:

Now these new folders are empty, we must add some **.swift** files inside our **MVVM.xctemplate** folder. Again, go back to the **Finder** window that displays the contents of our **MVVM.xctemplate**.

I will add the required files respectively in their specific subfolder:

Here you can observe the content of our ViewController:

___FILEBASENAME___ViewController.swift

The ViewController above contains some basic initialization for UI and ViewModel. You can see that is very easy to understand. There are lots of comments that the developer should use to remember what he have to do and where he should put things such as outlets, initialization of objects etc.

The most important thing here, is the usage of macros, such as **___FILEHEADER___** that we use to add info about the file and project name, copyright and the author name.

Then, at row 7 where we declare the class name you can find:

___VARIABLE_productName:identifier___ ViewController

See also row number 12, we declared a lazy property that represents our ViewModel. We used the same macro to name it.

This kind of macro above retrieves the string typed by the user in the wizard used to create our MVVM module. We can access that value because in **TemplateInfo.plist** file we set an **Option** with an **identifier** called **productName**. That value is retrieved as a sort of variable. So, if the user writes **Login** as the module name, at the end of the process of creating files and folders, **Xcode** will make the necessary substitutions and the class will be called **LoginViewController**.

Same thing for the **ViewModel** below, the class will be named: **LoginViewModel**.

___FILEBASENAME___ViewModel.swift

With patient and small effort you can setup your templates to best fit your needs adding more files and folders such as Protocols, Storyboards or XIBs and or customizing the content of your files.

What comes next

> *It is available **here** a second article that will describe how to implement **code variants**. We will setup our MVVM template in order to give the user the ability to enter a name for the module then make a choice selecting the type of view he wants to be generated. We want Xcode to generate the files according to the choice of the user. In short, we want to find specialized versions of our views containing eventually an UICollectionView or an UITableView or a common one without particular type of controls.*

References

Xcode 4 Template Documentation, by Steffen Itterheim (book)

Simple Xcode 9 file Template, by Jean-Étienne's Blog

XCode Templates tutorial - How to create custom template step by step, by Marcin Rabieko

Creating your own templates in Xcode by Christoffer Winterkvist

Developer Tools: How to create custom MVC module template for Xcode, by Maxim Vialyx

iOS Swift Programming Xcode Template