Overview of Magento 2's Architecture

Magento's architecture was designed with the intent of making the source code as modularized and extensible as possible. The end goal of that approach is to allow it to be easily adapted and customized according to each project's needs.

Customizing usually means changing the behavior of the platform's code. In the majority of systems, this means changing the "core" code. In Magento, if you are following best practices, this is something you can avoid most of the time, making it possible for a store to keep up to date with the latest security patches and feature releases in a reliable fashion.

Magento 2 is a **Model View ViewModel** (MVVM) system. While being closely related to its sibling Model View Controller (MVC), an MVVM architecture provides a more robust separation between the Model and the View layers. Below is an explanation of each of the layers of a MVVM system:

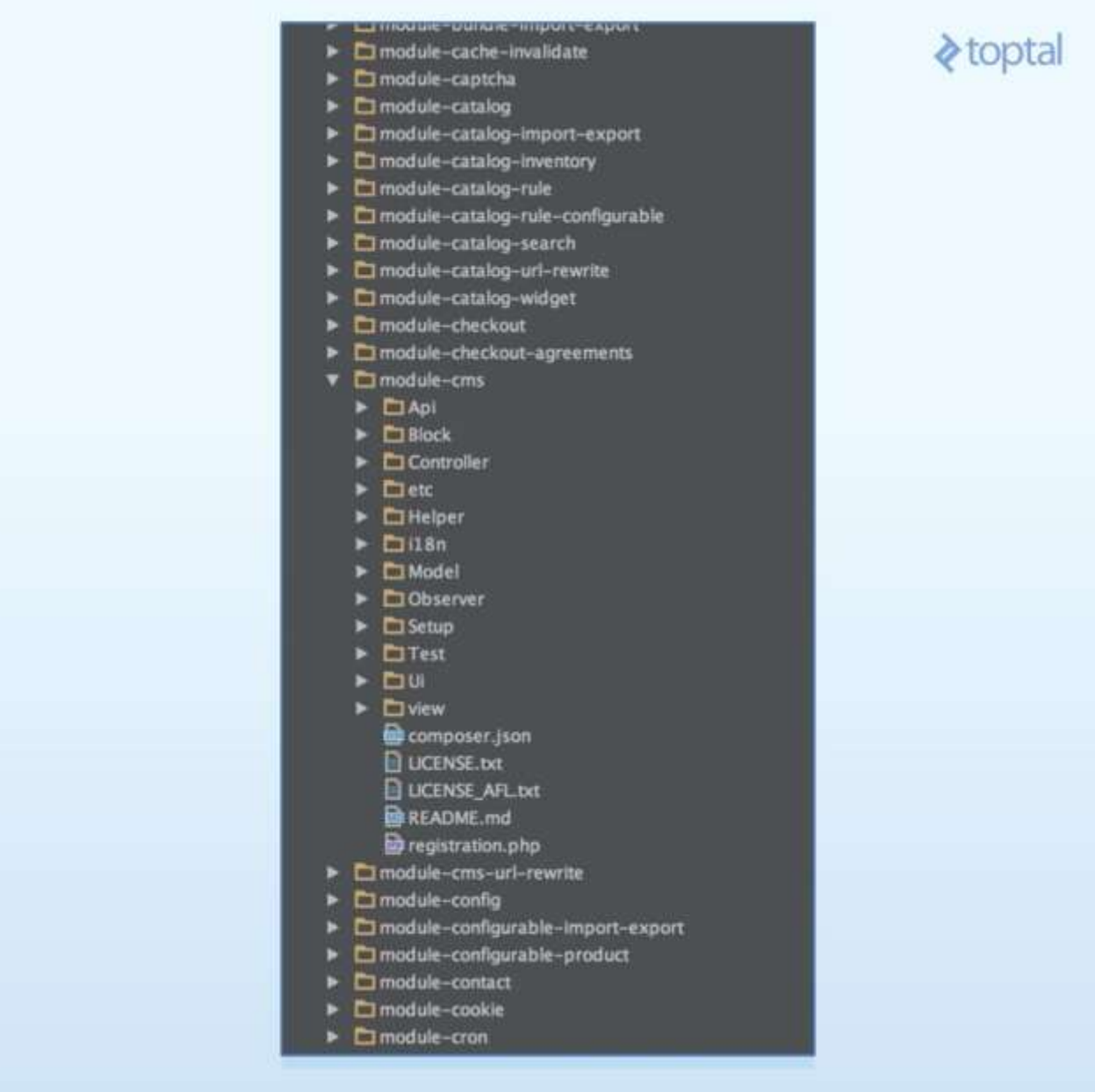
- The **Model** holds the *business logic* of the application, and depends on an associated class—the ResourceModel—for database access. Models rely on *service contracts* to expose their functionality to the other layers of the application.
- The **View** is the structure and layout of what a user sees on a screen - the actual HTML. This is achieved in the PHTML files distributed with modules. PHTML files are associated to each ViewModel in the **Layout XML files**, which would be referred to as ***binders*** in the MVVM dialect. The layout files might also assign JavaScript files to be used in the final page.

- The **ViewModel** interacts with the Model layer, exposing only the necessary information to the View layer. In Magento 2, this is handled by the module's **Block** classes. Note that this was usually part of the Controller role of an MVC system. On MVVM, the controller is only responsible for handling the user flow, meaning that it receives requests and either tells the system to render a view or to redirect the user to another route.

A Magento 2 module consists of some, if not all, elements of the architecture described above. The overall architecture is described below (source):

A Magento 2 module can in turn define external dependencies by using Composer, PHP's dependency manager. In the diagram above, you see that the Magento 2 core modules depend on the Zend Framework, Symfony as well as other third-party libraries.

Below is the structure of Magento/Cms, a Magento 2 core module responsible for handling the creation of pages and static blocks.



Each folder holds one part of the architecture, as follows:

- **Api:** Service contracts, defining service interfaces and data interfaces
- **Block:** The ViewModels of our MVVM architecture
- **Controller:** Controllers, responsible for handling the user's flow while interacting with the system

- **etc:** Configuration XML files—The module defines itself and its parts (routes, models, blocks, observers, and cron jobs) within this folder. The ***etc*** files can also be used by non-core modules to override the functionality of core modules.
- **Helper:** Helper classes that hold code used in more than one application layer. For example, in the Cms module, helper classes are responsible for preparing HTML for presentation to the browser.
- **i18n:** Holds internationalization CSV files, used for translation
- **Model:** For Models and ResourceModels
- **Observer:** Holds Observers, or Models which are "observing" system events. Usually, when such an event is fired, the observer instantiates a Model to handle the necessary business logic for such an event.
- **Setup:** Migration classes, responsible for schema and data creation
- **Test:** Unit tests
- **Ui:** UI elements such as grids and forms used in the admin application
- **view:** Layout (XML) files and template (PHTML) files for the front-end and admin application

It is also interesting to notice that, in practice, all of Magento 2's inner workings live inside a module. In the image above, you can see, for instance, `Magento_Checkout`, responsible for the checkout process, and `Magento_Catalog`, responsible for the handling of products and categories. Basically, what this tells us is that learning how to work with modules is the most important part of becoming a Magento 2 developer.

All right, after this relatively brief introduction to the system architecture and module structure, let's do something more concrete, shall we? Next, we will go through the traditional Weblog tutorial in order to get you comfortable with Magento 2 and on track to become a Magento 2 Developer. Before that, we need to set up a development environment. Let's get to it!

Setting up the Magento 2 Module Development Environment

At the time of this writing, we were able to use the official Magento 2 DevBox, which is a Magento 2 Docker container. Docker on macOS is something I still consider to be unusable, at least with a system which heavily depends on fast disk I/O such as Magento 2. So, we will do it the traditional way: Install all packages natively on our own machine.