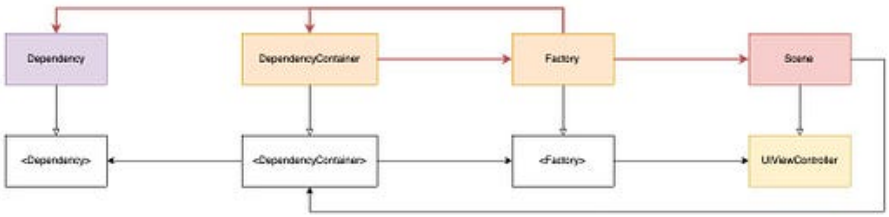

Dependency management done manually in Swift

Resolve dependencies easily by writing your own containers

Sven Korset

Follow

Better Programming · ~7 min read · August 16, 2019 (Updated: December 11, 2021) · Free: No

Pieces of a scalable iOS app architecture

If you depend on something, you lose some freedom. But if you can easily replace that dependency, you'll at least have the freedom to decide what you are depending on! 🙃

In code, this means defining only abstract dependencies via protocols so you can easily exchange the concrete implementation. The concrete dependency is then assigned from the outside at runtime. This is called Dependency Injection (DI) or, in a broader sense, Dependency Management (DM).

Alexey Kuznetsov explains very clearly how this can be done in his article "Dependency Injection in swift". And Joe Masilotti shows why it should be done in "Better Unit Testing with Swift".

dependencies in it. When creating objects, the container is queried, which resolves the dependencies by returning these objects.

An interesting alternative is Weaver. Unlike Swinject, Weaver is a command-line tool and is, therefore, not included as a framework in the project. During the build process, it will automatically generate the dependency code.

As Théophane Rupin explains in his article "Weaver: A Painless Dependency Injection Framework for Swift", Weaver has several advantages over Swinject, e.g. a compile-time resolution, while Swinject may experience run-time crashes due to non-optional options. Also, dependency definition using comments is a very interesting approach, but it also poses some problems because you suddenly write "code" in comments. 😬

Anyone who is happy with a framework or command-line tool should, of course, use one. However, there is a certain irony in making oneself dependent on something new for the purpose of solving dependencies. 😏

Manual Dependency Injection

In principle, you need a container that holds all dependencies and that you pass to the classes that need the dependencies. To avoid being directly dependent on this container, you use protocols and also for the dependencies themselves.

Line 5 defines the dependency: here a *Server* class whose exemplary task is to communicate with a server.

In order not to be directly dependent on this specific class, the associated protocol is defined in line 1. In this case, protocols are used in the classical sense as interfaces and are, therefore, named as such. Back to the good old days where header files were used. 😅

The same happens on lines 9 and 13, where the container is defined. The *DependencyContainer* represents the container holding the *Server* dependency in the variable `server`. The container already knows nothing about the concrete *Server* class because it only ever references the *ServerInterface*.

A fictitious *ViewController* then gets the container injected in the `init` method. Again, the *ViewController* does not know the concrete *DependencyContainer* class, only its protocol. This is sufficient, however, to start a server request via the dependent *Server* class in line 29.

A root object, such as an *AppDelegate*, must then create concrete instances. Here, this happens as an example in line 38. A concrete instance of type *DependencyContainer* and *Server* is created.

Line 39 finally creates the concrete *ViewController* and injects the dependencies. If you do not manually instantiate your *ViewController*, you can also pass the container via property

If a *ViewController* calls a new one, this is done again via DI, which is outlined as an example in a `navigate` method in line 32.

If further dependencies are required, all you have to do is extend the *DependencyContainer* and pass the concrete dependencies on the instantiation.

In a *UnitTest* environment, you would not give the *DependencyContainer* a specific *Server* instance but maybe a *ServerMock* instance. The mock then simply implements the *ServerInterface* and checks whether the correct methods are called during the test or returns test data.

And because the *DependencyContainer* has its own interface protocol, it is easy to replace it with a special test stub that creates all dependency mocks as shown in the following example.

That's actually all. Pretty manageable, right? But what does a DI framework actually do? 🤔

> A dependency injection framework merely provides a container. You still have to write the rest yourself.

So, does it really need a framework? Everyone has to decide that for themselves, but in my opinion, it works very well without one, as

Transient Dependencies

Wait, line 33 creates a new *NextViewController*. Isn't that also a dependency? 🧐

Indeed!

That's why this also belongs in the *DependencyContainer*. However, *UIViewController* dependencies are somewhat special because we really do not want to hold an instance of a *NextViewController* in the container. If we request a *NextViewController* via a container, it should be recreated each time. This dependent component is, therefore, transient.

Also, we do not really want to know what specific *NextViewController* class it is. It would be enough for navigation if we know that it is a *UIViewController*. This will prevent *ViewController* from knowing anything about *NextViewController*.

Factory pattern to the rescue! 🤠

> The factory pattern describes how an object can create new objects through an interface without having to know the specific class of these objects.

input, we get back the desired concrete *UIViewController*.

Defined in line 1, *Scene* is a simple enum mapped to the concrete *UIViewController* class. The instantiation takes place in the *DependencyContainer*, where the container injects itself as a dependency reference to the new *NextViewController* instance.

This allows the *ViewController* in line 31 to create an instance from *NextViewController* without being directly dependent on the class. By the way, the direct dependency of the *AppDelegate* root object on the *ViewController* is also resolved that way now in line 37.

Instead of being dependent on the concrete *UIViewController* classes, one is dependent now on the *Scene* enum. However, this is justifiable, because a *Logic* module or a *ViewController* must finally say to which new scene to navigate. This way, it's a simple, independent type.

So, if more view controllers are added, you have to extend the *Scene* enum and the *DependencyContainer*. Potential merge conflicts should then be easy to fix. Usually. 😅

Scene Setup Parameter

You often want to initialize a scene with additional data, not just global dependencies. For example, the user in the *ViewController* could make an input, which should then be passed to *NextViewController* for further processing.

This value is, therefore, also a dependency for *NextViewController*, but only for this and not for all other controllers. These parameters

Quite simply, you define a struct as in line 25 that could hold any data you want to pass to the new controller. Then this dependency is to be expected on line 28.

Of course, the *ViewController* must create it in line 19 and fill it accordingly, before passing it as an associated value to the *Scene* enum on line 20.

The *DependencyContainer* then retrieves this associated value from the enum in line 11 and passes it as an extra parameter to the *NextViewController* during instantiation.

Et voilà, no other *UIViewController* needs to know anything about this parameter. 🤫

Nested Dependencies

In the first example, there was a server dependency that could then be passed through to any number of *UIViewControllers*. But what if, during runtime, a new dependency occurs? For example, beginning from a certain point, we have a logged-in user who becomes a new dependency for all the following *UIViewControllers* down the flow? 🤔

John Sundell shows this use case in his lecture "The Lost Art of System Design" and calls his solution "Lock & Keys".

is done as a dependency on the old container, so the old container creates the new container.

Here a distinction is made between a pre-login app state and a post-login app state. Accordingly, there are two containers defined in lines 13 and 36. In the beginning, the app is in the pre-login state, and, therefore, only the *PreLoginDependencyContainer* exists there.

At a particular time, a login is made via line 58. From there on, there is a concrete user object, and the app should now switch to the post-login state with a corresponding transition. For this purpose, the *PostLoginDependencyContainer* with the required user parameter is created in line 60.

The *PostLoginDependencyContainer* now knows all scenes of the post-login states, which are defined in line 26. So you can also create the *NextViewController* in line 61 and present it. In any case, the user dependency exists for the *NextViewController* and any subsequent controllers.

If further dependencies are still needed, e.g. the server object, this would also have to be passed to the *PostLoginDependencyContainer*. Of course, via injection. 😊

Conclusion

have to constantly reinvent the wheel, but sometimes it's faster than installing an existing one. 😉

In my DemoApp Project (DAP), you can see the manual use of DI in a more complex environment. There, the factory part is outsourced in its own *Factory* class, which is held by the *DependencyContainer* as a dependency. But that's just icing. 🤓

Incidentally, this is a post from the "Pieces of a scalable iOS app architecture" series.

#ios #swift #programming #mobile #software-development