Modular Architecture on iOS/macOS

Building Large Scalable iOS Apps and Frameworks With Domain Driven Design

Cyril Cermak

Modular Architecture on iOS/macOS

Building large scalable iOS apps and frameworks with Domain Driven Design

Dedication

"To my Mom and Dad, because they really tried."

&&

"To all my non-tech friends whom I am fixing problems with not enough disk space, printers, forgotten passcodes etc. who will never read this line."

&&

"To all passionate engineers who are solving tough problems on a daily basis with a smile, it is great pleasure for everyone to work with you!"

&&

"Finally, to whoever my current girlfriend is…"

About the author

Cyril Cermak is a software engineer by heart and author of this book. Most of his professional career was spent by building iOS apps or iOS frameworks. Beginning by Skoda Auto Connect App in Prague, continuing building iOS platform for Freelancer Ltd in Sydney, numerous start-ups on the side, and as of now being an iOS TechLead in Stuttgart for Porsche AG. In this book, Cyril describes different approaches for building modular iOS architecture so as some mechanisms and essential knowledge that should help you, the reader, to decide which approach would fit in the best or should be considered on your project.

Reviewers

Joerg Nestele … Ask Joerg whether he wants be the reviewer.

Contents

Introduction

In the software engineering field, people are going from project to project, gaining different kind of experience out of it. Especially, on iOS mostly the monolithic approaches are used. In some cases it makes totally sense, so nothing against it. However, scaling up the team, or even better, team of teams on monolithically built app is horrifying and nearly impossible without some major time impacts on a daily basis. Numerous problems will rise up, that are limiting the way how iOS projects are built or managed on the organisational level.

Scaling up the monolithic approach to a team of e.g 10+ developers will most likely result in hell. By hell, I mean, resolving xcodeproj issues, where in the worst case both parties renamed or edited and deleted the same file, touched the same {storyboard|xib} file, or typically both worked on the same file which would resolve in classic merge conflicts. Somehow, those issues we all got used to live with…

The deal breaker comes when your PO/PM/CTO/CEO basically anybody higher on the company's food-chain than you are will come to the team to announce that they are planning to release a new flavour of the app or to divide the current app into two separate parts. Afterwards, the engineering decision needs to be made. Either, to continue with this monolithic approach, create different targets, assign files towards the new flavour of the app and continue living in multiplied hell and hoping that some requirement such as shipping core components of the app to a subsidiary or open-sourcing it as a framework will not come into play.

Not surprisingly, a better approach would be to start refactoring the app into a modular approach, where each team can be responsible for particular frameworks (parts of the app) that are then linked towards final customer facing apps. That will most certainly take time as it will not be easy to transform it but the future of company's mobile engineering will be faster, scalable, maintainable and even ready to distribute or open source some SDKs of it to the outer world.

Another scenario could be, that you are already working on an app which is done in a modular way but your app takes at around 20 mins to compile. As it is a huge legacy codebase that was in development past 8 or so years and linked every possible 3rd party library along the way. The decision was made to modularise it with Cocoapods therefore, you cannot link easily already pre-compiled libraries with Carthage and every project clean you can take double shot of espresso. I had been there, trust me, it is another type of hell, definitely not a place where anyone would like to be. I described the whole migration process of such project here. Of course, in this book you will read about it in more detail.

Nowadays, as an iOS tech lead, I am often getting asked some questions all over again from new teams or new colleagues regards those topics. Thereafter, I decided to sum it up and tried to get the whole subject covered in this book. The purpose of it is to help developers working on such architectures to gain the speed, knowledge, ideas and understanding faster.

Hopefully, this introduction gave enough motivation to deep dive further into this book.

What you Need

The latest version of Xcode for compiling the demo examples, brew to install some mandatory dependencies, Ruby and bundler for running scripts and downloading some ruby gems.

What is this book about

This book describes essentials about building modular architecture on iOS. You will find examples of different approaches, framework types, pros and cons, common problems and so on. By the end of this book you should have a very good understanding of what benefits will bring such architecture to your project, whether it is necessary at all or which would be the best way for modularising the project.

At the end of this book, you can read experience from the top notch iOS engineers working across numerous different projects from different countries and continents.

What is this book NOT about

SwiftUI.

Modular Architecture

```
- Modular -

adjective - employing or involving a module or modules as the basis of
design or construction: "modular housing units"
```

In the introduction, I briefly touched the motivation for building the project in the modular way. To summaries it, modular architecture will give you much more freedom when it comes to the product decisions that will influence the overall app engineering. Such as, building another app for the same company, open-sourcing some parts of the existing codebase, scaling the team of developers and so on. With the already existing mobile foundation, it will be done way faster and cleaner.

To be fair, maintaining such software foundation of a company might be also really difficult. By maintaining, I mean, taking care of the CI/CD, old projects developed on top of the foundation that was heavily refactored in the meantime, legacy code, keeping it up-to date with the latest development tools and so on. No need to say, that on a very large project, this could be a work of one standalone team.

This book describes building such large scalable architecture with domain driven design by example; The software foundation for the International Space Station.

Design

In this book, I chose to use the architecture that I think is the most evolved. It is a five layer architecture that consists of those layers:

- Application
- Domain
- Service
- Core
- Shared

Each layer is explained in the following chapter.

Nevertheless, the same principles can be applied for other architectural types as well for example; feature oriented architecture, where the layers could be defined as:

- Application
- Feature
- Core
- Shared

Now to the specific layers.

Layers

Let us have a look now on each layer and its purpose. Modules within layers are then demonstrated with the example in the following chapter.

Application Layer

Application layer consists of the final customer facing products; applications. Applications are linking domains and via configurations and Scaffold module glueing all different parts together. In such architecture, the App is basically a container that puts pieces together.

Nevertheless, App might also contain some necessary Application implementations like receiving push notifications, handling deep linking, permissions and so on.

Patterns, that will help achieve such goals will be described later.

For example an app in an e-commerce business could be `The Shop` for online customer and `Cashier` for the employees of that company.

Domain Layer

Domain layer links services and other modules from layers below and uses them to implement business domain needs of the company or the project. Domains will contain for example the users flow within the particular domain part of the app. So as, the necessary components for it like; view controllers, views, models and view models. No need to say, that it depends on the teams preferences and technical experience which pattern will be used for creating screens. Personally, the reactive MVVM+C is my favourite but more on that later.

For example a domain in an e-commerce app could be a `Checkout` or `Store Items`.

Service Layer

Services are modules supporting domains. Each domain, can link several services in order to achieve desired outcomes. Such service will most likely talk to the backend, get the data out of it, persist them in its own storage and expose them to domains.

For example a service in an e-commerce app could be a `Checkout Service` which will handle all the necessary communication with the backend so as proceeding with the credit card payments etc.

Core Layer

Core layer is the enabler for the whole app. Services will link the necessary modules out of it for e.g communicating with the backend or providing general abstraction of persisting the data. Domains will link e.g uicomponents for easier implementation of screens and so on.

For example a core module in an e-commerce app could be `Network` or `UIComponents`

Shared Layer

Shared layer is a supporting layer for the whole framework. It can happen that this layer might not need to exist, therefore, it is not considered in all diagrams. However, a perfect example for the shared layer is some logging mechanism. Where even core layer modules will want to log some output. That would potentially lead to duplicates, which could be solved by the shared layer.

For example a shared module in an e-commerce app could be `Logging` or `AppAnalytics`

Example: International Space Station

Now in this example, we will have a look at how such architecture could really look like for International Space Station. Diagram below shows the four layer architecture with the modules and linking.

While this chapter is rather theoretical, in the following chapters everything will be explained and showcased in practice.
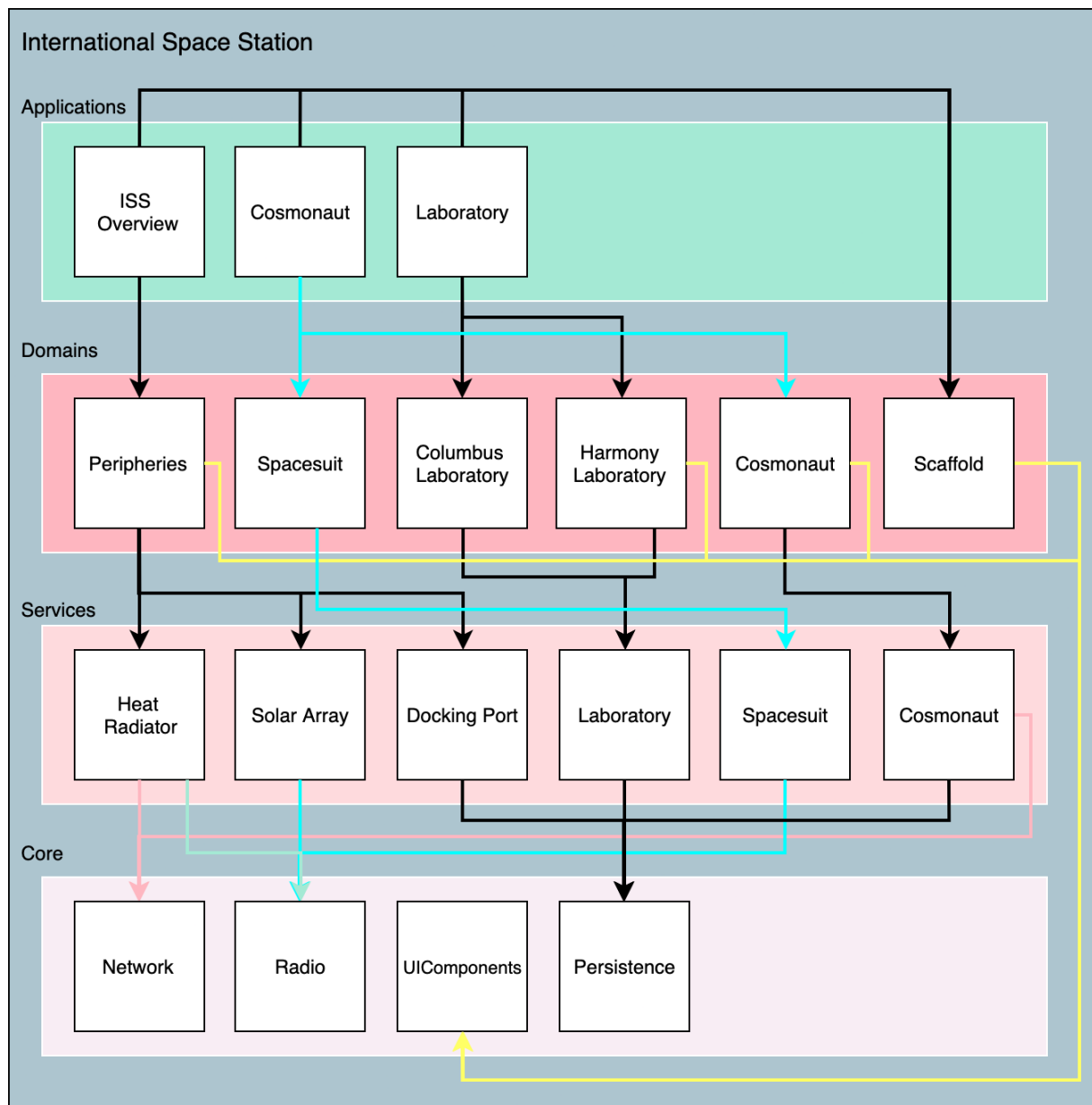
Figure 1: Overview

The example has three applications. - `ISS Overview`: app that shows astronauts the overall status of the space station - `Cosmonaut`: app where Cosmonaut can control his spacesuit so as his supplies and personal information - `Laboratory`: app from which the laboratories on the space station can be controlled

As described above, all apps are linking Scaffold module which provides the bootstrapping for the app as the app itself is the container.

ISS Overview

`ISS Overview` app links `Peripheries` domain which implements logic and screens for peripheries of the station.

The `Peripheries` domain links `Heat Radiator`, `Solar Array` and `Docking Port` services from whom the data about those peripheries are gathered so as `UIComponents` for bootstrapping the screens development.

Linked services are using `Network` and `Radio` core modules which are providing the foundation for the communication with other systems via network protocols. `Radio` in this case could implement some communication channel via BLE or other technology which would connect to the solar array or heat radiator. Diagram below describes the concrete linking of modules for the app.
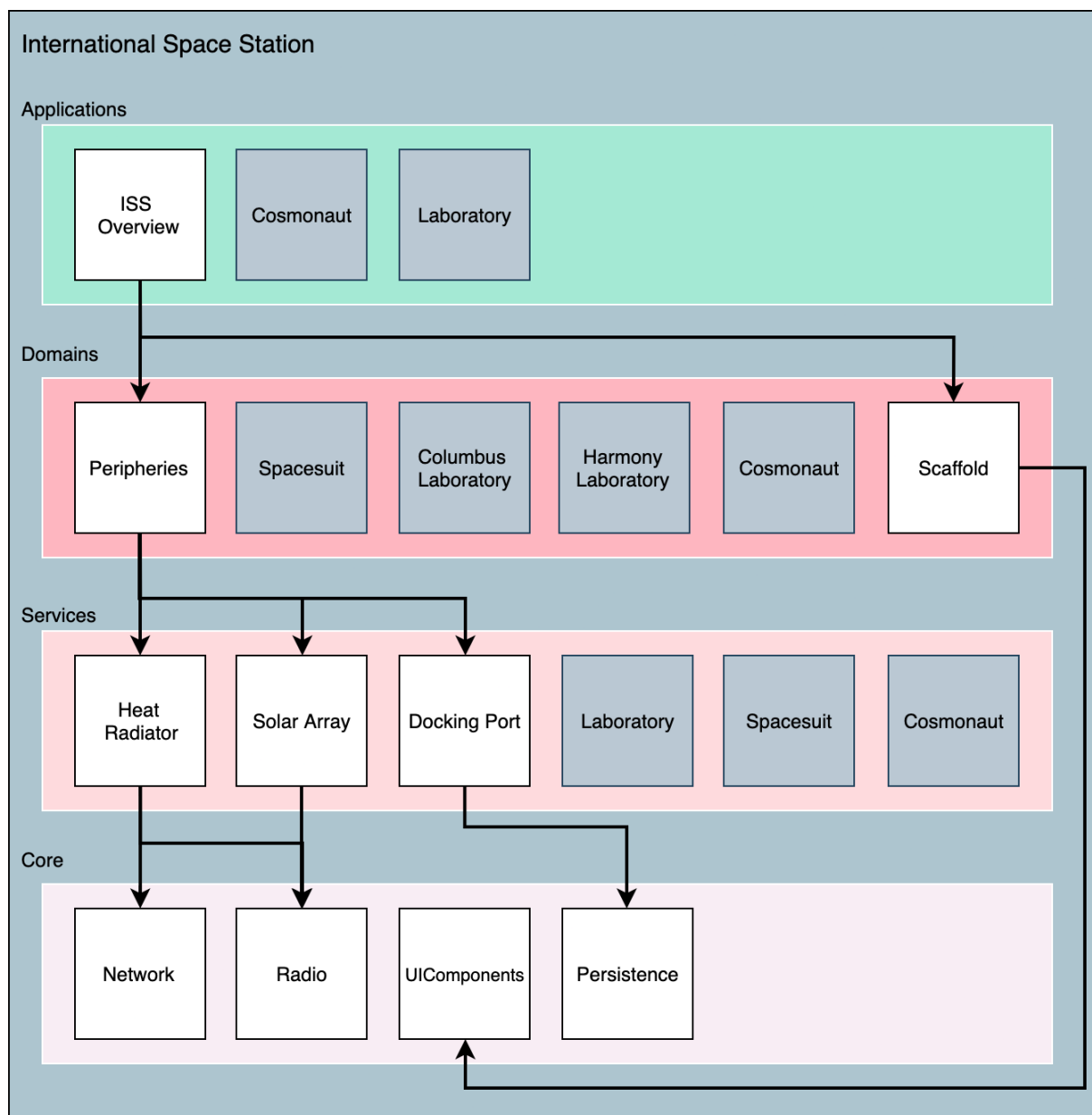
Figure 2: ISS Overview

Cosmonaut

`Cosmonaut` app links `Spacesuit` and `Cosmonaut` domains. Same as for every other domain, each module is responsible for screens and users flow through the part of the app.

`Spacesuit` and `Cosmonaut` domains link `Spacesuit` and `Cosmonaut` services that are providing data for domain defined screens so as `UIComponents` who are providing the UI parts.

`Spacesuit` service is using `Radio` for communication with cosmonauts spacesuit via BLE or other type of radio technology. `Cosmonaut` service is using `Network` for updating Huston about the current state of the `Cosmonaut` so as `Persistence` for storing the data of the cosmonaut for offline usage.
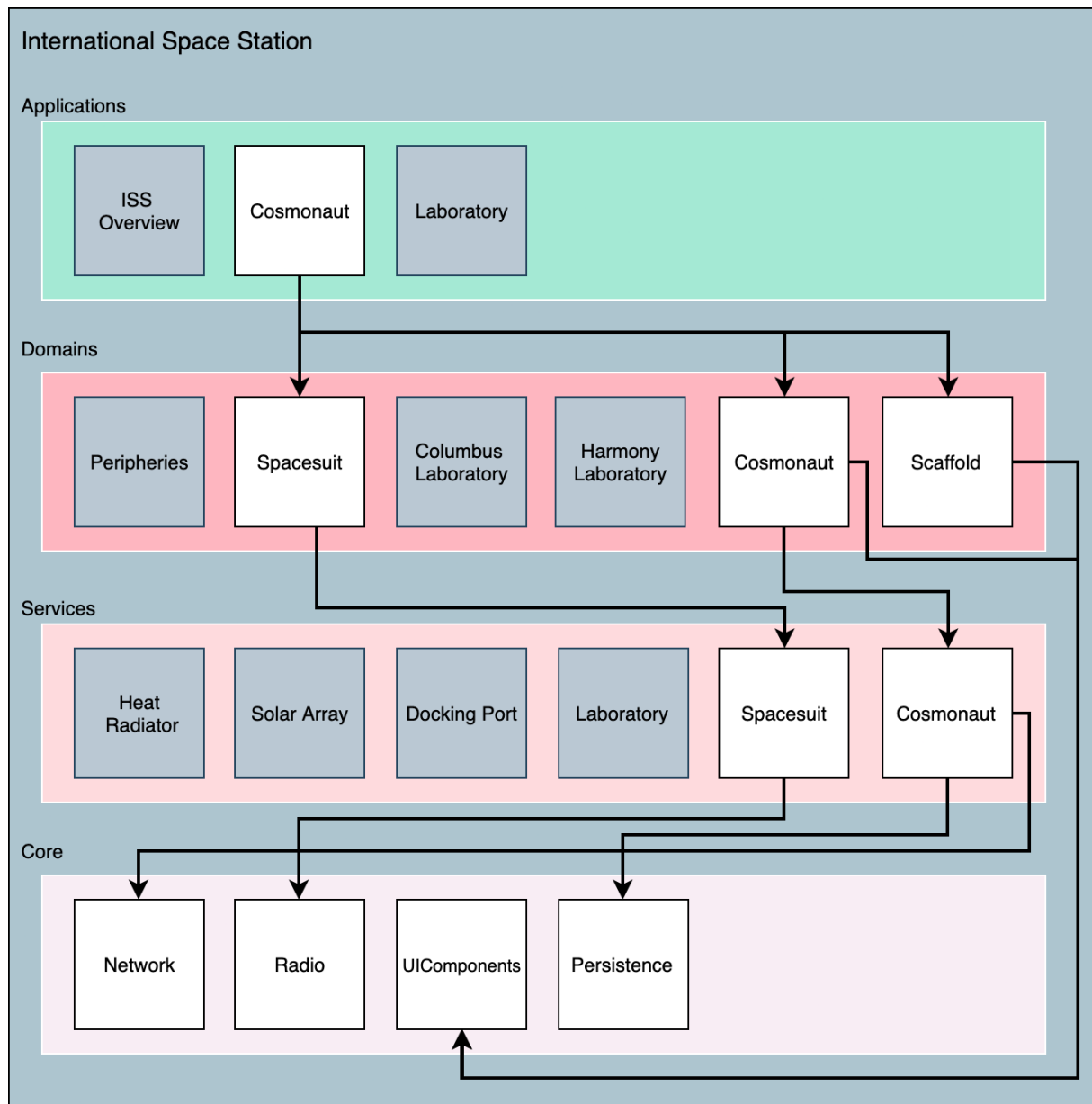
Figure 3: Cosmonaut App

Laboratory

I will leave this one for the reader to figure it out.

Conclusion

As you can probably imagine, scaling of the architecture described above should not be a problem. When it comes to extending for example the ISS Overview app for another ISS periphery, a new domain module can be easily added with some service modules etc.

When a requirement comes for creating a new app for e.g. cosmonauts, the new app can already link the battlefield proven and tested Cosmonaut domain module with other necessary modules that are required. Development of the new app will become way more easier due to that.

The knowledge of the software remains in one repository where developers have access to and can learn from is also very beneficial.

There are of course some disadvantages as well. For example, onboarding new developers on such architecture might take a while, especially when there is already huge existing codebase. There, the pair programming comes into play so as proper project onboarding, software architecture documents and the overall documentation of modules and the whole project.

Libraries on Apple's ecosystem

Before we deep dive into the development of previously described architecture there is some essential knowledge that needs to be explained. Especially, the type of library that is going to be used for building such project and its behaviour.

In Apple's ecosystem as of today we have two main options when it comes to creating a library. The library can either be statically or dynamically linked. Dynamic library previously known as `Cocoa Touch Framework`, nowadays simplified to `Framework` and the statically linked, the `Static Library`.
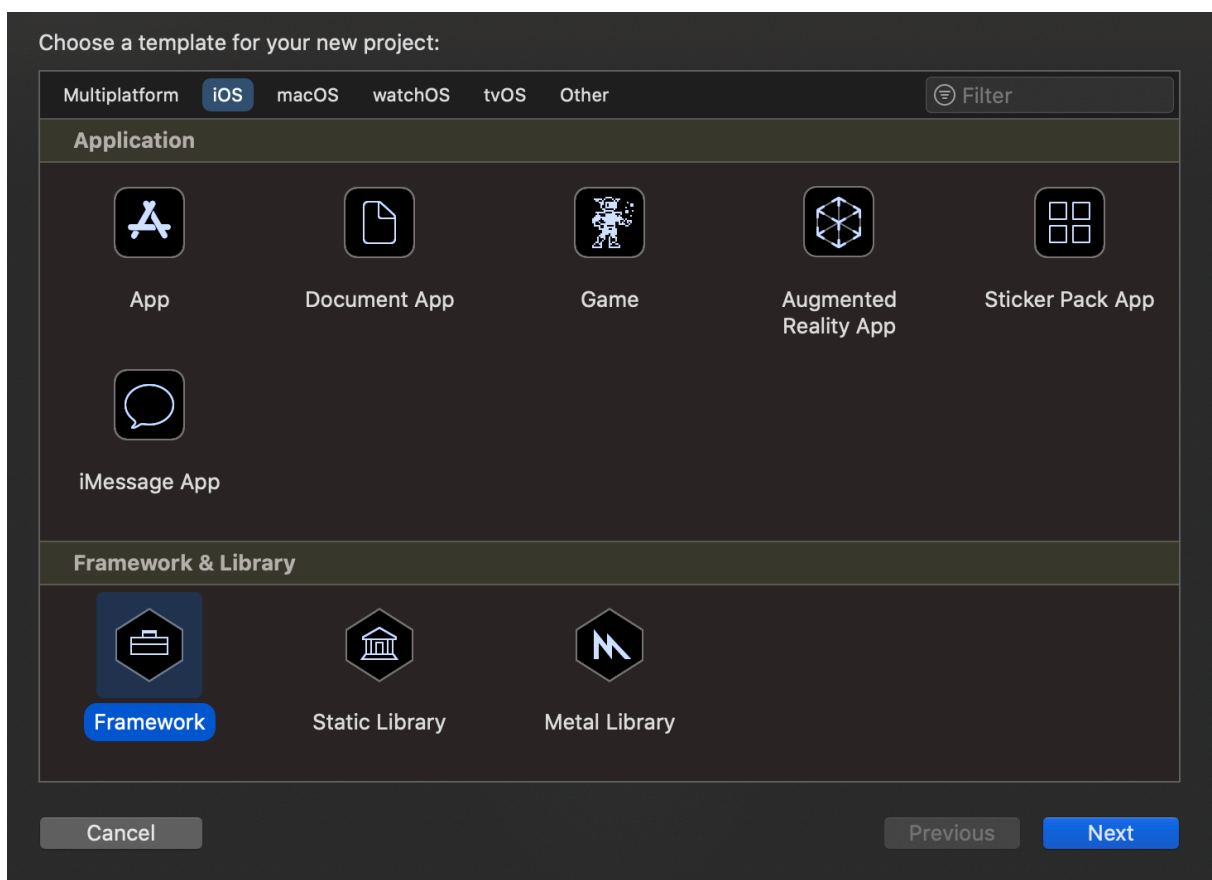
Figure 4: Xcode Framework Types

What is a library?

To quote Apple: *"Libraries define symbols that are not built as part of your target."*

What are symbols? *Symbols reference to chunks of code or data within binary.*

Types of libraries:

1) Dynamicaly linked

- **Dylib**: Library that has its own Mach-O (explained later) binary. (`.dylib`)
- **Framework**: Framework is a bundle that contains the binary and other resources the binary might need during the run time. (`.framework`)
- **TBDs**: Text Based Dynamic Library Stubs is a text stubbed library (symbols only) around a binary without including it as the binary resides on the target system, used by Apple to ship lightweight SDKs for development. (`.tbd`)
- **XCFramework**: From Xcode 11 the XCFramework was introduced which allows to group a set of frameworks for different platforms e.g `macOS`, `iOS`, `iOS simulator`,`watchOS` etc. (`.xcframework`)

2) Statically linked

- **Archive**: Archive of a compiler produced object files with object code. (`.a`)
- **Framework**: Framework contains the static binary or static archive with additional resources the library might need. (`.framework`)
- **XCFramework**: Same as for dynamically linked library the XCFramework can be used with statically linked. (`.xcframework`)

We can look at a framework as some bundle that is standalone and can be attached to a project with its own binary. Nevertheless, the binary cannot run by itself, it must be part of some runnable target. So what is exactly the difference?

Dynamic vs static library?

The main difference between a static and dynamic library is in the Inversion Of Control (IoC) and how they are linked towards the main executable. When you are using something from a static library, you are in control of it as it becomes part of the main executable during the build process (linking). On the other hand when you are using something from a dynamic framework you are passing responsibility for it to the framework as framework is dynamically linked on the app start to the executables process. I'll delve more into IoC in the paragraph below. Static libraries, at least on iOS, cannot contain anything other than the executable code unless they are wrapped into a static framework. A framework (dynamic or static) can contain everything you can think of e.g storyboards, xibs, images and so on…

As mentioned above, the way dynamic framework code execution works is slightly different than in a classic project or a static library. For instance, calling a function from the dynamic framework is done through a frameworks interface. Let's say a class from a framework is instantiated in the project and

then a specific method is called on it. When the call is being done you are passing the responsibility for it to the dynamic framework and the framework itself then makes sure that the specific action is executed and the results then passed back to the caller. This programming paradigm is known as Inversion Of Control. Thanks to the umbrella file and module map you know exactly what you can access and instantiate from the dynamic framework after the framework was built.

A dynamic framework does not support any Bridging-Header file; instead there is an umbrella.h file. An umbrella file should contain all Objective-C imports as you would normally have in the bridging-Header file. The umbrella file is basically one big interface for the dynamic framework and it is usually named after the framework name e.g myframework.h. If you do not want to manually add all the Objective-C headers, you can just mark `.h` files as public. Xcode generates headers for ObjC for public files when building. It does the same thing for Swift files as it puts the ClassName-Swift.h into the umbrella file and exposes the Swift publicly available interfaces via swiftmodule definition. You can check the final umbrella file and swiftmodule under the derived data folder of the compiled framework.

On the other hand, a statically linked library is attached directly to the main executable during linking as the library contains already pre-compiled archive of the source files with symbols. That being said, there is no need for umbrella file so as IoC like in dynamic framework.

No need to say, classes and other structures must be marked as public to be visible outside of a framework or a library. Not surprisingly, only objects that are needed for clients of a framework or a library should be exposed.

PROS & CONS

Now let's have a look at some pros & cons of both.

Dynamic:

- **PROS**
    - Faster app start time as library is linked during app launch time or runtime, therefore the main executable has lesser memory footprint to load.
    - Can be opened on demand, therefore, might not get opened at all if user do not open specific part of the app (`dlopen`).
    - Can be linked transitively to other dynamic libraries without any difficulty.
    - Can be exchanged without the recompile of the main executable just by replacing the framework with a new version.
    - Is loaded into a different memory space than the main executable.
    - Can be shared in between applications especially useful for system libraries.
    - Can be loaded partially, only the needed symbols can be loaded into the memory (`dlsym`).

- – Can be loaded lazily, only objects that are referenced will be loaded.
- – Library can perform some cleanup tasks when it is closed (`dlclose`).

- **CONS**

  - – The target must copy all dynamic libraries else the app crashes on the start or during runtime with `dyld library not found`.
  - – The overall size of the binary is bigger than the static one as compiler can strip of symbols from the static library during the compile time while in dynamic library the symbols at least the public ones must remain.
  - – Potential replacement of a dynamic library with a new version with different interfaces can break the main executable.
  - – Slower library API calls as it is loaded into a different memory space and called via library interface.
  - – Launch time of the app might take longer if all dynamic libraries are opened during the launch time.

Static:

- **PROS**

  - – Is part of the main executable therefore, the app cannot crash during launch or runtime due to a missing library.
  - – Overall smaller size of the final executable as the symbols can be stripped of.
  - – In terms of calls speed there is no difference in between the main executable and the library as the library is part of the main executable.
  - – Compiler can provide some extra optimisation during the build time of the main executable.

- **CONS**

  - – The library must NOT be linked transitively as each link of the library would add it again. The library must be present only once in the memory either in the main executable or one of its dependencies else the app on the start needs to decide which library is going to be used.
  - – The main executable must be recompiled when the library has an update even though the library's interface remains the same.
  - – Memory footprint of the main executable is bigger which implies the load time of the App is slower.

Essentials

When building any kind of modular architecture, it is crucial to keep in mind that a static library is attached to the executable while dynamic one is opened and linked at the start time. Thereafter, if there are two frameworks linking the same static library the app will launch with warnings `Class loaded twice ... one of them will be used`. issue. That causes much slower app start as the app needs to decide which of those classes will be used. So as, in the worst case when two different versions of the same static library are used the app will use them interchangeably. The debugging will become a horror if that case happens that being said, it is very important to be sure that the linking was done right and no warnings appears.

All that is the reason why using dynamically linked frameworks for internal development is the way to go. However, working with static libraries is, unfortunately, inevitable especially when working with 3rd party libraries. Big companies like Google, Microsoft or Amazon are using static libraries for distributing their SDKs. For example: `GoogleMaps`, `GooglePlaces`, `Firebase`, `MSAppCenter` and all subsets of those SDKs are linked statically.

When using 3rd party dependency manager like Cocoapods for linking one static library attached to more than one project (App or Framework) it would fail the installation with `target has transitive dependencies that include` **`static`** `binaries`. Therefore, it takes an extra effort to link static binaries into multiple frameworks.

Let's have a look how to link such static library into a dynamically linked SDK.

Exposing static 3rd party library

As mentioned above, it takes an extra effort to link a static library or static framework into a dynamically linked projects correctly. The crucial part is to make sure that it is linked only at one place. Either towards one dynamic framework where the static library can be exposed via umbrella file and then everywhere where the framework is linked the static library can be accessed through it as well. Or, only towards the app target from where it cannot be exposed anywhere else but via some level of abstraction it can be passed through to other frameworks on the code level. The same applies for static framework.

As an example of such umbrella file exposing GoogleMaps library that was linked to it could be:

```
1  // MyFramework.h – Umbrella file
2  #import <UIKit/UIKit.h>
3  #import "GoogleMaps/GoogleMaps.h"
```

The import of the header file of `GoogleMaps` into the frameworks umbrella file exposes all public headers of the GoogleMaps because of the `GoogleMaps.h` has all the GoogleMaps public headers.

```
1  // GoogleMaps.h
2  #import "GMSIndoorBuilding.h"
3  #import "GMSIndoorLevel.h"
4  #import "GMSAddress.h"
5  ...
```

The library becomes available as soon as the MyFramework import precedes the GoogleMaps one.

```
1  // MyFileInApp.swift
2  import MyFramework
3  import GoogleMaps
4  ...
```

In case of the static GoogleMaps framework, it is necessary to copy its bundle towards the app as there the GoogleMaps binary is looking for its resources like translations, images and so on.

Examining library

Let us have a look at some of the commands that comes in handy when solving some problems when it comes to compiler errors or receiving compiled closed source dynamic framework or a static library. To give it a quick start let's have a look at a binary we all know very well; UIKit. The path to the UIKit.framework is: `/Applications/Xcode.app/Contents/Developer/Platforms/iPhoneOS.platform/Library/Developer/CoreSimulator/Profiles/Runtimes/iOS.simruntime/Contents/Resources/RuntimeRoot/System/Library/Frameworks/UIKit.framework`

Apple ships various different tools for exploring compiled libraries and frameworks. On the UIKit framework I will demonstrate only essential commands that I find useful quite often.

Mach-O file format

Before we start, it is crucial to know what we are going to be exploring. In Apple ecosystem, file format of any binary is called Mach-O (Mach object). Mach-O has a pre-defined structure starting with Mach-O header following by segments, sections, load commands and so on.

Since you are surely a curious reader, you are now having tons of questions about where it all comes from. The answer to that is quite simple. Since it is all part of the system you can open up Xcode and look for a file in a global path `/usr/include/mach-o/loader.h`. In the `loader.h` file for example the Mach-O header struct is defined.

```
1  /*
2   * The 64-bit mach header appears at the very beginning of object files
         for
3   * 64-bit architectures.
```

```
 4   */
 5  struct mach_header_64 {
 6      uint32_t    magic;       /* mach magic number identifier */
 7      cpu_type_t  cputype;     /* cpu specifier */
 8      cpu_subtype_t   cpusubtype; /* machine specifier */
 9      uint32_t    filetype;    /* type of file */
10      uint32_t    ncmds;       /* number of load commands */
11      uint32_t    sizeofcmds;  /* the size of all the load commands */
12      uint32_t    flags;       /* flags */
13      uint32_t    reserved;    /* reserved */
14  };
```

When compiler produces final executable the Mach-O header is placed at a concrete byte position in it. Therefore, tools that are working with the executables knows exactly where to look for desired information. The same principle applies to all other parts of Mach-O as well.

//TODO: Redirect to an article or explain here?

For further exploration of Mach-O file, I would recommend reading the following article.

Fat headers

First, let's have a look on what Architectures the binary can be linked on (fat headers). For that, we are going to use `otool`;the utility that is shipped within every macOS. To list fat headers of a compiled binary we will use the flag `-f` and to produce a symbols readable output I also added the `-v` flag.

```
 1  otool -fv ./UIKit
```

Not surprisingly, the output produces two architectures. One that runs on the Intel mac (`x86_64`) when deploying to simulator and one that runs on iPhones so as on recently introduced M1 Mac (`arm64`).

```
 1  Fat headers
 2  fat_magic FAT_MAGIC
 3  nfat_arch 2
 4  architecture x86_64
 5      cputype CPU_TYPE_X86_64
 6      cpusubtype CPU_SUBTYPE_X86_64_ALL
 7      capabilities 0x0
 8      offset 4096
 9      size 26736
10      align 2^12 (4096)
11  architecture arm64
12      cputype CPU_TYPE_ARM64
13      cpusubtype CPU_SUBTYPE_ARM64_ALL
14      capabilities 0x0
15      offset 32768
```

```
16      size 51504
17      align 2^14 (16384)
```

When the command finishes successfully while not printing any output it simply means that the binary does not contain the fat header. That being said, the library can run only on one architecture and to see what the architecture is we have to print out the mach-o header of the executable.

```
1  otool -hv ./UIKit
```

From the output of the Mach-O header we can see that the `cputype` is `X86_64` so as some extra information like which `flags` the library was compiled with, `filetype` and so on.

```
1  Mach header
2      magic cputype cpusubtype  caps    filetype ncmds sizeofcmds
          flags
3  MH_MAGIC_64  X86_64        ALL  0x00        DYLIB    21       1400
    NOUNDEFS DYLDLINK TWOLEVEL APP_EXTENSION_SAFE
```

Executable type

Second, let us determine what type of a library we are dealing with. For that we will use again the `otool` as mentioned above. Mach-O header specifies `filetype`. So running it again on the UIKit.framework with the `−hv` flags produces the following output:

```
1  Mach header
2      magic cputype cpusubtype  caps    filetype ncmds sizeofcmds
          flags
3  MH_MAGIC_64  X86_64        ALL  0x00        DYLIB    21       1400
    NOUNDEFS DYLDLINK TWOLEVEL APP_EXTENSION_SAFE
```

From the output's `filetype` we can see that it is a dynamically linked library. From its extension we can say it is a dynamically linked framework. Like described before, framework can be dynamically or statically linked. The perfect example of statically linked framework is `GoogleMaps.framework`. When running the same command on the binary of `GoogleMaps` from the output we can see that the binary is NOT dynamically linked as its type is `OBJECT` aka object files which means that the library is static and linked to the attached executable at the compile time.

```
1  Mach header
2      magic cputype cpusubtype  caps    filetype ncmds sizeofcmds
          flags
3  MH_MAGIC_64  X86_64        ALL  0x00        OBJECT    4       2688
    SUBSECTIONS_VIA_SYMBOLS
```

The reason for wrapping the static library into a framework was the necessary include of `GoogleMaps` `.bundle` which needs to be copied to the target in order the library to work correctly with its re-

sources.

Now, let's try to run the same command on the static library archive. As an example we can use again one of the Xcode's libraries located at /Applications/Xcode.app/Contents /Developer/Toolchains/XcodeDefault.xctoolchain/usr/lib/swift/iphoneos/ libswiftCompatibility50.a path. From the library extension we can immediately say the library is static. Running the otool -hv libswiftCompatibility50.a just confirms that the filetype is OBJECT.

```
1  Archive : ./libswiftCompatibility50.a (architecture armv7)
2  Mach header
3      magic cputype cpusubtype  caps    filetype ncmds sizeofcmds
          flags
4    MH_MAGIC     ARM       V7  0x00      OBJECT    4        588
        SUBSECTIONS_VIA_SYMBOLS
5  Mach header
6      magic cputype cpusubtype  caps    filetype ncmds sizeofcmds
          flags
7    MH_MAGIC     ARM       V7  0x00      OBJECT    5        736
        SUBSECTIONS_VIA_SYMBOLS
8  ...
```

While static library archive ending with .a is clearly static one with a framework to be really sure that the library is dynamically linked it is necessary to check the binary for its filetype in the Mach-O header.

Dependencies

Third, let's have a look at what the library is linking. For that the otool provides -l flag.

```
1  otool -L ./UIKit
```

The output lists all dependencies of UIKit framework. For example, here you can see that UIKit is linking Foundation. That's why the **import** Foundation is no longer needed when importing UIKit into a source code file.

```
1  ./UIKit:
2      /System/Library/Frameworks/UIKit.framework/UIKit (compatibility
          version 1.0.0, current version 3987.0.109)
3      /System/Library/Frameworks/FileProvider.framework/FileProvider (
          compatibility version 1.0.0, current version 1.0.0, reexport)
4    /System/Library/Frameworks/Foundation.framework/Foundation (
        compatibility version 300.0.0, current version 1751.108.0)
5    /System/Library/PrivateFrameworks/DocumentManager.framework/
        DocumentManager (compatibility version 1.0.0, current version
        200.0.0, reexport)
```

```
6        /System/Library/PrivateFrameworks/UIKitCore.framework/UIKitCore (
             compatibility version 1.0.0, current version 3987.0.109,
             reexport)
7        /System/Library/PrivateFrameworks/ShareSheet.framework/ShareSheet (
             compatibility version 1.0.0, current version 1564.6.0, reexport)
8        /usr/lib/libobjc.A.dylib (compatibility version 1.0.0, current
             version 228.0.0)
9        /usr/lib/libSystem.B.dylib (compatibility version 1.0.0, current
             version 1292.0.0)
```

Symbols table

Fourth, it is also useful to know what are the symbols that are defined in the framework. For that the nm utility is available. To print all symbols including the debugging ones I added -a flag so as -C to print them demangled. The name mangling is a technique of adding extra information about the language data type (class, struct, enum …) to the symbol during compile time in order to pass more information about it to the linker. With mangled symbol the linker will know that this symbols is for a class, getter, setter etc and can work with it accordingly.

```
1   nm -Ca ./UIKit
```

Unfortunately, the output here is very limited as those symbols listed are the ones that defines the dynamic framework itself. The limitation is because of Apple ships the binary obfuscated and when reverse engineering the binary with for example Radare2 disassembler, all we can see is couple of add byte assembly instructions. It is still possible to dump the list of symbols but for that we would have to either use lldb and have the UIKit framework loaded in the memory space or dump the memory footprint of the framework and explore it decrypted. That is unfortunately not part of this book.

```
1   0000000000000ff0 s _UIKitVersionNumber
2   0000000000000fc0 s _UIKitVersionString
3                    U dyld_stub_binder
```

Just to give an example how the symbols would look like I printed out compiled realm framework by running nm -Ca ./Realm.

```
1   ...
2   00000000002c4650 T realm::Table::do_move_row(unsigned long, unsigned
        long)
3   00000000002cb1e8 T realm::Table::do_set_link(unsigned long, unsigned
        long, unsigned long)
4   00000000004305e0 S realm::Table::max_integer
5   00000000004305e8 S realm::Table::min_integer
6   00000000002c44b4 T realm::Table::do_swap_rows(unsigned long, unsigned
        long)
7   00000000002ce9bc T realm::Table::find_all_int(unsigned long, long long)
```

```
 8  00000000002cb3ac T realm::Table::get_linklist(unsigned long, unsigned
        long)
 9  00000000002c4d64 T realm::Table::set_subtable(unsigned long, unsigned
        long, realm::Table const*)
10  00000000002ba4f0 T realm::Table::add_subcolumn(std::__1::vector<
        unsigned long, std::__1::allocator<unsigned long> > const&, realm::
        DataType, realm::StringData)
11  00000000002bd9f8 T realm::Table::create_column(realm::ColumnType,
        unsigned long, bool, realm::Allocator&)
12  00000000002bf3fc T realm::Table::discard_views()
13  ...
```

It seams like Realm was developed in C++ but it can be clearly seen what kind of symbols are available within the binary. One more example for Swift with Alamofire. There we can unfortunately see that the nm was not able to demangle the symbols.

```
 1  ...
 2  0000000000034d00 T _$s9Alamofire7RequestC8delegateAA12TaskDelegateCvM
 3  0000000000034dc0 T _$s9Alamofire7RequestC4taskSo16NSURLSessionTaskCSgvg
 4  0000000000034e20 T _$s9Alamofire7RequestC7sessionSo12NSURLSessionCvg
 5  0000000000034e50 T
        _$s9Alamofire7RequestC7request10Foundation10URLRequestVSgvg
 6  00000000000350c0 T
        _$s9Alamofire7RequestC8responseSo17NSHTTPURLResponseCSgvg
 7  00000000000351e0 T _$s9Alamofire7RequestC10retryCountSuvpfi
 8  ...
```

To demangle swift manually following command can be used.

```
 1  nm -a ./Alamofire | awk '{ print $3 }' | xargs swift demangle {} \;
```

Which produces the mangled symbol name with the demangled explanation.

```
 1  ...
 2  _$s9Alamofire7RequestC4taskSo16NSURLSessionTaskCSgvg ---> Alamofire.
        Request.task.getter : __C.NSURLSessionTask?
 3  _$s9Alamofire7RequestC4taskSo16NSURLSessionTaskCSgvgTq ---> method
        descriptor for Alamofire.Request.task.getter : __C.NSURLSessionTask?
 4  _$s9Alamofire7RequestC10retryCountSuvpfi ---> variable initialization
        expression of Alamofire.Request.retryCount : Swift.UInt
 5  ...
```

Strings

Last but not least, it can be also helpful to list all strings that the binary contains. That could help catch developers mistakes like not obfuscated secrets and some other strings that should not be part of the binary. To do that we will use strings utility again on the Alamofire binary.

```
1  strings ./Alamofire
```

The output is a list of plain text strings found in the binary.

```
1  ...
2  Could not fetch the file size from the provided URL:
3  The URL provided is a directory:
4  The system returned an error while checking the provided URL for
5  reachability.
6  URL:
7  The URL provided is not reachable:
8  ...
```

Build system

Last piece of information that is missing now is; how it all gets glued together. As Apple developers, we are using Xcode for developing apps for Apple products that are then distributed via App Store or other distribution channels. Xcode under the hood is using Xcode Build System for producing final executables that runs on X86 and ARM processor architectures.

The Xcode build system consists of multiple steps that depends on each other. Xcode build system supports C based languages (C, C++, Objective-C, Objective-C++) compiled with clang so as Swift language compiled with swiftc.

Let's have a quick look at what Xcode does when the build is triggered.

1. **Preprocessing**

Preprocessing resolves macros, removes comments, imports files and so on. In a nutshell it prepares the code for the compiler. Preprocessor also decides which compiler will be used for which source code file. Not surprisingly, swift source code file will be compiled by swiftc and other C like files will use clang.

2. **Compiler** (swiftc, clang)

As mentioned above, Xcode build system uses two compilers; clang and swiftc. Compiler consists of two parts, front-end and back-end. Both compilers are using the same back-end, LLVM (Low Level Virtual Machine) and language specific front-end. The job of a compiler is to compile the preprocessed source code files into object files which contains object code. Object code is simply human readable assembly instructions that can be understood by the CPU.

3. **Assembler** (asm)

Assembler takes the output of the compiler (assembly) and produces relocatable machine code. Machine code is recognised by a concrete type of a processor (ARM, X86). The opposite of relocatable machine code would be absolute machine code. While relocatable code can be placed at any position in memory by loader the absolute machine code has its position set in the binary.

4. **Linker** (`ld`)

Final step of the build system is linking. Linker is a program that takes object files (multiple compiled files) and links (merges) them together based on the symbols those files are using so as links static and dynamic libraries if needed. In order to be able to link libraries linker needs to know the paths where to look for them. Linker produces final single file; Mach-O executable.

5. **Loader** (`loader`)

After the executable was built the job of a loader is to bring the executable into memory and start the program execution. Loader is a system program operating on the kernel level. Loader assigns the memory space and loads Mach-O executable to it.

Now you should have a high level overview of what phases Xcode build system goes through when the build is started.

Conclusion

I hope this chapter gave the essentials of what is the difference in between static and dynamic library so as some examples of how to examine them. It was quite a lot to grasp so now it's time for a double shot of espresso or any kind of preferable refreshment.

I would highly recommend to deep dive into this topic even more. Here are some resources I would recommend;

How does the executable structure looks like:

https://medium.com/@cyrilcermak/exploring-ios-es-mach-o-executable-structure-aa5d8d1c7103

Static and dynamic libraries and manual compile examples:

https://pewpewthespells.com/blog/static_and_dynamic_libraries.html

Difference in between static and dynamic library from our beloved StackOverflow:

https://stackoverflow.com/questions/15331056/library-static-dynamic-or-framework-project-inside-another-project

The official Apple documentation about dynamic libraries:

https://developer.apple.com/library/archive/documentation/DeveloperTools/Conceptual/DynamicLibraries/000-Introduction/Introduction.html#//apple_ref/doc/uid/TP40001908-SW1

To know more about the Xcode build system: https://medium.com/flawless-app-stories/xcode-build-system-know-it-better-96936e7f52a

https://www.objc.io/issues/6-build-tools/mach-o-executables/

https://llvm.org/

https://developer.apple.com/videos/play/wwdc2018/415/

https://developer.apple.com/videos/play/wwdc2018/408/

Used binaries:

GoogleMaps: https://developers.google.com/maps/documentation/ios-sdk/v3-client-migration#install-manually

Alamofire: https://github.com/Alamofire/Alamofire

Realm: https://realm.io/docs/swift/latest

Swift Compiler (optional)

Since we touched the Xcode's build system in the previous chapter it would be totally unfair to the `swiftc` to leave it untouched. Even though knowing how compiler works is not a mandatory knowledge it is really interesting and it gives a good closure of the whole process from writing human readable code till running it on a bare metal.

While other chapters are rather essential for having good understanding for the development of the modular architecture, this chapter is optional.

Compiler Architecture

To fully understand the swift's compiler architecture and it's process let us have a look at the documentation provided by swift.org and do some practical examples out of it.

The following image describes the swiftc architecture. It consists of 7 steps, which are explained in subchapters.
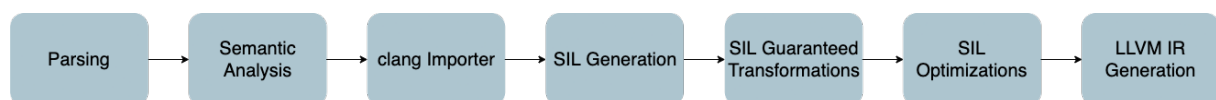
Figure 5: Swiftc Architecture

For the demonstration purposes, I prepared two simple swift source code files. First, Employee.swift and second main.swift. Employee.swift is a standalone source code while main.swift requires the Employee being linked to it as a library. All compiler steps are explained on the Employee.swift but in the end the employee source code will be created as a library that the main file will consume and use.

`Employee.swift`

```swift
1   import Foundation
2
3   public protocol Address {
4       var houseNo: Int { get }
5       var street: String { get }
6       var city: String { get }
7       var state: String { get }
8   }
9
10  public protocol Person {
11      var firstName: String { get }
12      var lastName: String { get }
13      var address: Address { get }
14  }
```

```swift
15
16  public class Employee: Person {
17      public let firstName: String
18      public let lastName: String
19      public let address: Address
20
21      public init(firstName: String, lastName: String, address: Address)
            {
22          self.firstName = firstName
23          self.lastName = lastName
24          self.address = address
25      }
26
27      public func printEmployeeInfo() {
28          print("\(firstName) \(lastName)")
29          print("\(address.houseNo). \(address.street), \(address.city),
                \(address.state)")
30      }
31  }
32
33  public struct EmployeeAddress: Address {
34      public let houseNo: Int
35      public let street: String
36      public let city: String
37      public let state: String
38
39      public init(houseNo: Int, street: String, city: String, state:
            String) {
40          self.houseNo = houseNo
41          self.street = street
42          self.city = city
43          self.state = state
44      }
45  }
```

`main.swift`

```swift
1  import Foundation
2  import Employee
3
4  let employee = Employee(firstName: "Cyril",
5                          lastName: "Cermak",
6                          address: EmployeeAddress(houseNo: 1, street: "
                              PorschePlatz", city: "Stuttgart", state: "
                              Germany"))
7
8  employee.printEmployeeInfo()
```

Parsing

> The parser is a simple, recursive-descent parser (implemented in lib/Parse) with an integrated, hand-coded lexer. The parser is responsible for generating an Abstract Syntax Tree (AST) without any semantic or type information, and emit warnings or errors for grammatical problems with the input source.

Source: swift.org

First of all comes the parsing. As the definition says, parser is responsible for lexical syntax check without any type check. The following command prints the parsed AST.

```
1  swiftc ./employee.swift -dump-parse
```

In the output you can notice that the types are not resolved and are resolved with errors.

```
1  (source_file "./employee.swift"
2  // Importing Foundation
3    (import_decl range=[./employee.swift:1:1 - line:1:8] 'Foundation')
4  // Address protocol declaration
5    (protocol range=[./employee.swift:3:8 - line:8:1] "Address" <Self :
         Address> requirement signature=<null>
6      (pattern_binding_decl range=[./employee.swift:4:5 - line:4:28]
7        (pattern_typed
8          (pattern_named 'houseNo')
9          (type_ident
10           (component id='Int' bind=none))))
11 // houseNo variable declaration
12      (var_decl range=[./employee.swift:4:9 - line:4:9] "houseNo" type='<
           null type>' readImpl=getter immutable
13        (accessor_decl range=[./employee.swift:4:24 - line:4:24] '
             anonname=0x7fc1e408db80' get_for=houseNo
14 // Not recognized Int type
15          (parameter "self"./employee.swift:4:18: error: cannot find type
               'Int' in scope
16      var houseNo: Int { get }
17                   ^~~
18 )
19 ...
20 // Employee class declaration
21 (class_decl range=[./employee.swift:16:8 - line:31:1] "Employee"
       inherits: <null>
22      (pattern_binding_decl range=[./employee.swift:17:12 - line:17:27]
23        (pattern_typed
24          (pattern_named 'firstName')
25          (type_ident
26           (component id='String' bind=none))))
27 // Employee class vars declaration
28      (var_decl range=[./employee.swift:17:16 - line:17:16] "firstName"
           type='<null type>' let readImpl=stored immutable)
```

```
29        (pattern_binding_decl range=[./employee.swift:18:12 - line:18:26]
30          (pattern_typed
31            (pattern_named 'lastName')
32            (type_ident
33              (component id='String' bind=none))))
34  ...
```

From the parsed AST we can see that it is really descriptive. The source code of Employee.swift has 47 lines of code while its parsed AST without type check has 270.

Out of curiosity, let us have a look at how the tree would look like with syntax error. In order to do so, I added the winner of all times in hide and seek ; to the protocol declaration.

```
1  public protocol Address {
2      var houseNo: Int; { get }
```

After running the same command we can see a syntax error at the declaration of houseNo variable. That is the error Xcode would show as soon as it type checks the source file.

```
1  ...
2  (var_decl range=[./employee.swift:4:9 - line:4:9] "houseNo" type='<null
       type>'./employee.swift:4:9: error: property in protocol must have
     explicit { get } or { get set } specifier
3      var houseNo: Int; { get }
4  ...
```

Semantic analysis

> Semantic analysis (implemented in lib/Sema) is responsible for taking the parsed AST and trans-
> forming it into a well-formed, fully-type-checked form of the AST, emitting warnings or errors
> for semantic problems in the source code. Semantic analysis includes type inference and, on
> success, indicates that it is safe to generate code from the resulting, type-checked AST.

Source: swift.org

Second comes the semantic analysis, from the definition of it, we should see fully type checked parsed
AST. Executing the following command will give us the answer.

```
1  swiftc ./employee.swift -dump-ast
```

In the output all types are resolved and recognised by the compiler, errors no longer appear.

```
1  // Address protocol with resolved types
2  (protocol range=[./employee.swift:3:8 - line:8:1] "Address" <Self :
       Address> interface type='Address.Protocol' access=public non-
       resilient requirement signature=<Self>
3    (pattern_binding_decl range=[./employee.swift:4:5 - line:4:18]
         trailing_semi
4      (pattern_typed type='Int'
5        (pattern_named type='Int' 'houseNo')
6        (type_ident
7          (component id='Int' bind=Swift.(file).Int))))
8  ...
9  // EmployeeAddress conforming to the Address protocol
10 (struct_decl range=[./employee.swift:33:8 - line:45:1] "EmployeeAddress
       " interface type='EmployeeAddress.Type' access=public non-resilient
       inherits: Address
11   (pattern_binding_decl range=[./employee.swift:34:12 - line:34:25]
12     (pattern_typed type='Int'
13       (pattern_named type='Int' 'houseNo')
14       (type_ident
15         (component id='Int' bind=Swift.(file).Int))))
16 // Variable declaration on EmployeeAddress
17   (var_decl range=[./employee.swift:34:16 - line:34:16] "houseNo" type
       ='Int' interface type='Int' access=public let readImpl=stored
       immutable
18     (accessor_decl implicit range=[./employee.swift:34:16 - line
         :34:16] 'anonname=0x7fd2821409e8' interface type='(
         EmployeeAddress) -> () -> Int' access=public get_for=houseNo
19       (parameter "self" type='EmployeeAddress' interface type='
           EmployeeAddress')
20       (parameter_list)
21       (brace_stmt implicit range=[./employee.swift:34:16 - line:34:16]
22         (return_stmt implicit
23           (member_ref_expr implicit type='Int' decl=employee.(file).
```

```
                    EmployeeAddress.houseNo@./employee.swift:34:16
                    direct_to_storage
24                  (declref_expr implicit type='EmployeeAddress' decl=
                       employee.(file).EmployeeAddress.<anonymous>.self@./
                       employee.swift:34:16 function_ref=unapplied))))))
```

Not surprisingly, when modifying a type for some unknown one the command results in error.

```
1  public protocol Address {
2      var houseNo: Foo { get }
```

```
1  /employee.swift:4:18: error: cannot find type 'Foo' in scope
2      var houseNo: Foo { get }
3                   ^~~
4  (source_file "./employee.swift"
5    (import_decl range=[./employee.swift:1:1 - line:1:8] 'Foundation')
6    (protocol range=[./employee.swift:3:8 - line:8:1] "Address" <Self :
        Address> interface type='Address.Protocol' access=public non-
        resilient requirement signature=<Self>
7      (pattern_binding_decl range=[./employee.swift:4:5 - line:4:28]
8        (pattern_typed type='<<error type>>'
```

Clang importer

> The Clang importer (implemented in lib/ClangImporter) imports Clang modules and maps the C or Objective-C APIs they export into their corresponding Swift APIs. The resulting imported ASTs can be referred to by semantic analysis.

Source: swift.org

Third is the clang importer. This is the well-known bridging of C/ObjC languages to the Swift API's and wise versa.

SIL generation

> The Swift Intermediate Language (SIL) is a high-level, Swift-specific intermediate language suitable for further analysis and optimization of Swift code. The SIL generation phase (implemented in lib/SILGen) lowers the type-checked AST into so-called "raw" SIL. The design of SIL is described in docs/SIL.rst.

Source: swift.org

Fourth comes the Swift Intermediate Language. Are you curious about how it looks like? In order to print it we can use the following command.

```
1  swiftc ./employee.swift -emit-sil
```

In the output, we can see the `witness tables`, `vtables` and `message dispatch` tables along side with other intermediate declarations. Unfortunately, explanation of this is out of the scope of this book. More about these topics can be obtained here.

```
1   ...
2   // protocol witness for Address.state.getter in conformance
        EmployeeAddress
3   sil shared [transparent] [serialized] [thunk]
        @$s8employee15EmployeeAddressVAA0C0A2aDP5stateSSvgTW : $@convention(
        witness_method: Address) (@in_guaranteed EmployeeAddress) -> @owned
        String {
4   // %0                                           // user: %1
5   bb0(%0 : $*EmployeeAddress):
6     %1 = load %0 : $*EmployeeAddress              // user: %3
7     // function_ref EmployeeAddress.state.getter
8     %2 = function_ref @$s8employee15EmployeeAddressV5stateSSvg :
          $@convention(method) (@guaranteed EmployeeAddress) -> @owned
          String // user: %3
9     %3 = apply %2(%1) : $@convention(method) (@guaranteed EmployeeAddress
          ) -> @owned String // user: %4
10    return %3 : $String                           // id: %4
11  } // end sil function '
        $s8employee15EmployeeAddressVAA0C0A2aDP5stateSSvgTW'
12
13  sil_vtable [serialized] Employee {
14    #Employee.init!allocator: (Employee.Type) -> (String, String, Address
          ) -> Employee :
          @$s8employee8EmployeeC9firstName04lastD07addressACSS_SSAA7Address_ptcfC
            // Employee.__allocating_init(firstName:lastName:address:)
15    #Employee.printEmployeeInfo: (Employee) -> () -> () :
          @$s8employee8EmployeeC05printB4InfoyyF  // Employee.
          printEmployeeInfo()
16    #Employee.deinit!deallocator: @$s8employee8EmployeeCfD   // Employee
          .__deallocating_deinit
17  }
18
19  sil_witness_table [serialized] Employee: Person module employee {
20    method #Person.firstName!getter: <Self where Self : Person> (Self) ->
          () -> String :
          @$s8employee8EmployeeCAA6PersonA2aDP9firstNameSSvgTW // protocol
          witness for Person.firstName.getter in conformance Employee
21    method #Person.lastName!getter: <Self where Self : Person> (Self) ->
          () -> String : @$s8employee8EmployeeCAA6PersonA2aDP8lastNameSSvgTW
            // protocol witness for Person.lastName.getter in conformance
          Employee
22    method #Person.address!getter: <Self where Self : Person> (Self) ->
          () -> Address :
          @$s8employee8EmployeeCAA6PersonA2aDP7addressAA7Address_pvgTW  //
```

```
             protocol witness for Person.address.getter in conformance Employee
23  }
24
25  sil_witness_table [serialized] EmployeeAddress: Address module employee
        {
26    method #Address.houseNo!getter: <Self where Self : Address> (Self) ->
          () -> Int :
          @$s8employee15EmployeeAddressVAA0C0A2aDP7houseNoSivgTW  //
          protocol witness for Address.houseNo.getter in conformance
          EmployeeAddress
27    method #Address.street!getter: <Self where Self : Address> (Self) ->
          () -> String :
          @$s8employee15EmployeeAddressVAA0C0A2aDP6streetSSvgTW // protocol
          witness for Address.street.getter in conformance EmployeeAddress
28    method #Address.city!getter: <Self where Self : Address> (Self) -> ()
           -> String : @$s8employee15EmployeeAddressVAA0C0A2aDP4citySSvgTW
          // protocol witness for Address.city.getter in conformance
          EmployeeAddress
29    method #Address.state!getter: <Self where Self : Address> (Self) ->
          () -> String :
          @$s8employee15EmployeeAddressVAA0C0A2aDP5stateSSvgTW   // protocol
           witness for Address.state.getter in conformance EmployeeAddress
30  }
31  ...
```

Furthermore the SIL gets through next two phases; guaranteed transformation and optimisation.

> SIL guaranteed transformations: The SIL guaranteed transformations (implemented in lib/SILOptimizer/Mandatory) perform additional dataflow diagnostics that affect the correctness of a program (such as a use of uninitialized variables). The end result of these transformations is "canonical" SIL.

Source: swift.org

> SIL Optimizations: The SIL optimizations (implemented in lib/Analysis, lib/ARC, lib/LoopTransforms, and lib/Transforms) perform additional high-level, Swift-specific optimizations to the program, including (for example) Automatic Reference Counting optimizations, devirtualization, and generic specialization.

Source: swift.org

LLVM IR (Intermediate Representation) Generation

> IR generation (implemented in lib/IRGen) lowers SIL to LLVM IR, at which point LLVM can continue to optimize it and generate machine code.

Source: swift.org

Last comes the IR for LLVM. To get the IR from the swiftc we can use the following command:

```
1  swiftc ./employee.swift -emit-ir | more
```

Here we can see a snippet of the LLVM's familiar declaration which in the next step would be transformed by it into machine code.

```
1  ...
2  entry:
3    %1 = getelementptr inbounds %__opaque_existential_type_1, %
         __opaque_existential_type_1* %0, i32 0, i32 1
4    %2 = load %swift.type*, %swift.type** %1, align 8
5    %3 = getelementptr inbounds %__opaque_existential_type_1, %
         __opaque_existential_type_1* %0, i32 0, i32 0
6    %4 = bitcast %swift.type* %2 to i8***
7    %5 = getelementptr inbounds i8**, i8*** %4, i64 -1
8    %.valueWitnesses = load i8**, i8*** %5, align 8, !invariant.load
         !64, !dereferenceable !65
9    %6 = bitcast i8** %.valueWitnesses to %swift.vwtable*
10   %7 = getelementptr inbounds %swift.vwtable, %swift.vwtable* %6, i32
         0, i32 10
11   %flags = load i32, i32* %7, align 8, !invariant.load !64
12   %8 = and i32 %flags, 131072
13   %flags.isInline = icmp eq i32 %8, 0
14   br i1 %flags.isInline, label %inline, label %outline
15 ...
```

Exporting dylib

Finally, we can explore how to manually create a library out of the source code and link it towards the executable.

The following command will export the employee.swift file as an Employee.dylib with it's module definition. Instead of using `-emit-module` we could use `-emit-object` parameter to get a statically linked library.

```
1  swiftc ./employee.swift -emit-library -emit-module -parse-as-library -
       module-name Employee
```

After executing the command, following files should be created.

```
1  380B Apr  2 13:36 Employee.swiftdoc
2   19K Apr  3 20:52 Employee.swiftmodule
3  2.9K Apr  3 20:52 Employee.swiftsourceinfo
4  1.1K Apr  3 20:52 employee.swift
5   57K Apr  3 20:52 libEmployee.dylib
```

Now we can import the Employee library into the main.swift file and proceed with the compile. However, here we have to tell the compiler and linker where to find the Employee library. In this example, I placed the library into Frameworks directory which is on the same level as the main.swift.

```
1  swiftc main.swift -emit-executable -lEmployee -I ./Frameworks -L ./
       Frameworks
```

To give it a bit more explanation the swiftc -h prints:

```
1  ...
2  -emit-executable       Emit a linked executable
3  -I <value>             Add directory to the import search path
4  -L <value>             Add directory to library link search path
5  -l<value>              Specifies a library which should be linked
       against
6  ...
```

Hurrray, the executable was created with the linked library! Unfortunately, it crashes right on start with;

```
1  dyld: Library not loaded: libEmployee.dylib
2    Referenced from: /Users/cyrilcermak/Programming/iOS/
       modular_architecture_on_ios/example/./main
3    Reason: image not found
4  [1]    92481 abort      ./main
```

Using the knowledge from the previous chapter we can check where the binary expects the library to be with `otool -l ./main`.

```
1  Load command 15
2           cmd LC_LOAD_DYLIB
3       cmdsize 48
4          name libEmployee.dylib (offset 24)
5    time stamp 2 Thu Jan  1 01:00:02 1970
6       current version 0.0.0
7 compatibility version 0.0.0
```

The binary expects the libEmployee.dylib to be at the same path. This can be easily fixed with one more tool, install_name. It changes the path to the linked library in the main executable. It can be used as follows;

```
1  install_name_tool -change libEmployee.dylib @executable_path/Frameworks
       /libEmployee.dylib main
```

Running it again prints the desired output:

```
1  Cyril Cermak
2  1. PorschePlatz, Stuttgart, Germany
```

Conclusion

In this chapter the basics of swift compiler architecture were explored. I hope this (optional) chapter gave a high level overview and brought more curiosity into the topic of how compilers work. For more study I would refer to the following sources:

Swift Compiler

Understanding Swift Performance

Understanding method dispatch in Swift

Method Dispatch in Swift

Getting Started with Swift Compiler Development

Development of the modular architecture

The necessary theory about Apple's libraries and some essentials were explained. Finally, it is time to deep dive into the building phase.

First, let us do it manually and automate the process of creating libraries later on so that the new comers do not have to copy-paste much of the boilerplate code when starting a new team or new part of the framework.

For the demonstration purposes, I chose the Cosmonaut app with all its necessary dependencies. Nevertheless, the same principle applies for all other apps within our future iOS/macOS ISS foundational framework.

You can download the pre-build repository here and fully focus on the step by step explanations in the book or you can build it on your own up until certain point.

As a reminder the following schema showcases the Cosmonaut app with its dependencies.
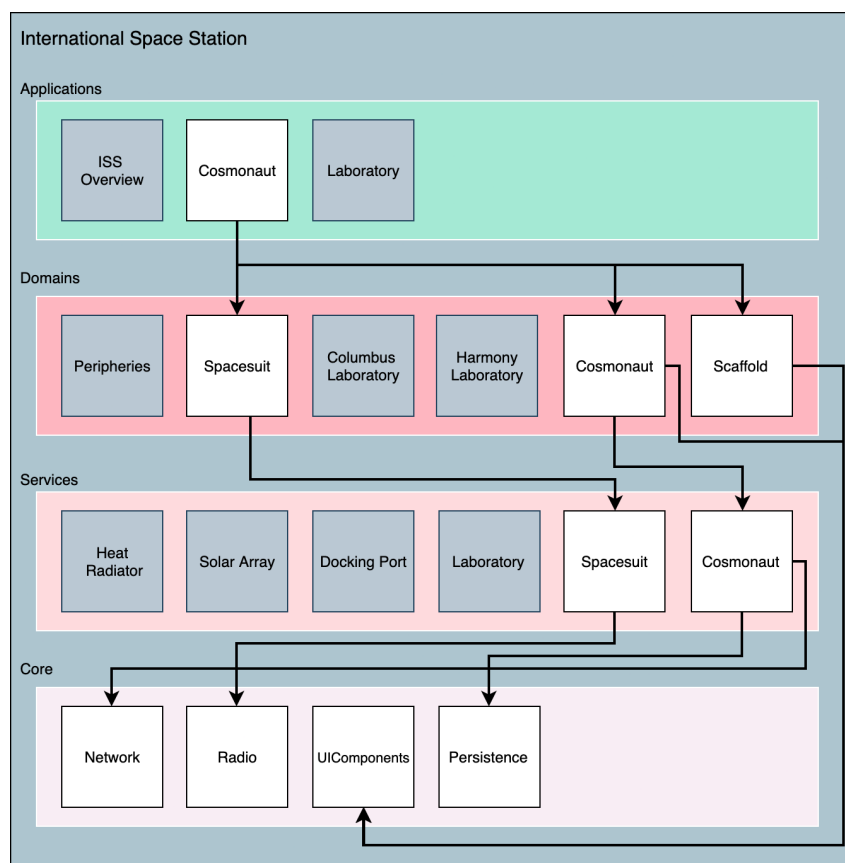
Figure 6: Cosmonaut App

Creating workspace structure

First let us manually create the Cosmonaut app from Xcode under the `iss_application_framework` `/app/` directory. To achieve that, simply create a new App from the Xcode's menu and save it under the predefined folder path with the `Cosmonaut` name. An empty project app should be created, you can run it if you want. Nevertheless, for our purposes the project structure is not optimal. We will be working in a workspace which will contain multiple projects(apps and frameworks).
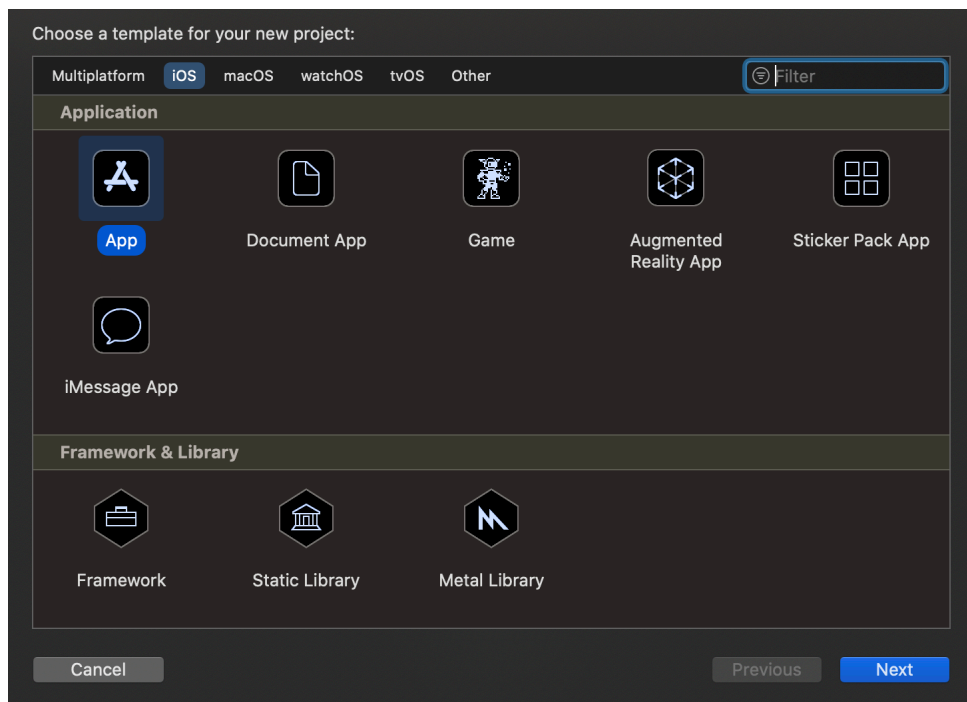
Figure 7: Create New App

Since we do not have `Cocopods` yet which would convert the project to the workspace we have to do it manually. In Xcode under `File` select the option `Save As Workspace`, close the project and open the newly created Workspace by Xcode. So far the workspace contains only the App. Now it is time to create the necessary dependencies for the Cosmonaut app.

Going top down through the diagram, first comes the `Domain` layer where `Spacesuit`, `Cosmonaut` and `Scaffold` is needed to be created. For creating the `Spacesuit` let us use Xcode one last time. Under the new project select the framework icon, name it `Cosmonaut` and save it under the `iss_application_framework`/`domain`/ directory.
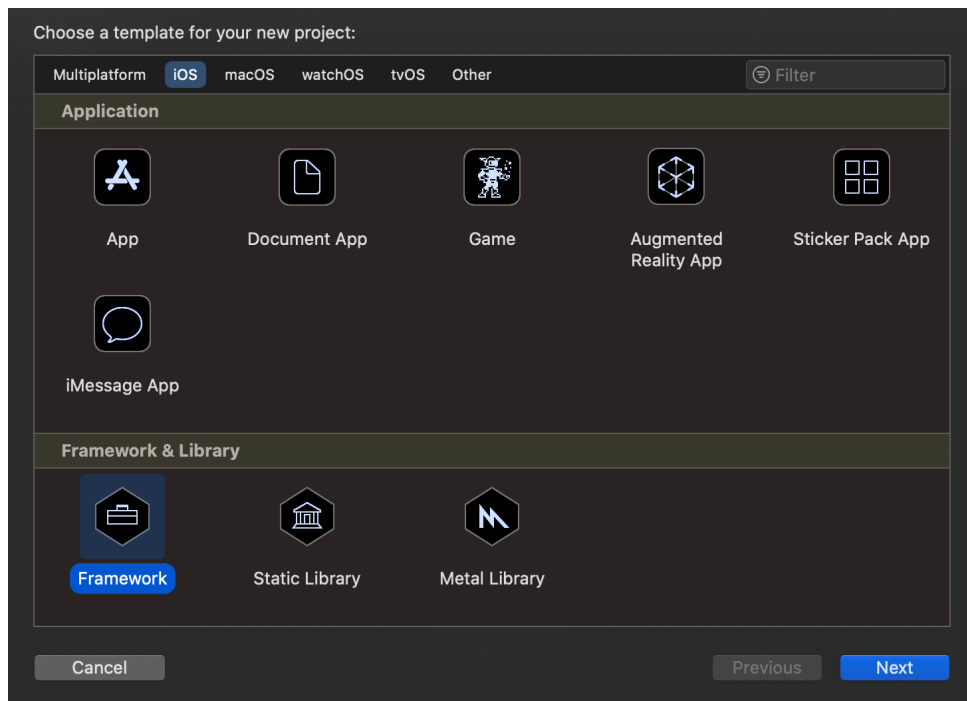
Figure 8: Create New Framework

Automating the process

While creating new frameworks and apps is not a daily business the process still needs to assure that correct namespaces and conventions are used across the whole application framework. This usually leads to a copy-pasting already created framework or app in order to create a new one with the same patterns. Now is good time to create first script that will support the development of the application framework.

If you are building the application framework from scratch please copy the `{PROJECT_ROOT}`/ `fastlane` directory from the repository into your `root` directory.

The scripting around the framework with `Fastlane` is explained later on in the book. However, all you need to know now is that Fastlane contains lane `make_new_project` that takes three arguments; `type` {app|framework}, `project_name` and `destination_path`. The lane in Fastlane simple uses the instance of the `ProjectFactory` class located in the `{PROJECT_ROOT}`/`fastlane`/`scripts`/ `ProjectFactory`/`project_factory`.`rb` file.

The `ProjectFactory` creates new framework or app based on the `type` parameter that is passed to it from the command line. As an example of creating the Spacesuit domain framework the following command can be used.

```
1  fastlane make_new_project type:framework project_name:Spacesuit
       destination_path:../domain/Spacesuit
```

In case of Fastlane not being installed on your mac you can install it via `brew install fastlane` or later on via Ruby `gems` defined in `Gemfile`. For installation please follow the official manual.

Furthermore, we can continue creating all dependencies via the script up until the point where all dependencies were created is reached.

The overall ISS Application Framework should look as follows:


```
→  iss_modular_architecture git:(master) ✗ tree -L 2
.
├── app
│   ├── Cosmonaut
│   ├── Laboratory
│   └── Overview
├── core
│   ├── Network
│   ├── Persistence
│   ├── Radio
│   └── UIComponents
├── domain
│   ├── Columbus\ Laboratory
│   ├── Cosmonaut
│   ├── Harmony\ Laboratory
│   ├── Peripheries
│   ├── Scaffold
│   └── Spacesuit
├── fastlane
│   ├── Fastfile
│   ├── README.md
│   ├── build_settings.yml
│   ├── report.xml
│   └── scripts
└── service
    ├── CosmonautService
    ├── Docking\ Port
    ├── Heat\ Radiator
    ├── Laboratory
    ├── Solar\ Array
    └── SpacesuitService

25 directories, 4 files
```

Figure 9: Tree structre

Each directory contains Xcode project which is either a framework or an app created by the script. From now on, every onboarded team or developer should use the script to create a framework or an app that will be developed.

Xcode's workspace

Last but not least, let us create the same directory structure in the Xcode's Workspace so that we can later on link those frameworks together and towards the app. In the Cosmonaut app our `Cosmonaut.xcworkspace` resides. An `xcworkspace` is simply a structure that contains; - `xcshareddata`: Directory that contains schemes, breakpoints and other shared information - `xcuserdata`: Directory that contains information about the current users interface state, opened/modified files of the user and so on - `contents.xcworkspacedata`: An XML file that describes what projects are linked towards the workspace such that Xcode can understand it

The workspace structure can be created either by drag and dropping all necessary framework projects for the `Cosmonaut` app or by directly modifying the `contents.xcworkspacedata` XML file. No matter which way was chosen the final `xcworkspace` should look as follow:
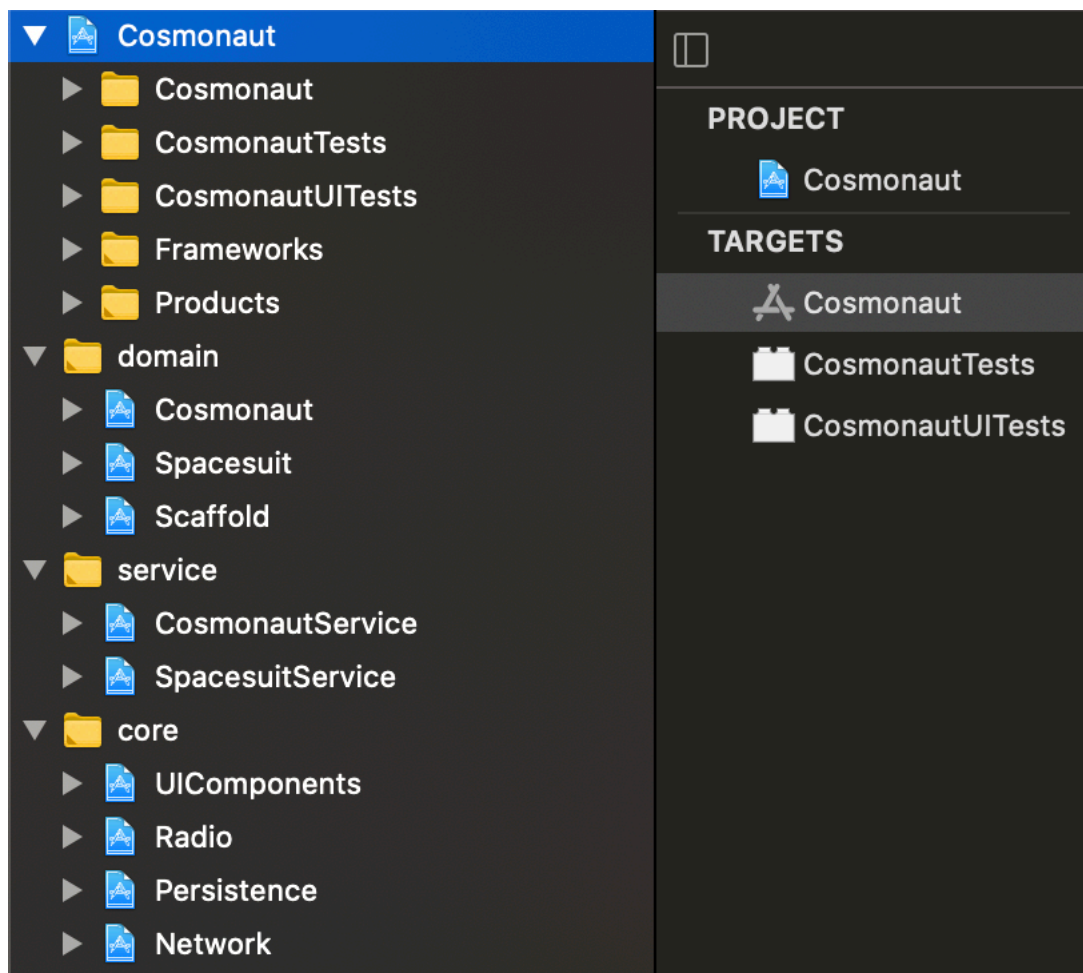
Figure 10: Workspace strucutre

Generating projects

You might have noticed `project.yml` file that was created with every framework or app. This file is used by `XcodeGen` (will be introduced in a second) to generate the project based on the settings described in the yaml file. This will avoid conflicts in the Apple's infamous `project.pbxproj` files that are representing each project. In the modular architecture this is particularly useful as we are working with many projects across the workspace.

Conflicts in the `project.pbxproj` files are very common when more than one developer are working on the same codebase. Besides the build settings for the project, the file also contains and tracks files that are included for the compilation so as which target they belongs to. A typical conflict happens when one developer removes a file from the Xcode's structure while another developer was modifying it in a separate branch. This will resolve in a merge conflict in the `pbxproj` file which is very time consuming to fix as the file is using Apple's mystified language no one can understand.

Since programmers are lazy creatures, very often also happens that the file that was removed from the Xcode's project still remains in the repository as it was not moved to the trash. That could lead to a git tracking of those unused files inside of the repository so as re-adding the deleted file to the project by the developer who was modifying it.

Hello XcodeGen

Fortunately, in the Apple ecosystem we can use xcodegen, a program that generates the `pbxproj` file for us based on the well-arranged yaml file. In order to use it we have to first install it via `brew install xcodegen` or via other ways described on its homepage.

As an example let us have a look at the Cosmonaut app project.yml. `app/Cosmonaut/project.yml`

```yaml
1  # Import of the main build_settings file
2  include:
3    - ../../fastlane/build_settings.yml
4
5  # Definition of the project
6  name: Cosmonaut
7  settings:
8    groups:
9      - BuildSettings
10
11 # Definition of the targets that exists within the project
12 targets:
13
14   # The main application
15   Cosmonaut:
```

```
16        type: application
17        platform: iOS
18        sources: Cosmonaut
19        dependencies:
20          # Domains
21          - framework: ISSCosmonaut.framework
22            implicit: true
23          - framework: ISSSpacesuit.framework
24            implicit: true
25          - framework: ISSScaffold.framework
26            implicit: true
27          # Services
28          - framework: ISSSpacesuitService.framework
29            implicit: true
30          - framework: ISSCosmonautService.framework
31            implicit: true
32          # Core
33          - framework: ISSNetwork.framework
34            implicit: true
35          - framework: ISSRadio.framework
36            implicit: true
37          - framework: ISSPersistence.framework
38            implicit: true
39          - framework: ISSUIComponents.framework
40            implicit: true
41
42    # Tests for the main application
43    CosmonautTests:
44      type: bundle.unit-test
45      platform: iOS
46      sources: CosmonautTests
47      dependencies:
48        - target: Cosmonaut
49      settings:
50        TEST_HOST: $(BUILT_PRODUCTS_DIR)/Cosmonaut.app/Cosmonaut
51
52    # UITests for the main application
53    CosmonautUITests:
54      type: bundle.ui-testing
55      platform: iOS
56      sources: CosmonautUITests
57      dependencies:
58        - target: Cosmonaut
```

Even though the yaml file speaks for itself, some explanation is needed.

First of all, the `include` in the very beginning.

```
1  # Import of the main build_settings file
2  include:
3    - ../../fastlane/build_settings.yml
```

Before xcodegen starts generating the pbxproj project it processes and includes other yaml files if the include keyword is found. In case of the application framework this is extremely helpful as the build settings for each project can be described just by one yaml file.

Imagine a scenario where the iOS deployment version must be bumped up for the app. Since the app links also many frameworks which are being compiled before the app, their deployment target also needs to be bumped up. Without XcodeGen, each project would have to be modified to have the new deployment target. Even worse, when trying some build settings out instead of modifying it on each project a simple change in one file that is included into the others will do the trick.

A simplified build settings yaml file could look like this: `fastlane/build_settings.yml`

```
 1  options:
 2    bundleIdPrefix: com.iss
 3    developmentLanguage: en
 4  settingGroups:
 5    BuildSettings:
 6      base:
 7      # Architectures
 8        SDKROOT: iphoneos
 9      # Build Options
10        ALWAYS_EMBED_SWIFT_STANDARD_LIBRARIES: $(inherited)
11        DEBUG_INFORMATION_FORMAT: dwarf-with-dsym
12        ENABLE_BITCODE: false
13      # Deployment
14        IPHONEOS_DEPLOYMENT_TARGET: 13.0
15        TARGETED_DEVICE_FAMILY: 1
16  ...
```

Worth mentioning is that in the `BuildSettings` the key in the yaml matches with Xcode build settings which can be seen in the inspector side panel. As you can see the `BuildSettings` key is then referred inside the `project.yml` file under the settings right after the project name.

```
 1  name: Cosmonaut
 2  settings:
 3    groups:
 4      - BuildSettings
```

The following key is `targets`. In case of the Cosmonaut application we are setting three targets. One for the app itself, one for unit tests and finally one for ui tests. Each key sets the name of the target and then describes it with `type`, `platform`, `dependencies` and other parameters XcodeGen supports.

Next, let us have a look at the dependencies.

```
 1  dependencies:
 2    # Domains
 3    - framework: ISSCosmonaut.framework
 4      implicit: true
```

```
5      - framework: ISSSpacesuit.framework
6        implicit: true
7      - framework: ISSScaffold.framework
8        implicit: true
9    ...
```

Dependencies links the specified frameworks towards the app. On the snippet above you can see which dependencies the app is using. The `implicit` keyword with the framework means that the framework is not pre-compiled and requires compilation in order to be found. That being said, the framework needs to be part of the workspace in order the build system to work. Another parameter that can be stated there is `embeded: {true|false}`. This parameter sets whether the framework will be embedded with the app and copied into the target. By default XcodeGen has `embeded: true` for applications as they have to copy the compiled framework to the target in order the app to launch successfully and `embeded: false` for frameworks. Since framework is not a standalone executable and must be part of some application it is expected that the application copies it.

Full documentation of XcodeGen can be found on its GitHub page:

Finally, let's generate the projects and build the app with all its frameworks. For that a simple lane in Fastlane was created.

```
1  lane :generate do
2    # Finding all projects within directories
3    Dir["../**/project.yml"].each do |project_path|
4      # Skipping the template files
5      next if project_path.include? "fastlane"
6
7      UI.success "Generating project: #{project_path}"
8      `xcodegen -s #{project_path}`
9    end
10 end
```

Simply executing the `fastlane generate` command in the root directory of the application framework generates all projects and we can open the workspace and press run. The output of the command should look as follows:

Figure 11: fastlane generate output

Ground Rules

Looking at the ISS architecture, there are two very important patterns that are being followed.

First of all, any framework does NOT allow to link modules on the same layer. That is a prevention for creating cross linking cycles in between frameworks. For example, if Network module would link Radio module and Radio module would link Network module we would be in a serious trouble. Surprisingly, not every time Xcode build fails in such set up, however, it will have a really hard time with compiling and linking, up until one day it starts failing.

Second of all, each layer can link frameworks only from its sublayer. This ensures the vertical linking pattern. That being said, the cross linking dependencies will also not happen on the the vertical level.

Let us have a look at some examples of cross-linking dependencies.

Cross linking dependencies

Let us say, the build system will jump on compiling the Network module where the Radio is being linked to. When it comes to the point where the Radio needs to be linked it jumps to compile Radio module without finishing the compilation of the Network. The Radio module now requires Network module in order to continue compiling, however, the Network module has not finished compiling yet, therefore, no `swiftmodule` and other files were yet created. The compiler will continue compiling up until one file will be referencing some part(e.g a class in a file) of the other module and the other module will be referencing the caller.

That's where the compiler will stop.

No need to say, each layer is defined to contain stand alone modules that are just in need of some subdependencies. While, in theory this is all nice and makes sense but in practice it can happen that for

example the Cosmonaut domain will require something from the Spacesuit domain. It can be some domain only logic, views or even the whole flow of screens. In that case there are some options how to tackle that issue. Either, creating a new module on the service layer and moving there the necessary source code files that are shared across multiple domains or shifting those files from the domain layer to the service layer. Third option would be to use abstraction and achieving the same not from the module level but on the code level. Chosen solution solely depends on the use-case.

A simple example could be that some flow is represented by a protocol that has a `start` function on it. That could for example be a `coordinator` pattern that would be defined for the whole framework and all modules would be following it. That protocol must then be defined in one of the lower layers frameworks in this case since it is related to a flow of view controllers, the UIComponents could be a good place for it. Due to that, in the framework we can count that all domains understands it. Thereafter, the Cosmonaut app could instantiate the coordinator from the Spacesuit domain and pass it down or assign it as a child coordinator to the Cosmonaut domain.

Vertical linking

Same as the horizontal layer linking, the vertical linking is also very important and must be followed in order to avoid above mentioned compiler issues. In practice, such scenario can also happen very easily. Imagine, that your team designed a new framework on the Core layer that will provide some extent logging functionality so as data analytics and so on. After a while, some team will want to use the logging functionality for example in the Radio module in order to provide more debugging details for developers for the Bluetooth module.

Unlike in the cross linking dependencies scenario, in this case the abstraction was defined on the core level already. Thereafter, there is no way of passing it in the code from the top down. In this case, the new layer needs to be created, let us say shared or common. The supporting layer that will contain mostly some shared functionality for the Core layer so as some protocols that would allow passing references from the top down.

Another solution would be to separate the core public protocols and models out the framework so that it can be exposed and linked towards more frameworks on the same layer. On the higher level the instantiation would take place and the instances would be passed to the implementations on the lower layer as they both linked towards the newly created core framework of that module. This, however, have the downside of having an extra framework that needs to be linked and maintained. However, with this approach the so called `Clean Architecture` would be followed. More about that later.

No need to say, any higher level layer framework can link any framework from any lower layer. So for example, the Cosmonaut app, can link anything from the Core or the newly defined Shared layer.

App secrets

Considering, many developers are working in the same repository on the same project, handling project secrets in the secure way is inevitable. Project secrets could be for example API keys, SDK keys, encryption keys, so as configuration files or certificates containing sensitive information that could cause potential harm to the app. Essentially, any piece of sensitive information that should not be exposed to anyone not working directly on the project. By any means, secrets should NOT be stored in the repository so as included in the compiled binary as a plain strings.

The app ideally should decrypt encrypted secret in its runtime. Even though, on the jailbroken iPhone the potential attacker could gain runtime access and print out the secrets while debugging or bypass SSLPinning and sniff the secrets from the network, considering the SSLPinning was in place like it should. In any case, it will take much more effort than just dumping binary strings that contain secrets.

In about two years ago me and my colleague Jörg Nestele had a look at the problem and over few weekends we came out with an open-source project written purely in Ruby called Mobile Secrets which solves this problem in a swifty way.

Now, let us have a look how to handle secrets in the project.

How to handle secrets

First thing first, as mentioned above any secret must be obfuscated, without a doubt. String obfuscation is a technique that via XOR, AES or other encryption algorithms modifies the confidential string or a file such that it cannot be de-obfuscated without the initial encryption key.

Unfortunately, obfuscating strings or files and committing them to the repository might not be enough. What if there is a colleague who has access to the source repository or someone who might want to steal these secrets and hand it out? Simply downloading the repository and printing the de-obfuscated string into a console would do it.

Essentially, the secret should be visible only for the right developer in any circumstances. Especially, for mono-repository projects where many teams are contributing to simultaneously. That is where GPG comes into play.

The GnuPG (GPG)

GPG is asymmetric key management system that creates a hash for encryption from public keys of all participants. In the initialisation process GPG will generate a private and public key. The public key is

saved in the .gpg folder under user's email visible to everyone. The private key is saved in ~/.gnupg and is protected by a password chosen by the user.

In order to add a developer into the authorised group, the developer needs to provide a public key from his machine, simply executing `dotgpg key` will print the key. This key must then be added by the already authorised person `via dotgpg add` .
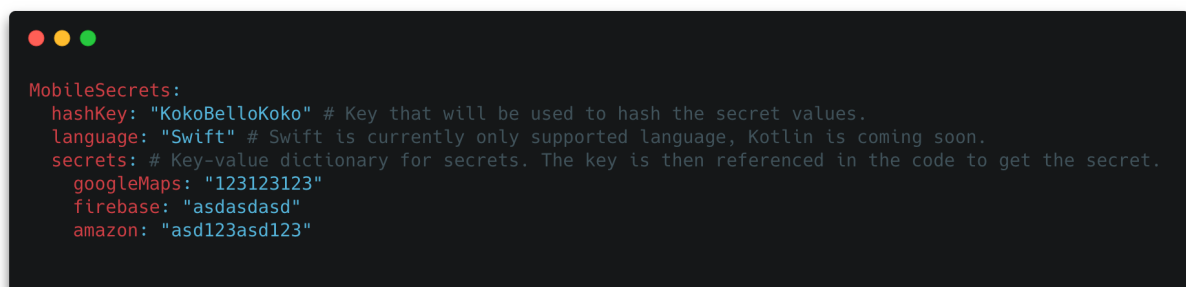
The encrypted file by GPG containing all project secrets can be then thoughtfully committed to the repository, since only authorised developers who posses the private key can decrypt it.

GEM: Mobile Secrets

Mobile Secrets gem is using XOR cipher alongside with GPG to handle the whole process. To install MobileSecrets simply execute `gem install mobile-secrets`.

Mobile Secrets itself can then initialise the GPG for the current project if it has not been done yet by running: `mobile-secrets --init-gpg` ..

When the GPG is initialised a template yaml file can be created by running: `mobile-secrets --create-template`

```
MobileSecrets:
  hashKey: "KokoBelloKoko" # Key that will be used to hash the secret values.
  language: "Swift" # Swift is currently only supported language, Kotlin is coming soon.
  secrets: # Key-value dictionary for secrets. The key is then referenced in the code to get the secret.
    googleMaps: "123123123"
    firebase: "asdasdasd"
    amazon: "asd123asd123"
```

Figure 12: MobileSecrets.yml

The `MobileSecrets.yml` file contains the hash key used for obfuscation of secrets with the key-value dictionary of secrets that must be adjusted to the project needs. All secrets of the project including the initial hashing key are then being organised in this structure encrypted by GPG. When everything was edited simply import the configuration file by running command. `mobile-secrets --`**`import`** `./MobileSecrets.yml` and it will be stored under `secrets.gpg`.

Finally, we can run: `mobile-secrets --export ./Output/Path/` to export the swift file with obfuscated secrets.

Mobile Secrets exported Swift source code will look like to the example below. Last but not least, that

path to this file must be added to the `gitignore`. The secrets source code must be generated locally alongside with generating the projects.

```swift
//
//  Autogenerated file by Mobile Secrets
//

import Foundation

class Secrets {
    static let standard = Secrets()
    private let bytes: [[UInt8]] = [[75, 111, 107, 111, 66, 101, 108, 108, 111, 75, 111, 107, 111],
                                    [103, 111, 111, 103, 108, 101, 77, 97, 112, 115],
                                    [122, 93, 88, 94, 112, 86, 93, 94, 92],
                                    [102, 105, 114, 101, 98, 97, 115, 101],
                                    [42, 28, 15, 14, 49, 1, 13, 31, 11],
                                    [97, 109, 97, 122, 111, 110],
                                    [42, 28, 15, 94, 112, 86, 13, 31, 11, 122, 93, 88]]

    private init() {}

    func string(forKey key: String) -> String? {
        guard let index = bytes.firstIndex(where: { String(data: Data($0), encoding: .utf8) == key }),
            let value = decrypt(bytes[index + 1]) else { return nil }
        return String(data: Data(value), encoding: .utf8)
    }

    private func decrypt(_ input: [UInt8]) -> [UInt8]? {
        let key = bytes[0]
        guard !key.isEmpty else { return nil }
        var output = [UInt8]()
        for byte in input.enumerated() {
            output.append(byte.element ^ key[byte.offset % key.count])
        }
        return output
    }
}
```

Figure 13: Exported Swift secrets source file

The ugly and brilliant part of the Secrets source code

What happened behind the hood? Out of the yaml configuration, secrets were obfuscated with the specified hash key via XOR and converted into bytes. Therefore, it ended up with an array of UInt8 arrays.

The first item in the bytes array is the hash key. The second item is the key for a secret, the third item contains the obfuscated secret, fourth is again the key and fifth is the value, and so forth.

To get the de-obfuscated key just call the string(forKey key: String) function. It will iterate over the bytes array, convert the bytes into a string and compare it with the given key. If the key was found the

decrypt function will be called with a value on the next index.

Since we have the array of UInt8 arrays(`[[UInt8]]`) mixed with the hash key, keys and obfuscated secrets, it would take a significant effort to reverse-engineer the binary and get the algorithm. Even to get the bytes array of arrays would take a significant effort. Even though, it made it really hard to get the secrets out of the binary. Yet, the secrets can still be obtained when the attacker gains control over the runtime of the app like mentioned before.

Workflow

While this book is mostly focused on the development aspects of the modular architecture some management essentials must also be mentioned. Not surprisingly, developing software for large organisation can be a real challenge, imagine a scenario where at around 30 to 50 developers per platform (iOS/Android/Backend) are working towards the same goal, the same project. Those developers are usually divided into multiple cross-functional teams that are working standalone towards the team's goal.

Cross-functional team usually consists of a product owner or a product manager, that highly depends on the company and its structure so as the agile methodologies the company is using. Furthermore, designers, who are defining the UI/UX and behaviour on each platform, developers from each platform and last but not least, the quality assurance.

The team goal can be for example developing some specific business domain, the team then becomes the domain expert and is further responsible for developing, improving and integrating that domain into the final customer facing application.

It could also be that the team develops a standalone application on top of the framework. If the team followed the same patterns defined in the framework, usually by technical leads, it can also be easily later on integrated as a part of some bigger application that for example groups those functionalities in one app. Or the other way around, splits one big app into multiple smaller ones. Like for example Facebook did with their Messenger app.

Teams

Teams and their management plays crucial role in the success of the project. The modular architecture by any means helps to find boundaries of teams. From the developers POV, each developer is responsible for developing the frameworks belonging to the team. In our Cosmonaut example, it could be Cosmonaut domain along side with Cosmonaut service. While Spacesuit domain and Spacesuit service would be developed by another team.

That does not necessary mean that the team Cosmonaut, let us say, cannot develop or modify the source code in the domain or service of Spacesuit, however, it surely should not be able to merge their changes into the repository without the given permission of that affected team.

Luckily, there is one simple solution for it. The code owners definition, which most of the CI/CD platforms supports. The code owner file simply defines who is the owner of what part of the codebase. This already briefly touched the topic of git which is covered in the following subchapter.

Git & Contribution

While there are many different approaches how to contribute to the repository via git in our modular architecture I find out one particular the most helpful. The GitFlow. I am sure you have heard of it at some point or if you have not you could be using it without knowing.

Most likely, on projects developed by a single team it heavily depends on the team how they will decide to contribute to the repository. Nevertheless, in the case of team of teams, contributing into mono-repository with GitFlow alongside with four eyes principle is the way to go.

Four eyes principles simply means that in order to merge a pull request it needs to be reviewed first. That being said, in the team, it is essential that on each platform works at least two developers so that the team can gain autonomy and work independently.

In case of making changes in other team's code a dedicated code-owner from the team must approve the changes. Only with that, the team can stay ahead and have the ultimate overview of their part of the codebase and domain knowledge. Without defining the code-owners soon everyone would be working everywhere and it would all turn into chaos.

Looking at the framework structure, it is quite clear where each team has its boundaries. However, there is one part that's very difficult to maintain and takes the most effort. That part will be the core layer. While, core layer could be develop by the creators of the application framework in the very early stages, it is surely the part everybody is relying on. Therefore, great test coverage plays crucial role when developing anything in the core layer. Later on, any change in any interface of an object will affect everybody who is using it. Since it is the lowest layer, if we don't count the utils layer, it will be highly likely used by many frameworks on the higher layer.

The core layer, after all the desired functionality was implemented and bugs were fixed will not need much of a focus. However, teams might need to extent the functionality on that layer, which could lead as mentioned above towards opening a PR with their suggestions for extension. In this case, the core ownership should be made of tech leads who are having the overall vision over the project.

Scalability

As mentioned already in the book, the modular architecture is designed to be highly scalable. With the pre-defined scripts new team can simply create new framework or an app and start the implementation. Nevertheless, the onboarding of a new colleague or the whole team plays significant time that needs to be taken into account.

Most likely, scaling for example to a 6th team working on the framework will require quite extensive onboarding session. Due to the large amount of code, design patterns, CI/CD, code style etc. will take a lot of time for new comers to get the speed. In such case, usually platform technical leads are doing pair programming, code reviews and further onboarding up until the new comers are familiar with the development concept, patterns and so on.

Application Framework & Distribution

Architecture-wise, the whole application framework can be used as the software foundation for the whole company. Different products can be easily created with the already pre-built foundation which contains the whole company knowledge in terms of the software development. No need to say, from engineering point of view, this is a big win for the whole company. For example, thanks to that, SW engineers can much more precisely estimate how much of the development time can the new product cost, how long will it take so as probably what would be the biggest challenges there.

Nevertheless, there are much more use cases where such architecture would be helpful. Due to the modularisation, standalone frameworks can be exported and used for development without affecting the current development workflow. This could be particularly helpful when, for example, a subsidiary would be developing another software product. With already pre-built core components, most importantly the UI part facing the customers, new product in the subsidiary can be built much faster and without gaining the confidential parts held in service or domain layers. However, in such case the team responsible for developing the distributed components or let us say the SDK is becoming the support team for the customers of the SDK. In that case a new process and workflow must be established. The responsible team could be opened for submitting bug reports and distributing the new versions of the SDK bi-weekly, or whenever it is suitable.

Common Problems

While praising such architecture pretty much all the time, like everything, it comes with it's disadvantages as well.

Maintenance

First things first, the maintenance of the application framework can be very inefficient. The whole framework will not go far without technical leads who are having the company strategy aligned vision for the development of the framework or products and can, thereafter, give the directions for the development, also technically backed up by solution architects. Since there can be many teams working and contributing to the repository, the maintenance might be happening on a daily basis. Most effected part is probably the CI/CD chain, where things can break quickly, failing unit-tests, legacy code, supporting apps that are no longer in development etc.

In case of the maintenance, particularly hard could be maintaining apps or frameworks that are no longer in development. Let us say, an app consists of domains and services was successfully delivered to the customers and there is no more development planned for it. This resolves in code that runs in production, therefore, it is very important. However, since it relies on the e.g core frameworks that are still in development the tests will start failing, the interfaces needs to be updated and so on. In the end, this will require additional effort for one of the teams to just take care of it until further decision is made for development. Or, archive it, remove it from the actively developed codebase and when time comes, put it back, update all interfaces and changes that happened in the framework and then happily continue development.

Code style

Since there are many different developers working, following the same code style, principles and patterns can be a challenge. Everybody has a different preferences, different experience and getting sometimes people on the same board is quite difficult.

Nevertheless, following the ground rules and overall framework patterns is what matters the most. In this case what really helps is well documented code, framework documentations so as proper onboarding that ideally consists of pair programming, code reviews and ad-hoc one on one sessions. Changes in the code-style, importing new libraries, new patterns and so on can be discussed in developers guild meetings where everybody can vote for what seams to be the best option. In guilds, everybody can make suggestions for improvements and vote for changes he or she likes.

Not fully autonomous teams

In theory, each team should have its autonomy. Nevertheless, in practice it is slightly different, in some cases, teams might depend on each other which, furthermore, brings more teams communication, and worst case failing the teams goal because of the unfulfilled promises of the dependent team.

More teams are dependent on each other, more meetings and alignments are needed which unfortunately, slows down the teams.

Conclusion

In this chapter we had a look, how in practice development of modular architecture could look like. We explored ground rules, project generations with XcodeGen, handling secrets in the secure way so as some common problems people working on such project will be facing.

I hope it all gave a good understanding of how to work in such setup.

Dependency Managers

Generally, good practice when working on large codebases is not to import many 3rd party libraries. Especially the huge ones, for example, RxSwift, Alamofire, Moja etc. Those libraries are extraordinarily big and highly likely some of their functionality will never be used. This will resolve in having a dead code attached to the project. No need to say, the binary size, compile time and most importantly the maintenance will increase heavily. With each API change of the library every part of the codebase will have to be adapted. Obviously, essential vendor SDKs, like GoogleMaps, Firebase, AmazonSDK and so on will still have to be linked to the project, however, using libraries to provide wrappers around the native iOS code should be avoided and developed specifically to the project needs.

In one of my projects I worked on, the compile time of the whole application was at about 20-25 minutes. The project was in development for about 8 years and had approximately 80 3rd party dependencies linked via Cocopods. That was the biggest problem for taking so much of the compile time. I do remember that I spent nearly three full months refactoring the huge codebase into the modular architecture described here. Before the project was modularised with internally developed frameworks maintained as Cocoapods. Along the way of refactoring, I also removed some of the unused libraries, and most importantly removed the Alamofire, RxSwift, RxCocoa, and other big libraries from Cocoapods and linked them via Carthage which decreased the compile time drastically.

Cocopods libraries are compiled every time the project is cleaned while Carthage is compiled only once. Carthage produces binaries that are then linked to a project. Cleaning a project was pretty common thing to do, Xcode has improved in that sense a lot, but with first Swift versions it was nowhere near to be perfect and with clean build most issues disappeared immediately. Due to that refactoring, the compile time after cleaning the project was decreased to 10 minutes which was mostly compile of the project source code with a few 3rd party libraries that remained linked via Cocoapods for convenient.

The way third party libraries are managed and linked to the project matters a lot, especially, when the project is big or is aiming to be big. Now let us have a look at the three most commonly used dependency managers on iOS and how to use them in the modular architecture. By the most common, I obviously mean Cocopods, Carthage and the new Apple's SwiftPM.

Cocopods

The most used and well known dependency manager on iOS is Cocoapods. Cocoapods are great to start with, it is really easy to integrate a new library so as to remove it. Cocoapods manage everything for the developer under the hood, therefore, there is no further work required in order to start using the library. When Cocoapods are installed, with `pod install`, they are attached to the workspace

as Xcode project that contains all libraries that are specified in the Podfile. During the compile of the project dependencies are compiled as they are needed. This is really good for small projects or even big ones but the libraries must be linked carefully as every library takes some compile time and further maintenance, like mentioned before.

Quite often Cocoapods are also used for in-house framework development which is very convenient, however, all the fun stops when the project grows and the internal dependencies are using many of big libraries. Then the whole project depends on the in-house developed pods who are internally linking the 3rd party pods. This scenario can easily result in a huge compile time as there is no legit way of replacing the linked 3rd party frameworks via their compiled versions. No need to say, Cocoapods also will not let you integrate a static library to more than one framework because of transitive dependencies, therefore, some dynamic library wrappers might need to be introduced to avoid it.

In such cases, it might be necessary to move away from internally developed Cocoapods and integrate similar approach like described in this book which gives the project complete freedom. This could lead to days or even weeks of work, depends on how the project is big, structured etc. Nevertheless, moving away from such design will improve everyday compile-time of each developer when it comes to that point. Like mentioned before, for small projects it could really be the way to go but it definitely has its limits.

Integration with the application framework

Surprisingly, to integrate Cocoapods in the whole application framework might also not be as easy as you might think. Cocoapods must keep the same versions of libraries across all frameworks so as each app developed on top of it. This will require a little bit of Ruby programming. Essentially, the application framework must have one shared Podfile that will define Pods for each framework, thereafter, every app can easily reuse it. Furthermore, each app has its own Podfile that specifies what Pods must be installed for which framework to avoid unnecessary linking for frameworks the app will not need.

Let us have a look now how App's Podfile could look like for the Cosmonaut example. app/
Cosmonaut/Podifle

```
1  # Including the shared podfile
2  require_relative "../../fastlane/Podfile"
3
4  platform :ios, '13.0'
5  workspace 'CosmonautApp'
6
7  # Linking pods for desired frameworks from the shared Podfile
8  # Domain
9  spacesuit_sdk
```

```
10  cosmonaut_sdk
11  scaffold_sdk
12  # Service
13  spacesuitservice_sdk
14  cosmonautservice_sdk
15  # Core
16  network_sdk
17  radio_sdk
18  uicomponents_sdk
19  persistence_sdk
20
21
22  # Installing pods for the Application target
23  target 'CosmonautApp' do
24    use_frameworks!
25
26    pod $snapKit.name, $snapKit.version
27
28    # Linking all dynamic libraries required from any used framework
          towards the main app target
29    # as only app can copy frameworks to the target
30    add_linked_libs_from_sdks_to_app
31
32    # Dedicated tests for the application
33    target 'CosmonautAppTests' do
34      inherit! :search_paths
35    end
36
37    target 'CosmonautAppUITests' do
38      # Pods for testing
39    end
40  end
```

Firstly, the shared Podfile that defines pods for all frameworks is included. After setting the platform and workspace, the installation for all linked frameworks takes place. Last but not least, the well known app target is defined, potentially with some extra pods. Here, a special attention goes to the `add_linked_libs_from_sdks_to_app` function which will be explained in a second.

To fully understand what is happening inside of the app's Podfile we have to have a look at the shared Podfile. `fastlane/Podifle.rb`

```
1  require 'cocoapods'
2  require 'set'
3
4  Lib = Struct.new(:name, :version, :is_static)
5  $linkedPods = Set.new
6
7  ### available libraries within the whole Application Framework
8  $snapKit = Lib.new("SnapKit", "5.0.0")
9  $siren = Lib.new("Siren", "5.8.1")
```

64

```ruby
10   ...
11
12   ### Project paths with required libraries
13   # Domains
14   $scaffold_project_path = '../../domain/Scaffold/Scaffold.xcodeproj'
15   $spacesuit_project_path = '../../domain/Spacesuit/Spacesuit.xcodeproj'
16   ...
17
18   # Linked libraries
19   $network_libs = [$trustKit]
20   $cosmonaut_libs = [$snapKit]
21   ...
22
23   ### Domain
24   def spacesuit_sdk
25     target_name = 'ISSSpacesuit'
26     install target_name, $spacesuit_project_path, []
27   end
28
29   def cosmonaut_sdk
30     target_name = 'ISSCosmonaut'
31     test_target_name = 'CosmonautTests'
32     install target_name, $cosmonaut_project_path, $cosmonaut_libs
33
34     install_test_subdependencies $cosmonaut_project_path, target_name,
35         test_target_name, []
     end
36   ...
37
38   # Helper wrapper around Cocoapods installation
39   def install target_name, project_path, linked_libs
40     target target_name do
41       use_frameworks!
42       project project_path
43
44       link linked_libs
45     end
46   end
47
48   # Helper method to install Pods that
49   # track the overall linked pods in the linkedPods set
50   def link libs
51     libs.each do |lib|
52       pod lib.name, lib.version
53       $linkedPods << lib
54     end
55   end
56
57   # Helper method called from the App target to install
58   # dynamic libraries, as they must be copied to the target
59   # without that the app would be crashing on start
```

```
60  def add_linked_libs_from_sdks_to_app
61    $linkedPods.each do |lib|
62      next if lib.is_static
63      pod lib.name, lib.version
64    end
65  end
66
67  # Maps the list of dependencies from YAML files to the global variables
         defined on top of the File
68  # e.g ISSUIComponents.framework found in subdependencies will get
       mapped to the uicomponents_libs.
69  # From all dependencies found a Set of desired libraries is taken and
       installed
70  def install_test_subdependencies project_path, target_name,
       test_target_name, found_subdependencies
71    ...
72  end
```

Here we can see, on top of the file, the struct `Lib` that represents a Cocoapod library. On next lines, it is used to describe the libraries that can be used within the whole framework and apps. Furthermore, each framework is defined by a function, e.g. `spacesuit_sdk`, that is then called from the main app Podfile to install the libraries for those required frameworks. Finally, helper functions are defined to simplify the whole workflow.

Two functions requires some explanation, first the `add_linked_libs_from_sdks_to_app` mentioned in the app's Podfile. The function must be called from within the app's target to add all the dependencies of the linked frameworks. Without it, we would end up in the so called dependency hell. The app would be crashing with e.g `TrustKit library not loaded... referenced from: ISSNetwork`, because the libraries were linked towards the frameworks, however, frameworks do not copy linked libraries into target. Therefore, the App must do it for us. Frameworks then can find their libraries at the `@rpath`(Runpath Search Paths).

The second function is `install_test_subdependencies`, this the same scenario as for the previous function, however, in this case for tests. In order to launch tests, they also have to link all dependencies of the linked frameworks towards it. Lucky enough, thanks to Xcodegen, we can iterate over all `project.yml` files and find the linked frameworks and within the shared Podfile then use the defined pods for those frameworks.

In the source code everything is well commented so it should be easy to understand.

Carthage

While Cocoapods is true 3rd party dependency manager that does everything for the developer under the hood, Carthage leaves developers with free hands. Carthage pre-build libraries described in

the Cartfile and produces compiled binaries for pre-defined architectures. Executing Carthage's build command with `carthage update --platform iOS` will fetch all the dependencies and produces the compiled versions. Such task can be really time consuming, especially, when 3rd party libraries are one of the big ones. Nevertheless, such command is usually executed only ones. Compiled libraries can then be stored in some cloud storage where each developer or CI will pull them from into the pre-defined git ignored project's folder or possibly update them if it is necessary. That being said, some dependency maintenance and sharing strategy needs to be in place.

As mentioned above, Carthage only builds the libraries, it is on the developer how they are then linked towards the frameworks and apps. Luckily, XcodeGen helps a lot when linking Carthage libraries. In the yaml file it can be easily defined where the Carthage executables are stored so as which ones we want to link. In case the linked framework is a static one it can also be easily specified with linkType.

```
1  # Definition of the targets that exists within the project
2  targets:
3
4    # The main application
5    Cosmonaut:
6      type: application
7      platform: iOS
8      sources: Cosmonaut
9      # Considering the Carthage is stored in the root folder of the
           project
10     carthageBuildPath: ../../Carthage/build
11     carthageExecutablePath: ../../Carthage
12     dependencies:
13       - carthage: Alamofire
14       - carthage: SnapKit
15       - carthage: MSAppCenter
16         linkType: static
17       # Domains
18       - framework: ISSCosmonaut.framework
19         implicit: true
20       - framework: ISSSpacesuit.framework
21         implicit: true
22       - framework: ISSScaffold.framework
23         implicit: true
24     ...
```

Eventually, the compiled binaries are linked to the frameworks and apps which no longer requires the expensive compile time. When the build is started Xcode looks at the library search paths for linking the compiled binaries. In comparison to compiling the whole 3rd party libraries from source code linking takes only seconds.

By the end of the day, Carthage will require much more work for having it properly setup and maintained in the project, updating one library will require recompile the library and its dependencies,

uploading it somewhere to the server where it can be accessible by all developers and finally, down-loading the latest Carthage builds by developers locally. Surely, each developer can also recompile the libraries locally but that can take away again a lot of time from each developer, depends on the amount of libraries used.

One last thing worth mentioning is ABI (Application Binary Interface) interoperability. Since with Carthage the binaries are being linked, the compiler must be interoperable with the compiler who produced the binaries. That in the end might be a big problem, as the whole team will have to update Xcode in the same time so as the vendors of those libraries might need to update their source code to be compatible with the higher version of Swift. ABI is a very interesting topic in language development. I would highly encourage reading about it in the official Swift repository.

SwiftPM

Swift Package Manager is official dependency manager provided by Apple. It works on a similar basis as Cocoapods. Each framework or app is described with its dependencies in Package.swift file and compiled during the build. The benefit of using SwiftPM is that it can be used for scripts development or even cross-platform development, while Cocopods are AppleOS dependent. Nevertheless, SwiftPM can be used for iOS/macOS development with an ease. In comparision with Cocoapods, SwiftPM does not require extra actions in order to fetch the dependencies, Xcode simply takes care of it. SwiftPM also does not require to use Workspace for development as it directly adds the dependencies to the current Xcode project while Cocopods requires to work in a Xcode's workspace as the Pods are at-tached as a standalone project that contains all the binaries.

One of the example projects using the SwiftPM for cross-platform development is the Swift server-side web framework, Vapor. Vapor runs happily on both, macOS and linux instances. Nevertheless, devel-oping Swift code for both platform might not be as easy as you imagine. There is a lot of differences in the Foundation developed for Apple OSes and Linux versions. Therefore, some alignments might be necessary in order to run the code on both systems which can be the reason why some libraries cannot be simply used on both systems, but that was just a side note.
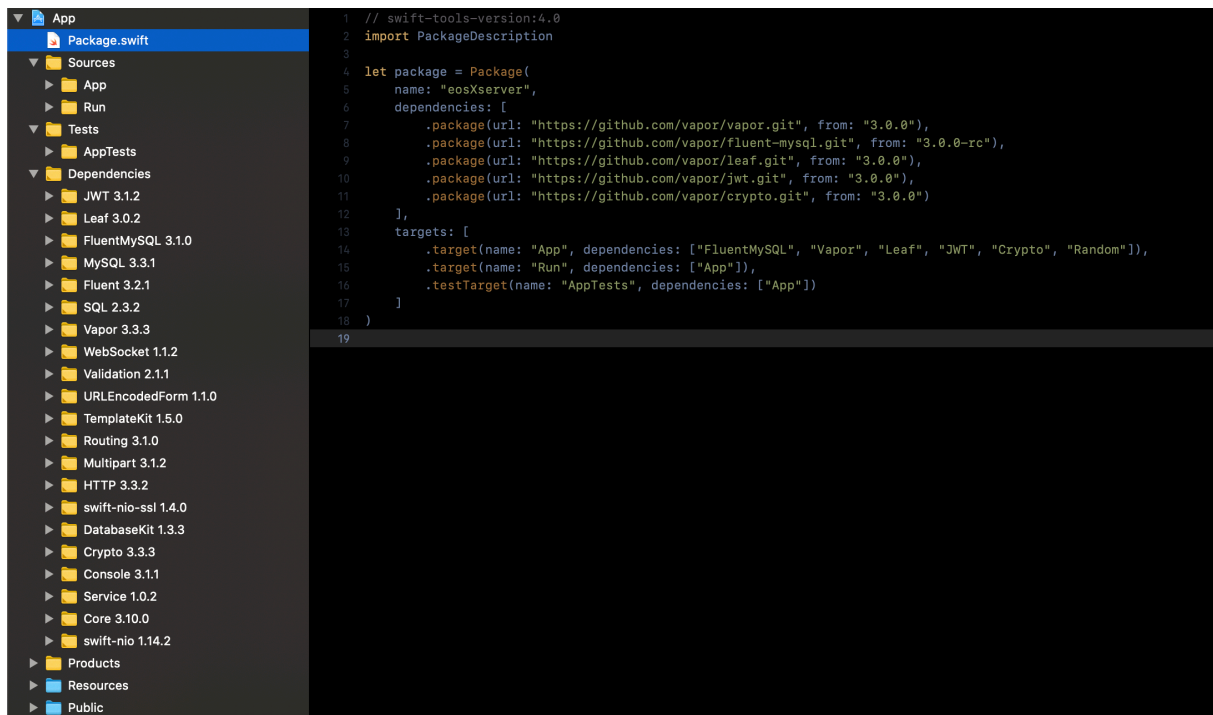
Figure 14: SwiftPM with Vapor application

Conclusion

This chapter gave an introduction into the most common package managers that could be used for managing 3rd party frameworks with an ease. Choosing the right one might, unfortunately, not be as obvious as we would wish for. There are trade-offs for each one of them, however, choosing Cocopods or SwiftPM at start and then potentially replacing some of the big libraries with Carthage, such that the compile time can be decreased might be a good way to go. That being said, with the hybrid approach, project benefits from both which could speed up the everyday development dramatically.

Design Patterns

Design patterns help developers to solve complex problems in a known, organised and structural way. Furthermore, when new developers are onboarded, they might already know the patterns used for solving such problem which helps them to gain the speed and confidence for development in the new codebase.

Purpose of this book is not to focus on design patterns in detail as there are plenty of books about them already. However, some patterns that are particularly useful when developing on such modular architecture are highlighted in here.

Coordinator

First of all, let us have a look at Coordinator pattern. One of the well known navigation pattern of all times when it comes to iOS development. Coordinator as it's name says takes care of coordinating the user's flow through out the app. In our Application Framework, each domain framework can be represented by its coordinator as an entry point to that domain. Coordinator can then internally instantiate view controllers so as their view models and coordinate the presentation flow. For the client, who is using it, all the necessary complexity is abstracted and held at one place. Coordinator's usually needs to be triggered to take the charge with `start` method. Such method could also provide an option for a `link` or a `route` which is a deep link whose the coordinator can decide to handle or not.

While there are many different implementations of such pattern, for the sake of the example and our CosmonautApp I chose the simplest implementation.

First, let us have a look at some protocols: File: `core`/`UIComponents`/`UIComponents`/`Source`/`Coordinator`.`swift`

```swift
1  // DeepLink represents a linking within the coordinator
2  public protocol DeepLink {}
3
4  public protocol Coordinator: AnyObject {
5
6      var childCoordinators: [Coordinator] { get set }
7      var finish: ((DeepLink?) -> Void)? { get set }
8
9      func start()
10     func start(link: DeepLink) -> Bool
11 }
12
13 // Representation of a coordinator who is using navigationController
14 public protocol NavigationCoordinator: Coordinator {
15     var navigationController: UINavigationController { get set }
16 }
```

```
17
18   // Representation of a coordinator who is using tabBarController
19   public protocol TabBarCoordinator: Coordinator {
20       var tabBarController: UITabBarController { get set }
21       var tabViewController: TabBarViewController { get }
22   }
23
24   public protocol TabBarViewController: UIViewController {
25       var tabBarImage: UIImage { get }
26       var tabBarName: String { get }
27   }
```

To see how those protocols are used in action we can have a look at the CosmonautCoordinator. File:

domain/Cosmonaut/Cosmonaut/Source/CosmonautCoordinator.swift

```
 1   public class CosmonautCoordinator: NavigationCoordinator {
 2       public enum CosmonautLink {
 3         case info
 4       }
 5
 6       public lazy var navigationController: UINavigationController =
 7           UINavigationController()
 8
 9       public var childCoordinators: [Coordinator] = []
10
11       public init() {}
12
13       public func start() {
14           navigationController.setViewControllers([
15               makeCosmonautViewController()
16           ], animated: false)
17       }
18
19       public func start(link: DeepLink) -> Bool {
20           guard let link = link as? CosmonautLink else { return false }
21
22           // TODO: handle rute
23           return true
24       }
25
26       private func makeCosmonautViewController() -> UIViewController {
27           return ComsonautViewController()
28       }
29   }
```

After hooking up the coordinator into the App window, with for example the main AppCoordinator defined in the Scaffold module, simply calling start() or start(link: CosmonautLink.info) will take over the flow of the particular domain or user's flow.

Strategy

One of my favourite pattern is Strategy, even though I create it in a slightly different way than it was originally meant to. Strategy pattern is particularly helpful when developing reusable components, like for example views. Such view can be initialised with a certain `strategy` or a `type`. In traditional book examples, strategy pattern is often described and defined via protocols and the ability to exchange the protocol with a different implementation that conforms to it. However, there is much more Swiftier way to achieve the same goal with an ease, `enum`. Enum, can simply represent strategy for each case for the object and via enum functions the necessary logic can be implemented.

// TODO implement example with one of the views

Configuration

Configuration is great for all the services and components that serve more than one purpose which will highly likely happen as we are developing many apps on top of the same reusable services. Configuration, is a simple object that describes how an instance of the desired object should look like or behave. That could be as simple as setting SSLPinning for network service or setting a name to a CoreData context and so on.

As an example in the Application Framework we can have a look at the `AppCoordinator` where the `Configuration` is defined in its extension. File: `domain/Scaffold/Scaffold/Source/AppCoordinator.swift`

```
1  extension AppCoordinator {
2      public struct Configuration {
3          public enum PresentatinStyle {
4              case tabBar, navigation
5          }
6
7          public var style: PresentatinStyle
8          public var menuCoordinator: Coordinator?
9
10         public init(style: PresentatinStyle, menuCoordinator:
                Coordinator?) {
11              self.style = style
12              self.menuCoordinator = menuCoordinator
13          }
14      }
15  }
```

AppCoordinator takes the configuration object and based on it's values performs necessary actions to provide desired behaviour.

File: `domain/Scaffold/Scaffold/Source/AppCoordinator.swift`

```
1  public class AppCoordinator: Coordinator {
2    ...
3      public init(window: UIWindow, configuration: Configuration) {
4          self.configuration = configuration
5          self.window = window
6      }
7    ...
8  }
```

Decoupling

It can surely happen that at some point a different implementation of some protocol must be used, however, that might be much harder than you might think. Imagine a scenario where a Cosmonaut-Service (protocol) is used all over our ComsonautApp, however it was decided that this app will have two different flavours, one for US cosmonaut and one for Russian cosmonaut. The cosmonaut service logic can be really huge, 3rd party libraries that are used might also differ and surely we do not want to include unused libraries with our US ComsonautApp or vice versa, that would be shipping a dead code! In that case we have to decouple those two frameworks and provide a common interface to them in a separate framework.

In such case we have to make one exception to our Application Framework linking law. For example, let us call it `CosmonautServiceCore` framework would be representing the public interfaces for the higher layers, it would contain protocols and necessary objects that needs to be exposed out of the framework to the outer world. `USCosmonautService` and `RUCosmonautService` would then link the `CosmonautServiceCore` on the same hierarchy level and would provide the implementations of protocols defined there.

In such case, since there is no cross linking in between those interfaces framework and its implementation, it is fine to link it that way. The higher level framework or even better the main App itself, in our example `CosmonautApp` would then based on the availability of the linked framework instantiated the objects represented in the `CosmonautServiceCore` from the framework that was linked to it. Which would be either `USCosmonautService` or `RUCosmonautService`.

This example is not part of the source code demo, however, such scenario can happen so it is important to keep the solution for such problem in mind.

MVVM + C

Probably no need much of explanation for Model, View, ViewModel with Coordinator pattern, however, there are many different approaches and implementations. First of all, sometimes it's not necessary

to use MVVM as the plain old classic MVC do the trick as well without the necessity of having an extra object. Second of all, the way how objects are binded together matters.

Protocol Oriented Programming (POP)

The most beautiful way of extending any kind of functionality of an object is probably via protocols and their extensions. In some cases, like for structs or enums it is also the only way. In Swift, structs and enums cannot use inheritance. On the other hand, protocols can use inheritance so as composition for defining the desired behaviour which gives the developer the freedom to design fine granular components with all OOP features.

//TODO: Think of more helpful related paterns

Conclusion

//TODO:

Project Automation

When it comes to a project where many developers are contributing to simultaneously automation will become a crucial part of it. It might be hard in the very beginning to imagine what kind of tasks might be automated but it will become crystal clear during the development phase. It can be simply generating the `xcodeproj` projects by XcodeGen like in our example to avoid conflicts, pulling new translation strings, generating entitlements on the fly, up to building the app on the CI and publishing it to the AppStore or other distribution centre.

Fastlane

First and foremost, iOS developers beloved `Fastlane`. Fastlane is probably the biggest automation help when it comes to iOS development. It contains countless amount of plugins that can be used to support the project automation. With Fastlane it is also easy to create your own plugins that will be project specific only. Fastlane is developed in ruby so as it's plugins are. However, since all is built with ruby fastlane gives the freedom to import any other ruby projects or classes developed in ruby and directly call them from the Fastlane's recognisable function so called `lane`.

As an example we can have a look at the Fastfile's `make_new_project` lane introduced in the very beginning where in this case the so called `ProjectFactory` class is implemented in `/fastlane` `/scripts/ProjectFactory/` and imported into the Fastfile and then used as a normal ruby program.

`/fastlane/Fastfile`

```
 1  require_relative "scripts/ProjectFactory/project_factory"
 2
 3  lane :make_new_project do |options|
 4    type = options[:type] # Project type can be either app or framework
 5    project_name = options[:project_name]
 6    destination_path = options[:destination_path]
 7
 8    UI.error "⛔app_name and destination_path must be provided⛔." unless
         project_name && destination_path
 9
10    factory = ProjectFactory.new project_name, destination_path
11
12    type == "app" ? factory.make_new_app : factory.make_new_framework
13  end
```

Another option would be to create a proper Fastlane's plugin out of it which could be easily done following this documentation.

Fastlane gives the ultimate home for the whole project automation which will prevent script duplications and help with organising and finding necessary actions. To check what is available available alredy in the house of automation, fastlane can be executed out of the command line and it will give you option to choose from the publicly available `lanes` that are already implemented.

- CI/CD
- Translations, Stages, Configurations etc.
- Distribution

  - Frameworks
  - XCFrameworks
  - Static Library

Ruby, programmer's best friend

If you have not learnt it yet, do so, it's great.

//TODO: ?