

Uniwersytet Jagielloński w Krakowie
Wydział Fizyki, Astronomii i Informatyki Stosowanej



WITOLD KAROL BOBROWSKI

NUMER ALBUMU: 1115454

CZYTNIK KSIĄŻEK ELEKTRONICZNYCH NA IOS

PRACA LICENCJACKA NA KIERUNKU INFORMATYKA

PRACA WYKONANA POD KIERUNKIEM:
dra Karola Przystalskiego
Zakład Technologii Informatycznych

Kraków 2017

Oświadczenie autora pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Kraków, dnia

Podpis autora pracy

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Kraków, dnia

Podpis kierującego pracą

Streszczenie

Poniższa praca powstała w celu opisania biblioteki **EPUBKit**. Biblioteka ta ma na celu obsługę plików w formacie *EPUB*, oraz dostarczenie możliwości ich wyświetlenia. **EPUBKit** jest napisany w języku Swift na platformę iOS. Rozpoczynając od opisu użytych narzędzi oraz technologii, przechodzę do charakteryzacji standardu *EPUB*, szczegółowo omawiając jego strukturę. Kolejny rozdział zawiera dogłębną analizę biblioteki **EPUBKit**. Pracę kończy omówienie procesu tworzenia aplikacji na iOS oraz wykorzystania w niej biblioteki **EPUBKit**.

Słowa kluczowe: iOS, Swift, EPUB, parser, czytnik książek elektronicznych, aplikacja.

Abstract

The purpose of this thesis is to present the **EPUBKit** library. The goal of the library is to handle *EPUB* documents and give the possibility of displaying them on an iOS device.

EPUBKit is written using swift language and available exclusively for iOS platform. Starting off with the description of used tools and technologies, I will move onto covering the *EPUB* standard in greater depth. Afterwards, deep analysis of my library will follow. Thesis ends with the description of iOS applications creation process with special emphasis on the **EPUBKit** and how it fits in this landscape.

Keywords: iOS, Swift, EPUB, parser, e-reader, application.

Spis treści

1. Wstęp.....	7
2. Wykorzystane technologie.....	9
2.1. Xcode i Developer Tools	9
2.1.1. Xcode IDE.....	10
2.1.2. Simulator.....	11
2.1.3. Instruments.....	11
2.2. Swift	12
2.3. iOS SDK.....	14
2.3.1. CocoaTouch.....	14
2.4. Zewnętrzne biblioteki	14
2.4.1. Zip	15
2.4.2. AXML	15
3. Charakterystyka EPUB.....	16
3.1. Specyfikacja.....	16
3.1.1. EPUB Open Container Format	17
3.1.2. EPUB Content Documents.....	18
3.1.3. EPUB Core Media Types	19
4. Biblioteka EPUBKit.....	21
4.1. Model.....	22
4.1.1. EPUBDocument.....	23
4.1.2. EPUBManifest	25
4.1.3. EPUBMetadata.....	27
4.1.4. EPUBSpine	28
4.1.5. EPUBTableOfContents	29
4.2. Parser	30
4.2.1. EPUBParsable	30
4.2.2. EPUBParser	31
4.2.3. EPUBParserError	37

4.3.	Widok	39
4.3.1.	EKViewController.....	40
4.3.2.	EKTableOfContentsViewController	43
4.3.3.	EKInfiniteScrollView	45
4.3.4.	EKTableOfContentsDataSource	47
4.3.5.	Protokoły	50
4.4.	Dystrybucja.....	51
5.	Aplikacja demonstracyjna	53
5.1.	Wykorzystanie biblioteki w aplikacji	56
5.1.1.	BookTableViewCell	56
5.1.2.	ViewController	57
6.	Podsumowanie	59

1. Wstęp

Celem pracy jest opisanie stworzonego przeze mnie projektu jakim jest biblioteka **EPUBKit**, której zadaniem jest obsługa (parsowanie oraz wyświetlanie) książek elektronicznych w formacie EPUB (Electronic Publication), a następnie wykorzystanie jej w aplikacji mobilnej na platformie iOS. Biblioteka ta została stworzona z myślą o udostępnieniu jej publicznie w formie open-source na portalu GitHub. EPUBKit jest lekkim narzędziem w języku Swift, które uprości pracę innym programistom tworzącym aplikacje na iOS rozwiązuając problem jakim jest obsługa formatu EPUB. Na dzień dzisiejszy natywne biblioteki iOS nie zapewniają programistom takiego narzędzia, a publicznie istnieje niewiele rozwiązań, które cieszą się mniejszą lub większą popularnością. Problem parsowania oraz wyświetlenia publikacji w formacie EPUB nie jest trywialny. Parser musi sobie poradzić ze skomplikowaną strukturą i bardzo bogatą zawartością takiego dokumentu, który choć posiada szczegółową specyfikację, to ze względu na urozmaicenie technologii w nim wykorzystanych, czyni pracę parsera znacznie trudniejszą. EPUBKit pokonuje te trudności analizując zawartość i kolektywizując informacje, dzięki czemu finalnie tworzy instancję klasy, która reprezentuje dokument EPUB. Biblioteka dodatkowo dostarcza widok, który można wykorzystać w celu podglądu dokumentu. Cały projekt jest stworzony tak, aby jak najlepiej wpasowywał się w konwencje programowania w Swiftie na platformę iOS i opiera się na wzorcu architektonicznym Model-Widok-Kontroler, który zgodnie z wytycznymi Apple jest szeroko stosowanym przy programowaniu aplikacji na iOS. Biblioteka wykorzystuje w pełni język Swift i wszelkie jego nowinki, które sprawiają że jest językiem bardzo praktycznym a przy tym lekkim, szybkim i przyjaznym. Biblioteka EPUBKit choć nie jest idealna, a na pewno nie będąca w swojej finalnej wersji, to już w aktualnej formie oferowane przez nią API (Application Programming Interface) czyni bibliotekę funkcjonalną oraz konkurencyjną. Ze względu na wciąż ewoluującą oraz szeroką specyfikację formatu EPUB dostarczanie takiej biblioteki, która gwarantowała by taką funkcjonalność oraz była by tak rozwinięta jak, dla przykładu program iBooks na iOS wymagało by ogromnego nakładu pracy ze strony znacznego zespołu programistów. Głęboko wierzę w słuszność ruchu wolnego oprogramowania, dlatego też postępuję zgodnie z jego duchem i decyduję się upublicznić moją pracę, która już w obecnej formie może zaoszczędzić wiele czasu komuś, kto natknie się na problem obsługi dokumentu EPUB, a przy tym może zostać przez tą osobę ulepszona i rozwinięta.

Mobilny system od Apple, iOS niedługo będzie obchodził 10 lat od wprowadzenia go na rynek. W 2015 roku Apple ogłosiło, że dostarczyli konsumentom już ponad miliard urządzeń z tym właśnie systemem a dziś jest ich z pewnością znacznie więcej[10]. Każdy kolejny model telefonu komórkowego marki Apple, prezentowany z roczną częstotliwością cieszy się coraz większym powodzeniem. Oprócz

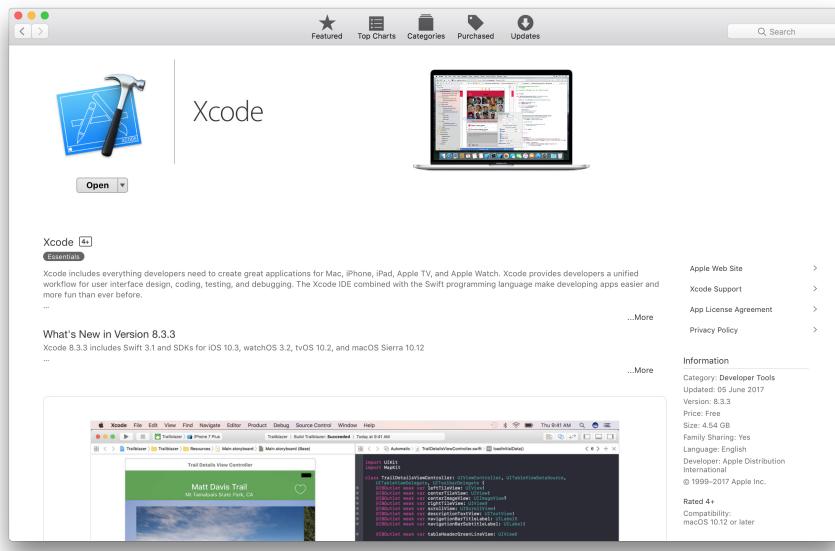
nowych urządzeń dostajemy w pakiecie nową wersję systemu iOS, która jest nie tylko udoskonaleniem poprzedniej wersji, ale również jej pełnoprawnym następcą wprowadzając nowy zbiór zarówno funkcjonalności jak i elementów wizualnych. Tak dynamicznie rozwijający się system jest bardzo atrakcyjny dla użytkownika, który dzięki darmowym aktualizacjom dla starszych urządzeń wciąż może cieszyć się najnowszym oprogramowaniem. iOS Development staje się coraz popularniejszy wśród programistów oraz przyciąga wielu młodych ludzi zainteresowanych technologią. Język Swift, nad którym prace zostały rozpoczęte w 2010 roku przez Chrisa Lattnera oraz wielu innych programistów z Apple, a w 2014 roku miał swój debiut, dzisiaj jest już głównym językiem programowania mobilnych aplikacji na platformę iOS oraz aplikacji *desktopowych* na macOS i wyparł dotychczas używany w tych celach język Objective-C, który swoją historię ma również ścisłe związaną z Apple. Korzenie fundamentalnych frameworków z iOS, takich jak CocoaTouch, sięgają lat 80-tych poprzedniego stulecia, a przez ten czas były bardzo silnie rozwijane i wykorzystywane w systemie macOS. Nowoczesny język oraz potężne SDK (Software Development Kit) stanowią dziś podstawę pracy z aplikacjami na te platformy. Z roku na rok, wraz z nową wersją systemu, Apple uaktualnia istniejące API oraz dostarcza nowe biblioteki zapewniające dostęp do najnowszych elementów systemu.

Ta praca dokumentuje bibliotekę **EPUBKit**, rozpoczynając od dokładnego opisu środowiska, wykorzystanych narzędzi, scharakteryzowano format EPUB i jego specyfikację techniczną, opisano proces tworzenia biblioteki, jej strukturę oraz możliwości dystrybucji biblioteki jako moduł gotowy do wykorzystania przez developerów. Następnie w celu demonstracji funkcjonalności biblioteki opisano proces tworzenia aplikacji z jej wykorzystaniem.

2. Wykorzystane technologie

W świecie technologii pierwszym krokiem do osiągnięcia sukcesu przez system jest przyciągnięcie chętnych, którzy będą tworzyć na nią oprogramowanie. Apple dobrze zdaje sobie z tego sprawę, więc robią wszystko aby zjednać sobie programistów i zapewnić im wygodne, funkcjonalne narzędzia. Nie ma żadnych wątpliwości, że iPhone osiągnął sukces między innymi dzięki dostępnym aplikacjom, które bez programistów z pewnością nigdy by nie powstały. Udostępniane przez Apple narzędzia z pewnością są dalekie od ideału, jednakże dostarczają ogromne możliwości dzięki pełnemu pakietowi, z którego pomocą sprawiają, że jedynym ograniczeniem jest wyobraźnia programisty. W tym rozdziale opiszę narzędzia wykorzystane do tworzenia projektu **EPUBKit**. Są to głównie technologie Apple, co daje pogląd na to jak zaawansowane są ich narzędzia oraz pod koniec rozdziału przedstawiłem dodatkowe biblioteki, które zostały wykorzystane w projekcie.

2.1. Xcode i Developer Tools



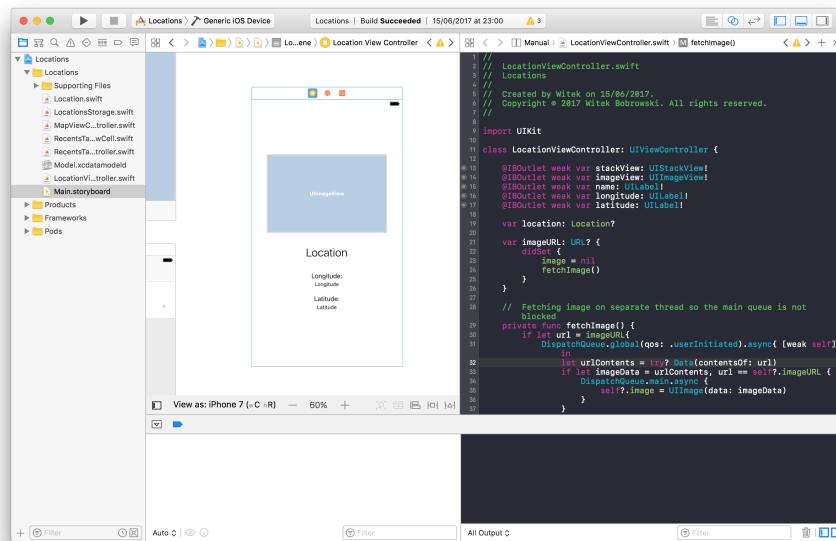
Rysunek 2.1: Xcode w App Store

Xcode jest IDE (Integrated Development Environment) stworzonym przez Apple i dostępnym za darmo do pobrania z App Store (patrz: rysunek 2.1), sklepu z aplikacjami do którego dostęp mają wyłącznie użytkownicy komputerów z systemem Mac OS. Jest wyposażony w pakiet wszystkich narzędzi (Deve-

loper Tools) potrzebnych dla programistów, aby tworzyć aplikacje na platformę iOS. Główną aplikacją pakietu jest Xcode IDE, który wraz z wspomagającymi aplikacjami dostępnymi w pakiecie takimi jak Simulator czy Instruments, czyni pracę przy tworzeniu aplikacji płynną i efektywną. W tym rozdziale przedstawię właśnie te narzędzia ze względu na ich rolę w procesie tworzenia aplikacji.

2.1.1. Xcode IDE

Xcode jako nowoczesne, produktywne środowisko jest miejscem w którym programista aplikacji na iOS spędza większość swojego czasu. Całość prac wykonywanych przy produkcji aplikacji może zostać wykonana właśnie tutaj. Najbardziej podstawowy element jakim jest edytor tekstu dobrze współgra z takimi narzędziami jak Interface Builder, który pozwala w prosty sposób zaprojektować stronę wizualną aplikacji przy użyciu *Storyboardów*, a następnie stworzyć referencję w kodzie do wybranych przez nas elementów przez proste przeciągnięcie myszką. Storyboardy są opcjonalnym, aczkolwiek bardzo pozytycznym narzędziem szczególnie dla programistów stawiających swoje pierwsze kroki na tej platformie. Zapewniają one wizualne wyobrażenie interfejsu aplikacji, nad którą wykonywana jest praca, a projektowanie dowolnego widoku, który będzie wyglądał dobrze na każdym urządzeniu w dowolnej orientacji, jest relatywnie proste po zapoznaniu się z kilkoma elementarnymi zasadami. Interface Builder jest widoczny na rysunku 2.2, po lewej stronie widoku *Assistant editor* w Xcode.



Rysunek 2.2: Xcode i widok *Assistant editor*

Storyboardy tworzy się w jednym pliku o formacie .storyboard. Dlatego często w profesjonalnej produkcji rezygnuje się z nich ze względu na konflikty w systemach kontroli wersji. Konflikty te powstają w wyniku pracy wielu programistów, a ponieważ plik .storyboard jest w rzeczywistości plikiem XML, który został wygenerowany automatycznie, rozwiązywanie konfliktów bywa kłopotliwe, a przy dużych projektach problematyczne. Dlatego rezygnuje się z nich na rzecz tworzenia widoków tylko przy użyciu kodu, oraz niezależnych plików Xib. Pliki te pozwalają na ustawienie elementów w stylu znany ze Storyboardów, lecz w przeciwnieństwie do nich reprezentują pojedynczy widok, dzięki czemu można

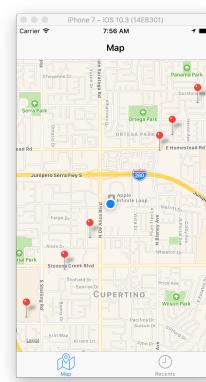
uniknąć konfliktów, a jednocześnie tworzenie bardziej skomplikowanych widoków jest znacznie ułatwione. Xcode zapewnia wsparcie dla systemu kontroli wersji git. Przy tworzeniu nowego projektu, gdy jest o to poproszony, inicjalizuje nowe repozytorium. Dodatkowo w nawigatorze projektu, w którym widać jego strukturę, Xcode oznaczy literą *M* pliki które git oznacza jako pliki w których dokonano zmian (modified), a literą *A* pliki które zostały dodane (new file) od czasu poprzedniego zachowania zmian. W najnowszej wersji 9.0, Xcode zyskał nową funkcjonalność – Source Control Navigator, który pozwala na eksplorowanie poszczególnych gałęzi repozytorium i podglądu dowolnego momentu w jego historii.

2.1.2. Simulator

Simulator pozwala na uruchomienie zbudowanej aplikacji na dowolnym urządzeniu z iOS, które jest w stanie symulować. Gdy chce się przetestować aplikację wystarczy w pasku narzędzi Xcode wybrać dowolny model urządzenia, które chcemy symulować, tak jak przedstawiono to na rysunku 2.3a. Oprócz telefonów iPhone znajdują się tam również tablety iPad. Jeżeli podłączymy do komputera fizyczne urządzenie, Xcode również je wykryje i pozwoli na zainstalowanie i uruchomienie na nim aplikacji. W kolejnym kroku Xcode przejdzie to etapu budowania aplikacji, w którym kompiluje pliki źródłowe, a następnie umieści aplikację w symulatorze wybranego przez nas urządzenia, co jest widoczne na rysunku 2.3b.



(a) Wybór docelowego urządzenia



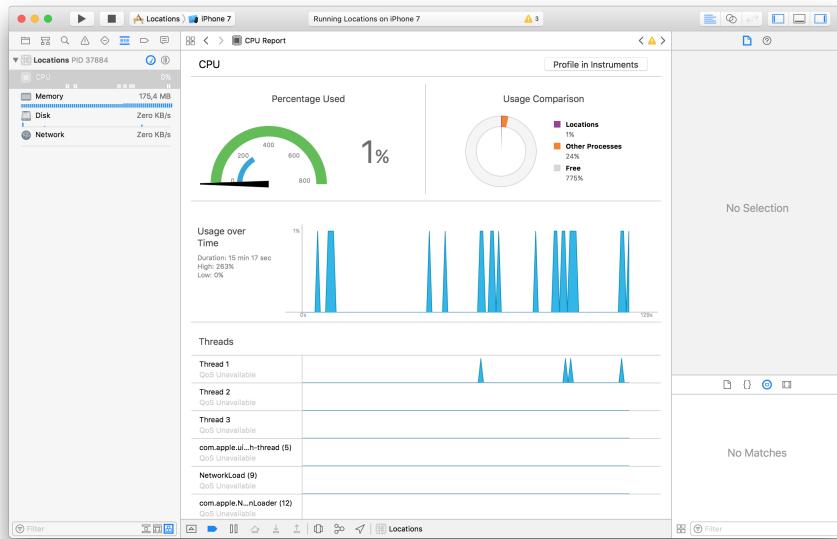
(b) Uruchomiona aplikacja na iPhone 7

Rysunek 2.3: Po wybraniu urządzenia w Xcode, Simulator uruchamia na nim aplikację

Podczas gdy aplikacja jest uruchomiona i testowana na symulatorze, Xcode pozwala na podgląd użycia zasobów takich jak procesor, pamięć RAM, pamięć dyskowa oraz sieć. Po wybraniu dowolnego z wyżej wymienionych zasobów z nawigatora *Debuggera* ukazuje nam się bardziej szczegółowy podgląd na to, w jakim stopniu aplikacja obciąża urządzenie, co pozwala na szczegółowe testowanie (patrz: rysunek 2.4).

2.1.3. Instruments

Widok poboru zasobów w Xcode jest bardzo pomocny na ogólną ocenę wydajności naszej aplikacji. Jeżeli jednak chcemy poddać ją prawdziwej próbie, musimy uruchomić kolejne narzędzie jakim



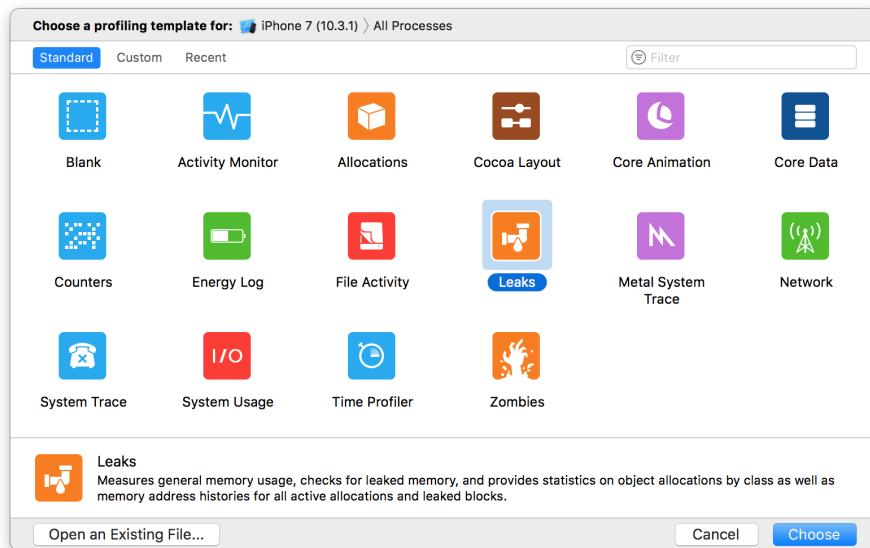
Rysunek 2.4: Użycie CPU na symulatorze odświeżane na bieżąco i wyświetcone w Xcode

jest Instruments. Instruments widoczne na rysunku 2.5 jest aplikacją, dzięki której dokonamy pomiaru nie tylko każdego zasobu na urządzeniu, ale również dostajemy możliwość nadzoru takich aktywności jak alokowanie pamięci dla obiektów, zmiany layoutu widoków czy zmiany w Core Data, natywnej bazie danych dla iOS.

Niezależnie od tego jak szybkie i wydajne są w dzisiejszych czasach telefony, optymalny kod nadal jest podstawą prawidłowego działania aplikacji i niedbałe projektowanie architektury może być fatalne w skutkach. Cykle referencji mogą powodować niechciane wycieki pamięci, a problemy powstające podczas przerysowywania się widoków spowodują nieczytelny interfejs. Jeżeli takie błędy nie zostaną wykryte na etapie produkcyjnym, a dogłębne testy aplikacji nie zostaną przeprowadzone przed oddaniem jej do recenzji, aplikacja najprawdopodobniej zostanie odrzucona, co wiąże się z dodatkowymi opóźniami. Aby nasza aplikacja mogła znaleźć się w sklepie AppStore i być dostępna dla każdego, musi ona otrzymać pozytywną recenzję Apple. Pakiet narzędzi dostarczanych przez Apple spełnia swoje zadanie i dla większości programistów są one wystarczające. Istnieją rozwiązania firm trzecich, JetBrains dostarcza alternatywne IDE do produkcji aplikacji – AppCode, które cieszy się bardzo dobrą reputacją wśród użytkowników.

2.2. Swift

Gdy w 1996 roku Apple przejęło NeXT wraz z ich oprogramowaniem, Objective-C stało się głównym językiem do tworzenia oprogramowania na najnowszy system operacyjny OS X. Wraz z OpenStepem Apple znalazło się w posiadaniu takich narzędzi jak Interface Buildier oraz Project Buildier, które z biegiem czasu ewoluowały w Xcode, który znamy dzisiaj. Objective-C służył programistom przez wiele lat, na początku tym tworzącym aplikacje desktopowe, a od ukazania się pierwszego iPhone'a, programistom mobilnym. Objective-C jest bardzo mocno rozwiniętym językiem, co zważając na jego wiek



Rysunek 2.5: Instruments, widok menu głównego

nie powinno nikogo dziwić. Jednak jako język, którego początki sięgają wczesnym latom osiemdziesiątym dwudziestego wieku, z biegiem lat postarzał się, co bardzo upraszczając można by przedstawić stwierdzeniem, iż nie jest wystarczająco nowoczesny.

Twórca LLVM (Low Level Virtual Machine) Chris Lattner[11], w 2010 roku rozpoczął prace nad nowym językiem, który z założenia miał czerpać inspirację z takich języków jak Objective-C, Haskell, Python czy Ruby. Cztery lata później, Apple na WWDC (Worldwide Developer Conference) zaprezentowało Swifta i wypuściło wersję beta[1]. Swift jest bezpiecznym, szybkim językiem, który agreguje wiele paradygmatów znanych z innych nowoczesnych języków. Jego najważniejszymi cechami są:

- automatycznie zarządzana pamięć,
- typ opcjonalny, który zapewnia wsparcie dla /textit{nullowych} wskaźników,
- zaawansowana obsługa błędów, która zapewnia kontrolowane wyjście z niespodziewanych niepowodzeń,
- zmienne, które zawsze są zainicjalizowane przed użyciem,
- silne typowanie które po połączeniu z nowoczesną, lekką składnią czyni go językiem zarówno potężnym jak i łatwo dostępnym.

Dynamiczny rozwój zawdzięcza inżynierom z Apple oraz społeczności open-source, którzy bardzo intensywnie pracują dodając nowe elementy, naprawiając niedoskonałości oraz optymalizując jego architekturę. Swift dzisiaj jest jednym z najbardziej lubianych języków programowania. Jego popularność wciąż rośnie, a odkąd stał się projektem open-source'owym oprócz systemów Apple, zaczyna być używany do takich celów jak tworzenie aplikacji internetowych. Zdominował platformę iOS i zdetronizował Objective-C, stając się językiem pierwszego wyboru. Wszystkie powstające aplikacje są obecnie tworzone przy jego użyciu, i wiele napisanych w Objective-C jest dzisiaj przepisywanych na nowy język. Warto

wspomnieć, że kod w Objective-C i kod w Swiftcie może być użyty w tej samej aplikacji, dzięki czemu może się tam znaleźć również C i C++. Jednak, aby użyć kodu napisanego w C czy w C++ w Swiftcie musimy go najpierw opakować, aby był dla niego dostępny. W przypadku Objective-C, Swift ma dostęp do jego obiektów oraz może dziedziczyć z jego klas. A to zapewnia nam dostęp do wszystkich natywnych bibliotek, czyniąc Swift pełnoprawnym językiem i narzędziem gotowym do użytku już w dniu jego premiery. W projekcie wykorzystano Swift w wersji 3.1.

2.3. iOS SDK

Jak już wspomniano wcześniej, iOS dziedziczy wiele po systemie macOS. W skład iOS SDK wchodzą biblioteki znane już od wielu lat oraz biblioteki stworzone z myślą o urządzeniach mobilnych. W tej sekcji omówiono pokrótkę framework CocoaTouch, który dostarcza elementy interfejsu będące podstawą każdej aplikacji i zapewniające wizualną spójność platformy.

2.3.1. CocoaTouch

CocoaTouch jest potomkiem frameworku Cocoa dostępnego na systemach macOS, który jest rozszerzony o interfejs obsługi narzędzi dostępnych w urządzeniu mobilnym takich jak rozpoznawanie gestów, serwis lokalizacji czy obsługa kamery. W skład CocoaTouch wchodzą między innymi biblioteki Foundation, UIKit, MapKit, EventKit i wiele innych. Dzięki temu pakietowi Apple zdefiniowało jak powinny być tworzone aplikacje na iOS. Dostarczony jest zbiór wielu elementów interfejsu użytkownika, które można dowolnie rozszerzać i modyfikować, aby stworzyć unikalny wygląd aplikacji trzymając się wtycznych wyznaczonych przez Apple. Dostęp do gestów zapewni naszej aplikacji lekkość obsługi oraz intuicyjność, a niezliczona ilość innych bibliotek wchodzących w skład CocoaTouch sprawi, że aplikacja nabierze życia. Implementowanie funkcjonalności staje się bardzo proste dzięki wysoko poziomowym interfejsom dającym dostęp do poszczególnych elementów systemu oraz fizycznego urządzenia. CocoaTouch jest najbardziej elementarnym frameworkm na iOS, ponieważ to on zapewnia na poziomie podstawowym to co potrzebne do stworzenia funkcjonalnej aplikacji. Zakładając że aplikacja powinna agregować, w jakiś sposób przetwarzać, a następnie wyświetlić w odpowiedniej formie zbiór pewnych danych, CocoaTouch w parze ze Swiftem to zagwarantują, oczywiście do pewnego stopnia. W większości przypadków, gdy potrzebujemy wykonać jakieś specyficzne zadanie, lub chcemy je po prostu uprościć unikając implementacji własnego rozwiązania problemu, co było by czasochłonne, będziemy sklaniać się ku zewnętrznym bibliotekom.

2.4. Zewnętrzne biblioteki

Biblioteki te, tworzone przez środowiska open-source'owe, pojedynczych programistów czy wielkie korporacje dadzą nam to czego nie otrzymaliśmy wraz z natywnymi frameworkami. Przykładem jest projekt który opisuje ta praca. Rozwiązuje problem programisty tworzącego własną aplikację, który nie chce tracić czasu na implementowanie funkcjonalności, która zajęła by mu relatywnie dużo czasu, a która nie-

koniecznie jest głównym celem jego aplikacji. Istnieje wiele sposobów na rozszerzenie naszej aplikacji o wybrane moduły, które bardziej szczegółowo zostaną opisane pod koniec czwartego rozdziału podczas omawiania możliwościach dystrybucji biblioteki. W tej sekcji przedstawiono biblioteki wykorzystane do projektu EPUBKit, które nie wchodzą w skład iOS SDK, a zostały udostępnione do publicznego użytku.

2.4.1. Zip

Zip jest swiftową biblioteką open-source dostępną na licencji MIT, której kod źródłowy znajdziemy na GitHubie¹. Zip jest narzędziem do archiwizowania plików oraz rozpakowywania archiwów. Wspiera on formaty archiwów .zip, .cbz oraz daje możliwość dodania rozszerzenia pliku który chcemy rozpakować. W przypadku tego projektu dodano format .epub, aby rozpakować książkę elektroniczną w tym właśnie formacie. Zip bazuje na narzędziu minizip napisanym w języku C, również na wolnej licencji, oraz dostępnym na portalu GitHub. Dzięki bibliotece zip, parser biblioteki EPUBKit może w prosty sposób otrzymać dostęp do struktury plików publikacji elektronicznej. Szczegółowe opisanie wykorzystania biblioteki Zip znajduje się w czwartym rozdziale podczas omawiania kodu źródłowego parsera.

2.4.2. AEXML

Kolejną wykorzystaną biblioteką jest AEXML, prosty i lekki parser XML, dostępny publicznie na licencji MIT, również udostępniony na GitHubie². Ze względu na strukturę elektronicznej publikacji EPUB, parser XML jest niezbędny do rozpoznania zawartości całej struktury dokumentu, identyfikacji elementów oraz określenia ich lokalizacji. AEXML jest swiftowym frameworkm, a jego API jest czytelne i proste w wykorzystaniu, dzięki czemu idealnie spełnia swoje zadanie w projekcie który opisuje ta praca.

¹Adres url: github.com/marmelroy/Zip

²Adres url: github.com/tadija/AEXML

3. Charakterystyka EPUB

EPUB jest standardem formatu dystrybucji cyfrowych publikacji i dokumentów opartych na standarach technologii webowej. EPUB definiuje formę reprezentacji, organizacji struktury oraz kodowania określonej zawartości webowej, na co składają się XHTML, CSS, SVG, obrazy i inne zasoby sprowadzone do formy pojedynczego pliku. EPUB daje wydawcom możliwość stworzenia cyfrowej publikacji, a następnie dystrybuowania go, a odbiorcy łatwy dostęp do pliku niezależnie od urządzenia jakim操作. Jako następca OEB (Open eBook Publication Structure, zaprezentowanego w 1999 roku, EPUB 2 został ustanowiony w roku 2007, a aktualną jego wersją jest EPUB 3.1 (styczeń 2017). Dzisiaj jest on standardem wykorzystywany na szeroką skalę przez wszystkich wydawców. Obok MOBI oraz PDF dominuje na rynku, dzięki jego popularności wśród wydawców oraz wsparciu urządzeń. W przeciwieństwie do MOBI, które zostało spopularyzowane przez Amazon, właściciela sklepu amazon.com, giganta dystrybucji książek elektronicznych oraz producenta czytników elektronicznych marki Kindle, EPUB jest standardem uniwersalnym, nieograniczonym do jednej platformy. EPUB zarówno jak i MOBI charakteryzuje się tym, że jego zawartość nie jest statyczna, co oznacza, że ilość która jest wyświetlana dopasowana jest do wielkości ekranu urządzenia, dzięki czemu jest ona bardziej przyjazna dla odbiorcy. PDF natomiast jest już podzielony na strony których nie da się podzielić. To co najbardziej różni EPUB od MOBI, to wsparcie EPUB dla multimedialnych (od wersji 3.0) oraz CSS, który stylizuje cały dokument. Dzięki temu jest znacznie bardziej elastyczny i nowoczesny. Popularną praktyką wśród dystrybutorów książek elektronicznych, jest dostarczanie książki klientowi, który ją zakupił we wszystkich trzech wcześniej wymienionych formatach. EPUB jako standard jest szeroko udokumentowany dzięki International Digital Publishing Forum¹, grupie która nadzoruje rozwój formatu. W następnej sekcji zostanie szczegółowo opisana struktura formatu EPUB.

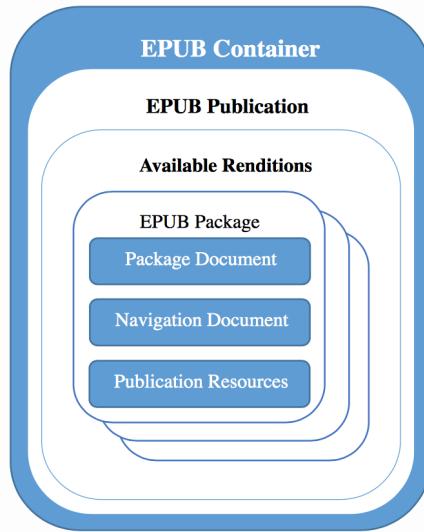
3.1. Specyfikacja

Poniższa specyfikacja jest podziobrem najważniejszych informacji wyselekcjonowanych ze specyfikacji EPUB 3.1 z dnia 5-go stycznia 2017 roku, dostępnej na stronie International Digital Publishing Forum². Przedstawiono tutaj najbardziej istotne elementy formatu EPUB w celu zrozumienia problemu jakiego dotyczy projekt EPUBKit.

¹Adres url: idpf.org

²Adres url najnowszej wersji specyfikacji: <http://www.idpf.org/epub3/latest>

3.1.1. EPUB Open Container Format



Rysunek 3.1: Wizualna reprezentacja struktury formatu EPUB[4]

Format EPUB definiuje jego najwyższy abstrakcyjny model jakim jest EPUB Publication (patrz: rysunek 3.1). Model ten składa się z interpretacji jego zawartości. Interpretacja, która jest przedstawiona za pomocą EPUB Package, zawiera już bezpośrednio zawartość dokumentu, oraz poboczne zasoby mające na celu wspomagać system czytający (z ang. ze specyfikacji *Reading System*, jakim jest EPUBKit). Kluczowym elementem jest Package Document, który zawiera wszystkie metadane, które są używane następnie przez system czytające do zaprezentowania publikacji użytkownikowi. Zawiera on również kompletny manifest zasobów publikacji oraz *kregostup* (z ang. ze specyfikacji *Spine*), który reprezentuje sekwencję w jakiej system czytający ma wyświetlać poszczególne elementy. EPUB Package zawiera również Navigation Document pełniący rolę spisu treści, przeznaczony dla użytkownika do poruszania się po dokumencie. Wszystko to opakowane jest w archiwum ZIP z rozszerzeniem .epub. Rozszerzenie informuje o charakterze pliku oraz dostarcza informację o archiwum w ZIPowskim stylu za pomocą pliku *mimetype*, oraz zapewnia system o posiadaniu przez niego folderu /META-INF, w którym dostępny jest plik *container.xml*, niezbędny systemowi do określenia lokalizacji zawartości publikacji.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <container version="1.0" xmlns="urn:oasis:names:tc:opendocument:xmlns:container">
3   <rootfiles>
4     <rootfile full-path="OEBPS/content.opf" media-type="application/oebps-
      package+xml"/>
5   </rootfiles>
6 </container>
  
```

Listing 3.1: Przykładowy plik container.xml

To w jaki sposób zawartość publikacji jest zorganizowana, określa standard EPUB Open Container Format (OPF), który definiuje reguły enkapsulacji zasobów w pojedynczym kontenerze abstrakcyjnym (EPUB Container) zawartym w archiwum ZIP. Struktura OPF to tylko jedna część składająca się na

EPUB Publication, druga część to zawartość przedstawiona użytkownikowi, która jest oparta o dokumenty określone w specyfikacji jako EPUB Content Documents. Zawartość ta jest rozszerzona o wiele dodatkowych zasobów potrzebnych do prawidłowego wyświetlenia publikacji jakimi mogą być obrazy, pliki audio lub video, dodatkowe czcionki, skrypty oraz style nazywane w oficjalnej specyfikacji *EPUB Core Media Types*.

3.1.2. EPUB Content Documents

Ta sekcja opisuje rolę jaką odgrywają standardy HTML, SVG i CSS w elektronicznej publikacji w formacie EPUB. Wizualna kompozycja publikacji w znacznej mierze oparta jest o pliki XHTML. Specyfikacja EPUB Content Documents 3.1 szczegółowo opisuje semantykę atrybutów wspieranych przez EPUB, które mają na celu wzbogacenie doświadczenia użytkownika. Tak jak pokazano na przykładzie 3.2 atrybuty te nadają dodatkową naturę i znaczenie elementom XHTML, przy tym nie nadpisując ich pierwotnej funkcjonalności. Atrybuty te są przeznaczone wyłącznie dla systemów czytających i przekazują istotne informacje odnośnie struktury i zawartości dokumentu. Wsparcie standardu EPUB dla HTML nie odnosi się do konkretnej jego wersji, natomiast pozostawia tę kwestię twórcom systemów czytających. To w ich obowiązkach leży upewnienie się, że każda publikacja zostanie prawidłowo przez system obsłużona.

```
1 <html ... xmlns:epub="http://www.idpf.org/2007/ops">
2   ...
3   <p> ... <span epub:type="pagebreak" title="234" id="p234"/> ...
4   ...
5 </html>
```

Listing 3.2: Przykładowe wykorzystanie atrybutu epub:type aby oznaczyć zakończenie linii.[5]

Kolejnym typem dokumentu ważnym dla formatu EPUB jest SVG (Scalable Vector Graphics). Co prawda uniwersalność XHTML przyczynia się do tego iż jest to w znacznej mierze dominujący format przedstawiania treści w publikacji, SVG oferuje udogodnienia dzięki czemu również znajduje zastosowanie. Format ten jest zazwyczaj wykorzystywany w specyficznych przypadkach, takich jak prezentacja zawartości publikacji z gatunku manga czy komiks. Użytkownik oczekwał by od tego typu dokumentu, aby zawartość w tym przypadku głównie graficzna, prezentowała się dobrze na każdym urządzeniu niezależnie od rozdzielczości jego ekranu, i to właśnie SVG może zagwarantować. Dodatkowo, treść tych dokumentów jest zazwyczaj z góry podzielona na strony, i narzuca systemom czytającym wyświetlenie ich w określonej formie.

CSS odgrywa ogromną rolę w praktyce programistycznej jako technologii webowej od wielu lat, i jego możliwości wciąż rosną. Nietrudno się domyślić, że swoje miejsce CSS znajduje również w standardzie EPUB. Prezentacja publikacji jest sztywnie ustylizowana, chociaż nadpisywanie stylu przez systemy czytające nie jest zabronione. Specyfikacja EPUB sugeruje, aby systemy dawały możliwość zmian w stylu publikacji użytkownikowi, który dopasuje jej wygląd pod swoje upodobania. Zazwyczaj będzie to kolor tła, rozmiar czy typ czcionki. Specyfikacja również przestrzega twórców publikacji w formacie

EPUB, aby rozważnie stylizować dokument. Niektóre systemy czytające mogą nie wspierać pewnych elementów CSS co może być problematyczne [5].

To o czym również należy wspomnieć opisując EPUB Content Documents jest to, że EPUB wspiera skrypty w dokumentach HTML czy SVG. Opcjonalne jest wspieranie skryptowania w systemach czytających, natomiast te systemy, które wspierają skryptowane dokumenty, muszą wziąć pod uwagę pewne niebezpieczeństwa z tym związane. Specyfikacja przestrzega, aby systemy czytające zapewniały izolację publikacji w celu uniknięcia niebezpieczeństw zagrażającym podatnej na ataki zawartości. System powinien monitorować wszelkie aktywności, aby zawartość pozostała nienaruszona. Należy założyć, że może zostać przeprowadzony atak na zawartość innych plików w strukturze publikacji, atak na sam system czytający, np. próba zdobycia danych użytkownika, lub atak na sieć czy wszechepianie innych złośliwych skryptów w niezaszyfrowane fragmenty dokumentu. Ponadto systemy które zezwalają na stałe przechowywanie publikacji muszą dostarczyć metody zezwalające na podgląd i kasowanie danych publikacji. W sytuacji usunięcia publikacji przez użytkownika, system musi skasować wszelkie pliki z nią związane [5].

3.1.3. EPUB Core Media Types

Zaobserwować można, że EPUB jako format jest bardzo ściśle wyspecyfikowany odnośnie jego struktury i wspieranych technologii. W przypadku samej zawartości nie jest inaczej. Specyfikacja EPUB 3 Core Media Types³ szczegółowo opisuje jakie typy mediów mogą zostać załączone do publikacji i określa dla nich unikalny typ (*Media Type*), który jest wykorzystywany w celach poinformowania systemu czytającego o typie danego elementu. Ta sekcja opisuje poszczególne typy mediów wspierane przez EPUB, nie zagłębiając się przy tym w techniczne szczegóły konkretnego typu.

Image Types

Wspierane typy obrazów (*Image Types*) to GIF, JPEG, PNG oraz SVG. Ich typy zadeklarowane w standardzie EPUB to kolejno *image/gif*, *image/jpeg*, *image/png*, *image/svg+xml*. Notacja ta wykorzystywana jest w atrybutach elementów wskazujących na konkretny obiekt odpowiedniego typu. Oznaczenia te możemy spotkać w manifeście publikacji.

```
1 <manifest>
2 ...
3   <item href="Images/image-1.jpg" id="id1" media-type="image/jpeg"/>
4   <item href="Images/image-2.jpg" id="id2" media-type="image/jpeg"/>
5 ...
6 </manifest>
```

Listing 3.3: Przykładowy fragment manifestu znajdującego się w pliku content.opf

Sposób oznaczenia elementów zademonstrowany na przykładzie 3.3, wykorzystywany jest w deklarowaniu każdego obiektu o wspierany przez EPUB typie.

³Adres url: <https://idpf.github.io/epub-cmt/v3/>

Audio Types

EPUB wspiera pliki audio w formacie MP3 (*audio/mpeg*) oraz MP4 (*audio/mp4*). Zapewne zdecydowano się na te dwa konkretne standardy ze względu na ich popularność oraz mały rozmiar dzięki stosunkowo wysokiej kompresji.

Video Types

Ze względu na rozbieżność preferencji wśród autorów oraz twórców specyfikacji, EPUB nie definiuje preferowanego typu wideo dla publikacji w swoim standardzie. IDPF proponuje dwa formaty kodowania: H.264 oraz WebM, ale nie kładzie nacisku na jeden z nich. Tę kwestię pozostawia na ten moment autorom. Spór dotyczący określenia konkretnego formatu plików wideo rozchodzi się o szerokie wsparcie H.264 przeciwko nieopatentowanemu, wolnemu formatowi WebM, który jednak wciąż nie jest spopularyzowanym standardem. IDPF deklaruje, że najprawdopodobniej w pewnym momencie preferowany standard zostanie określony, jednakże teraz jest jeszcze na to zbyt wcześnie [6].

Application, Text, Font Types

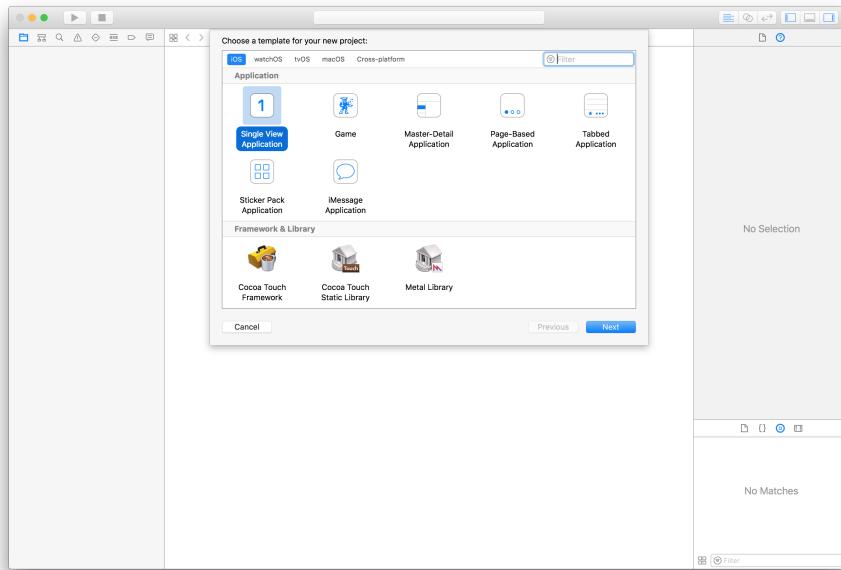
Pozostałe typy które wspiera EPUB zdecydowano się opisać razem w tej sekcji w skrócie ze względu na to, że wszystkie służą jednemu celowi jakim jest prezentacja treści. Są to czcionki w formacie WOFF(*application/font-woff*), WOFF2(*font/woff2*) oraz OpenType/TrueType(*application/font-sfnt*), pliki CSS(*text/css*), skrypty(*application/javascript*), dokumenty XHTML(*application/xhtml+xml*), nawigacyjne pliki NCX(*application/x-dtbncx+xml*) oraz nakładki MediaOverlays3(*application/smil+xml*) wraz z plikami PLS(*application/pls+xml*), które mają na celu odtworzenia tekstu w formie dźwiękowej[3].

Foreign Resources

Należy wspomnieć, iż EPUB zezwala na użycie niewspieranych formatów, jednakże nie gwarantuje, że zostaną prawidłowo obsłużone przez systemy czytające, od których wymaga jedynie wsparcia wcześniej wymienionych formatów. W przypadku użycia przez autora publikacji niewspieranych formatów, specyfikacja EPUB wymaga od niego zadeklarowanie tak zwanego *Core Media Type fallback*, czyli typu wspieranego, który zastępco będzie przypisany do obiektu jeżeli system czytający nie będzie wspierał pierwotnie wskazanego typu [4].

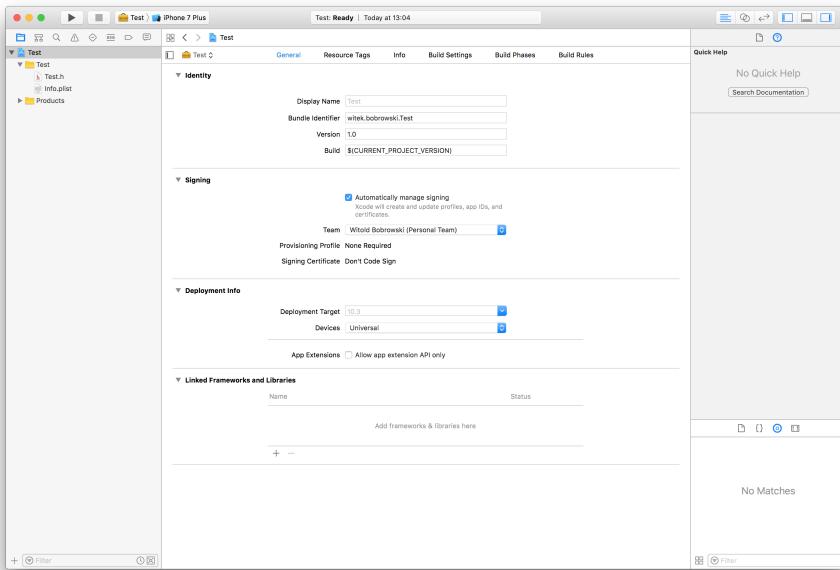
4. Biblioteka EPUBKit

Tworzenie biblioteki, którą zamierzamy następnie wykorzystać we własnej aplikacji lub udostępnić publiczne, jest stosunkowo prostym rozwiązaniem. Wszystko sprowadza się do stworzenia nowego projektu w Xcode, a następnie dołączenie go do przestrzeni roboczej (Xcode Workspace) w której znajdzie się projekt aplikacji nad którą pracujemy oraz projekt biblioteki. Aby stworzyć projekt biblioteki należy uruchomić Xcode IDE i na powitalnym ekranie wybrać opcję *Create a new Xcode project*, która przeniesie nas do kolejnego ekranu z możliwością wybrania konkretnego szablonu projektu nad jakim chcemy pracować, tak jak jest to zaprezentowane na ilustracji 4.1.



Rysunek 4.1: Szablony projektów które znajdują się w Xcode

W naszym przypadku interesuje nas szablon *Cocoa Touch Framework*, a po jego wybraniu jesteśmy proszeni o uzupełnienie formularza ze szczegółowymi informacjami na temat projektu, który zamierzamy stworzyć, poczynając od nazwy, po język w którym będzie napisany, tj. Swift lub Objective-C. Następnie Xcode poprosi o wskazanie lokalizacji na dysku, w której chcemy zapisać projekt, oraz zapyta nas czy chcemy stworzyć repozytorium systemu kontroli wersji git. W tym momencie mamy projekt, który można już w prosty sposób dołączyć do aplikacji. Zostanie to dokładnie opisane w kolejnym rozdziale przy okazji omawianie wykorzystania biblioteki EPUBKit w demonstracyjnej aplikacji. Teraz już możemy tworzyć klasy, które mają składać się na funkcjonalność biblioteki.



Rysunek 4.2: Widok nowego, pustego projektu

Plik projektu, którego widok jest widoczny na ilustracji 4.2 pozwala nam na szczegółowygląd w preferencje oraz informacje jego dotyczące, które można zmienić w dowolnej chwili. Jest możliwość zmiany wersji biblioteki, docelowego urządzenia, preferowanej wersji systemu który ma wspierać bibliotekę, oraz co bardzo istotne, mamy możliwość użycia innych niezależnych bibliotek w naszym projekcie. Wystarczy wyeksportować plik projektu jako plik przestrzeni roboczej do której można dodać inne projekty, a następnie połączyć je z naszym w menu *General* w polu *Linked Frameworks and Libraries* w pliku projektu. Dzięki tak funkcjonalnym i prostym w obsłudze narzędziom jak Xcode, po szybkiej konfiguracji swoją uwagę można skupić na samej logice którą chcemy zaimplementować.

W tym rozdziale zostanie opisana stworzona przez mnie biblioteka EPUBKit. Biblioteka ta jest oparta na architekturze MVC (Model View Controller), dlatego omówienie zaczyna się od opisania jej modelu oraz parsera, a następnie przejdę do zawartych w niej widokach, które pozwalają na wyświetlenie danych zawartych w modelu. Zakończę rozdział przedstawiając możliwości dystrybuowania takiej biblioteki, przy pomocy szeroko stosowanych i popularnych narzędzi, które są podstawą programowania na platformę iOS.

4.1. Model

Struktura klas modelu zaprezentowana na wykazie 4.1, została zaprojektowana w ten sposób, aby z jednej strony odzwierciedlała strukturę dokumentu EPUB oraz format OPF który EPUB wykorzystuje, a z drugiej by trzymała się konwencji „swiftowych” i intuicyjnie reprezentowała instancję, która następnie będzie wykorzystana w kolejnych klasach biblioteki.

```
1 Model
2 |-- EPUBDocument.swift
3 |-- EPUBManifest.swift
```

```
4 | -- EPUBMetadata.swift  
5 | -- EPUBSpine.swift  
6 '-- EPUBTableOfContents.swift
```

Listing 4.1: Struktura modelu EPUBKit

4.1.1. EPUBDocument

EPUBDocument jest klasą publiczną reprezentującą całą publikację EPUB i agregującą pozostałe struktury modelu biblioteki jako jej stałe własności. Ze względu na nomenklaturę stosowaną w języku Swift, *properties* będą nazywane własnościami). Tak jak przedstawiono na wykazie 4.2, w języku Swift odróżniamy dwa rodzaje własności, są to stałe `let` do których wartość może zostać przypisana wyłącznie jednokrotnie oraz zmienne `var` których wartość może być modyfikowana w dowolnym momencie.

```
let (constant name): (type) = (expression)  
var (variable name): (type) = (expression)
```

Listing 4.2: Deklaracje własności w Swifcie.[8]

Własności zostały oznaczone jako stałe ze względu na statyczną naturę struktury publikacji EPUB. Ciężko sobie wyobrazić powód dla którego któreś z metadanych publikacji miały by zostać zmienione albo któreś z dokumentów XML usunięte z manifestu publikacji. Dlatego biorąc pod uwagę kontekst, w którym klasa się znajduje, zdecydowałem się oznaczenie jej własności jako stałe, co jest widoczne na wykazie 4.3, aby zapewnić klasie niemodalność oraz zgodność z wytycznymi odnośnie projektowania klas i struktur w języku Swift, według których powinno oznaczać się własności jako stałe w każdej sytuacji, która nie wymaga od nas ich mutowania. Słowa „mutowania” użyłem tutaj nie bez powodu, co stanie się jasne w kolejnym akapicie.

```
1 public class EPUBDocument {  
2     public let directory: URL  
3     public let contentDirectory: URL  
4     public let metadata: EPUBMetadata  
5     public let manifest: EPUBManifest  
6     public let spine: EPUBSpine  
7     public let tableOfContents: EPUBTableOfContents  
8 }
```

Listing 4.3: Klasa EPUBDocument i jej stałe publiczne

W przeciwieństwie do C, struktury w Swiftie mogą posiadać metody. W przypadku, gdy metoda w jakiś sposób zmienia własności, musi ona zostać oznaczona słowem kluczowym `mutating` co tyczy się również metod enumeracji. Wspominam o tym, ponieważ chciałbym wytlumaczyć dlaczego typy własności klasy EPUBDocument są strukturami, a nie klasami. Ze względu na to, że instancje klasy są przekazywane przez referencję, a instancje struktur są przekazywane przez kopowanie wartości, oznacza to że są one przeznaczone do innych zadań. Zgodnie z wytycznymi Apple, strukturami powinno oznaczać się typy, których zadaniem jest enkapsulacja relatywnie prostych wartości [8], co jest prawdą w przypadku wcześniej wspomnianych typów, a które zostaną one opisane w kolejnych paragrafach.

Klasa EPUBDocument posiada dwa inicjalizatory widoczne na wykazie 4.4, które pozwalają tworzyć instancje tej klasy. Pierwszym z nich jest inicjalizator prywatny w nomenklaturze swiftowej *memberwise initializer* ze względu na kolejność argumentów które przyjmuje. Jest ona zgodna z kolejnością deklaracji własności. Inicjalizator ten został oznaczony jako prywatny, ponieważ jego przeznaczeniem jest inicjalizować instancję jedynie przy pomocy drugiego inicjalizatora. Drugi z inicjalizatorów jest dostępny publicznie i jest jedynym publicznym inicjalizatorem dla tej klasy.

```

1 private init (directory: URL, contentDirectory: URL, metadata: EPUBMetadata,
               manifest: EPUBManifest, spine: EPUBSpine, toc: EPUBTableOfContents) {
2     self.directory = directory
3     self.contentDirectory = contentDirectory
4     self.metadata = metadata
5     self.manifest = manifest
6     self.spine = spine
7     self.tableOfContents = toc
8 }
9
10 public convenience init?(named: String) {
11     let parser = try? EPUBParser(named: named)
12     guard let directory = parser?.directory,
13         let contentDirectory = parser?.contentDirectory,
14         let metadata = parser?.metadata,
15         let manifest = parser?.manifest,
16         let spine = parser?.spine,
17         let tableOfContents = parser?.tableOfContents else { return nil }
18     self.init(directory: directory, contentDirectory: contentDirectory, metadata:
19             metadata, manifest: manifest, spine: spine, toc: tableOfContents)
20 }
```

Listing 4.4: Inicjalizatory klasy EPUBDocument

Zważając na naturę publikacji EPUB jako spójnej całości, zdecydowałem ograniczyć się inicjalizowaniem klasy EPUBDocument do inicjalizatora pomocniczego, który wykorzystuje do tego parser. Ten inicjalizator wykorzystuje w pełni możliwości Swifta. Oznaczając go słowem kluczowym *convenience*, zmuszam go do wykorzystania wyznaczonego (ang. designated) inicjalizatora, ponieważ pomocniczy inicjalizator nie może samemu tworzyć instancji, musi do tego wykorzystać wyznaczony inicjalizator. W tym przypadku jest to pierwszy inicjalizator, który jest prywatny. Dodatkowo pomocniczy inicjalizator, jest oznaczony znakiem zapytania `init?` co oznacza, że inicjalizacja może się nie powieść, a w takiej sytuacji inicjalizator zwróci...nic, czyli `nil` w Swiftie. Konsekwencją tego jest to, że typ który zwraca ten inicjalizator to `EPUBDocument?`, a nie `EPUBDocument`, co oznacza że może on nie mieć żadnej wartości, co trzeba w odpowiedni sposób obsłużyć. Przeanalizujmy więc krok po kroku operacje, które wykonuje inicjalizator pomocniczy.

```
let parser = try? EPUBParser(named: named)
```

Listing 4.5: Inicjalizacja EPUBParser

Inicjalizator zaczynając od instrukcji widocznej na wykazie 4.5, tworzy instancję parsera, i jako argument inicjalizatora podaje własny parametr który wskazuje na nazwę publikacji EPUB. Słowo kluczowe `try` oznacza, że inicjalizator może zwrócić błąd, a dzięki znakowi zapytania błąd ten gdy zostanie rzucony, będzie interpretowany jako zwrócenie wartości `nil` przez inicjalizator. W ten sposób unikamy umieszczenia bloku `do-catch`, co znacznie upraszcza kod. Na końcu znajdujemy się w posiadaniu stałej `parser`, która jest typu `EPUBParser?` czyli opcjonalny `EPUBParser`.

```
guard let directory = ... else { return nil }
```

Listing 4.6: Wyrażenie `guard let`

Wyrażenie `guard let` przedstawione na wykazie 4.6 jest jednym ze sposobów obsłużenia typu opcjonalnego. Jest to odmiana wyrażenia `if let`, które pozwala nam na przypisanie wartości zmiennej `a`, do nowej stałej `b`, jeżeli zmienna `a` takową posiada. Wadą takiego rozwiązania jest to, że nowo powstała zmienna `b`, znajduje się jedynie w zasięgu bloku `if`, co w pewien sposób ogranicza dostęp do niej. Z pomocą przychodzą wyrażenia `guard`, dzięki którym zadeklarujemy nową stałą, która będzie przyjmowała wartość zmiennej, którą chcemy *rozpakować* (ang. *unwrap* – co odnosi się do czynności wywłaszczenia wartości z typu opcjonalnego) i będzie ona dostępna w obrębie tego samego bloku co wyrażenie `guard`. Dodatkowo mamy możliwość wykonania jakiejś czynności w sytuacji gdy zmienna, którą rozpakowujemy nie ma wartości, co w tym konkretnym przypadku będzie oznaczało niepowodzenie wywłaszczenia którejś z wartości parsera, a więc inicjalizator `EPUBDocument` zwróci `nil`. Wyrażenie `guard` działa w podobny sposób co `if`, dlatego otrzymuje on również te same funkcjonalności co `if` w Swiftie, czyli możliwość kolejkowania wyrażeń zwracających wartość boolowską, tj. wypisujemy je kolejno po przecinku. W przypadku zwrócenia fałszu przez jedno z nich, instrukcja natychmiast zostaje przerwana, a pozostałe wyrażenia nie zostają ewaluowane. W tym przypadku program przechodzi do bloku `else`.

```
self.init(...)
```

Listing 4.7: Inicjalizacja danymi z parsera

Jeżeli udało się wywalczyć wszystkie potrzebne wartości z parsera, to można przejść do tworzenia instancji `EPUBDocument`. Inicjalizator pomocniczy wywołuje inicjalizator wyznaczony (wykaz 4.7), i dokument zostaje pomyślnie stworzony, a wszystkie informacje otrzymane dzięki parserowi zostają przypisane na stałe do jednej instancji `EPUBDocument`. W ten sposób zostaje zachowana niemutowalność instancji oraz gwarancja, że wszystkie wartości w których posiadaniu znajduje się instancja, pochodzą z jednego źródła, z którego czerpie parser. Omówienie działanie samego parsera zostawiam na kolejny podrozdział.

4.1.2. EPUBManifest

Jak już wspomniano w rozdziale opisującym format EPUB, jego struktura jest oparta o standard OPF a to oznacza, że znajduje się w nim wykaz (manifest) wszystkich dokumentów oraz zasobów na które składa się dokument. Każdy element wymieniony w manifeście posiada swoje ID, wskazaną ścieżkę w

strukturze dokumentu oraz typ. Parser starannie analizuje manifest i tworzy strukturę w której posiadaniu następnie znajduje się instancja EPUBDocument, o czym więcej przy okazji omawiania parsera.

```
1 public struct EPUBManifest {
2     public struct Item {
3         public var id: String
4         public var path: String
5         public var mediaType: EPUBMediaType
6         public var property: String?
7     }
8
9     public var id: String?
10    public var items: [String:Item]
11
12    public func path(forItemWithId id: String) throws -> String {
13        if let item = items[id] {
14            return item.path
15        } else {
16            throw EPUBParserError.noPathForItem(id)
17        }
18    }
19 }
```

Listing 4.8: Struktura EPUBManifest

Przedstawiona na wykazie 4.8 struktura EPUBManifest deklaruje własną strukturę Item, na którą składa się kilka własności opisujących daną pozycję w manifeście. EPUBManifest posiada dwie własności. Pierwszą z nich jest opcjonalne id manifestu, które może się pojawić w publikacji EPUB, ale w specyfikacji nie jest określone jako wymagane pole. Drugą własnością jest słownik, którego kluczem jest id elementu, a wartością jest instancja struktury Item. Manifest nie musi być listą posortowaną, dlatego zdecydowałem się na użycie słownika jako struktury danych która ma przechowywać wszystkie jego elementy. Dzięki temu dostęp do nich jest natychmiastowy. Należy zwrócić uwagę na to, że nie został zadeklarowany inicjalizator dla EPUBManifest, tak jak miało to miejsce przy EPUBDocument. Powód jest następujący, Swift w przypadku, gdy żaden inicjalizator nie został zadeklarowany, dostarcza domyślny *memberwise initializer* dzięki czemu w EPUBManifest dostajemy go „za darmo”, natomiast w przypadku struktury EPUBDocument został zadeklarowany pomocniczy inizjalizator przez co domyślny inicjalizator nie został dostarczony przez swift. EPUBManifest posiada publiczną metodę, która przyjmuje jako argument id elementu zwraca do niego ścieżkę w przypadku, gdy element znajduje się w słowniku items. Słownik w swiftcie zwraca wartość o typie opcjonalnym, ponieważ nie ma żadnej gwarancji, że do podanego przez nas kluczu przypisana jest jakaś wartość. Zastosowano tutaj rozpakowanie typu opcjonalnego przy pomocy wyrażenia if let, aby otrzymać ścieżkę elementu. W przypadku, gdy znajduje się on w słowniku zostanie on przypisany do nowej stałej ,której typ już nie jest opcjonalny, więc posiadanie przez niej jakiejś wartości jest zagwarantowane. Dodatkowo w przypadku, gdyby taki element manifestu o wskazanym id nie istniał, zostanie rzucony błąd co zostało oznaczone słowem kluczowym throws przy deklaracji zwracanego typu. Sposób deklaracji takiej metody został zaprezentowany na wykazie 4.9.

```
func (function name) ((parameters)) throws -> (return type) {
    (statements)
}
```

Listing 4.9: Funkcje i metody które mogą rzucać błędy, przy deklaracji muszą zostać oznaczone słowem kluczowym **throws**[8]

W kontekście EPUBManifest pozostało jeszcze wspomnieć o typie enumeracji, który został stworzony aby określić charakter elementu znajdującego się w publikacji EPUB, i wymienionego w manifestie. Mowa tutaj o EPUBMediaType, enumeracji która posiada powiązany typ (ang. Associated Type), którym jest typ String.

```
1 public enum EPUBMediaType: String {
2     case gif = "image/gif"
3     case jpeg = "image/jpeg"
4     case png = "image/png"
5     case svg = "image/svg+xml"
6     case xHTML = "application/xhtml+xml"
7     case rfc4329 = "application/javascript"
8     case opf2 = "application/x-dtbncx+xml"
9     case openType = "application/font-sfnt"
10    case woff = "application/font-woff"
11    case mediaOverlays = "application/smil+xml"
12    case pls = "application/pls+xml"
13    case mp3 = "audio/mpeg"
14    case mp4 = "audio/mp4"
15    case css = "text/css"
16    case woff2 = "font/woff2"
17    case unknown
18 }
```

Listing 4.10: Enumeracja EPUBMediaType.

Enumeracja ta deklaruje wszystkie przypadki typu mediów wspieranych przez standard EPUB i wymienionych w specyfikacji. Dzięki tej enumeracji element manifestu posiada własność, która jest ograniczona do kilku przypadków, a w sytuacji potrzeby obsłużenia takiego elementu w prosty sposób można zdeterminować jego rodzaj. Powiązana wartość dla każdego przypadku jest ciągiem znaków reprezentującym typ określony w specyfikacji EPUB, i gwarantuje nam ona prostą inicjalizację przez podanie wartości jako argument inicjalizatora enumeracji.

4.1.3. EPUBMetadata

Kolejny elementem wymaganym przez OPF jest metadata, który enkapsuluje meta informacje na temat konkretnej interpretacji zawartej w publikacji. W celach reprezentacji tych meta danych w bibliotece EPUBKit została stworzona struktura EPUBMetadata.

```
1 public struct EPUBMetadata {
2     public struct Creator {
3         public var name: String?
```

```
4     public var role: String?
5     public var fileAs: String?
6 }
7     public var contributor: Creator?
8     public var coverage: String?
9     public var creator: Creator?
10    public var date: String?
11    public var description: String?
12    public var format: String?
13    public var identifier: String?
14    public var language: String?
15    public var publisher: String?
16    public var relation: String?
17    public var rights: String?
18    public var source: String?
19    public var subject: String?
20    public var title: String?
21    public var type: String?
22    public var coverId: String?
23 }
```

Listing 4.11: Struktura EPUBMetadata

Struktura EPUBMetadata, której deklaracja została przedstawiona na wykazie 4.11, definiuje własny publiczny typ pomocniczy `Creator`, aby w lepszy sposób reprezentować element twórcy i współtwórcy publikacji, którzy mogą zostać wymienieni w metadanych publikacji EPUB. Wszystkie właściwości EPUBMetadata posiadają typ opcjonalny, ponieważ ich obecność w dokumencie EPUB nie jest zagwarantowana. Inicjalizator nie jest obecny, ponieważ po raz kolejny jest dostarczony domyślny inicjalizator przez Swift. Właściwości są oznaczone jako zmienne publiczne, ponieważ udostępnienie ich globalnie ma sens ze względu na ich informatywny cel. Właściwość `metadata` w klasie EPUBDocument jest stałą, a więc pomimo tego iż w deklaracji struktury EPUBMetadata jej właściwości są zmiennymi, to w momencie przypisania jej instancji do instancji klasy EPUBDocument wartości zmiennych nie mogą zostać zmienione, co wynika z natury struktury w swiftcie, która jest kopiwana przez wartość. Mutowanie wartości zmiennych w instancji struktury EPUBMetadata, która nie znajduje się w kontekście całego dokumentu, ma sens, a przynajmniej nie jest niedopuszczalne. W przypadku, gdyby parser znajdujący się w posiadaniu takiej instancji, w pewnym momencie wywalał pewne dodatkowe dane z dokumentu, warto pozwolić mu na uaktualnienie ich w strukturze. Niemutowalność zostaje zagwarantowana dopiero w momencie przypisania jej jako stałej w klasie EPUBDocument.

4.1.4. EPUBSpine

Element `spine` w publikacji EPUB definiuje kolejność w jakiej elementy manifestu są uporządkowane, czyli w jakiej kolejności należy je wyświetlać. Element ten znajduje swoją reprezentację w bibliotece EPUBKit, jako struktura EPUBSpine (wykaz 4.12).

```
1 public struct EPUBSpine {
2     public struct Item {
```

```

3     public var id: String?
4     public var idref: String
5     public var linear: Bool
6 }
7
8     public var id: String?
9     public var toc: String?
10    public var pageProgressionDirection: EPUBPageProgressionDirection?
11    public var items: [Item]
12 }
```

Listing 4.12: Struktura EPUBSpine

EPUBSpine podobnie do EPUBManifest deklaruje własny typ pomocniczy Item, który odzwierciedla itemref znajdujący się w spine publikacji EPUB. Item posiada własności, które odnoszą się do identyfikatora elementu manifestu idref, dodatkowego identyfikatora, oraz istotnej informacji, czy element powinien zostać wyświetlony w kolejności liniowej, czy nie. Struktura EPUBSpine składa się z pola id, toc, które opcjonalnie może posiadać identyfikator spisu treści jeżeli takowy się znajduje w dokumencie, własności która definiuje kierunek w którym powinno się wyświetlać elementy dokumentu, oraz tablicę elementów własnego typu Item.

```

1 public enum EPUBPageProgressionDirection: String {
2     case leftToRight = "ltr"
3     case rightToLeft = "rtl"
4 }
```

Listing 4.13: Enumeracja EPUBPageProgressionDirection

Zmienna pageProgressionDirection posiada typ, który został zadeklarowany przeze mnie jako enumeracja EPUBPageProgressionDirection (wykaz 4.13) składająca się z dwóch przypadków. Pierwszego, który reprezentuje kolejność czytania od lewej do prawej oraz drugiego, który mówi o kierunku przeciwnym. W przypadku, gdy kolejność nie zostanie zdefiniowana w publikacji, decyzja jak dokument powinien zostać wyświetlony zostaje pozostawiona systemowi czytającemu. W celu szybkiej inicjalizacji i zdefiniowaniu odpowiedniego kierunku dzięki powiązaniu typowi, parser podaje jako argument inicjalizatora enumeracji wartość, która wyświetzczył z publikacji EPUB, która przyjmuje wartość rtl lub ltr.

4.1.5. EPUBTableOfContents

Ostatnia struktura modelu reprezentuje spis treści zawarty w publikacji EPUB. Pomimo tego iż w wersji trzeciej standardu nie jest wspierany w takiej formie jak w poprzednich wersjach, to ze względu na zapewnienie wsparcia tym starszym wersjom publikacji zdecydowałem się zaimplementować tą relatywnie prostą strukturę. Spis treści przedstawiony w formie pliku typu NCX służył w poprzednich wersjach EPUB użytkownikowi do nawigacji w obrębie dokumentu, jednak specyfikacja wymaga od systemu czytającego ignorowania tego pliku w publikacjach wersji trzeciej.

```
1 public struct EPUBTableOfContents {
```

```
2  public var label: String
3  public var id: String
4  public var item: String?
5  public var subTable: [EPUBTableOfContents]?
6 }
```

Listing 4.14: Struktura EPUBTableOfContents

EPUBTableOfContents (wykaz 4.14) posiada właściwości, które znamy już z poprzednich struktur, jednakże to co ją od nich wyróżnia to możliwość zagnieżdżenia. Rozdziały mogą posiadać podrozdziały, a te z kolei mogą być podzielone na jeszcze mniejsze części, co może zostać uwzględnione w takim spisie treści. Konieczne jest pozwolenie przechowywania przez każdą pozycję swojej własnej tablicy. A skoro sam spis posiada takie same atrybuty co jej elementy, postanowilem nie tworzyć dodatkowej struktury Item jak we wcześniej omówionych strukturach. Swift zapewnia nam domyślny inizjalizator, więc po raz kolejny unikamy konieczności implementowania go ręcznie.

4.2. Parser

Pierwotnie planowałem opisać parser w rozdziale poświęconym modelowi, ponieważ w architekturze MVC biblioteki EPUBKit jego rolna mocno wpasowuje się w definicję modelu, ponieważ to właśnie on go tworzy. Jednakże zdecydowałem, że rozsądniej będzie, gdy przeznaczę na to osobny podrozdział ze względu na to jak obszerny jest to temat ze względu na jego architekturę, oraz ilość wykonywanych operacji. Ten rozdział będzie opisywał typy parsera wypisane na wykazie 4.15.

```
1  Utils
2  |-- EPUBParsable.swift
3  |-- EPUBParser.swift
4  '-- EPUBParserError.swift
```

Listing 4.15: Struktura folderu narzędzi służących modelowi EPUBKit

4.2.1. EPUBParsable

Aby praca parsera była dobrze zorganizowana, zdecydowałem się zaimplementować wzorzec projektory *Budowniczy* (ang. *Builder*), ponieważ jest on zgodny z naturą parsera, który wywala informacje z publikacji EPUB, a następnie na ich podstawie buduje model EPUBKit, którym jest EPUBDocument. Pierwszym krokiem w celu zaimplementowania wzorca budowniczego było stworzenie protokołu (wykaz 4.16), który zostanie zastosowany przez parser.

```
1 protocol EPUBParsable {
2     func unzip(archive name: String) throws -> URL
3     func getContentPath(from bookDirectory: URL) throws -> URL
4     func getMetadata(from xmlElement: AEXMLElement) -> EPUBMetadata
5     func getManifest(from xmlElement: AEXMLElement) -> EPUBManifest
6     func getSpine(from xmlElement: AEXMLElement) -> EPUBSpine
7     func getTableOfContents(from xmlElement: AEXMLElement) -> EPUBTableOfContents
```

```
8 }
```

Listing 4.16: Protokół EPUBParsable

Protokół ten deklaruje metody, które każda stosująca go klasa lub struktura będzie musiała zaimplementować. Konwencja nazewnictwa protokołów w swiftcie jest taka, aby nazywać je tak, aby dobrze określały naturę klasy którą go stosuje, aby pasowały do cech które nadają klasom. Jeżeli protokół opisuje czym klasa jest, to powinien być czytany jako rzeczownik jak np. protokół standardowej biblioteki swifta `MutableCollection`. Protokoły, które opisują funkcjonalność, pewne dodatkowe możliwości powinny być kończyć się `-ible` lub `-able`. Dzięki tej konwencji, od razu staje się jasne jakich cech nabywa klasa już w momencie oznaczenia, że stosuje ona dany protokół. Zaimplementowanie przez stosującą protokół klasę jego metod jest wymagane, ponieważ na ten moment nie istnieje żadna implementacja domyślnej. Swift pozwala na dostarczenie tak zwanych rozszerzeń deklarowanych typów przy pomocy słowa kluczowego `extension`, tak jak zademonstrowano to na wykazie 4.17. Dzięki tym rozszerzeniom deklarację typu można podzielić na schlundnie wyglądające i dobrze zorganizowane bloki kodu, co zostanie przedstawione przy omawianiu klasy `EPUBParser`.

```
extension (type name): (adopted protocols) {
    (declarations)
}
```

Listing 4.17: Deklaracja wyrażenia `extension`[8]

Takie rozszerzenie można wykorzystać również w protokołach, które są typem tak samo jak klasa czy struktura, i dostarczyć im domyślną implementację metody lub zadeklarować nowe. Jednakże w tym przypadku nie będę tego robił ze względu na to, iż `EPUBParser` powinien zaimplementować każdą z nich w kontekście konkretnego dokumentu.

4.2.2. EPUBParser

Zadaniem parsera jest zebranie wszystkich informacji na temat publikacji i enkapsulacji ich w zmiennych o typach, które posiada `EPUBDocument` i pełni rolę budowniczego. Jego deklaracja to zbiór właściwości o typie opcjonalnym oraz inicjalizator, w którym odbywa się cały proces budowania modelu, poprzez parsowanie publikacji EPUB.

```
1 class EPUBParser {
2     var directory: URL?
3     var contentDirectory: URL?
4     var metadata: EPUBMetadata?
5     var manifest: EPUBManifest?
6     var spine: EPUBSpine?
7     var tableOfContents: EPUBTableOfContents?
8
9     init(named: String) throws {
10         do {
11             directory = try unzip(archive: named)
12             let contentPath = try getContentPath(from: directory!)
13             contentDirectory = contentPath.deletingLastPathComponent()
```

```

14     let data = try Data(contentsOf: contentPath)
15     let content = try AEXMLDocument(xml: data)
16     metadata = getMetadata(from: content.root["metadata"])
17     manifest = getManifest(from: content.root["manifest"])
18     spine = getSpine(from: content.root["spine"])
19     let tocPath = contentDirectory!.AppendingPathComponent(try manifest!.
20             path(forItemWithId: spine?.toc ?? ""))
21     let tocData = try Data(contentsOf: tocPath)
22     let tocContent = try AEXMLDocument(xml: tocData)
23     tableOfContents = getTableOfContents(from: tocContent.root)
24 } catch {
25     print(error.localizedDescription)
26     throw error
27 }
28 }
```

Listing 4.18: Klasa EPUBParser

W przeciwieństwie do wcześniej zadeklarowanych typów, `EPUBParser` (wykaz 4.18) tak jak i protokół `EPUBParsable`, nie są oznaczone jako publiczne, więc nie mogą zostać użyte poza biblioteką. Domyślnym parametrem dostępu w Swiftcie jest `internal`, który ogranicza typ, czy właściwość do użytku w obrębie jednego projektu, w tym przypadku biblioteki. Powodem tego jest sposób w jaki zorganizowana została inicjalizacja `EPUBDocument`. Osobie z zewnątrz najprawdopodobniej nie zależy, aby się dowiedzieć w jaki sposób powstaje instancja dokumentu, dla niej ważne jest to, że może otrzymać ją podając jako argument inicjalizatora nazwę pliku, a cały proces będzie odbywał się poza jego zasięgiem. `EPUBParser` odgrywa tutaj rolę budowniczego i w momencie rozpoczęcia inicjalizacji dokumentu, jego własna instancja zostaje stworzona i „oddelegowana” do wykonania zadania do którego jest przeznaczona. Klasa `EPUBParser` nie ma innego zastosowania i nie ma absolutnie żadnego powodu aby używać jej niezależnie i do innych celów niż ten jeden, z góry narzucony. Dlatego też ani klasa, ani jej właściwości nie są oznaczone publicznymi. Warto zwrócić uwagę na to, iż w swojej deklaracji `EPUBParser` nie stosuje protokołu `EPUBParsable`, co wynika ze stylu i wytycznych apple odnośnie projektowania klas. Klasa powinna deklarować to co dla niej najbardziej elementarne, a dopiero przy użyciu `extension` funkcjonalność powinna zostawać do niej dodawana. I tak jest w tym przypadku, gdzie zadeklarowane są zmienne oraz inicjalizator, który już wykonuje metody których jeszcze są dla niego nie znane, a zostaną dopiero zaimplementowane po zastosowaniu protokołu tak jak zademonstrowano to na wykazie 4.19.

```

30 //MARK: - EPUBParsable
31 extension EPUBParser: EPUBParsable {
32     func unzip/archive named: String) throws -> URL { ... }
33     func getContentPath(from bookDirectory: URL) throws -> URL { ... }
34     func getMetadata(from content: AEXMLElement) -> EPUBMetadata { ... }
35     func getManifest(from content: AEXMLElement) -> EPUBManifest { ... }
36     func getSpine(from content: AEXMLElement) -> EPUBSpine { ... }
37     func getTableOfContents(from toc: AEXMLElement) -> EPUBTableOfContents { ... }
```

```
38 }
```

Listing 4.19: Klasa EPUBParser stosuje protokół EPUBParsable

Dobrą praktyką jest dołączenie komentarza przy każdym rozszerzeniu aby oznaczyć dodatkowo co oferuje dane rozszerzenie. Należy również pamiętać, aby dla każdego stosowanego protokołu przeznaczyć osobne rozszerzenie w celu zwiększenia czytelności kodu. Omówię teraz, krok po kroku w jaki sposób postępuje inicjalizacja parsera i w jaki sposób wywala się on dane z publikacji w formacie EPUB prezentując kod implementacji każdej z metod protokołu.

```
init(named: String) throws {
    do {
        ...
    } catch {
        print(error.localizedDescription)
        throw error
    }
}
```

Listing 4.20: Block *do-catch* w inicjalizatorze EPUBParser

Jak widać na wykazie 4.20, inicjalizator przyjmuje jako argument nazwę pliku publikacji i składa się bloku *do-catch* w którym odbywa się parsowanie. W przypadku, gdy ktoś z metod rzuci błędem, zostanie on rzucony dalej i zostanie wyświetlona na konsoli informacja o błędzie, który został napotkany. Błędy, które mogą zostać rzucone są typu `EPUBParserError`, który zostanie omówiony pod koniec podrozdziału dotyczącego parsera. Przejdźmy zatem do instrukcji znajdujących się w bloku *do-catch*.

```
11 directory = try unzip(archive: named)
```

Listing 4.21: Próba rozpakowania archiwum

Pierwszą czynnością parsera jest rozpakowanie archiwum `.epub` (wykaz 4.21), które w rzeczywistości jest archiwum w stylu ZIP, co omówiono dokładnie w rozdziale trzecim. Dzięki wykorzystaniu open-sourcowej biblioteki ZIP, proces jest ten wyjątkowo prosty.

```
func unzip(archive named: String) throws -> URL {
    Zip.addCustomFileExtension("epub")
    do {
        let filePath = Bundle.main.url(forResource: named, withExtension: "epub")!
        let unzipDirectory = try Zip.quickUnzipFile(filePath)
        return unzipDirectory
    } catch ZipError.unzipFail {
        throw EPUBParserError.unZipError
    }
}
```

Listing 4.22: Implementacja metody `unzip(archive named:)`

Jak można zaobserwować na wykazie 4.22, w pierwszej kolejności metoda dodaje do klasy `Zip` rozszerzenie `.epub`, aby biblioteka rozpoznała plik jako archiwum. Następnie w bloku *do-catch* dochodzi do ustalenia lokalizacji pliku w paczce aplikacji oraz rozpakowania archiwum metodą, która

zwraca ścieżkę do docelowego folderu, w którym znajduje się rozpakowana publikacja EPUB. Jeżeli proces rozpakowania przebiegnie bezbłędnie, metoda zwróci ścieżkę.

```
12 let contentPath = try getContentPath(from: directory!)
13 contentDirectory = contentPath.deletingLastPathComponent()
```

Listing 4.23: Próba uzyskania ścieżki do pliku *content.opf*

Inicjalizator wywołuje następnie metodę (wykaz 4.23), która lokalizuje w publikacji obowiązkowy plik `content.opf` w folderze `META-INF`, a następnie szuka w nim elementu `rootfile`, który jest plikiem OPF, opisującym zawartość całej publikacji.

```
func getContentPath(from bookDirectory: URL) throws -> URL {
    do {
        let path = bookDirectory.appendingPathComponent("META-INF/container.xml")
        let data = try Data(contentsOf: path)
        let container = try AEXMLDocument(xml: data)
        let content = container.root["rootfiles"]["rootfile"].attributes["full-path"]
        ]
        return bookDirectory.appendingPathComponent(content!)
    } catch {
        throw EPUBParserError.containerParseError
    }
}
```

Listing 4.24: Implementacja metody `getContentPath(from bookDirectory:)`

Po raz kolejny instrukcje metody (wykaz 4.24) znajdują się w bloku `do-catch`, aby zapewnić obsługę ewentualnych błędów. Jak przedstawiono na przykładzie 3.1, w dokumencie `container.xml` znajduje się zagnieżdżona ścieżka do pliku OPF. W przypadku zlokalizowania pliku OPF, ścieżka do niego zostaje zwrócona, a w przeciwnym wypadku zostanie rzucony błąd. Jeżeli funkcja zwróci ścieżkę pliku OPF, inicjalizator przypisze do zmiennej `contentDirectory` ścieżkę do folderu w którym się on znajduje przez usunięcie ostatniego komponentu ścieżki.

```
14 let data = try Data(contentsOf: contentPath)
15 let content = try AEXMLDocument(xml: data)
```

Listing 4.25: Parsowanie pliku *content.opf*

Znając już lokalizację pliku OPF, należy go sparsować przy użyciu zewnętrznej biblioteki AEXML (wykaz 4.25). Dzięki temu dostęp do elementów dokumentu będzie znacznie uproszczony, dokument zostanie przedstawiony w formie klasy `AEXMLDocument`. Jeżeli próba inicjalizacji klas `Data` oraz `AEXMLDocument` przebiegnie poprawnie, otrzymamy zmienna `content`, z której parser będzie wywalał kluczowe informacje dotyczące dokumentu.

```
16 metadata = getMetadata(from: content.root["metadata"])
17 manifest = getManifest(from: content.root["manifest"])
18 spine = getSpine(from: content.root["spine"])
```

Listing 4.26: Wywołanie metod protokołu

Inicjalizator, gdy znajdzie się w posiadaniu dokumentu OPF w formie dokumentu AEXMLDocument, będzie następnie podawał jako argumenty odpowiednim funkcjom jego elementy (wykaz 4.26).

```
func getMetadata(from xmlElement: AEXMLElement) -> EPUBMetadata {
    var metadata = EPUBMetadata()
    metadata.contributor = EPUBMetadata.Creator(name: xmlElement["dc:contributor"].value,
                                                role: xmlElement["dc:contributor"].attributes["opf:role"],
                                                fileAs: xmlElement["dc:contributor"].attributes["opf:file-as"])
    metadata.coverage = xmlElement["dc:coverage"].value
    metadata.creator = EPUBMetadata.Creator(name: xmlElement["dc:creator"].value,
                                            role: xmlElement["dc:creator"].attributes["opf:role"],
                                            fileAs: xmlElement["dc:creator"].attributes["opf:file-as"])
    metadata.date = xmlElement["dc:date"].value
    metadata.description = xmlElement["dc:description"].value
    metadata.format = xmlElement["dc:format"].value
    metadata.identifier = xmlElement["dc:identifier"].value
    metadata.language = xmlElement["dc:language"].value
    metadata.publisher = xmlElement["dc:publisher"].value
    metadata.relation = xmlElement["dc:relation"].value
    metadata.rights = xmlElement["dc:rights"].value
    metadata.source = xmlElement["dc:source"].value
    metadata.subject = xmlElement["dc:subject"].value
    metadata.title = xmlElement["dc:title"].value
    metadata.type = xmlElement["dc:type"].value
    for metaItem in xmlElement["meta"].all! {
        if metaItem.attributes["name"] == "cover" {
            metadata.coverId = metaItem.attributes["content"]
        }
    }
    return metadata
}
```

Listing 4.27: Implementacja metody getMetadata(**from** xmlElement:)

Działanie metody getMetadata(**from** xmlElement:) przedstawionej na wykazie 4.27, agregującej metadane jest relatywnie proste, musi ona przeanalizować elementy zawarte w przekazanej instancji AEXMLElement. Elementy te są oznaczone w sposób określony w specyfikacji, więc są wyszukiwane według klucza którym jest równy swojemu odpowiednikowi w specyfikacji co znacznie upraszcza proces budowy instancji EPUBMetadata. Ponieważ każda z własności EPUBMetadata posiada typ opcjonalny, nie ponosimy żadnego ryzyka w przypadku gdy oczekiwana wartość nie zostanie zwrócona.

```
func getManifest(from xmlElement: AEXMLElement) -> EPUBManifest {
    var items: [String:EPUBManifest.Item] = [:]
    for item in xmlElement["item"].all! {
```

```

        let id = item.attributes["id"]!
        let path = item.attributes["href"]!
        let mediaType = item.attributes["media-type"]
        let properties = item.attributes["properties"]
        items[id] = EPUBManifest.Item(id: id, path: path, mediaType: EPUBMediaType(
            rawValue: mediaType!) ?? .unknown, property: properties)
    }
    return EPUBManifest(id: xmlElement["id"].value, items: items)
}

```

Listing 4.28: Implementacja metody getManifest (from xmlElement:)

Metoda `getManifest (from content:)` (wykaz 4.28) w pierwszej kolejności tworzy instancję słownika, który znamy ze struktury `EPUBManifest`. Reprezentuje on wszystkie elementy wymienione w manifeście. Następnie iteruje po tablicy elementów zawartych w parametrze `xmlElement`, i inicjalizuje przy każdym z nich nową instancję struktury `EPUBManifest.Item` z informacjami, które udało jej się wywalczyć.

```

func getSpine(from xmlElement: AEXMLElement) -> EPUBSpine {
    var items: [EPUBSpine.Item] = []
    for item in xmlElement["itemref"].all! {
        let id = item.attributes["id"]
        let idref = item.attributes["idref"]!
        let linear = (item.attributes["linear"] ?? "yes") == "yes" ? true : false
        items.append(EPUBSpine.Item(id: id, idref: idref, linear: linear))
    }
    let pageProgressionDirection = xmlElement["page-progression-direction"].value ?? ""
    return EPUBSpine(id: xmlElement.attributes["id"], toc: xmlElement.attributes["toc"], pageProgressionDirection: EPUBPageProgressionDirection(rawValue: pageProgressionDirection), items: items)
}

```

Listing 4.29: Implementacja metody getSpine (from xmlElement:)

W identycznym stylu działa metoda `getSpine (from content:)` (wykaz 4.29), której zadaniem jest stworzenie instancji struktury `EPUBSpine`. W pierwszej kolejności iteracja po elementach i inicjalizacja `EPUBSpine.Item` przy użyciu uzyskanych danych w celu przechowania ich tablicy przypisanej następnie do instancji `EPUBSpine`. Po zebraniu informacji o elementach metoda determinuje kierunek przeglądania stron po czym zwraca instancję `EPUBSpine`.

```

19 let tocPath = contentDirectory!.appendingPathComponent(try manifest!.path(
    forItemWithId: spine?.toc ?? ""))
20 let tocData = try Data(contentsOf: tocPath)
21 let tocContent = try AEXMLDocument(xml: tocData)
22 tableOfContents = getTableOfContents(from: tocContent.root)

```

Listing 4.30: Parsowanie dokumentu NCX

Ostatnimi czynnościami parsera są odszukanie w publikacji dokumentu typu NCX, który reprezentuje spis treści (table of contents) i przedstawienie go w formie struktury `EPUBTableOfContents` poprzez

parsowanie pliku NCX. Widoczna na wykazie 4.30 sekwencja instrukcji rozpoczyna się od odszukania dokumentu w manifeście przy pomocy jego *ID*, co jest wykonywane w czasie **O(1)** ponieważ elementy manifestu przechowywane są w słowniku. Gdy już zostanie ustalona ścieżka do pliku NCX, należy użyć biblioteki AEXML w celu przekonwertowania dokumentu na klasę AEXMLDocument i dopiero do udanej konwersji zostaje wywołana metoda `getTableOfContents (from xmlElement:)`.

```
func getTableOfContents(from xmlElement: AEXMLElement) -> EPUBTableOfContents {
    let item = xmlElement["head"]["meta"].all(withAttributes: ["name": "dtb=uid"])?.
        first?.attributes["content"]
    var tableOfContents = EPUBTableOfContents(label: xmlElement["docTitle"]["text"].
        value!, id: "0", item: item, subTable: [])

    func evaluateChildren(from map: AEXMLElement) -> [EPUBTableOfContents] {
        if let _ = map["navPoint"].all {
            var subs: [EPUBTableOfContents] = []
            for point in map["navPoint"].all! {
                subs.append(EPUBTableOfContents(label: point["navLabel"]["text"].
                    value!, id: point.attributes["id"]!, item: point["content"].
                    attributes["src"]!, subTable: evaluateChildren(from: point)))
            }
            return subs
        } else {
            return []
        }
    }
    tableOfContents.subTable = evaluateChildren(from: xmlElement["navMap"])
    return tableOfContents
}
```

Listing 4.31: Implementacja metody `getTableOfContents (from xmlElement:)`

Ze względu na możliwość zagnieżdżania elementów metoda `getTableOfContents (from xmlElement:)` (wykaz 4.31) deklaruje własną funkcję, która będzie odpowiedzialna za zbieranie informacji o elementach i penetrowanie w głąb kolejnych poziomów zagnieżdżenia.

W tym miejscu praca parsera kończy się. Jegoinicjalizacja dobiegła końca, a metody z powodzeniem lub bez zebrały odpowiednie informacje o publikacji EPUB. To co następnie dzieje się z danymi parsera jest dokładnie opisane w podrozdziale opisującym EPUBDocument.

4.2.3. EPUBParserError

Aby w razie niepowodzenia któregoś z etapów parsowania błąd był bardziej czytelny, zdecydowałem aby zaimplementować własny typ błędów.

```
1 enum EPUBParserError {
2     case unZipError
3     case containerParseError
4     case noPathForItem(String)
5     case noIdForTableOfContents
```

```
6 }
```

Listing 4.32: Enumeracja EPUBParserError

Enumeracja EPUBParserError, której implementacja jest widoczna na wykazie 4.32, przewiduje kilka sytuacji mogących wymagać wyrzucenia błędu ponieważ są one krytyczny, a dalsze przeprowadzenie procesu parsowania nie jest możliwe. Taką sytuacją może być niepowodzenie przy rozpakowywaniu archiwum, a ponieważ biblioteka która jest wykorzystana do tego celu posiada obsługę błędów, to w razie się jego pojawienia nie zostanie on przeoczony. Gdyby parser próbował kontynuować swoją pracę, oczywiście skończyło by się to niepowodzeniu każdej instrukcji. Dzięki zadeklarowanemu typowi, można w prosty sposób wykryć w którym miejscu program się przerywa.

```
8 //MARK: - LocalizedError
9 extension EPUBParserError: LocalizedError {
10     var errorDescription: String? {
11         switch self {
12             case .unZipError:
13                 return "Error while trying to unzip the archive"
14             case .containerParseError:
15                 return "Error with parsing container.xml"
16             case .noPathForItem (let message):
17                 return "Error with getting ID for item: '\(message)'"
18             case .noIdForTableOfContents:
19                 return "Error with getting path for toc.ncx"
20         }
21     }
22     var failureReason: String? {
23         switch self {
24             case .unZipError:
25                 return "Zip module was unable to unzip this archive"
26             case .containerParseError:
27                 return "The path to container.xml may be wrong, or the file itself may
28                 be missing"
28             case .noPathForItem (let message):
29                 return "Path to item with id: '\(message)' was not found in the manifest
30                 !"
30             case .noIdForTableOfContents:
31                 return "Table of contents ID was probably not mentioned in the spine"
32         }
33     }
34     var recoverySuggestion: String? {
35         switch self {
36             case .unZipError:
37                 return "Make sure your archive is a proper .epub archive"
38             case .containerParseError:
39                 return "Make sure the path to container.xml is correct"
40             case .noPathForItem (let message):
41                 return "Make sure to check if the item with id: '\(message)' is in the
41                 manifest!"
```

```

42     case .noIdForTableOfContents:
43         return "Make sure to check if the '<spine>' contains the ID for TOC"
44     }
45 }
46 }
```

Listing 4.33: Rozszeżenie EPUBParserError z zastosowaniem protokołu LocalizedError

Jednakże deklaracje `EPUBParserError` jeszcze nie czynią typu błędem, który byłby tolerowany przez Swift, ponieważ nie stosuje protokołu `Error`. Natomiast aby zapewnić dodatkową wartość informacyjną, zdecydowałem się zastosować protokół `LocalizedError` (wykaz 4.33), który wymaga zadeklarowania zmiennych które dostarczą opisy błędu, jego przyczyny oraz sugestię jego rozwiązania. Wyrażenie `switch` w swiftcie pozwala na zdefiniowanie działania dla każdego przypadku błędu gdy jako argument poda się `self`, czyli samego siebie a w tym przypadku konkretną instancję enumeracji.

Tutaj kończy się opisywanie parsera, który w roli budowniczego przyczynia się do inicjalizacji `EPUBDocument` poprzez parsowanie publikacji EPUB. Jest on częścią modelu w architekturze MVC, ponieważ na dobrą sprawę nie odpowiada za to w jaki sposób dane są wyświetlane na widoku, więc nie sposób nazwać go kontrolerem. Prawdę mówiąc, rolą kontrolera zostanie opisana przy okazji omawiania widoku, ponieważ ściśle ze sobą współpracują, a jak już wcześniej wspomniano, biblioteka `EPUBKit` dostarcza bardzo prosty widok dla modelu.

4.3. Widok



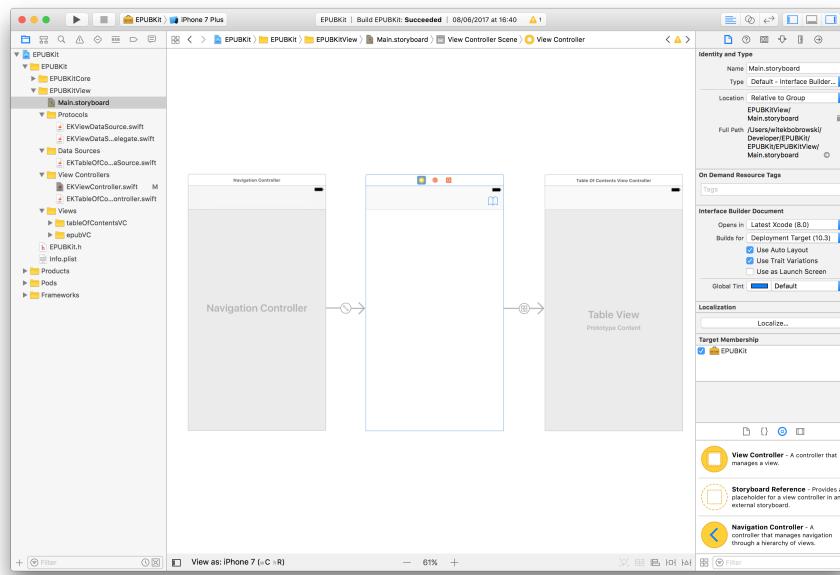
(a) Widok `EKViewController`

(b) Widok `EKTableOfContentsViewController`

Rysunek 4.3: Widoki dostępne w **EPUBKit**

Ostatnim elementem biblioteki **EPUBKit** jest jej widok. Jak wspomniano we wstępie do pracy, aby stworzyć widok znany z takich aplikacji jak *iBooks* potrzeba wielu tygodni oraz ogromnego nacisku pracy wielu programistów. Dlatego zdecydowałem się w obecnej wersji dostarczyć bardzo elementarny widok w celu zaprezentowania możliwości parsera oraz jak zręcznie można korzystać z klasy `EPUBDocument`. Widok ten będzie wyświetlał stronę w stylu *Scrolling View* często spotykanym w systemach czytających, co oznacza, że zawartość całej książki zostanie wyświetlona w jednej kolumnie, a

więc nie będzie podzielona na strony. Aby stworzyć widok, który dzieliłby tekst na strony należałyby dość mocno nadpisywać style dokumentu, aby zapewnić *paginację*, a następnie dostosować wyświetloną zawartość w formie widoku przypominającego książkę, czyli symulującego przerzucanie stron. Stworzenie tak zaawansowanego widoku pozostawiam na zaimplementowanie w którejś z przyszłych wersji biblioteki.



Rysunek 4.4: Plik *Main.storyboard* biblioteki **EPUBKit** wyświetlony w Xcode

Widok biblioteki **EPUBKit** w obecnej wersji ogranicza się do dwóch *View Controller*'ów. Czym są *View Controller*y zostało objaśnione w rozdziale piątym przy okazji omawiania widoku aplikacji demonstracyjnej. Pierwszy z nich jest głównym widokiem w którym zostaje wyświetlony dokument, a drugi przedstawia spis treści, w którym po wybraniu pozycji jesteśmy przekierowani do odpowiedniego punktu w tekście. Na to wszystko składają się własne protokoły, klasy dostarczające dane (*DataSource*'y), klasy i pliki *Xib* widoków, dwie klasy dziedziczące po *UIViewController* oraz plik *storyboard* w którym skomponowane są *View Controller*'y.

4.3.1. EKViewController

Klasa *EKViewController* reprezentuje główny *View Controller* biblioteki, w którym zostanie umieszczony widok dokumentu oraz, z którego będzie można przenieść się do spisu treści. Na wykazie 4.34 przedstawiono deklarację klasy w której znajduje się referencja do elementu widoku typu *UIView*, który został użyty w *storyboardach* widocznych na rysunku 4.4.

```

1 public class EKViewController: UIViewController {
2     @IBOutlet fileprivate weak var documentView: UIView!
3     public var epubDocument: EPUBDocument?
4     public override var prefersStatusBarHidden: Bool {
5         return navigationController?.isNavigationBarHidden == true
6     }

```

```
7     public override var preferredStatusBarUpdateAnimation: UIBarAnimation {
8         return UIBarAnimation.slide
9     }
10
11    public override func viewDidLoad() {
12        super.viewDidLoad()
13        configure()
14    }
15    public override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
16        if let tableOfContentsVC = segue.destination as?
17            EKTableOfContentsViewController {
18            tableOfContentsVC.delegate = self
19            tableOfContentsVC.epubDocument = epubDocument
20        }
21    }
22 }
```

Listing 4.34: Deklaracja klasy `textttEKViewController`

`EKViewController` deklaruje zmienną publiczną `epubDocument`, która posiada typ `EPUBDocument`. Przetrzymuje ona dokument który będzie wyświetlony, i należy go ustawić przed pojawieniem się widoku. Oznacza to, że programista musi samemu inicjalizować `EPUBDocument`, a następnie przekazać kontrolerowi `EKViewController` przed jego załadowaniem. Następnie zostają napisane dwie zmienne klasy nadrzędnej, aby umożliwić chowanie się paska nawigacji przy dotknięciu widoku. *View Controller* jest osadzony w kontrolerze nawigacji `UINavigationController`, dlatego domyślnie wyświetlał on będzie pasek u góry ekranu służący do nawigacji. Będzie się na nim znajdować przycisk służący do powrotu do poprzedniego widoku, oraz dołączony został przycisk umożliwiający wyświetlenie spisu treści.

Aby przeprowadzić konfigurację *View Controller'a*, nadpisuję metodę `viewDidLoad()` klasy nadrzędnej w celu wywołania własnej metody `configure()`, która jest zadeklarowana na rozszerzeniu widocznym na wykazie 4.35. Ostatnią nadisaną metodą jest `prepare(for:sender:)`, która zostaje wywołana w przypadku, gdy jeden *View Controller* będzie próbował przejść do kolejnego. W tym przypadku będzie to sytuacja gdy użytkownik naciśnie przycisk wyświetlający spis treści.

```
1 //MARK: - Configuration
2 extension EKViewController {
3     fileprivate func configure() {
4         let nibName = String(describing: EKInfiniteScrollView.self)
5         let nib = UINib(nibName: nibName, bundle: Bundle(for: EKInfiniteScrollView.
6             classForCoder()))
6         let infiniteScrollView = nib.instantiate(withOwner: EKInfiniteScrollView.
7             self, options: nil).first as! EKInfiniteScrollView
7         view.addSubview(infiniteScrollView)
8         documentView = infiniteScrollView
9         infiniteScrollView.epubDocument = epubDocument
10        navigationController?.navigationBar.barTintColor = .white
11        navigationController?.navigationBar.shadowImage = UIImage()
```

```

12     let tapGestureRecogniser = UITapGestureRecognizer(target: self, action: #
13         selector(hideNavigationBar(_:)))
14     view.addGestureRecognizer(tapGestureRecogniser)
15 }
16 @objc private func hideNavigationBar(_ sender: UITapGestureRecognizer) {
17     navigationController?.setNavigationBarHidden(navigationController?.
18         isNavigationBarHidden == false, animated: true)
19 }

```

Listing 4.35: Rozszerzenie klasy textttEKViewController o metody konfiguracji

W pierwszym rozszerzeniu klasy, definiuję metodę `configure()`, w której przeprowadzam konfigurację jej widoków. Inicjalizuję z pliku *Xib* widok w którym zostanie wyświetlony dokument, a następnie przypisuję do zmiennej `documentView` oraz dodaję jako pod-widok głównego widoku i przekazuję dokument EPUB w postaci zmiennej `epubDocument`. Pod koniec dodaję do głównego widoku `tapGestureRecogniser`, który jest instancją klasy odpowiedzialnej za rozpoznawanie wykonanego przez użytkownika gestu dotknięcia na ekran. Podczas inicjalizacji `UITapGestureRecognizer` przekazuje mu niżej zadeklarowaną metodę `hideNavigationBar(_:)`, która zostanie przez niego wywołana w momencie rozpoznania gestu. Metoda ta chowa lub pokazuje pasek nawigacji tak jak jest to widoczne na rysunku 4.3a.

```

1 //MARK: - EKTableOfContentsViewControllerDelegate
2 extension EKViewController: EKTableOfContentsViewControllerDelegate {
3     func tableOfContentsView(_ tableOfContentsView: EKTableOfContentsViewController,
4         didSelectRowAt indexPath: IndexPath) {
5         if let infiniteScrollView = documentView as? EKInfiniteScrollView {
6             infiniteScrollView.idOfElementToDisplay = (tableOfContentsView.
7                 dataSource.item(at: indexPath) as? EKTableOfContentsDataSource.Item)?
8                 .item
9         }
10    }
11 }

```

Listing 4.36: Rozszerzenie klasy textttEKViewController o protokół `EKTableOfContentsViewControllerDelegate`

W drugim i ostatnim rozszerzeniem widocznym na wykazie 4.36, stosuję na klasie protokół `EKTableOfContentsViewControllerDelegate`, który jest własnym protokołem stworzonym w celu zdefiniowania delegata dla `EKTableOfContentsViewController`. Protokół ten wymaga implementacji metody `tableOfContentsView(_:didSelectRowAt:)`. Wzorzec projektowy **Delegat** jest wyjątkowo często stosowanym wzorcem przy programowaniu na iOS. Większość klas należących do *iOS SDK* deklaruje swoich delegatów (w formie protokołu), a każdy z nich posiada szereg metod, które musi lub może zaimplementować przy stosowaniu protokołu.

4.3.2. EKTableOfContentsViewController

Widok wiczony na rysunku 4.3b wyświetla spis treści dokumentu EPUBDocument i jest reprezentowany przez klasę EKTableOfContentsViewController.

```

1 class EKTableOfContentsViewController: UIViewController {
2     @IBOutlet fileprivate weak var tableView: UITableView!
3     public weak var delegate: EKTableOfContentsViewControllerDelegate?
4     public var dataSource = EKTableOfContentsDataSource()
5     public var epubDocument: EPUBDocument?
6     didSet {
7         dataSource.epubDocument = epubDocument
8     }
9 }
10 override func viewDidLoad() {
11     super.viewDidLoad()
12     configure()
13 }
14 }
```

Listing 4.37: Deklaracja klasy EKTableOfContentsViewController

EKTableOfContentsViewController deklaruje referencję do elementu w *storyboardach* własnego delegata, oraz dotąd niespotkany w pracy *dataSource*. Kod źródłowy oraz jego rola zostaną omówione w podrozdziale 4.3.4, który będzie właśnie tej klasie zadektykowany. Po raz kolejny do widoku przypisana jest instancja EPUBDocument, z której widok ten czerpie dane, a zmienna do której jest przypisana, posiada tak zwany *Obserwator własności* (z ang. Property Observer), czyli blok *didSet{ ... }*, który zostanie wywołany w przypadku gdy wartość własności się zmieni. Istnieje drugi obserwator, jakim jest *willSet{ ... }*, ten natomiast wywoływany jest tuż przed zmianą wartości przez własność. W blokach tych istnieje możliwość wykorzystania specjalnej stałej *newValue* lub *oldValue* w zależności od rodzaju obserwatora, co daje możliwość dostępu do wartości, która zostanie zmieniona, lub zostanie przypisana. W tym przypadku, gdy wartość do zmiennej *epubDocument* zostanie przypisana, zostanie wykonana instrukcja *dataSource.epubDocument = epubDocument*, czyli przypisanie samej siebie do własności zmiennej *dataSource* o tej samej nazwie.

```

1 //MARK: - Configuration
2 extension EKTableOfContentsViewController {
3     fileprivate func configure() {
4         navigationController?.navigationBar.barTintColor = .white
5         navigationController?.navigationBar.shadowImage = UIImage()
6         navigationController?.navigationBar.backItem?.title = ""
7         dataSource.delegate = self
8         tableView.dataSource = dataSource
9         tableView.delegate = self
10        tableView.register(UINib(nibName: "EKTableOfContentsViewCell", bundle:
11            Bundle(for: classForCoder)), forCellReuseIdentifier: "EKTableOfContentsViewCell")
11    }
}
```

```
12 }
```

Listing 4.38: Rozszerzenie klasy EKTableOfContentsViewController o metodę konfiguracji

Konfiguracja klasy jest przeprowadzona w podobnym stylu co w poprzednim przypadku, i tak też będzie w kolejnych. Rozdzielenie kodu klasy jest praktyką mającą na celu pilnowanie porządku oraz rozczłonkowanie deklaracji na fragmenty odpowiedzialne za konkretne czynności. Wykaz 4.38 deklaruje metodę konfiguracji, która zostaje wywołana w momencie załadowania widoku. Konfiguruje ona widok paska nawigacji, oraz ustawie siebie (`self`) jako delegata tabeli oraz źródła danych. Na końcu tabela rejestruje widoki, które będzie mogła wykorzystać jako komórki. Klasa widoku komórki `EKTableOfContentsTableViewCell` jest bardzo prosta, ponieważ jedyne co zawiera to referencję etykiety, na której zostanie wyświetlona nazwa podrozdziału oraz rozszerzenie o trywialną metodę konfigurującą.

```
1 //MARK: - UITableViewDelegate
2 extension EKTableOfContentsViewController: UITableViewDelegate {
3     func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {
4         delegate?.tableOfContentsView(self, didSelectRowAt: indexPath)
5         navigationController?.popViewController(animated: true)
6     }
7 }
8 //MARK: - EKViewDataSourceDelegate
9 extension EKTableOfContentsViewController: EKViewDataSourceDelegate {
10    func dataSourceDidFinishBuilding(_ dataSource: EKViewDataSource) {
11        tableView.reloadData()
12    }
13 }
```

Listing 4.39: Rozszerzenie klasy EKTableOfContentsViewController o protokoły UITableViewDelegate oraz EKViewDataSourceDelegate

Jeżeli skonfigurowano kontroler `EKTableOfContentsViewController` jako delegata, musi on więc stosować jego protokół. Jak zademonstrowano na wykazie 4.39, zastosowano protokoły `UITableViewDelegate` oraz `EKViewDataSourceDelegate` w celu zaimplementowania ich metod. Pierwsza z nich, metoda `tableView(_:didSelectRowAt:)` zostaje wywołana w momencie naciśnięcia na dany rząd w tabeli, którą podaje jako parametr przez użytkownika. Jako parametr podaje również indeks rzędu, który reprezentowany jest strukturą `IndexPath`, którego następnie użyjemy przy okazji wywołania metody na delegacie, aby jej przekazać indeks. Delegat, którym w tym przypadku jest `EKViewController`, otrzyma indeks i z jego pomocą wyciągnie z `dataSource` odpowiednią informację w celu wyświetlenia pożądanego przez użytkownika fragmentu w tekście.

Drugą metodą jest `dataSourceDidFinishBuilding(_:)`, która powiadamia delegata, czyli w tym przypadku `EKTableOfContentsViewController`, o tym aby wymusił przeładowanie danych na tabeli. Wynika to z tego, że źródło danych zostało zbudowane, i jest gotowe dostarczać je widokowi.

4.3.3. EKInfiniteScrollView

EKInfiniteScrollView (wykaz refEKInfiniteScrollView-declaration) jest jedynym prototypowym widokiem dostarczonym w tej wersji EPUBKit, który potrafi wyświetlić dokument EPUB. Widok w tym stylu jest często spotykany w czytnikach, więc to jego zdecydowałem się zaimplementować w pierwszej kolejności.

```
1 class EKInfiniteScrollView: UIView {  
2     @IBOutlet fileprivate weak var contentView: UIView!  
3     fileprivate var webView: WKWebView!  
4     public var epubDocument: EPUBDocument? {  
5         didSet {  
6             if let epubDocument = epubDocument {  
7                 configure(with: epubDocument)  
8             }  
9         }  
10    }  
11    public var idOfElementToDisplay: String? {  
12        didSet {  
13            guard let id = idOfElementToDisplay else { return }  
14            if id.contains("#") {  
15                let idSubstring = id.substring(from: id.characters.index(of: "#")!)  
16                webView.evaluateJavaScript("location.href = \"\\"(idSubstring)\";")  
17            } else {  
18                webView.evaluateJavaScript("location.href = \"#\\"(id)\";")  
19            }  
20        }  
21    }  
22    override func awakeFromNib() {  
23        super.awakeFromNib()  
24        configure()  
25    }  
26}
```

Listing 4.40: Deklaracja klasy EKInfiniteScrollView

Klasa ta dziedziczy po UIView, czyli najbardziej podstawowej klasie reprezentującej widok. EKInfiniteScrollView deklaruje pod-widok, który jest kontenerem dla drugiego pod-widoku, którego typem jest WKWebView. Typ ten reprezentuje widok, który pozwala na wyświetlanie treści w technologiach internetowych, takich jak pliki *HTML* stylizowane plikami *CSS*. Ze względu na to, iż dokument EPUB między innymi zbudowany jest w oparciu o te technologie można przy pomocy WKWebView wyświetlić jego zawartość. Należy jednak zmodyfikować nieco zawartość dokumentu w celu otrzymania pożdanego widoku. EKInfiniteScrollView posiada również zmienną, do której może zostać przypisana referencja do instancji EPUBDocument (typ opcjonalny), a w momencie jej przypisania zostanie sprawdzona wartość. W przypadku gdy jest ona niepusta, zostaje wywołana metoda konfiguruająca widok. Kolejną zmienną jest idOfElementToDisplay, własność służąca do wywoływania nawigacji w przypadku jej nadpisania do określonego elementu w wyświetlonym dokumencie.

Ze względu na to, że widok jest stworzony w pliku *Xib*, ogólną metodę konfiguracyjną wywołuję w napisanej metodzie `awakeFromNib()`. Metoda ta jest wywołana w momencie zainicjalizowania widoku oraz wszystkich elementów z pliku *Xib*, dzięki czemu mamy gwarancję, że elementy konfigurowane są już w pełni zainicjalizowane.

```
1 //MARK: - Configuration
2 extension EKInfiniteScrollView {
3     fileprivate func configure() {
4         webView = WKWebView(frame: contentView.frame)
5         constrainView(view: webView, toView: contentView)
6         webView.allowsBackForwardNavigationGestures = false
7         webView.scrollView.showsHorizontalScrollIndicator = false
8         webView.scrollView.pinchGestureRecognizer?.isEnabled = false
9         webView.navigationDelegate = self
10        webView.scrollView.delegate = self
11    }
12    public func configure(with epubDocument: EPUBDocument) {
13        var htmlString = "<html><body>"
14        for spineItem in epubDocument.spine.items {
15            if let manifestItem = epubDocument.manifest.items[spineItem.idref] {
16                htmlString.append("(<div id=\\"(manifestItem.path)\\"></div>")
17                let itemURL = epubDocument.contentDirectory.appendingPathComponent(
18                    manifestItem.path)
19                let spineHtmlString = (try? String(contentsOf: itemURL)) ?? ""
20                htmlString.append(spineHtmlString)
21            }
22            htmlString += "</html></body>"
23            webView.loadHTMLString(htmlString, baseURL: epubDocument.contentDirectory)
24        }
25        fileprivate func constrainView(view: UIView, toView contentView: UIView) {
26            contentView.addSubview(view)
27            view.translatesAutoresizingMaskIntoConstraints = false
28            view.leadingAnchor.constraint(equalTo: contentView.leadingAnchor).isActive =
29                true
30            view.topAnchor.constraint(equalTo: contentView.topAnchor).isActive = true
31            view.trailingAnchor.constraint(equalTo: contentView.trailingAnchor).isActive =
32                true
33        }
34    }
```

Listing 4.41: Rozszerzenie klasy `EKInfiniteScrollView` o metody konfiguracji

Podstawowa metoda konfiguracji `configure()` stylizuje widoki w pliku *Xib*, oraz przypisuje siebie jako delegata widokowi wyświetlającemu publikację oraz jego pod-widokowi o typie `UIScrollView`. Druga metoda konfiguruje widok na podstawie przekazanego dokumentu EPUB. Zgodnie z jego specyfikacją, zbierze wszystkie elementy manifestu wymienione w kolejności określonej.

ne w *kręgostupie* czyli *spine*. Aby wyświetlić dokument w całości trzeba scaić wszystkie elementy dokumentu razem, w formie stringa, aby następnie załadować je w webView przy użyciu metody `loadHTMLString (_:baseURL:)`. Podczas konkatenacji każdego z elementów, dodaję do wyniku element `div`, który zapewni prawidłową nawigację po dokumencie. Ostatnia metoda konfiguracyjna pozwala na *przyczepienie* jednego widoku do drugiego przy pomocy *constraints*, które służą do ustalania odległości elementów interfejsu między sobą.

```

1 //MARK: - WKNavigationDelegate
2 extension EKInfiniteScrollView: WKNavigationDelegate {
3     func webView(_ webView: WKWebView, didCommit navigation: WKNavigation!) {
4         webView.evaluateJavaScript("var meta = document.createElement('meta');meta.
5             setAttribute('name', 'viewport');meta.setAttribute('content', 'width=
6             device-width, initial-scale=1.0, minimum-scale=1.0, maximum-scale=1.0,
7             user-scalable=no');document.getElementsByTagName('head')[0].appendChild(
8             meta);")
9     }
10 }
11 //MARK: - UIScrollViewDelegate
12 extension EKInfiniteScrollView: UIScrollViewDelegate {
13     func scrollViewDidScroll(_ scrollView: UIScrollView) {
14         if scrollView.contentOffset.x != 0 {
15             scrollView.contentOffset.x = 0
16         }
17     }
18 }
```

Listing 4.42: Rozszerzenie klasy `EKInfiniteScrollView` o protokoły `WKNavigationDelegate` oraz `UIScrollViewDelegate`

`EKInfiniteScrollView` dzięki swojej delegacji w pod-widokach, deklaruje ważne dla niego czynności w metodach protokołów (patrz: wykaz 4.42). Pierwsza z nich jest wywoływana gdy `webView` otrzyma polecenie nawigacji do strony, czyli rozpoczęcie proces jej ładowania. W tym czasie delegat załaduje *javascript*, którym zmodyfikuje styl wyświetlonej strony. Ustawi jej odpowiednią szerokość oraz zabroni przybliżania lub oddalania, aby widok był statyczny. Druga metoda również zablokuje możliwość przesuwania w przypadku, gdyby rozmiar zawartości strony przekraczał z jakiegoś względem oczekiwane wymiary. W ten sposób otrzymujemy efekt widoczny na rysunku 4.3. Nie jest to rozwiązanie idealne i zapewne brakuje mu kilku funkcjonalności znanych z takich czytników jak *iBooks*, lecz jak na wersję pierwszą biblioteki, jest zdecydowanie wystarczający. Przewiduję ulepszenie widoku w kolejnych wersjach biblioteki.

4.3.4. EKTableOfContentsDataSource

Data source jest stosowany w wielu klasach widoku znajdujących się w *iOS SDK*. Ich celem jest dostarczenie modelu dla widoku, którego struktura będzie do niego dopasowana. Zadaniem jego jest zebranie danych, organizacja ich w określony sposób i zbudowanie na nich podstawie modelu.

```
1 class EKTableOfContentsDataSource: NSObject {
```

```
2   public weak var delegate: EKViewDataSourceDelegate?
3   public var epubDocument: EPUBDocument? {
4       didSet {
5           if epubDocument != nil {
6               build(from: epubDocument!)
7           }
8       }
9   }
10  fileprivate var model: [[Item]] = []
11  struct Item {
12      let item: String
13      let title: String
14  }
15 }
```

Listing 4.43: Deklaracja klasy EKTableOfContentsDataSource

Takim źródłem danych (z ang. *data source*) jest klasa EKTableOfContentsDataSource, dostarczająca model danych widokowi odpowiedzialnemu za spis treści. Klasa ta (wykaz 4.43) deklaruje swojego delegata, własne źródło danych jakim jest instancja EPUBDocument oraz własność *model*, której struktura jest skomponowana w sposób taki aby pasowała do tabeli, którą będzie zasiedlała. Tabela w postaci klasy UITableView składa się z sekcji, które posiadają swoje rzędy, a typem modelu jest tablica tablic typu Item. Typ ten reprezentuje pojedynczy rząd, a tablica w której się znajduje reprezentuje jedną sekcję. Ze względu na to iż sekcji może w tabeli być wiele, umieszczamy sekcję w tablicy i w ten sposób typem modelu źródła danych jest [[Item]].

```
1 //MARK: - EKViewDataSource
2 extension EKTableOfContentsDataSource: EKViewDataSource {
3     func build(from epubDocument: EPUBDocument) {
4         var section: [Item] = []
5         func evaluate(tableOfContents: [EPUBTableOfContents], space: String) {
6             for item in tableOfContents {
7                 section.append(Item(item: item.item ?? "", title: space + item.
8                     label))
9                 if let children = item.subTable {
10                     evaluate(tableOfContents: children, space: space + "    ")
11                 }
12             }
13             if let children = epubDocument.tableOfContents.subTable {
14                 evaluate(tableOfContents: children, space: "")
15             }
16             self.model = [section]
17         }
18         public func item(at indexPath: IndexPath) -> Any {
19             return model[indexPath.section][indexPath.row]
20         }
21     }
22 }
```

```
21 }
```

Listing 4.44: Rozszerzenie klasy EKTableOfContentsDataSource o protokół EKViewDataSource

Podążając za ustalonymi konwencjami, mowa o własnych protokołach, szczegółowo omówionych w kolejnym podrozdziale, rozszerzam klasę EKTableOfContentsDataSource o protokół EKViewDataSource (patrz: wykaz 4.44) co wymusza implementację jego metod. Metoda budująca model będzie analizowała każdą pozycję we własności tableOfContents instancji EPUBDocument, aby otrzymać jej tytuł oraz referencję do pliku w strukturze dokumentu. Aby przeszukać spis treści w głębi, zaimplementowałem dodatkową metodę, która dzięki rekurencji dotrze do każdego elementu w strukturze. Aby ta głębokość została odwzorowana w widoku, przekazuję funkcji pusty string, który następnie z każdym poziomem zostaje powiększony o kilka znaków spacji aby tytuł był odpowiednio odsunięty, co pozwoli na lepszą orientację użytkownika w strukturze dokumentu. Druga metoda tego protokołu będzie natomiast zwracać element w określonych sekcji i rzędzie modelu.

```
1 //MARK: - UITableViewDataSource
2 extension EKTableOfContentsDataSource: UITableViewDataSource {
3     func numberOfSections(in tableView: UITableView) -> Int {
4         return model.count
5     }
6     func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) ->
7         Int {
8         return model[section].count
9     }
10    func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) ->
11        UITableViewCell {
12        let chapter = item(at: indexPath) as! Item
13        let cell = tableView.dequeueReusableCell(withIdentifier: "
14            EKTableOfContentsViewCell",
15            for: indexPath) as!
16            EKTableOfContentsViewCell
17
18        cell.configure(with: chapter.title)
19        return cell
20    }
21 }
```

Listing 4.45: Rozszerzenie klasy EKTableOfContentsDataSource o protokół UITableViewDataSource

Jak przedstawiono na wykazie 4.45, ostatnim protokołem stosowanym przez to źródło danych jest UITableViewDataSource, który wymaga zaimplementowania metod, dzięki którym tabela dowiaduje się, jakie dane modelu będą znajdowały się w danym miejscu. Pierwszą rzeczą o której pyta UITableView swojego delegata, to ile w modelu znajduje się sekcji. Delegat, czyli klasa EKTableOfContentsDataSource zwróci tabeli model.count, czyli ilość elementów w najwyższej tablicy. Gdy tabela dowie się ile będzie musiała wyświetlić sekcji, zapyta metodą tableView(_:numberOfRowsInSection:) o ilość rzędów w sekcji, którą podaje jako para-

metr, a delegat odpowie: `model[section].count`, a więc zwróci ilość elementów w podrzędnej tablicy. Ostatnią wymaganą metodą przez protokół jest zapytanie o komórkę którą ma wyświetlić w danym rzędzie. Zanim zwrócimy komórkę o odpowiednim typie tabeli, należy przy pomocy parametru indeksu wyciągnąć z modelu metodą `item(at:)` odpowiedni element i skonfigurować nim komórkę. Ze względów wydajnościowych, tabela posiada funkcję kolejkowania komórek, które nie znajdują się na ekranie w celu ich ponownego użycia. W ten sposób nawet w przypadku ogromnej ilości danych tabela może operować zaledwie kilkoma komórkami, które wciąż uzupełnia nowymi danymi. Metoda `dequeueReusableCell(withIdentifier:for:)` pozwoli nam uzyskać taką komórkę ze wspólnej puli, i w końcu skonfigurować ją odpowiednim elementem i finalnie ją zwrócić.

Systemowe klasy takie jak `UITableView` czy `UICollectionView`, to potężne elementy widoku wykorzystywane w praktycznie każdej aplikacji, a prawidłowe ich wykorzystanie jest kluczem do stworzenia szybkiej i wydajnej aplikacji. Na przykładzie klasy `UITableViewDataSource` widać jak dobrze zorganizowany i zaprojektowany kod jest kluczowy w prawidłowym działaniu aplikacji. Podążając za wytycznymi, trzymamy się ścieżki wyznaczonej w celu utrzymania stabilności oraz uzyskania jak największej wydajności i niezawodności kodu.

4.3.5. Protokoły

Widok **EPUBKit** deklaruje własne protokoły pomocnicze, aby przyjąć pewne normy, które będą miały zostać uszanowane przez nowo powstające klasy. Podejście te zapewnia spójność kodu pośród określonego rodzaju klas, lecz co najważniejsze, pozwala to na przyjęcie pewnych konwencji obowiązujących w projekcie[2]. Każdy z poniższych protokołów oznaczony jest słowem kluczowym *class*, więc stosować je będą mogły wyłącznie klasy.

```
1 protocol EKViewDataSource: class {
2     func build(from epubDocument: EPUBDocument)
3     func item(at indexPath: IndexPath) -> Any
4 }
```

Listing 4.46: Deklaracja protokołu `EKViewDataSource`

Wykaz 4.46 przedstawia deklarację protokołu `EKViewDataSource`, który powinien być stosowany przez każdą klasę która pełni rolę źródła danych w bibliotece **EPUBKit**. Na ten moment definiuje on dwie metody, pierwsza z nich jest metodą w której z założenia ma się odbywać budowanie z instancji `EPUBDocument`. Druga powinna zwracać obiekt znajdujący się na pozycji określonej przez indeks w formie instancji `IndexPath`.

```
1 protocol EKViewDataSourceDelegate: class {
2     func dataSourceDidFinishBuilding(_ dataSource: EKViewDataSource)
3 }
```

Listing 4.47: Deklaracja protokołu `EKViewDataSourceDelegate`

Kolejnym protokołem jest delegatem klasy stosującej protokół `EKViewDataSource`. Protokół `EKViewDataSourceDelegate` którego implementacja przedstawiona została na wykazie 4.47, definiuję metodę wywoływaną przez źródło danych na delegacie w momencie ukończenia

budowy modelu. To pozwala na reakcję ze strony delegata, tak jak miało to miejsce w klasie EKTableOfContentsViewController, która wymuszała przeładowanie danych w tabeli.

```
1 protocol EKTableOfContentsViewControllerDelegate: class {
2     func tableOfContentsView(_ tableOfContentsView: EKTableOfContentsViewController,
3                               didSelectRowAt indexPath: IndexPath)
```

Listing 4.48: Deklaracja protokołu EKTableOfContentsViewControllerDelegate

Ostatnim z przedstawionych protokołów jest EKTableOfContentsViewControllerDelegate (patrz: wykaz 4.48). Implementuje on metodę, której działanie zostało opisane przy okazji omawiania klasy stosującej ten protokół w podrozdziale 4.3.1.

4.4. Dystrybucja

Ta część będzie opisywała najpopularniejsze metody dystrybucji biblioteki na iOS. Najczęściej dystrybucja odbywa się poprzez menadżery paczek (ang. *Package Manager*), ponieważ traktowanie dodatkowych bibliotek czy narzędzi jako paczki czy moduły jest bardzo wygodną konwencją. Systemy takie jak CocoaPods czy Carthage wykorzystują jeden plik w którym programista może deklarować wykorzystywane moduły, a po wykonaniu odpowiedniej komendy moduły te zostaną pobrane z bazy menadżera paczek dołączone do przestrzeni roboczej dzięki czemu będą dostępne do użycia w plikach źródłowych projektu poprzez zainportowanie ich tak jak zademonstrowano to na wykazie 4.49.

```
import (module)
```

Listing 4.49: Sposób importowania dodatkowego modułu

Bardzo często zdarza się, że biblioteki w celach zyskania popularności zostają udostępnione bezpośrednio na portalu GitHub, w którym prezentowane są sposoby instalacji, oraz demonstracja API. Udostępnianie w tym miejscu swojej biblioteki jest dobrym pomysłem ze względu na to, że programiści zainteresowani jej wykorzystaniem mają wgląd w jej kod źródłowy jeszcze zanim dołączają ją do swojego projektu za pomocą menadżerów. GitHub pozwala na tak zwane *forkowanie* repozytorium, co oznacza przypisanie go do swojego profilu wraz z całą historią systemu kontroli wersji. W ten sposób każdy może zaimplementować własne funkcjonalności a następnie poprosić właściciela repozytorium o dołączenie ich do głównej gałęzi repozytorium za pomocą *pull-request*. Społeczność open-source na GitHub'ie jest obecna, i w momencie podchwycenia popularności przez repozytorium, niezależni programiści często dostosowują je pod swoje potrzeby, a potem często prezentują zmiany, które mogą zostać zatwierdzone przez właściciela. System kontroli wersji pozwala na używanie modułów, więc istnieje możliwość wykorzystania publicznie dostępnych bibliotek z GitHuba bez integracji projektu z któryms z menadżerów paczek.

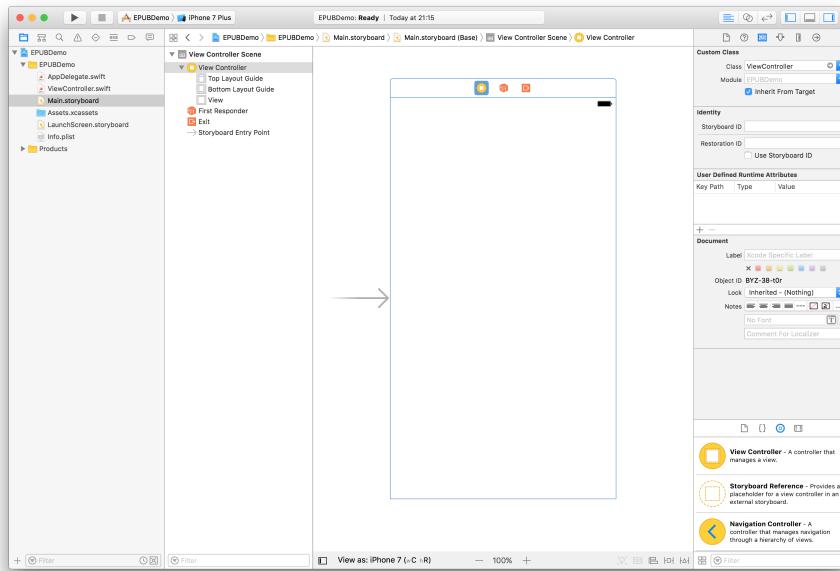
Aby udostępnić projekt na GitHubie jedyne co potrzebujemy to własny profil oraz repozytorium lokalne w którym znajduje się projekt. Jest to oczywiście najprostsze rozwiązanie, chociaż w innych przypadkach wcale nie jest to skomplikowane. Sprowadza się to zazwyczaj do wykonania pewnych poleceń z lini komend oraz wykonania innych równie trywialnych czynności. Dobrą praktyką jest udostępnianie

biblioteki na GitHubie oraz na wszystkich możliwych menadżerach paczek, zapewni to instalację w każdym projekcie niezależnie z jakim menadżerem jest zintegrowany. Niewątpliwie taki los czeka bibliotekę EPUBKit, która została stworzona z myślą o udostępnieniu jej publicznie.

Biblioteka **EPUBKit** została udostępniona na portalu GitHub pod adresem url: <https://github.com/witekbobrowski/EPUBKit>. Jest również dostępna do zainstalowania przy pomocy CocoaPods po dołączeniu jej w pliku *Podfile*. Szczegóły instalacji i wykorzystania w kolejnym rozdziale.

5. Aplikacja demonstracyjna

W tym rozdziale zaprezentuję jak stworzyć prostą aplikację na iOS wykorzystującą wcześniej przedstawioną bibliotekę **EPUBKit**. Pierwszym krokiem będzie otwarcie Xcode i stworzenie nowego projektu, analogicznie do tego co opisano w rozdziale czwartym. W tym przypadku jednak wybierzemy szablon *Single View Application* i damy nazwę projektowi **EPUBDemo**[9]. Po wybraniu lokalizacji w której chcemy zapisać aplikację, ukazuje nam się widok projektu w Xcode IDE tak jak miało to miejsce w przypadku tworzenia biblioteki. Biorąc pod uwagę demonstracyjne przeznaczenie aplikacji zamierzam ograniczyć się do minimum i skupić się na samej integracji z biblioteką oraz wykorzystanie *API* które jest przez nią dostarczane. Chcę przedstawić proces tworzenia widoków z których składa się aplikacja na iOS, rozpoczynając od omówienia wykorzystywanego wzorca architektonicznego *MVC*, przejdę do opisania elementów interfejsu użytkowych w aplikacji oraz klas które je reprezentują.

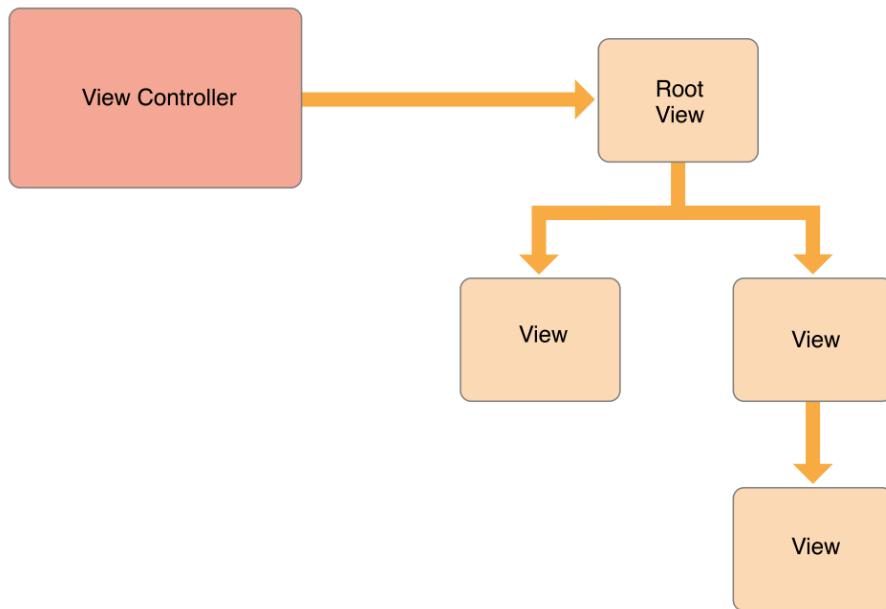


Rysunek 5.1: Plik *Main.storyboard* wyświetlony w Xcode.

Aplikacja będzie zbudowana przy pomocy *storyboardów* widocznych na rysunku 5.1, dzięki którym w prosty sposób można zaprojektować interfejs użytkownika przy pomocy gotowych komponentów, które można modyfikować według upodobań. Zanim jednak wy tłumaczę jak zaprojektować widok w *storyboardach*, należy poruszyć ważną sprawę, którą należy dobrze przemyśleć przed podejściem do programowania i komponowania widoku. Mam na myśli schemat *Model-Widok-Kontroler*, który każdy

widok aplikacji posiada przynajmniej jeden. Tworzenie widoku należy rozpocząć od ustalenia co będzie naszym modelem, a co będzie widokiem. Należy pamiętać, aby model i widok były rozdzielone i nigdy się ze sobą nie komunikowały, ponieważ kontroler decyduje w jaki sposób wyświetlić dane modelu na widoku. Między modelem, a widokiem istnieje niewidoczna linia, która nigdy nie może zostać przekroczena. To podejście pozwala nam w efektywny sposób zaprojektować architekturę aplikacji składającą się z wielu *MVC*, a to zapewni znakomitą organizację kodu.

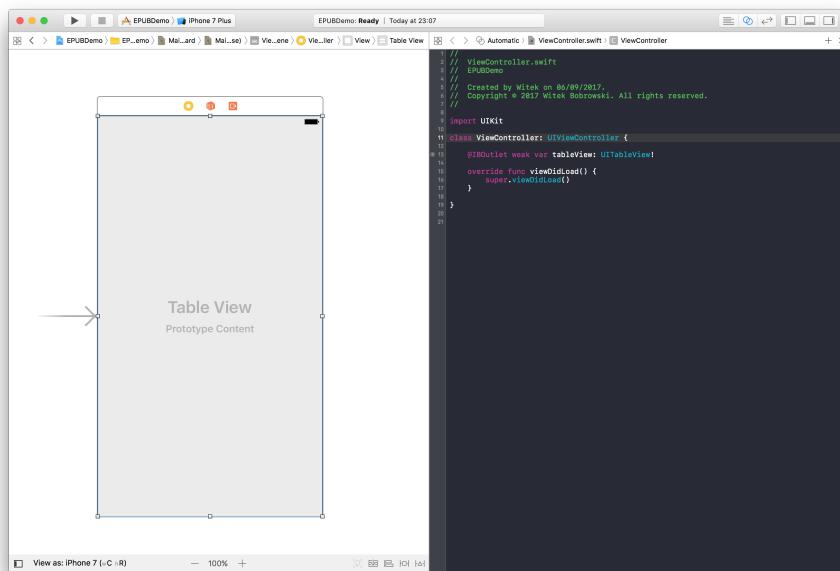
Zatem zastanówmy się co będzie naszym *MVC*. Pierwszy widok będzie przedstawiał listę książek, które znajdują się w aplikacji. W przypadku wybrania jednej z nich zostaniemy przeniesieni do widoku który dostarcza **EPUBKit**. Modelem pierwszego widoku więc będą książki zawarte w aplikacji, a książki naturalnie będą w formie instancji klasy `EPUBDocument`, którą również dostarcza **EPUBKit**. Widokiem jak już wspomniano będzie lista, a więc użyjemy komponentu `UITableView`, który pozwala na wyświetlenie komórek jedna pod drugą. `UITableView` dziedziczy z klasy `UIScrollView` co zapewnia naszej tabeli możliwość przewijania. Chcielibyśmy aby każda komórka w tabeli reprezentowała jedną z książek zawartych w aplikacji. Dobrze by było aby komórka ta poza tytułem książki wyświetlała również autora oraz miniaturkę okładki. W tym celu rozszerzymy klasę `UITableViewCell` w celu stworzenia własnej klasy która będzie spełniała założone wymagania, reprezentując prototypową komórkę zaprojektowaną przy użyciu *Xibów*. Kontrolerem będzie własna klasa `ViewController` rozszerzająca klasę `UIViewController` (*View Controllerem*, która określa się kontenerem agregującym widoki, tak jak przedstawiono na rysunku 5.2). Deklaruje między innymi metody wywoływane w każdym z momentów cyklu życia *View Controllerem* reprezentującego jedno *okno* widoczne w *storyboardach*. Nie należy nazywać *View Controller'a* oknem. Użyłem tego słowa, aby określić wizualną reprezentację *View Controller'a*. *Okno* reprezentowane przez klasę `UIWindow`, w nomenklaturze programowania na iOS jest instancją w której znajdują się wszystkie *View Controller'y*.



Rysunek 5.2: Związek *View Controllera* z jego widokami[7]

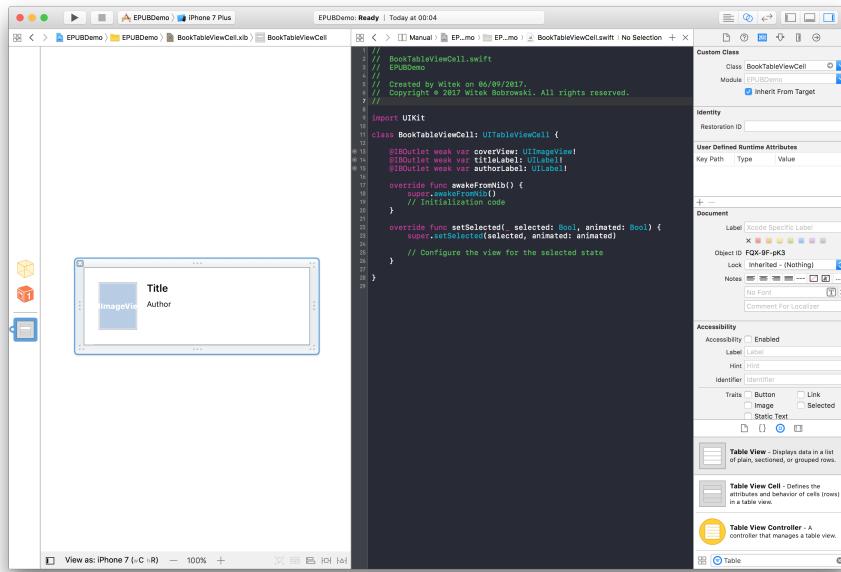
Skoro *MVC* widoku zostało ustalone, można rozpoczęć jego tworzenie, a ponieważ wykorzystanie

biblioteki zostanie przedstawione w kolejnym podrozdziale, zacznę od kompozycji widoku. Na głównym widoku *View Controllera* upuszczam element interfejsu *UITableView*, a następnie ustawiam go względem widoku nadzielnego, przyczepiam go do jego granic z odległością **0 pikseli**, co oznacza, że *UITableView* będzie rozszerzało się wraz z nim, przez co zawsze będzie rozciągnięte na całym ekranie i zachowa swoje proporcje niezależnie od rozdzielczości ekranu urządzenia oraz jego orientacji. Następnie włączam *Assistant editor* w Xcode, dzięki czemu obok pliku *Main.storyboard* widoczny jest plik *ViewController.swift* tak jak jest to zaprezentowane na rysunku 5.3. Klasa *ViewController* reprezentuje *View Controller* widoczny w *storyboardach* (tak jest ustawione w jego właściwościach), dlatego Xcode pozwoli na stworzenie referencji między świeżo umieszczonym *UITableView*, a klasą *ViewController*. W tym celu należy przytrzymać klawisz *crtl* na klawiaturze i przeciągnąć myszką chwytyając element w *storyboardach* i upuszczając go w klasie odpowiadającej danemu *View Controllerowi*. Zostanie wyświetcone okienko dialogowe na którym trzeba ustawić szczegóły referencji która ma zostać stworzona. Finalnie w klasie pojawi się zmienna oznaczona atrybutem `@IBOutlet` odnosząca się do wcześniej stworzonego elementu w *storyboardach*.



Rysunek 5.3: Dzięki asystującemu edytorowi, Xcode pozwala na podgląd dwóch plików jeden obok drugiego

Następnie można przejść do zaprojektowania komórki, której instancję będą wyświetlane w tabeli. W tym celu zostanie stworzona nowa klasa *BookTableViewCell* dziedzicząca po *UITableViewCell*, oraz plik *BookTableViewCell.xib* w którym zaprojektuję widok komórki sposobem znanym ze *storyboardów*. Widok ten będzie posiadał instancję *UIImageView* reprezentującą widok przeznaczony do wyświetlania obrazów oraz dwie instancje *UILabel*, elementu przedstawiającego etykietę. Jak można zaobserwować na rysunku 5.4, widok w pliku *Xib* ma określona klasę reprezentującą siebie, co jest absolutnie wymagane jeżeli zamierzamy użyć tego widoku w aplikacji.



Rysunek 5.4: Pliki *Xib* pozwalają na zaprojektowanie niezależnego widoku

5.1. Wykorzystanie biblioteki w aplikacji

Aby zaimplementować logikę aplikacji, należałoby w tym miejscu dołączyć bibliotekę **EPUBKit** do przestrzeni roboczej aplikacji. W tym celu należy zainstalować menadżera paczek *CocoaPods* i zainicjalizować *pods*'y w folderze zawierającym projekt **EPUBDemo** przy pomocy komendy *pod init*. Ta instrukcja stworzy plik *Podfile*, w którym będziemy trzymać referencje zależności aplikacji. Pierwszą z nich będzie naturalnie **EPUBKit**, więc plik *Podfile* powinien wyglądać tak jak na wykazie 5.1. Następnie należy po raz kolejny przejść do terminalu i uruchomić komendę *pod install*, która zainstaluje wszystkie wymienione w *Podfile* zależności i stworzy dedykowaną dla projektu przestrzeń roboczą. W tym momencie należy wyłączyć Xcode jeżeli uruchomiony jest plik projektu, i otworzyć nową przestrzeń roboczą. **EPUBKit** został zainstalowany, i jest gotowy do użytku, a więc można go zimportować instrukcją `import EPUBKit` w dowolnym pliku źródłowym swift.

```

1 source 'https://github.com/CocoaPods/Specs.git'
2 platform :ios, '10.3'
3 use_frameworks!
4 target 'EPUBDemo' do
5   pod 'EPUBKit', :git => 'https://github.com/witekbobrowski/EPUBKit.git'
6 end

```

Listing 5.1: Struktura modelu EPUBKit

5.1.1. BookTableViewCell

Komórka tabeli wyświetlającej książkę, jest relatywnie prostym widokiem. Klasa nie wymaga skomplikowanej konfiguracji, a poza referencjami widoków znajduje się tutaj wyłącznie jedna własność, która odnosi się do konkretnej instancji **EPUBDocument**. Jak przedstawiono na wykazie 5.2, w momencie

nadpisania właściwości `epubDocument`, wywołana zostanie metoda konfiguracji, i wypełni ona widok danymi.

```
1 class BookTableViewCell: UITableViewCell {
2     @IBOutlet weak var coverView: UIImageView!
3     @IBOutlet weak var titleLabel: UILabel!
4     @IBOutlet weak var authorLabel: UILabel!
5     public var epubDocument: EPUBDocument?
6     didSet {
7         configure()
8     }
9 }
10 }
11 //MARK: Configuration
12 extension BookTableViewCell {
13     fileprivate func configure() {
14         titleLabel.text = epubDocument?.title
15         authorLabel.text = epubDocument?.author
16         coverView.contentMode = UIViewContentMode.scaleAspectFit
17         if let cover = try? UIImage(data: Data(contentsOf: epubDocument?.cover ??
18             URL(fileURLWithPath: ""))
19             coverView.image = cover
20     }
21 }
```

Listing 5.2: Deklaracja klasy `UITableViewCell`

5.1.2. ViewController

Główny widok aplikacji, czyli lista książek została zadeklarowana w sposób widoczny na wykazie 5.3. Definiuje on zmienną `availableBooks`, która jest tablicą przechowującą nazwy plików .epub dostępnych w aplikacji. W tym rozdziale nie będę omawiał ani konfiguracji widoku, ani organizacji modelu danych dla tabeli.

```
1 class ViewController: UIViewController {
2     @IBOutlet fileprivate weak var tableView: UITableView!
3     fileprivate var availableBooks: [String] = []
4     fileprivate var model: [EPUBDocument] = [] {
5         didSet {
6             tableView.reloadData()
7         }
8     }
9     override func viewDidLoad() {
10         super.viewDidLoad()
11         configure()
12         rebuildModel()
13     }
}
```

```
14 }
```

Listing 5.3: Deklaracja klasy `UIViewController`

To co natomiast chciałbym poruszyć, to sposób inicjalizowania widoku z biblioteki **EPUBKit**. Otóż w celu przeniesienia się do widoku dokumentu, w pierwszej kolejności użytkownik musi wybrać odpowiednią pozycję w tabeli, co wywoła metodę delegata, której implementacja została przedstawiona na wykazie 5.4.

```
1 func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {
2     let bundle = Bundle(identifier: "org.cocoapods.EPUBKit")
3     if bundle != nil {
4         let ekViewController = UIStoryboard(name: "Main", bundle: bundle).
5             instantiateViewController(withIdentifier: "EKViewController") as!
6             EKViewController
7         ekViewController.epubDocument = model[indexPath.row]
8         navigationController?.pushViewController(ekViewController, animated: true)
9     }
10 }
```

Listing 5.4: Deklaracja metody delegata `UITableView`

W momencie wybrania któregoś z rzędów w tabeli, metoda delegata zainicjalizuje kontroler `EKViewController` przy pomocy instancji klasy `Bundle`, dzięki której otrzymujemy dostęp do plików znajdujących się w bibliotece. Następnie przypisujemy do właściwości stałej `ekViewController` dokument `EPUBDocument`, aby na końcu nałożyć nowy kontroler na kontroler nawigacji, co skutkuje pojawiением się widoku już znanego i omówionego w poprzednim rozdziale. Jedyne co pozostało to przerzucić do projektu aplikacji pliki EPUB, i uwzględnić ich nazwę bez rozszerzenia, w tablicy `availableBooks` oraz uruchomić aplikację.

To jest cała logika, którą wykonać musi aplikacja. Zainicjalizować instancję `EPUBDocument` oraz stworzyć widok z paczki biblioteki. Pierwotne założenie zostało spełnione, to znaczy udało się stworzyć bibliotekę, dzięki której można w bardzo prostu sposób stworzyć obiekt dokumentu EPUB, a następnie go wyświetlić.

6. Podsumowanie

Celem tej pracy było opisanie stworzonego przeze mnie projektu **EPUBKit**, który jest biblioteką napisaną w swiftcie na iOS. Starałem się również w całej pracy przedstawić sam język swift, ponieważ dla wielu jest on nieznany ze względu na to, że jeszcze jest nowością. Ta praca dokumentuje również specyfikację formatu publikacji elektronicznej EPUB, oraz pokazuje w jaki sposób należy obchodzić się z tym formatem w przypadku, gdy chcemy wyciągnąć z niego pewne informacje. **EPUBKit** na jego podstawie budował zupełnie nową strukturę danych w formie klasy EPUBDocument, którą można następnie wyświetlić przy pomocy klas widoku biblioteki.

Pomimo tego, że EPUB jest tak mocno wyspecyfikowanym formatem, różnice między wersjami (głównie 2.0 versus 3.0) są dość znaczące co nie powinno dziwić zważając na okres czasu, które je dzie- li. Największą trudność przy parsowaniu i wyświetlaniu takiego dokumentu, sprawia jednak nie różnica między specyfikacjami, lecz luźne interpretacje formatu stosowane przez twórców. W wielu aspektach publikacje różnią się od siebie co znacznie utrudniało moją pracę, szczególnie w przypadku implemen-tacji widoku. Koniec końców musiałem uciekać się do takich środków jak chociażby umieszczanie wła-snych elementów *HTML* pomiędzy dokumentami publikacji, w celu ujednolicenia sposobu nawigacji po dokumencie. Format EPUB jest fascynującą technologią, myślę, że szczególnie ze względu na to iż jest rozwijany przez środowisko *open-source*, lecz przede wszystkim ze względu na jego możliwości. Wyprzedza on znacznie takie formaty jak MOBI, i na pewno będzie on standardem, który nadal będzie dominował na rynku publikacji elektronicznych.

Programowanie na iOS jest jednak ważniejszym tematem tej pracy, niż sam format EPUB. Przed-stawiłem stan tej technologii na dzień dzisiejszy, i starałem się wytlumaczyć wszystkie jej najbardziej elementarne paradygmaty w ramach rzetelnego raportu. Platforma ta rozwija się coraz mocniej z roku na rok, i nic nie wskazuje na to iż miało by się to zmienić.

Bibliografia

- [1] Apple Inc. *Apple Releases iOS 8 SDK With Over 4,000 New APIs*, JUNE 2, 2014.
- [2] A. V. Chris Eidhof, Ole Begemann. *Advanced Swift*. objc.io, 2017.
- [3] I. D. P. Forum. Epub 3 core media types, OCT 2016.
- [4] I. D. P. Forum. Epub 3.1, recommended specification, JAN 2017.
- [5] I. D. P. Forum. Epub content documents 3.1, recommended specification, JAN 2017.
- [6] M. Garrish. *What is EPUB 3?* O'Reilly Media, 2011.
- [7] A. Inc. View controller programming guide for ios, 2015.
- [8] A. Inc. *The Swift Programming Language (Swift 3.1)*. Apple Inc., 2017.
- [9] A. Inc. *Using Swift with Cocoa and Objective-C (Swift 3.1)*. Apple Inc., 2017.
- [10] N. Ingraham. Apple shipped its billionth ios device this past november. *The Verge*, 2015.
- [11] C. Lattner. Llvm. In A. Brown and G. Wilson, editors, *The Architecture of Open Source Applications*, chapter LLVM. Lulu, 2011.