

**ĐẠI HỌC QUỐC GIA HÀ NỘI
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ**



Nguyễn Việt Minh Nghĩa

ỨNG DỤNG ĐỌC TRUYỀN TRANH

**KHÓA LUẬN TỐT NGHIỆP ĐẠI HỌC HỆ CHÍNH QUY
Ngành: Công nghệ Thông tin**

HÀ NỘI - 2021

**ĐẠI HỌC QUỐC GIA HÀ NỘI
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ**

Nguyễn Việt Minh Nghĩa

ỨNG DỤNG ĐỌC TRUYỀN TRANH

**KHÓA LUẬN TỐT NGHIỆP ĐẠI HỌC HỆ CHÍNH QUY
Ngành: Công nghệ Thông tin**

Cán bộ hướng dẫn: Nguyễn Văn Vinh

HÀ NỘI - 2021

LỜI CAM ĐOAN

Tôi cam đoan khóa luận này là do cá nhân tôi tự viết, không sao chép công trình của ai khác. Tuy nhiên, trong khóa luận này, có nhiều đoạn được tham khảo, tổng hợp từ nhiều nguồn. Các đoạn tham khảo đó được đánh dấu, và ghi rõ ở phần Tài liệu Tham khảo ở cuối khóa luận.

Người viết

LỜI CHẤP THUẬN CỦA CÁN BỘ HƯỚNG DẪN

Cán bộ hướng dẫn

LỜI CẢM ƠN

Phần này sẽ được viết khi phù hợp.

TÓM TẮT

Tóm tắt: Nhờ Internet, truyền hình và điện ảnh, văn hóa truyện tranh đã trở nên phổ biến toàn cầu, nhất là trong giới trẻ. Nhu cầu tiêu thụ truyện tranh tăng cao thúc đẩy sự phát triển của các trang web truyện tranh với ưu điểm là thuận tiện, tốc độ cập nhật truyện nhanh. Tuy nhiên, nguồn truyện tranh của những trang web này có chất lượng không cao. Một bộ phận người đọc kĩ tính chọn đọc và lưu trữ những tệp truyện được số hóa chất lượng cao, thường ở dạng nén đuôi CBR và CBZ. Xuất phát từ nhu cầu này, tôi muốn viết ứng dụng *yacv* có thể đọc các tệp truyện nén trên điện thoại Android. Khóa luận sẽ trình bày một số nền tảng của *yacv* như Android và Kotlin; sau đó là các ca sử dụng chính cùng với thiết kế và cài đặt của ứng dụng.

Từ khóa: *Android, coroutine, zip, comic*

Mục lục

1	Giới thiệu	1
1.1	Đặt vấn đề	1
1.2	Ứng dụng tương tự	2
1.3	Kết quả đạt được	3
1.4	Cấu trúc khóa luận	3
2	Kiến thức nền tảng	4
2.1	Hệ điều hành Android	4
2.1.1	Android Jetpack	6
2.1.2	Storage Access Framework	6
2.2	Ngôn ngữ lập trình Kotlin	7
2.2.1	Coroutine	8
2.3	Mẫu thiết kế MVVM và Kiến trúc khuyên dùng	11
2.3.1	Mẫu thiết kế MVVM	11
2.3.2	Kiến trúc Google khuyên dùng	13
2.4	Cơ sở dữ liệu SQLite	13
2.4.1	Thư viện ORM Room	14
2.4.2	Tìm kiếm văn bản	14
2.5	Định dạng tệp nén ZIP và CBZ	15
2.5.1	Định dạng tệp nén ZIP	15
2.5.2	Định dạng tệp truyen CBZ	16
3	Phân tích yêu cầu	17
3.1	Mô tả chung	17
3.1.1	Người dùng	17
3.1.2	Mục đích	17
3.2	Yêu cầu đặt ra	18
3.2.1	Yêu cầu chức năng	18
3.2.2	Yêu cầu phi chức năng	18

3.3	Phân tích yêu cầu	19
3.3.1	Quét các tệp truyện trên thiết bị	20
3.3.2	Duyệt truyện	21
3.3.3	Đọc truyện	24
3.3.4	Xem metadata truyện	25
3.3.5	Tìm kiếm truyện	27
3.3.6	Xóa truyện	29
4	Thiết kế	30
4.1	Module Database	31
4.2	Module View	33
4.2.1	Nguồn dữ liệu - Repository - DAO - ComicParser	34
4.2.2	Màn hình Quyền đọc	36
4.2.3	Màn hình Thư viện	36
4.2.4	Màn hình Thư mục	37
4.2.5	Màn hình Đọc truyện	38
4.2.6	Màn hình Metadata	38
4.2.7	Màn hình Tìm kiếm	39
4.2.8	Màn hình Danh sách truyện	41
4.3	Module Parser & Scanner	42
4.3.1	ComicParser	42
4.4	Module Image Loader	48
4.4.1	Image Extractor	48
4.4.2	Image Cache	50
5	Lập trình & Kiểm thử	52
5.1	Lập trình	52
5.1.1	Coroutine	52
5.1.2	CBZParser	55
5.2	Kiểm thử	56
5.2.1	ZipBuffer	56
5.2.2	Unit test	57
5.3	Sản phẩm	57
5.3.1	Màn hình Quyền đọc	57
5.3.2	Màn hình Duyệt truyện	58
5.3.3	Màn hình Đọc truyện	59
5.3.4	Màn hình Metadata	59

5.3.5 Màn hình Tìm kiếm	60
Phụ lục	62
A Giải thích các trường metadata	62
B Lược đồ XSD ComicInfo	64

Danh sách bảng

2.1	Hai kiểu API tương tranh hay dùng [8]	9
3.1	Cách yacy làm phẳng thư mục	23
4.1	Ba nguồn dữ liệu tương đương với Hình 2.9	34
4.2	Hai kiểu parser trong <code>ComicParser</code>	42
4.3	Danh sách các tệp ảnh trong một tệp truyện nén	48

Danh sách hình vẽ

2.1	Các phân lớp của Android [7]	5
2.2	Lập trình phi cấu trúc với GOTO [8]	8
2.3	Sự lộn xộn của lập trình phi cấu trúc [8]	8
2.4	Ba cấu trúc cơ bản: rẽ nhánh if, lặp for và gọi hàm [8]	9
2.5	Tương tranh phi cấu trúc với goroutine - API kiểu tương tranh [8]	9
2.6	Tương tranh cấu trúc dùng coroutine trong Kotlin [8]	10
2.7	Kiến trúc MVP [4]	12
2.8	Kiến trúc MVVM [5]	12
2.9	Kiến trúc Google khuyên dùng [2]	13
2.10	Cấu trúc tệp nén ZIP [9]	15
3.1	Mô tả Màn hình Tìm kiếm	28
4.1	Kiến trúc tổng quan của ứng dụng	30
4.2	Lược đồ cơ sở dữ liệu của yacv	32
4.3	Tương tác của ba nguồn dữ liệu, mũi tên gạch đứt thể hiện tính năng data binding	35
4.4	Trạng thái cấp quyền của một ứng dụng Android	36
4.5	Biểu đồ lớp của Màn hình Quyền đọc	36
4.6	Trạng thái của Màn hình Thư viện	37
4.7	Biểu đồ lớp của Màn hình Thư viện	37
4.8	Biểu đồ lớp của Màn hình Đọc truyện	38
4.9	Biểu đồ tuần tự của Màn hình Đọc truyện	38
4.10	Biểu đồ lớp của Màn hình Metadata	39
4.11	Biểu đồ các lớp liên quan đến Màn hình Tìm kiếm	41
4.12	Biểu đồ lớp của Màn hình Tìm kiếm	41
4.13	Biểu đồ lớp của Màn hình Danh sách truyện	42
4.14	ComicParser và các thành phần của nó	45
4.15	Biểu đồ tuần tự của ca sử dụng quét mới tệp truyện	47

5.1	Màn hình Quyền đọc	58
5.2	Hai Màn hình Duyệt truyện	58
5.3	Màn hình Đọc truyện	59
5.4	Màn hình Metadata	59
5.5	Hai Màn hình Tìm kiếm	60

Chương 1

Giới thiệu

1.1 Đặt vấn đề

Tại Đông Á và Đông Nam Á, văn hóa truyện tranh gốc Á, nhất là truyện tranh Nhật (manga), được đón nhận khá tích cực, đặc biệt trong giới trẻ. Thế hệ những người dưới 40 tuổi hiện nay được tiếp xúc với truyện tranh từ sớm, thông qua những cuốn truyện truyền tay và phim hoạt hình dựa trên truyện tranh, và tiếp tục đọc dù đã qua tuổi thiếu niên. Một số tác phẩm manga còn có lượng người đọc lớn trên toàn cầu như Doraemon, One Piece. Ở bên kia bán cầu, với sự thành công của vũ trụ điện ảnh Marvel và DC, truyện tranh phương Tây (comic) cũng được hồi sinh phần nào sau một thập kỷ thiêng sáng tạo và suy giảm doanh số sách in. Các bộ truyện siêu anh hùng, vốn trước đây chỉ phổ biến ở Hoa Kỳ, nay đang trên đường trở thành một phần của văn hóa đại chúng như vị thế của manga. Có thể nói, văn hóa truyện tranh nói chung đang ở thời kì phát triển mạnh, xét theo tiêu chí về độ phổ biến và thái độ đón nhận của xã hội.

Hiện nay, hầu hết mọi người đọc truyện qua các trang web tổng hợp truyện tranh. Những trang web này có hai ưu điểm chính:

- Số lượng: Mỗi trang cung cấp ít nhất hàng nghìn đầu truyện.
- Tốc độ: Tốc độ ra truyện rất nhanh. Với các bộ truyện nổi tiếng, thường chỉ trong vòng một vài giờ sau khi ra mắt, chương mới đã xuất hiện.

Tuy vậy, nhược điểm chính của những trang web này là chất lượng ảnh của truyện. Để giảm thời gian tải và tránh tốn băng thông, hình ảnh của truyện thường được nén khá nhiều, gây vỡ hình, mờ nhòe. Một bộ phận người đọc, hoặc kĩ tính, hoặc muốn sưu tầm truyện, thường chọn đọc những tệp truyện chất lượng cao, thường có đuôi CBZ hoặc CBR. Bản chất tệp truyện này là các tệp nén zip bình thường, bên trong có các tệp ảnh thông dụng như JPG, PNG. Tuy nhiên, do được tải hẳn về máy rồi mới đọc, những tệp

truyện này không bị giới hạn về băng thông hay thời gian, do đó hình ảnh trong tệp có thể có chất lượng rất cao.

Trong khóa luận này, tôi viết một ứng dụng Android nhằm phục vụ số ít người dùng có nhu cầu đọc truyện tranh chất lượng cao đã giới thiệu ở trên. Tên của ứng dụng là *yacv*, viết tắt của cụm từ tiếng Anh “Yet Another Comic Viewer”, tạm dịch là “Lại một ứng dụng xem truyện tranh nữa”. Hai tính năng chính duy nhất của ứng dụng là đọc và quản lý cơ bản (tìm kiếm, xóa) tệp truyện tranh có sẵn trên điện thoại.

Cần chú ý rằng ứng dụng *yacv* chỉ bao gồm các tính năng liên quan đến đọc truyện ngoại tuyến, đọc các tệp truyện có sẵn trên điện thoại người dùng. Ứng dụng không phải là ứng dụng khách cho các trang đọc truyện hiện có, hay có máy chủ tập trung riêng để cung cấp truyện.

1.2 Ứng dụng tương tự

Hiện có nhiều ứng dụng đọc truyện tranh ngoại tuyến như *yacv* trên chợ ứng dụng Google Play. Hai ứng dụng phổ biến nhất trong số này là *ComicScreen* và *Astonishing Comic Reader*. *ComicScreen* là ứng dụng có nhiều người dùng hơn. Các tính năng của *ComicScreen* giống với các tính năng của *yacv*, tuy nhiên *ComicScreen* có thêm nhiều chức năng phụ, đáng kể nhất là khả năng đọc từ mạng FTP/SMB và khả năng sửa ảnh trong file. *Astonishing Comic Reader* cũng có chức năng tương tự *yacv*, không hơn, tuy nhiên giao diện khá trau chuốt. Cả hai đều miễn phí và có quảng cáo, được cập nhật có thể nói là thường xuyên.

Một ngoại lệ đáng kể ở đây là ứng dụng mã nguồn mở *Tachiyomi*. Ứng dụng này có hệ thống phần mở rộng, cho phép đọc truyện ở các trang web truyện tranh. Khi web truyện tranh thay đổi, hoặc hỗ trợ thêm trang mới, chỉ cần tải về phần mở rộng tương ứng ở dạng ứng dụng APK. Tính năng này cùng mô hình mã nguồn mở khiến *Tachiyomi* mạnh hơn, cập nhật nhanh hơn toàn bộ các ứng dụng đã có và sẽ có. Tuy nhiên, *Tachiyomi* lại không thể được đưa lên Play Store, vì chính tính năng phần mở rộng đã vi phạm chính sách của Play Store [3].

Một điểm khác biệt quan trọng của *yacv* với các ứng dụng có sẵn là việc hỗ trợ metadata của tệp truyện tranh, do các ứng dụng có sẵn trên Play Store đa số bỏ qua thông tin này trong tệp truyện. Một trong số rất ít những ứng dụng hỗ trợ tính năng này là *Kuro Reader*, tuy nhiên đây là một tính năng trả phí.

1.3 Kết quả đạt được

Ứng dụng có các tính năng đủ dùng theo mục đích đã đề ra:

- Đọc file truyện CBZ
- Tìm kiếm truyện theo metadata

Tính năng đọc tệp truyện CBR hiện mới chỉ được cài đặt một phần, do khó khăn trong việc tích hợp thư viện đọc định dạng này.

1.4 Cấu trúc khóa luận

Các phần còn lại của khóa luận có cấu trúc như sau:

- Chương 2 - Kiến thức nền tảng: Giới thiệu sơ lược về ba nền tảng của ứng dụng, gồm hệ điều hành Android, ngôn ngữ lập trình Kotlin, và mẫu thiết kế MVVM; định dạng tệp nén ZIP cũng được giới thiệu vì liên quan trực tiếp đến ứng dụng.
- Chương 3 - Phân tích yêu cầu: Phân tích nhu cầu và ca sử dụng để có đặc tả yêu cầu.
- Chương 4 - Thiết kế: Thiết kế ứng dụng, gồm thiết kế cơ sở dữ liệu, giao diện, logic nghiệp vụ.
- Chương 5 - Lập trình & Kiểm thử: Một số cài đặt và ca kiểm thử trong ứng dụng sẽ được nêu một cách có chọn lọc.
- Chương 6 - Kết luận: Kết thúc khóa luận.

Sau phần Tài liệu Tham khảo, khóa luận có hai phụ lục về metadata của truyện tranh.

Chương 2

Kiến thức nền tảng

Chương này giới thiệu sơ qua về các nền tảng trong quá trình xây dựng ứng dụng.

- Hai nền tảng đầu tiên liên quan đến nhau, là tiền đề cho toàn bộ ứng dụng sẽ được giới thiệu trước, gồm hệ điều hành Android và ngôn ngữ lập trình Kotlin.
- Tiếp theo, lựa chọn về kiến trúc tổng quan, liên quan đến giao diện của ứng dụng được trình bày.
- Sau đó, cơ sở dữ liệu SQLite cùng một số phần mở rộng của nó dùng trong ứng dụng sẽ được nhắc qua.
- Cuối cùng là thông tin về CBZ - định dạng tệp tin mà ứng dụng đọc.

2.1 Hệ điều hành Android

Android là một hệ điều hành di động do Google phát triển. Android dùng nhân Linux và được thiết kế cho màn hình cảm ứng. Cùng với iOS của Apple, Android trở thành một phần không thể thiếu của cuộc cách mạng di động bắt đầu vào cuối những năm 2000.

Google mua lại phiên bản Android đầu của công ty khởi nghiệp cùng tên vào năm 2005, và phát triển nó từ đó. Ngoài Google, Android còn nhận đóng góp lớn từ cộng đồng, do có mã nguồn mở (phần lớn dùng giấy phép Apache); tên chính thức của dự án là Android Open Source Project. Dù vậy, mọi thiết bị Android thương mại đều có ứng dụng độc quyền. Ví dụ đáng kể là bộ Google Mobile Services, chứa những ứng dụng thiết yếu như trình duyệt Chrome hay chợ Play Store. Về mặt này, Android khá giống Chrome: thành phần cốt lõi kỹ thuật được phát triển theo mô hình mã nguồn mở (AOSP và Chromium), còn thành phần liên quan đến trải nghiệm người dùng được phát triển riêng.

Android được phân lớp như sau:

- Nhân Linux (Linux Kernel):

Android dùng nhánh hỗ trợ dài hạn (LTS) của Linux. Khác kiểu phát triển distro trên máy tính (chủ yếu thay đổi ngoài nhân), Google sửa nhân khá nhiều trước khi tích hợp.

- Lớp phần cứng trừu tượng (Hardware Abstraction Layer):

Tầng này đưa ra giao diện chung cho mỗi kiểu phần cứng (máy ảnh, loa,...), giúp tầng trên có thể dùng phần cứng mà không cần quan tâm chi tiết riêng.

- Android Runtime (ART):

Trên máy bàn, máy ảo Java (JVM) dịch bytecode thành mã máy. Trên Android, Android Runtime nhận nhiệm vụ này.

Chúng khác nhau ở chỗ ART *bên dịch* bytecode thành mã máy (trước khi chạy - AOT), còn JVM *thông dịch* bytecode thành mã máy (trong khi chạy).

- Thư viện C/C++:

Tầng này phục vụ một số ứng dụng dùng NDK (từ Java gọi C) như trò chơi điện tử.

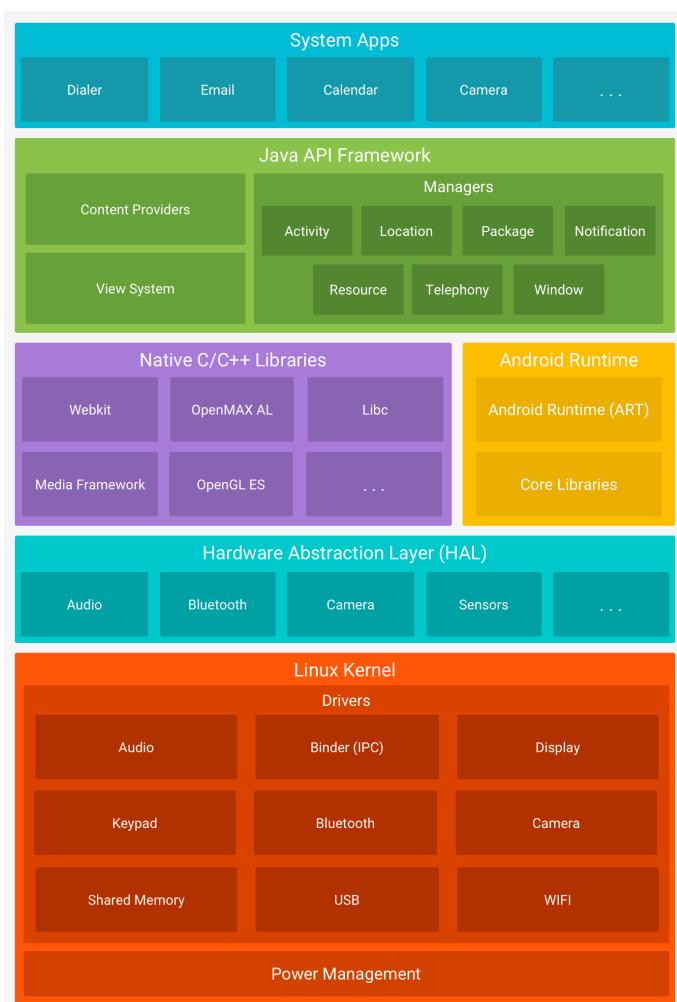
- Khung phát triển ứng dụng (Java API Framework):

Mọi ứng dụng Java được viết nhờ sử dụng các thành phần của tầng này thông qua API Java. Tầng này cung cấp toàn bộ tính năng của Android cho lập trình viên, bao gồm các yếu tố cơ bản như giao diện (View System), truy xuất,...

- Ứng dụng hệ thống (System Apps)

Android đi kèm với một số ứng dụng hệ thống như ứng dụng SMS, trình duyệt,...

Gần như mọi ứng dụng Android cơ bản đều dùng View System trong khung phát triển để viết giao diện, và yacy không là ngoại lệ. yacy còn sử dụng thành phần Content Provider, cụ thể là bộ Storage Access Framework, và sẽ được đề cập dưới đây.



Hình 2.1: Các phân lớp của Android [7]

2.1.1 Android Jetpack

Jetpack là bộ thư viện giúp viết ứng dụng Android nhanh gọn, ít lỗi hơn so với việc tự viết những đoạn mã tương tự. Jetpack gồm hai thành phần chính:

- AndroidX (trước gọi là Thư viện Hỗ trợ - Support Library): đưa API của hệ điều hành mới lên máy cũ
- Architecture Component: đưa ra *thư viện* hoàn toàn mới

Việc cập nhật Android rất khó khăn do phải chờ nhà sản xuất tối ưu. Do đó, Jetpack, nhất là AndroidX, rất cần thiết. Chú ý rằng Jetpack chỉ có ích cho lập trình viên (API mới tiện hơn thực ra là wrapper của API sẵn có), chứ không cập nhật tính năng hệ thống. yacy sử dụng nhiều thành phần của Jetpack, trong đó đáng kể đến ba thư viện sau:

- LiveData: giúp giao diện luôn được cập nhật theo dữ liệu mới nhất
- ViewModel: giúp tách dữ liệu và giao diện
- Room: đơn giản hóa việc lưu dữ liệu trong SQLite (sẽ được giới thiệu ở mục sau)

2.1.2 Storage Access Framework

Trước Android 4.4, Android không có sẵn hộp thoại chọn *tệp* (file picker). Mỗi ứng dụng lại phải tự viết hộp thoại chọn *tệp* cho riêng nó, dẫn đến nhiều vấn đề: giao diện không thống nhất, mã không được kiểm tra kĩ,... Từ bản 4.4, Android mới có hộp thoại này, là một phần trong bộ Storage Access Framework (từ đây gọi là SAF) [6].

Mục tiêu quan trọng nhất của SAF là trừu tượng hóa “nguồn dữ liệu tệp”. Ví dụ, SAF giúp ứng dụng không chỉ đọc tệp lưu sẵn trên điện thoại, mà còn đọc tệp lưu trong Google Drive, vì ứng dụng Drive khai báo với SAF nó là một nguồn dữ liệu tệp. Để làm được điều này, SAF ép buộc hai thay đổi sau:

- Chỉ cho phép đọc ghi tuần tự

Do không còn phân biệt được tệp lưu tại máy hay trên mạng, việc truy cập ngẫu nhiên vào tệp cũng không thể làm được.

- Dùng `DocumentFile` thay cho `File`

`DocumentFile` trừu tượng hóa tệp tin (tức ứng dụng không cần biết là tệp sẵn có hay tệp ở trên máy chủ), khác với `File` trong thư viện chuẩn chỉ hỗ trợ tệp lưu sẵn. SAF thậm chí không cho dùng `File` nữa.

Tương tự, đường dẫn thông thường chỉ cho tệp ngoại tuyến. Đường dẫn với SAF cần thể hiện nguồn dữ liệu là một ứng dụng. Do đó, SAF dùng URI làm đường dẫn.

Hai yêu cầu trên phù hợp với một hệ thống sẵn có trên Android là Content Provider. Hệ thống này cũng dùng URI làm đường dẫn, cũng dùng để cung cấp dữ liệu, cũng chỉ cho truy cập tuần tự. Do đó, SAF được cài đặt dưới dạng một Content Provider.

yacv chọn mốc Android cũ nhất còn hỗ trợ là 5.0 vì từ phiên bản này Android còn có hộp thoại chọn *thư mục*. Đây là một hộp thoại không thể thiếu với yacv, là bước đầu tiên để ứng dụng hoạt động, do đó cần dùng cài đặt hộp thoại tốt, “chính chủ” để tránh lỗi.

2.2 Ngôn ngữ lập trình Kotlin

Java là ngôn ngữ lập trình đầu tiên được hỗ trợ trên Android, nhưng không phải duy nhất. Từ 2019, Google khuyên lập trình viên viết ứng dụng trên Kotlin, một ngôn ngữ mới do JetBrains phát triển. Giới thiệu lần đầu vào năm 2011, Kotlin được định hướng trở thành lựa chọn thay thế cho Java. Điều đó thể hiện ở việc Kotlin tương thích hoàn toàn với Java (từ Java gọi được Kotlin và ngược lại), do cùng được biên dịch thành JVM bytecode.

Kotlin hơn Java ở tính ngắn gọn. Do được phát triển mới, Kotlin không cần tương thích ngược, cho phép dùng các cú pháp hiện đại, gọn ghẽ. Do được một công ty phát triển, Kotlin không cần chờ các cuộc họp phức tạp để đạt đồng thuận về tính năng mới. Đồng thời, công ty cũng mở mã nguồn của Kotlin và chương trình dịch, giúp đẩy nhanh quá trình phát triển và tạo thiện cảm cộng đồng cho một ngôn ngữ non trẻ.

Sau đây là tóm tắt một số đặc điểm kỹ thuật của Kotlin:

- Về mô hình, Kotlin hỗ trợ hướng đối tượng như Java, nhưng còn có hướng hàm, thể hiện ở tính năng hàm ẩn danh (lambda), và hàm được coi là first-class.
- Về hệ thống kiểu, Kotlin giống hệt Java:
 - Là kiểu mạnh (strongly typed), tức không cho phép chuyển kiểu ngầm
 - Là kiểu tĩnh (statically typed), tức kiểu được kiểm tra khi biên dịch (thay vì khi chạy, như Python, JavaScript,...)
- Về cú pháp, Kotlin gọn và hiện đại: bỏ dấu ; cuối dòng, template literal,...
- Về chống lỗi, Kotlin “né” `NullPointerException` do buộc người viết đánh dấu rõ ràng một đối tượng có thể bị `null` bằng hậu tố ? ở khai báo kiểu. Từ đó, Kotlin biết chính xác đối tượng có thể là `null` hay không, và buộc xử lí nếu có.

Do Google khuyên dùng Kotlin, tôi cho rằng khóa luận này là một cơ hội phù hợp để thử Kotlin thay vì dùng Java quen thuộc, và quyết định chọn viết yacv bằng Kotlin.

2.2.1 Coroutine

Giới thiệu

Một thư viện quan trọng của kotlin là *coroutine*. Coroutine giúp viết ứng dụng có tính tương tranh (concurrency) và bất đồng bộ (asynchronous) một cách đơn giản hơn.

Về cơ bản, coroutine giống với luồng (thread), nhưng nhẹ hơn. Coroutine dùng mô hình *đa nhiệm hợp tác* (cooperative multitasking), khác với luồng hay dùng đa nhiệm ưu tiên (preemptive multitasking). Mấu chốt khác biệt của chúng là đa nhiệm hợp tác có các “điểm dừng” do người viết tạo; khi chạy đến đó, coroutine có thể dừng, nhả CPU cho việc khác, rồi tiếp tục việc đang dở sau. Ngược lại, đa nhiệm ưu tiên có thể buộc luồng đang chạy ngừng lại bất kì lúc nào để ưu tiên chạy luồng khác. Đây cũng là điểm làm coroutine nhẹ hơn: việc chuyển ngữ cảnh (context switching) được kiểm soát và cắt giảm, do chuyển sang một coroutine khác chưa chắc đã chuyển sang một luồng hệ điều hành khác.

Roman Elizarov, trưởng dự án Kotlin, hướng coroutine trong Kotlin theo một ý tưởng mới: *tương tranh có cấu trúc* (structured concurrency, từ đây gọi tắt là SC). Ý tưởng này tiếp tục đơn giản hóa việc viết những đoạn mã tương tranh bằng cách áp đặt một cấu trúc cha-con. Kết quả là coroutine trong Kotlin hỗ trợ việc xử lý lỗi và ngừng tác vụ bất đồng bộ tốt hơn việc dùng luồng, hay các thư viện tương tranh như RxJava.

Coroutine giúp tăng tốc những đoạn mã chạy chậm trong yacc. SC giúp viết mã ngắn, rõ ràng. Do có tác động lớn, cả hai sẽ được giới thiệu kĩ hơn ở phần này.

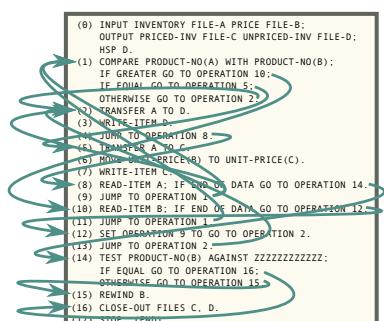
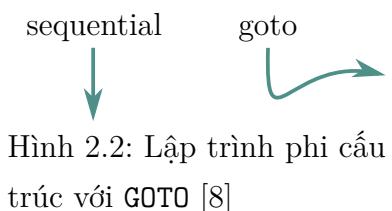
Bài học từ quá khứ: lập trình cấu trúc

Để hiểu SC, ta so sánh nó với *lập trình cấu trúc* (structured programming). Để hiểu về lập trình cấu trúc, ta phải tìm về *lập trình phi cấu trúc* (non-structured programming), với đặc điểm là lệnh nhảy GOTO.

Vấn đề của lập trình phi cấu trúc, hay của GOTO, gồm:

1. Khó nắm bắt luồng chương trình

Khi đã chạy GOTO, các lệnh sau nó không biết khi nào mới chạy, vì chương trình chuyển sang lệnh khác mà không trở lại. Luồng chạy trở thành một đống “mì trộn” như Hình 2.3, thay vì tuần tự từ trên xuống. Tệ hơn, tính trừu tượng bị phá vỡ: khi gọi hàm, thay vì có thể bỏ qua chi tiết bên trong, ta phải biết rõ để xem có lệnh nhảy bất ngờ không.



Hình 2.3: Sự lộn xộn của lập trình phi cấu trúc [8]

2. Không cài đặt được các chức năng mới (ngoại lệ, quản lý tài nguyên tự động,...)

Xét ví dụ Java (giả sử có GOTO) sau về quản lý tài nguyên tự động:

```
try (Scanner scanner = new Scanner(new File("f.txt"))) {  
    goto(SOMEWHERE); // Nhảy đi đâu mất  
}  
                           // Không chạy lệnh đóng
```

Như comment chỉ ra, không trả luồng điều khiển dẫn đến không thực hiện mục tiêu tự động đóng. Điều gần tương tự làm xử lý ngoại lệ và nhiều tính năng khác rất khó cài đặt, một khi ngôn ngữ cho phép GOTO.

Lập trình cấu trúc đơn giản hóa luồng chạy bằng cách giới hạn các lệnh nhảy còn if, for và gọi hàm. Khác biệt mấu chốt so với GOTO là chúng trả luồng điều khiển về điểm gọi, thể hiện rõ ở Hình 2.4. Theo định nghĩa, ba lệnh trên giải quyết được hậu quả 1. Đồng thời, do ngôn ngữ có cấu trúc (có call stack), hậu quả 2 cũng được giải quyết.

Ngày nay, ba cấu trúc trên đều có trong mọi ngôn ngữ lập trình, còn GOTO chỉ dùng trong hợp ngữ. Quá khứ cho thấy nếu áp dụng một số cấu trúc, ta có thể giải quyết vấn đề một cách tinh tế và gọn gàng. Ở đây, SC khắc phục một số điểm yếu của các API tương tranh truyền thống, giống cách lập trình cấu trúc đã làm.

Áp dụng vào hiện tại: tương tranh cấu trúc

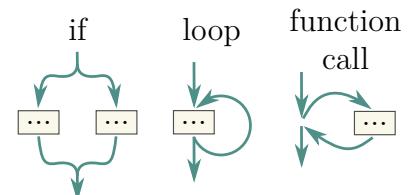
Trước hết, ta xem xét hai kiểu API tương tranh hay dùng hiện nay trong Bảng 2.1 [8]:

Bảng 2.1: Hai kiểu API tương tranh hay dùng [8]

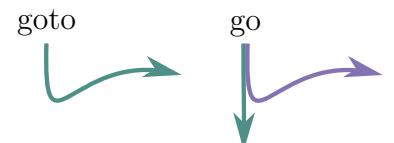
Tên	Giải thích	Ví dụ
Tương tranh	Chạy một hàm theo cách tương tranh với luồng chạy hiện tại	Thread(func).start() # Python
Bất đồng bộ	Chạy một hàm khi có sự kiện xảy ra (callback)	element.onclick = callback; // JS

Qua Hình 2.5, không khó để thấy rằng mọi vấn đề của lập trình phi cấu trúc đều lặp lại với hai API trên:

- Ta xem lại ví dụ quản lý tài nguyên tự động. Nếu có luồng thực thi khác tương tranh với luồng chính, thì



Hình 2.4: Ba cấu trúc cơ bản: rẽ nhánh if, lặp for và gọi hàm [8]



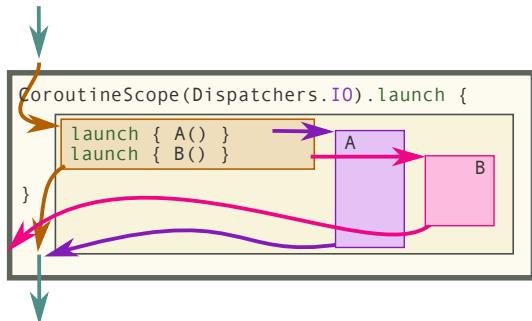
Hình 2.5: Tương tranh phi cấu trúc với goroutine - API kiểu tương tranh [8]

khi luồng chính đóng **Scanner**, có thể luồng kia đang đọc dở. Do đó, tính năng này không thể hoạt động.

- Với tính năng bắt ngoại lệ, nếu có ngoại lệ ở luồng tương tranh, ta cũng không có cách nào để biết, và buộc phải kệ nó.

Trên thực tế, có thể cài đặt chức năng trên với API hiện tại, tuy vậy đó đều là cách xử lí riêng, bất tiện. Ví dụ, ES6 có `catch()` để bắt ngoại lệ trong `Promise` mà không (thể) dùng cấu trúc `try-catch` sẵn có. Với SC, các vấn đề này đều được giải quyết.

Trong Hình 2.6, ta xét đoạn mã tương tranh dùng SC trong Kotlin.



Hình 2.6: Tương tranh cấu trúc dùng coroutine trong Kotlin [8]

Nguyên tắc của SC là: *coroutine cha chờ mọi coroutine con chạy xong, kể cả khi nó xong trước*. Nguyên tắc này đảm bảo rằng khi một hàm kết thúc, không còn tác vụ tương tranh nữa, và điểm gọi nhận lại luồng điều khiển hợp nhất (các mũi tên chụm lại ở góc trái dưới Hình 2.6). Đột nhiên, hai tính năng có vấn đề ở trên lại hoạt động:

- Quản lý tài nguyên tự động: do đảm bảo trả luồng điều khiển, tài nguyên đảm bảo

được đóng; do không còn tác vụ con, tài nguyên không bị đọc sau đóng.

- Bắt ngoại lệ: do cấu trúc cha-con (khác với các API hiện tại cho rằng hai tác vụ tương tranh là ngang hàng), coroutine con có thể ném ngoại lệ để coroutine cha bắt.

Chú ý là các API hiện tại không phải không làm được nguyên tắc trên, vấn đề là thực hiện một cách *tự động* và *đảm bảo*. Ví dụ, trong JS, để tuân theo SC, ta phải nhớ `await` mọi hàm `async`. Do không ràng buộc chặt chẽ, các tính năng ngôn ngữ mới vẫn khó cài đặt như đã phân tích.

Do trong các ngôn ngữ khác, nguyên tắc trên chỉ là một ca sử dụng, việc ép buộc viết theo ca sử dụng này đòi hỏi lập trình viên thay đổi suy nghĩ về tương tranh. Đổi lại, chương trình trở nên sáng rõ, giống những đoạn mã viết theo kiểu tuần tự truyền thống.

Một khi vấn đề tương tranh được giải quyết hoặc đơn giản hóa, việc song song hóa (parallelization) để tăng tốc ứng dụng chỉ còn là một chi tiết cài đặt.

Tóm tắt

Coroutine với SC là một trong những tính năng quan trọng nhất của Kotlin, giúp tăng tốc những đoạn mã chạy chậm trong yacy. Mẫu chốt của SC được tóm gọn trong Hình 2.6. Dù khá mới (Martin Sústrík, tác giả của ZeroMQ, nêu ý tưởng này năm 2016), mô hình này được cải thiện liên tục, có thư viện ở nhiều ngôn ngữ như Java (Loom), Python (Trio),... Điều này cho thấy ý tưởng có ý nghĩa lớn, giúp đơn giản hóa tư duy về tương tranh.

2.3 Mẫu thiết kế MVVM và Kiến trúc khuyên dùng

2.3.1 Mẫu thiết kế MVVM

Cũng như các tác vụ lập trình khác, lập trình giao diện sử dụng nguyên lý Separation of Concern, hiểu đơn giản là chia tách chức năng. Nhiều năm kinh nghiệm cho thấy giao diện nên được chia làm hai phần chính tách biệt nhau:

- Model: dữ liệu để hiển thị (trả lời câu hỏi “cái gì”); liên quan đến đối tượng, mảng,...
- View: cách để hiển thị dữ liệu đó (trả lời câu hỏi “như thế nào”); liên quan đến các yếu tố giao diện như nút, hộp thoại,...

Sự tách biệt thể hiện ở chỗ Model không được biết View. Khi này, giao diện và nghiệp vụ có thể phát triển khá độc lập với nhau, giúp giảm thời gian phát triển. Ngược lại, View có biết Model không là tùy vào cách triển khai cụ thể; có sự bất đối xứng này là do View luôn liên quan chặt chẽ đến framework, khác với Model thường đơn giản.

Khó khăn ở đây là làm sao để kết nối Model và View. Nhiều mô hình cố giải quyết vấn đề này, tiêu biểu là MVC, MVP và MVVM. Ta xem xét chúng để thấy rằng MVVM phù hợp nhất với Android, do đó được chọn làm nền tảng cho Kiến trúc Google khuyên dùng.

MVC: Model - View - Controller

Phương hướng đầu tiên được thử là MVC, vốn phổ biến vào thời điểm Android ra đời. Do được phát minh từ lâu và không được định nghĩa chặt chẽ, nên không có một mô hình cụ thể về cách ba thành phần trên tương tác. Tuy vậy, vẫn có vài điểm chung không đổi:

1. Controller nhận thao tác người dùng
2. Sau đó, controller cập nhật Model và View

Ngay ở đây, ta đã thấy điểm yếu của MVC khi áp dụng vào Android. Trong Android, ứng dụng sử dụng Activity và Fragment để viết giao diện. Hai đối tượng này cũng kiêm luôn việc nhận thao tác người dùng, tức chúng là cả *View* và *Controller*. Mục đích tách ra ba đối tượng do đó không thể làm được.

Hiện nay, MVC trên Android được coi là lỗi thời, không phù hợp.

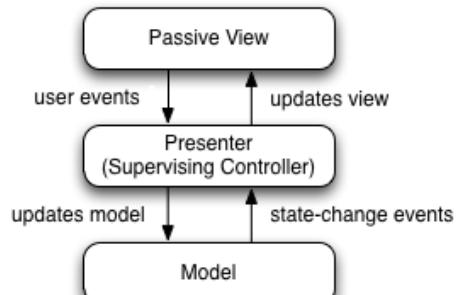
MVP: Model - View - Presenter

Năm 2012, Robert Martin “Uncle Bob” xuất bản một bài viết nổi tiếng về kiến trúc phần mềm: Clean Architecture. MVP, vốn được phát triển từ lâu, được đông đảo lập trình viên

chọn để triển khai Clean Architecture trên Android. Trước khi Google chọn MVVM, đây là hướng đi mới, có kì vọng cao sau nhiều thất bại trong việc đưa MVC vào Android.

Nhiệm vụ của ba thành phần như sau:

- Model: vẫn như trong MVC
- View: hiển thị dữ liệu; nhận tương tác người dùng để chuyển sang Presenter
- Presenter: trung gian giữa Model và View: nhận tương tác từ View, gọi/thay đổi Model, cập nhật View



Hình 2.7: Kiến trúc MVP [4]

Ta thấy điểm yếu View-Controller nhập nhằng được khắc phục, khi View kiêm luôn việc nhận tương tác. Đồng thời, Model và View hoàn toàn không biết nhau, đúng theo nguyên lý tách lớp của Clean Architecture.

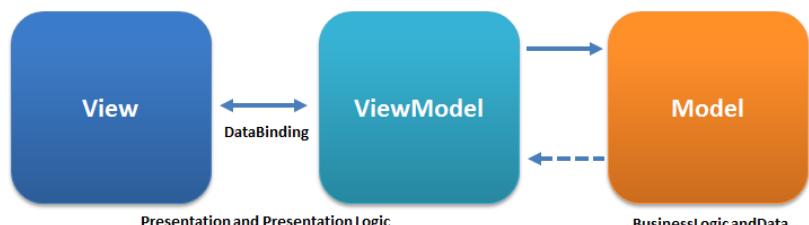
Ta xét một ứng dụng ToDo đơn giản, trong đó các công việc có thể được đánh dấu đã hoàn thành. Ứng dụng hiển thị số việc đã và chưa hoàn thành như sau:

1. Presenter lấy tất cả công việc trong Model
2. Presenter đếm số việc hoàn thành, chưa hoàn thành
3. Presenter gọi hàm của View, truyền hai số đếm được ở trên vào

Đến đây, thiết kế đã khá hoàn chỉnh và phù hợp với Android.

MVVM: Model - View - View Model

Ta quay về chủ đề chính: MVVM. MVVM giống MVP ở chỗ View Model (từ đây gọi tắt là VM) kết nối View và Model như Presenter. Điểm khác biệt là cách truyền dữ liệu [5]:



Hình 2.8: Kiến trúc MVVM [5]

- Trong MVP, Presenter gọi hàm của View để truyền dữ liệu cho View
- Trong MVVM, VM dùng *data binding* để truyền dữ liệu cho View

Data binding là cơ chế để *tự động* đưa dữ liệu vào thành phần hiển thị. Quay lại ví dụ ToDo ở trên, nếu dùng data binding để “gắn” (bind) danh sách công việc vào View, thì khi danh sách thay đổi, View cũng tự động thay đổi theo.

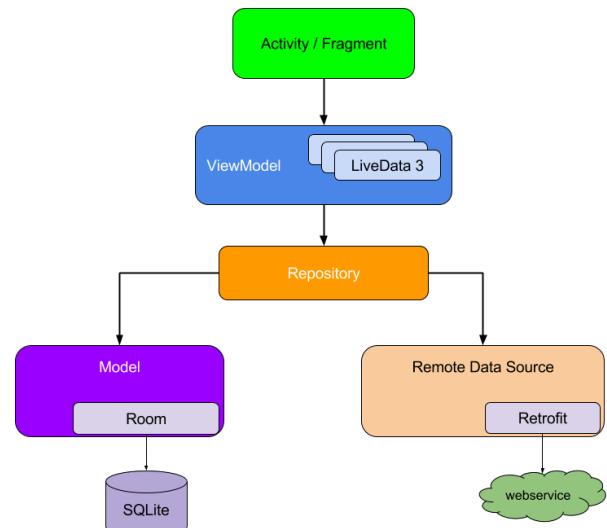
Do dùng data binding thay vì gọi hàm thủ công, VM không cần có tham chiếu tới View, khác với Presenter. Điều này giúp liên kết View - VM thêm *lỏng lẻo* (loose coupling), giúp kiểm thử dễ dàng hơn. Phần còn lại của hai mô hình giống nhau: View cần biết VM để chuyển tương tác; VM cần biết Model để lấy dữ liệu.

Do là mô hình phù hợp nhất trong cả ba với riêng Android, MVVM được chọn làm nền tảng cho Kiến trúc Google khuyên dùng.

2.3.2 Kiến trúc Google khuyên dùng

Kiến trúc Google khuyên dùng có gốc là mô hình MVVM, có dạng như Hình 2.9:

- Repository là Model trong MVVM, giúp VM lấy dữ liệu mà không cần quan tâm dữ liệu lấy từ đâu: cơ sở dữ liệu, gọi API qua mạng,...
- LiveData là cơ chế data binding dùng luồng dữ liệu (stream)
- Activity/Fragment là View



Repository là một điểm được chi tiết hóa so với MVVM. Google cho rằng một ứng dụng không nên hoàn toàn vô dụng nếu không có mạng. Do đó, cần có hai nguồn dữ liệu: dữ liệu từ máy chủ, và dữ liệu đệm, ngoại tuyến. Khi có nhiều nguồn dữ liệu, mẫu thiết kế Repository là lựa chọn hiển nhiên để trừu tượng hóa chúng.

yacyv sử dụng kiến trúc này, dù không có tính năng liên quan đến mạng. Lý do là yacyv có tính năng quét truyền hoạt động chậm giống như giao tiếp mạng, nên cần dữ liệu đệm và dữ liệu quét thực tế.

2.4 Cơ sở dữ liệu SQLite

SQLite là một hệ quản trị cơ sở dữ liệu quan hệ (RDBMS). Từ “Lite” trong tên có nghĩa là “nhỏ”, thể hiện mục tiêu thiết kế chính của nó là nhỏ gọn. SQLite có thể được nhúng vào phần mềm khác ở dạng thư viện, thay vì là một phần mềm riêng với kiến trúc khách-chủ

như MySQL,... Ngay từ những phiên bản đầu, Android đã tích hợp SQLite, giúp lập trình viên không phải nhúng SQLite vào từng ứng dụng.

Để đạt mục tiêu, SQLite chỉ giữ các tính năng SQL cốt lõi (tạo/đọc/sửa/xóa), giao dịch (có ACID), và chỉ tối ưu cho việc truy cập từ một ứng dụng một lúc. Các tính năng thường có trong RDBMS cho máy chủ, như nhân bản (replication), chia dữ liệu tự động (sharding), khóa dòng, đọc ghi nhiều luồng cùng lúc,... được loại bỏ. Do đó, với nhu cầu lưu trữ đơn giản, SQLite vừa nhanh vừa gọn.

yacyv dùng SQLite để lưu đệm thông tin truyền, tránh quét nhiều lần.

2.4.1 Thư viện ORM Room

Room là một thư viện thuộc Jetpack. Đây có thể xem là một thư viện ORM đơn giản cho SQLite. Room tự động làm nhiều công việc liên quan đến SQL:

- Tạo bảng: Người viết chỉ cần khai báo đối tượng dữ liệu như lớp (class) thông thường, rồi đánh dấu với Annotation của Room. Sau đó, Room sinh các bảng tương ứng.
- Truy vấn: Người viết chỉ cần viết lệnh SQL. Sau đó, Room sinh hàm truy vấn tương ứng, chuyển dữ liệu dạng đối tượng sang dạng để lưu trong bảng và ngược lại. Room còn bắt lỗi lệnh SQL mà không cần chờ đến khi chạy.
- Kiểm soát lược đồ (schema): Khi thêm/sửa/xóa bảng/cột, Room luôn phát hiện và ép viết cơ chế cập nhật. Do đó, ứng dụng dùng lược đồ cũ khi được cập nhật sẽ biết cách sửa cơ sở dữ liệu đến phiên bản lược đồ mới.
- Tương thích với LiveData: Giúp View cập nhật theo cơ sở dữ liệu.

2.4.2 Tìm kiếm văn bản

Tìm kiếm văn bản (full-text search, hay gọi tắt là FTS) là một trong số ít các tính năng nâng cao được giữ lại trong SQLite. Cũng như các thư viện tìm kiếm khác, SQLite cài đặt chức năng này bằng chỉ mục đảo (inverted index) - một cấu trúc giống từ điển:

1. Khi dữ liệu văn bản được ghi, nó được tách thành các từ.
2. Các từ được đưa vào chỉ mục đảo: khóa là từ đó, giá trị là khóa đại diện `rowid`.

FTS khác với đánh chỉ mục thường (cũng là chỉ mục đảo nhưng cho kiểu dữ liệu thông thường) ở bước 1: *từng từ* được tách ra, còn chỉ mục thường dùng *cả* văn bản. Do đó, khi tìm từ lẻ, FTS có thể tìm hàng có từ đó rất nhanh. Điểm yếu của FTS là ghi chậm, kích cỡ lớn hơn chỉ mục thường. Nếu bản thân dữ liệu trong cột là một khối, ví dụ như email,

chỉ mục thường là đủ. Ngoài ra, ở bước này có thể dùng kỹ thuật rút gọn từ (stemming; chỉ đúng với tiếng Anh, ví dụ tìm "run" ra được cả "runs",...), giúp tìm kiếm linh động hơn.

yacv có tính năng tìm kiếm tiêu đề, tên nhân vật,... đều là những câu văn, đoạn văn. Do đó, FTS có vai trò không thể thiếu để tăng tốc tìm kiếm trong ứng dụng.

2.5 Định dạng tệp nén ZIP và CBZ

Các tệp truyện mà yacv đọc có định dạng CBZ, bản chất chính là tệp nén ZIP. Do yêu cầu của các phần sau, định dạng tệp ZIP cũng cần được trình bày ở mức cơ bản.

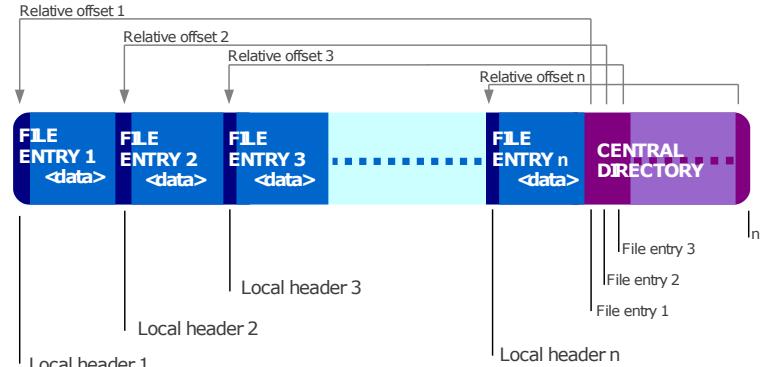
2.5.1 Định dạng tệp nén ZIP

ZIP là một định dạng tệp nén không mất mát (lossless). Được phát minh vào năm 1989 bởi Phil Katz, ZIP đã trở thành định dạng nén tiêu chuẩn, được hỗ trợ trên gần như mọi nền tảng, bao gồm Android.

ZIP thực chất là một định dạng chứa (container), chuyên chứa dữ liệu nén, chứ không phải thuật toán nén; thuật toán nén hay dùng nhất trong ZIP là DEFLATE. Một trong các mục tiêu của ZIP là giúp việc sửa tệp nén (thêm, sửa, xóa tệp con trong tệp ZIP) nhanh nhất có thể.

Mục tiêu trên dẫn đến thiết kế thể hiện trong Hình 2.10 [1]:

- Thuật toán nén mỗi tệp gốc thành một tệp nhị phân, ở đây gọi là *tệp nén lẻ* (data trong Hình 2.10). Sau đó, các tệp nén lẻ được nối thành tệp ZIP kết quả.



Hình 2.10: Cấu trúc tệp nén ZIP [9]

- Ở đầu mỗi tệp nén lẻ là một header gọi là *File Entry* để lưu thông tin liên quan, trong đó có *tên tệp gốc* và *vị trí bắt đầu* (offset), tức số byte tính từ đầu tệp ZIP đến tệp nén lẻ tương ứng.
- Ở cuối tệp ZIP, sau khi đã nối các tệp nén lẻ cùng header lại, các header được gom lại, lưu một lần nữa vào một cấu trúc gọi là *Central Directory*. Có thể so sánh File Entry như các *đề mục*, còn Central Directory là *mục lục*.

Ta phân tích kĩ hơn:

- Do nén riêng từng tệp, có thể dùng thuật toán khác nhau tối ưu với từng tệp.
- Cũng do nén riêng và có mục lục, việc sửa và đọc có thể được thực hiện với từng tệp lẻ, thay vì phải giải nén, sửa, rồi nén lại toàn bộ.
- Mục lục đặt ở cuối là tối ưu:
 - Giả sử mục lục đặt ở đầu. Khi nén hoặc sửa, toàn bộ các tệp nén lẻ phải di chuyển (thậm chí là nhiều lần) để tạo chỗ cho mục lục mới. Trường hợp này giống như thêm một phần tử vào đầu mảng: toàn bộ các phần tử sau bị đẩy lên để tạo chỗ trống.
 - Do mục lục nằm ở cuối tệp ZIP, nên khi sửa chỉ cần đẩy các tệp nén lẻ từ chỗ sửa. Trước đây tối ưu này rất quan trọng. Do đĩa mềm - phương tiện chia sẻ chủ yếu thời đó - có dung lượng nhỏ, tệp ZIP có thể phải cắt ra cho vừa [9]. Thiết kế này cho phép chỉ sửa lại dữ liệu ở một số đĩa, thay vì toàn bộ.

Tóm lại, cấu trúc tệp nén ZIP cho phép sửa và giải nén từng tệp gốc dễ dàng. Nhiều định dạng nén khác (TAR, 7z,...) không có tính năng này, do gộp toàn bộ các tệp vào rồi nén một thể. Khi đó, tệp nén được gọi là *đặc* (solid), và rất khó để đọc/sửa tệp lẻ.

2.5.2 Định dạng tệp truyện CBZ

Tệp truyện CBZ chỉ là một tệp nén ZIP thông thường, trong đó có:

- Các tệp ảnh trang truyện: Các tệp này là tệp ảnh JPEG, PNG,... thông thường. Tên tệp được đánh số tăng dần để biểu thị thứ tự trang. Tuy nhiên, không có một cấu trúc/định dạng tên tệp nào được thống nhất.
- (Tùy chọn) Một tệp metadata: Có nhiều định dạng metadata. Hiện nay, yacy chấp nhận định dạng ComicInfo, được trình bày ở Phụ lục B. Định dạng này lưu trong một tệp có tên `ComicInfo.xml`.

Chương 3

Phân tích yêu cầu

Chương này tìm hiểu đối tượng người dùng nhằm tới để tìm ra *Nhu cầu* của họ. Nhu cầu này sau đó được phân tích thành các *Yêu cầu chức năng* và *phi chức năng*. Cuối cùng, các chức năng được mổ xẻ, mô tả kĩ để được *Đặc tả ca sử dụng*, là bộ khung cho quá trình phát triển ứng dụng.

3.1 Mô tả chung

3.1.1 Người dùng

Ứng dụng yacy tập trung vào một số ít người dùng, là một trong hai nhóm sau:

- Người dùng sưu tầm truyện
- Người dùng có yêu cầu đọc truyện với chất lượng hình ảnh cao

Cả hai nhóm có điểm chung là kĩ tính, yêu cầu cao về trải nghiệm đọc truyện, cụ thể là về *chất lượng hình ảnh*. Cũng do kĩ tính, nên cả hai nhóm không cần nhiều chức năng, tuy nhiên từng chức năng cần hoàn thiện. Nhóm người dùng sưu tầm truyện còn có yêu cầu *xem thông tin (metadata)* của tệp truyện.

Tóm lại, ta thu được hai *Nhu cầu* của nhóm người dùng hướng đến:

- Đọc truyện trong tệp truyện
- Xem metadata

3.1.2 Mục đích

Trước khi đi vào chi tiết yêu cầu ở mục tiếp theo, tôi muốn làm rõ mục đích của sản phẩm đã nhắc ở Mục 1.1.

- Ứng dụng yacv chỉ bao gồm các tính năng liên quan đến đọc **truyện tranh** và là ứng dụng **ngoại tuyến** (tức đọc các tệp truyện có sẵn trên điện thoại).
- Ứng dụng *không phải* là ứng dụng khách cho các trang đọc truyện hiện có, hay có máy chủ tập trung riêng để cung cấp truyện.
- Ứng dụng *không có* khả năng đọc truyện đuôi PDF, cùng với các định dạng truyện thiên về chữ khác như TXT, EPUB.

Các giới hạn này nhằm tránh cho phần mềm quá phitic tạp với tôi, đồng thời phù hợp (không thừa thiếu chức năng) so với nhu cầu của nhóm người dùng mục tiêu đã nêu ở Mục 3.1.1.

3.2 Yêu cầu đặt ra

3.2.1 Yêu cầu chức năng

Ứng dụng có các chức năng chính sau:

- Quét các tệp truyện trên thiết bị
- Hiển thị danh sách truyện
- Đọc truyện
- Xem metadata truyện
- Tìm kiếm truyện
- Xóa truyện

3.2.2 Yêu cầu phi chức năng

Ứng dụng cần đạt một số tiêu chí sau:

- *Không trói buộc người dùng* (vendor lock-in): Đây là một tiêu chí quan trọng (trên thực tế nó có ảnh hưởng lớn đến thiết kế kỹ thuật của yacv). Điểm thể hiện rõ tiêu chí này là truyện phải được lưu trong *phân vùng chung*, để ứng dụng nào cũng có thể đọc, thay vì lưu trong phân vùng của riêng yacv. Do đó, người dùng có thể xóa, đổi ứng dụng bất kỳ lúc nào.

- Phản hồi nhanh: Các thao tác cần có thời gian phản hồi nhanh. Phản hồi nhanh không nhất thiết là thời gian thực thi ngắn, mà là luôn có các thông báo tiến độ cho người dùng, ví dụ như:
 - Luôn hiển thị thông báo chờ khi làm việc gì đó lâu
 - Nếu có nhiều kết quả tìm kiếm, hiển thị từ từ, đưa kết quả đã biết lên trước
- Tốc độ xem truyện chấp nhận được: Đây là một phần của phản hồi nhanh, nhưng được tách riêng vì độ quan trọng của nó. Tốc độ xem truyện gồm hai tiêu chí:
 - Tốc độ mở truyện, tức tốc độ xem trang đầu (có thể so với first contentful paint trong lập trình web)
 - Tốc độ cuộn trang tới-lui
- Chiếm ít bộ nhớ: Bộ nhớ chiếm dụng của ứng dụng gồm hai phần: bộ nhớ RAM và bộ nhớ tạm, cả hai cần sử dụng ít dung lượng nhất có thể. Đây là một yêu cầu đáng cân nhắc, lí do vì kích cỡ từng tệp truyện thường rất lớn (từ vài chục đến hơn một trăm megabyte), tuy nhiên cần chú ý cân bằng yêu cầu này với yêu cầu về tốc độ (đánh đổi không gian-thời gian).
- Giao diện đơn giản, trực quan: Người dùng hướng đến có thể xếp vào nhóm người dùng “say mê” (enthusiast), do đó giao diện chỉ cần đơn giản rõ ràng, không màu mè, tập trung vào tính năng.

3.3 Phân tích yêu cầu

Mỗi yêu cầu đã xác định trong Mục 3.2.1 được coi là một ca sử dụng, được trình bày trong các tiểu mục dưới đây.

Người dùng duy nhất trong các ca sử dụng là *người đọc*, do đó hai cụm từ này sẽ được dùng hoán đổi cho nhau. Do ứng dụng hoàn toàn ngoại tuyến, người đọc cũng không có tương tác với nhau.

Do ứng dụng đơn giản, các ca sử dụng tách biệt, nên mỗi ca sử dụng gắn với một *màn hình*. Có tổng cộng năm màn hình sẽ được mô tả, gồm:

- Màn hình Thư viện
- Màn hình Thư mục
- Màn hình Đọc truyện

- Màn hình Metadata
- Màn hình Tìm kiếm

3.3.1 Quét các tệp truyện trên thiết bị

- **Mô tả:**

Người đọc *chọn* một thư mục trong điện thoại làm thư mục gốc. Ứng dụng sẽ *quét* thư mục này và tìm các tệp truyện, rồi hiển thị những thư mục chứa tệp truyện cho người đọc chọn.

- **Luồng chính:**

1. Người đọc bật ứng dụng (tức ở Màn hình Thư viện).
2. Người đọc ấn vào nút thay đổi thư mục gốc.
3. Trình chọn thư mục của Android hiện ra, cho phép người đọc chọn thư mục làm thư mục gốc.
4. Màn hình Thư viện trở lại, quét và hiển thị các thư mục chứa truyện trong thư mục gốc, đồng thời lưu kết quả quét vào cơ sở dữ liệu.

Nếu người đọc đã chọn một thư mục gốc, ca sử dụng này *thay thế* thư mục gốc đã chọn bằng thư mục vừa chọn.

- **Luồng thay thế:**

Khi người dùng bật ứng dụng, và đã chọn một thư mục gốc từ lần sử dụng trước đó, tức ứng dụng cần *quét lại*, khác với Luồng chính là *quét mới*:

1. Người đọc bật ứng dụng (tức ở Màn hình Thư viện).
2. Ứng dụng quét lại truyện trong thư mục gốc, và cập nhật các thay đổi vào cơ sở dữ liệu.

- **Luồng ngoại lệ:**

Nếu người đọc không chọn thư mục nào, quay lại Màn hình Thư viện và không thay đổi gì.

Nếu có lỗi trong quá trình chọn thư mục, cần gợi ý người đọc chọn lại. Lỗi gồm:

- Thiếu quyền đọc
- Không tìm được thư mục gốc

- Thư mục gốc không có truyện

Nếu có lỗi trong quá trình quét cần phải giảm thiểu và giấu khỏi người đọc.

- **Kích hoạt khi:**

Người dùng ấn vào nút thay đổi thư mục gốc.

- **Điều kiện:**

- Tiền điều kiện:

- * Ứng dụng ở Màn hình Thư viện
- * Nếu là quét lại, cần có thư mục gốc

- Hậu điều kiện: Ứng dụng ở Màn hình Thư viện

- * Lưu truyện quét được vào cơ sở dữ liệu. Nếu là quét mới, cần xóa hẳn cơ sở dữ liệu cũ. Quét lại thì phải giữ cơ sở dữ liệu sẵn có.
- * Hiển thị thư mục truyện quét được
- * Hiển thị lỗi nếu có (ba loại lỗi ở trên), và gợi ý xử lý

- **Yêu cầu phi chức năng:**

Nếu đang quét, Màn hình Thư viện cần hiển thị danh sách thư mục theo tiến độ, ứng dụng quét đến đâu hiển thị đến đấy.

Đây là ca sử dụng đầu tiên khi người đọc chạy ứng dụng lần đầu. Các tệp truyện sẽ được quét từ thư mục gốc, rồi được gom lại theo thư mục như mô tả ở ca sử dụng kế tiếp.

Ca sử dụng này có luồng thay thế là trường hợp quét tự động chạy mỗi khi chạy ứng dụng mà không cần người dùng kích hoạt.

Khi quét, ứng dụng phải đọc luôn cả metadata của tệp truyện nếu có. Các trường trong metadata được giải thích chi tiết trong Phụ lục A. Hiện nay, yacv chấp nhận định dạng metadata ComicInfo, là một tệp XML trong tệp truyện. Phụ lục B trình bày lược đồ XSD của định dạng metadata này.

Một số metadata có thể được trích xuất ngay từ tên tệp truyện. Do không có quy chuẩn trong việc đặt tên tệp, cách trích xuất này không ổn định, tuy nhiên cũng không phải ý tưởng tồi. Nếu có thể, dựa vào Phụ lục A để thử trích xuất từ tên tệp truyện.

3.3.2 Duyệt truyện

- **Mô tả:**

Ứng dụng hiển thị những thư mục chứa tệp truyện, khi ấn vào sẽ hiển thị tệp truyện trong thư mục đó.

- **Luồng chính:**

1. Người đọc bật ứng dụng (tức ở Màn hình Thư viện).
2. Người đọc duyệt danh sách thư mục truyện, và chọn một thư mục.
3. Người đọc duyệt danh sách truyện trong thư mục.

- **Kích hoạt khi:**

Người dùng bật ứng dụng.

- **Điều kiện:**

Tiền điều kiện: Ứng dụng đã quét được ít nhất một thư mục chứa truyện.

- **Yêu cầu phi chức năng:**

Nếu đang quét, màn hình cần hiển thị danh sách thư mục theo tiến độ, ứng dụng quét đến đâu hiển thị đến đấy.

Mỗi thư mục cần hiển thị:

- Tên thư mục
- Ảnh đại diện cho thư mục: bìa một truyện bất kỳ tìm được trong thư mục

Mỗi tệp truyện cần hiển thị:

- Tên truyện
- Bìa truyện
- Tiến độ đọc
- Đánh giá yêu thích

Đây là một trong hai ca sử dụng chính của ứng dụng, cùng với ca sử dụng Đọc truyện. Ca sử dụng này liên quan đến hai màn hình.

Màn hình đầu tiên khi người đọc bật lên gọi là *Màn hình Thư viện* (Library screen). Màn hình này hiển thị các thư mục chứa truyện, hoặc thông báo lỗi nếu có của quá trình quét truyện (đã mô tả ở Luồng ngoại lệ của mục trước).

Màn hình khi người đọc chọn một thư mục gọi là *Màn hình Thư mục* (Directory screen). Màn hình này hiển thị các tệp truyện trong thư mục đó.

Trong yacv, truyện được quản lý và duyệt theo thư mục. Lý do cho lựa chọn thiết kế này là vì các phương pháp duyệt khác không trực quan:

- Các phương pháp duyệt khác chỉ bao gồm duyệt theo metadata, tức duyệt theo các thông tin đi kèm như Tác giả, Nhân vật, Bộ truyện,... thì yêu cầu truyện phải có đủ metadata. Trên thực tế, không phải tệp truyện nào cũng có đủ thông tin này, do vậy sẽ có trường hợp rất nhiều truyện bị gom vào mục “Không đủ thông tin”. Hơn nữa, giả sử truyện có đi kèm metadata, ta xem xét tiếp trường hợp dưới.
- Giả sử ta quản lí theo Nhân vật: Vậy để trực quan, yacv phải hiển thị ảnh nhân vật. Hiện nay, việc nhận diện và cắt đúng ảnh phần mặt nhân vật ra để tạo ảnh đại diện có thể nói là bất khả thi. Do vậy, khi duyệt theo Nhân vật, người đọc chỉ có thể thấy tên, không thấy một hình ảnh gợi ý nào khác, dẫn đến khó khăn khi sử dụng. Lập luận tương tự có thể dùng với các cách xếp khác.
- Một cách xếp có thể nói là tốt là xếp theo Bộ truyện, tuy nhiên ta lại quay về vấn đề thiếu metadata.

Hơn nữa, các thư mục cần được “làm phẳng”, tức là hiển thị thư mục con (cháu,...) ngang hàng với thư mục gốc. Bảng sau cho thấy cách yacv làm phẳng cây thư mục:

Bảng 3.1: Cách yacv làm phẳng thư mục

Cây thư mục gốc	yacv đã làm phẳng
thư mục gốc	thư mục gốc
Original Sin #1.cbz	Original Sin #1.cbz
House of M	House of M
House of M #1.cbz	House of M #1.cbz
House of M #3.cbz	House of M #3.cbz
Tie-ins	Tie-ins
Black Panther #7.cbz	Black Panther #7.cbz

Theo như bảng trên, các màn hình trong yacv được tổ chức như sau:

- Màn hình Thư viện: có 3 thư mục:
 - thư mục gốc
 - House of M
 - Tie-ins
- Khi chọn “House of M”: chuyển sang Màn hình Thư mục tương ứng, không có thư mục con, và có 2 tệp truyện:

- House of M #1.cbz
- House of M #3.cbz
- Tương tự với các thư mục khác.

Có ba lí do cho lựa chọn thiết kế này:

- Giảm độ phức tạp khi lập trình.
- Người đọc không phải đi qua nhiều tầng thư mục để đến được tệp truyện cần đọc.
- Không có ca sử dụng có ý nghĩa cho thư mục lồng nhau:

Trường hợp hợp lý nhất cho việc có thư mục lồng nhau là khi lưu các tệp truyện liên quan đến một bộ truyện (tie-ins), như cột trái Bảng 3:

- Thư mục cha (House of M) chứa tệp truyện trong bộ truyện cùng tên và thư mục tie-ins.
- Thư mục Tie-ins chứa các tệp truyện tie-in.

Tuy nhiên, bản thân các tệp tie-in lại là tệp truyện thông thường trong một bộ truyện khác, do đó nếu tổ chức thư mục như thế này sẽ dẫn đến tình trạng lặp tệp truyện, là điều không mong muốn ngay cả với máy tính.

3.3.3 Đọc truyện

- **Mô tả:**

Người đọc chọn một truyện để xem.

- **Luồng chính:**

1. Người đọc bật ứng dụng, đã chọn thư mục gốc, đã quét được ít nhất một thư mục chứa truyện, đã chọn một thư mục (tức ở Màn hình Thư mục).
2. Ứng dụng hiển thị danh sách truyện trong thư mục đó cho người đọc xem và chọn.
3. Người đọc chọn một truyện và đọc.
4. Màn hình Đọc truyện hiển thị trang truyện cho người đọc.
5. Người đọc vuốt qua lại theo phương ngang để chuyển trang.

- **Luồng thay thế:**

Xem phần Màn hình Tìm kiếm. Màn hình Đọc truyện có thể được kích hoạt bằng cách ấn vào truyện hiển thị trong màn hình này.

- **Luồng ngoại lệ:**

Nếu tệp truyện không tìm thấy được, báo cho người đọc và giữ nguyên ở Màn hình Thư mục.

- **Kích hoạt khi:**

Người dùng chọn một truyện trong danh sách truyện.

- **Điều kiện:**

- Tiền điều kiện: Ứng dụng ở Màn hình Thư mục.
- Hậu điều kiện: Ứng dụng ở Màn hình Đọc truyện.

- **Yêu cầu phi chức năng:**

- Nếu người đọc đã đọc truyện, ứng dụng cần đưa về chính trang truyện đang đọc dở. Nếu đã đọc đến trang cuối, tức đã đọc xong, ứng dụng cần đưa về trang đầu tiên.
- Trải nghiệm cuộn trang mượt mà nhất có thể.

Đây là một trong hai ca sử dụng chính của ứng dụng, bên cạnh (và là mục đích của) ca sử dụng duyệt truyện đã được miêu tả ở trên.

Màn hình khi người đọc đọc một truyện gọi là *Màn hình Đọc truyện*. Màn hình này cho phép người đọc duyệt các trang truyện theo phương ngang. Mục tiêu là thiết kế màn hình này sao cho có trải nghiệm gần giống nhất với ứng dụng Thư viện ảnh (Gallery) tích hợp trong mọi điện thoại Android.

3.3.4 Xem metadata truyện

- **Mô tả:**

Trong Màn hình Đọc truyện, người đọc ấn nút để xem metadata.

- **Luồng chính:**

1. Người đọc bật ứng dụng, chọn một truyện để vào đến Màn hình Đọc truyện.
2. Người đọc ấn nút Xem metadata.

3. Ứng dụng hiển thị mọi metadata, bao gồm cả những trường bị thiếu. Ảnh bìa của truyện cũng được hiển thị kèm.
4. Người dùng có thể đánh giá truyện bằng nút Yêu thích trong màn hình này, hoặc ngược lại (bỏ đánh giá Yêu thích).

- **Kích hoạt khi:**

Người dùng ấn vào nút Xem metadata trong Màn hình Đọc truyện.

- **Điều kiện:**

- Tiền điều kiện: Ứng dụng ở Màn hình Đọc truyện.
- Hậu điều kiện: Ứng dụng ở Màn hình Metadata.

- **Yêu cầu phi chức năng:**

- Màn hình Metadata phải hiển thị ảnh bìa, cùng các metadata của truyện
- Những trường metadata trống phải ghi rõ “Trống”, “Unknown”,...

Đây là một ca sử dụng phụ, có thể được kích hoạt khi người dùng đang ở Màn hình Đọc truyện.

Màn hình khi người đọc xem metadata gọi là *Màn hình Metadata*. Màn hình này có thể có chức năng sửa metadata, tùy theo tiến độ khóa luận để xem xét có cài đặt không.

Hệ thống đánh giá của ứng dụng chỉ ở mức cơ bản, gồm duy nhất tính năng Yêu thích. Tính năng này cũng chỉ phục vụ hai mục đích là thể hiện sự đánh giá của người dùng và lọc nhanh truyện về mặt thị giác (đã nhắc đến trong phần Mô tả từng bước của ca sử dụng hiển thị danh sách truyện).

Các tính năng nâng cao hơn như gợi ý không xuất hiện, do một số lí do sau:

- Giảm độ phức tạp khi lập trình.
- Người dùng không có nhu cầu: nhóm người dùng hướng đến có đặc điểm hiểu biết về truyện tranh, do đó việc gợi ý có thể coi là thừa thãi.
- Thiếu thông tin gợi ý: việc gợi ý chỉ có hiệu quả khi có một cơ sở dữ liệu về các bộ truyện liên quan, hoặc lựa chọn các truyện liên quan của cộng đồng người đọc, trong khi yacy là một ứng dụng hoàn toàn ngoại tuyến.

3.3.5 Tìm kiếm truyện

- **Mô tả:**

Trong Màn hình Thư viện, người đọc ấn nút Tìm kiếm để tìm truyện.

- **Luồng chính:**

1. Người đọc bật ứng dụng.
2. Người đọc ấn nút Tìm kiếm, gõ từ khóa cần tìm, và ấn nút Enter.
3. Ứng dụng hiển thị kết quả tìm kiếm theo metadata và tên tệp truyện.

- **Luồng ngoại lệ:**

Nếu không tìm thấy truyện, ứng dụng cần thông báo ở Màn hình Tìm kiếm.

- **Kích hoạt khi:**

Người dùng ấn nút Tìm kiếm trong Màn hình Thư viện.

- **Điều kiện:**

- Tiền điều kiện: Ứng dụng ở Màn hình Thư viện.
- Hậu điều kiện: Ứng dụng ở Màn hình Tìm kiếm.

- **Yêu cầu phi chức năng:**

- Kết quả tìm kiếm cần được gom theo nhóm dựa vào trường metadata tìm thấy được. Nếu không có kết quả, phải báo cho người dùng.
- Nếu có thể, hiển thị ảnh bìa của truyện.

Đây là ca sử dụng phụ, có thể được kích hoạt khi người dùng ở Màn hình Thư viện.

Màn hình khi người đọc xem kết quả tìm kiếm gọi là *Màn hình Tìm kiếm*. Màn hình này chỉ hiện ra khi người dùng ấn nút Enter để chính thức tìm kiếm; cho đến trước lúc đó, ứng dụng vẫn ở Màn hình Thư viện.

Màn hình Tìm kiếm nhóm kết quả theo trường metadata mà kết quả được tìm thấy. Ví dụ, người dùng tìm “Watchmen” sẽ thấy Màn hình Tìm kiếm gần như Hình 3.1.

Tương tác của người đọc với màn hình trên là như sau:

- Khi ấn vào một mục trong danh sách “Truyện”, người đọc được đưa đến trang Đọc truyện của truyện đó (và hiển thị ở trang đọc dở như đã mô tả trong ca sử dụng đọc truyện).

- Khi ấn vào một mục trong danh sách “Bộ truyện”, người đọc được đưa đến màn hình chứa danh sách truyện trong bộ truyện đã chọn. *Màn hình này cần giống với Màn hình Thư mục*. Sau đó, người dùng chọn một truyện để đọc như bình thường.
- Khi ấn vào nút “Xem thêm”, người đọc được đưa đến màn hình chứa danh sách kết quả đầy đủ của kiểu kết quả đó.

Ví dụ, nếu ấn vào “Xem thêm” trong màn hình trên, người dùng sẽ thấy một màn hình khác hiển thị đủ các bộ truyện tìm thấy được (*Watchmen* và *Watchmen Ultimate*). Tiếp tục ấn vào một kết quả sẽ ra màn hình danh sách truyện như trên.

Đây chỉ là ví dụ về một từ khóa có kết quả khi **Truyện** tìm theo tên tệp truyện và bộ truyện. Các trường metadata khác nếu có kết quả phù hợp cũng sẽ thể hiện theo hình thức trên.

Chú ý rằng bìa truyện luôn được thể hiện khi có thẻ. Trong ví dụ trên, chắc chắn phải có bìa truyện cho mọi mục con trong danh sách “Truyện”.

Còn mục con của “Bộ truyện” thì không, lí do là không có đủ dữ liệu (không có dữ liệu cho logo, banner,...của bộ truyện). Tương ứng, không có dữ liệu hiển thị cho danh sách “Nhân vật”, “Tác giả”,... Đây là một hạn chế quan trọng về giao diện mà hiện chưa có cách thiết kế hợp lý.

Nếu không có kết quả, cần thể hiện rõ cho người dùng biết.

Khi người dùng gõ từ khóa để tìm kiếm, một số thông tin gợi ý tìm kiếm (từ khóa cũ, từ khóa liên quan) có thể hiện ra; tính năng này tùy theo tiến độ khóa luận để xem xét có cài đặt không. Chỉ khi người dùng ấn Enter, quá trình tìm kiếm mới bắt đầu, và hiển thị kết quả, chứ không hiển thị kết quả theo quá trình người dùng gõ phím. Lí do cho lựa chọn này như sau:

- Giảm độ phức tạp khi lập trình.
- Không quá cần thiết: ví dụ, trang web tìm kiếm của Google cũng chỉ hiện gợi ý khi người dùng nhập từ khóa tìm kiếm, phải đến khi ấn Enter thì quá trình tìm kiếm mới diễn ra và kết quả chi tiết được hiển thị.

Với độ phức tạp dự kiến của việc hiển thị ảnh bìa truyện, đây có thể được xem là đánh đổi hợp lý để tăng hiệu năng ứng dụng.

- *Watchmen #1.cbz*
- *Watchmen #2.cbz*
- *Watchmen Noir #1.cbz*

Bộ truyện
- *Watchmen*
- *Xem thêm...*

(a) Khi mới tìm
- *Watchmen*
- *Watchmen Noir*

(b) Khi ấn “Xem thêm ...”

Hình 3.1: Mô tả Màn hình Tìm kiếm

3.3.6 Xóa truyện

- **Mô tả:**

Người dùng chọn một số truyện trong một màn hình chứa danh sách truyện để xóa.

- **Luồng chính:**

1. Người dùng truy cập vào một màn hình chứa danh sách truyện (là Màn hình Thư mục hoặc Màn hình Tìm kiếm).
2. Người dùng ấn và giữ vào một truyện.
3. Màn hình đó sẽ chuyển sang chế độ xóa, báo hiệu bằng biểu tượng Thùng rác trên màn hình, và ô đánh dấu để xóa ở cạnh mỗi truyện. Truyện mà người dùng ấn giữ phải được đánh dấu xóa ngay.
4. Người dùng có thể chọn thêm truyện để xóa nếu muốn.
5. Người dùng ấn nút xóa để xóa truyện.
6. Ứng dụng hiện ra hộp thoại xóa, ghi rõ rằng truyện sẽ được xóa khỏi bộ nhớ điện thoại, số truyện sẽ xóa, và hỏi người dùng có thực sự muốn xóa không.
7. Nếu người dùng ấn vào nút Đồng ý xóa, truyện sẽ được xóa khỏi bộ nhớ điện thoại và tắt hộp thoại, nếu không thì tắt hộp thoại.
8. Sau khi tắt hộp thoại, màn hình trở về chế độ ban đầu, biểu tượng thùng rác cũng biến mất, truyện được xóa cũng biến mất.

- **Kích hoạt khi:**

Người dùng ấn giữ một tệp truyện trong màn hình chứa danh sách truyện.

- **Điều kiện:**

Tiền và hậu điều kiện đều là màn hình chứa danh sách truyện, gồm:

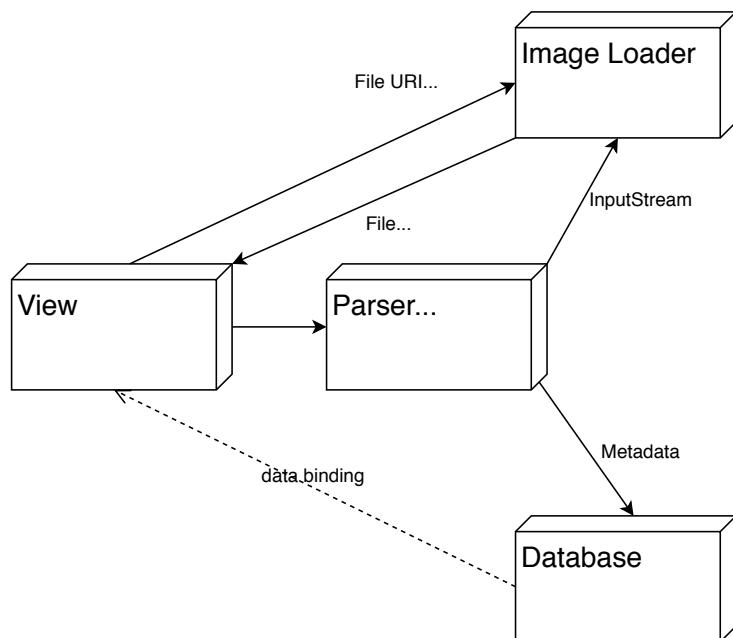
- Tiền điều kiện: Màn hình chứa danh sách truyện
- Hậu điều kiện:
 - * Màn hình chứa danh sách truyện
 - * Nếu người dùng đồng ý xóa, truyện được xóa khỏi bộ nhớ và cơ sở dữ liệu

Ca sử dụng này không có màn hình riêng biệt, mà sử dụng một chế độ của các màn hình hiển thị danh sách truyện.

Chương 4

Thiết kế

Chương này tập trung vào thiết kế của ứng dụng, là triển khai cụ thể của Chương 3.



Hình 4.1: Kiến trúc tổng quan của ứng dụng

Kiến trúc tổng quan của yacv rất đơn giản, gồm 4 module như hình:

1. yacv *quét* metadata tệp truyền bằng module **Parser & Scanner** và lưu kết quả quét vào *cơ sở dữ liệu*, tức module **Database**
2. Các *Màn hình* trong module **View** hiển thị dữ liệu cho người dùng
3. Khi người dùng đọc truyền, yacv trích xuất và lưu đệm tệp ảnh bằng module **Image Loader**

4. Khi người dùng xem metadata, yacv trích xuất và hiển thị thông tin tệp truyện lên Màn hình bằng data binding với module **Database**

Ở Chương 2 - Kiến trúc nền tảng, trong phần về MVVM, yacv được giới thiệu là có sử dụng kiến trúc này. Tuy nhiên, MVVM chỉ là một phần nhỏ của ứng dụng, chủ yếu liên quan đến việc hiển thị dữ liệu nên chỉ có ý nghĩa khi xét đến các thành phần trong module **View**.

Ở Chương 3, yêu cầu về tránh trói buộc người dùng được nêu lên đầu tiên trong các yêu cầu phi chức năng. Ở chương này, yêu cầu đó được hiện thực hóa bằng việc *yacv dùng phân vùng bộ nhớ chung*.

- Nếu dùng phân vùng bộ nhớ riêng, yacv phải chép các tệp truyện vào đó (tốn thời gian và dung lượng), hơn nữa các ứng dụng đọc truyện khác không thể thấy được tệp truyện ở khu vực này. Tuy nhiên, dữ liệu lưu ở phân vùng này có thể truy cập bằng API File của Java, giúp đơn giản đáng kể thiết kế ứng dụng.
- Dùng phân vùng bộ nhớ chung tránh được mọi điều trên, không buộc người dùng vào ứng dụng, tuy nhiên lại bị giới hạn chỉ được dùng API SAF.

yacv chỉ thiết kế cho *một người dùng*, do đó có rất ít tương tác, dẫn đến kiến trúc tối giản và rời rạc như trên. Các tiểu mục sau sẽ đi sâu vào các module này.

4.1 Module Database

Thông thường mục này được tách riêng ra, xếp vào mục *Thiết kế cơ sở dữ liệu*, ngang hàng với mục Thiết kế hướng đối tượng. Tuy nhiên, yacv còn cần xử lý dữ liệu khác quan trọng không kém là dữ liệu ảnh. Do không còn có vai trò trung tâm, duy nhất, phần cơ sở dữ liệu chỉ được coi là một module trong thiết kế hướng đối tượng của ứng dụng.

yacv chọn SQLite vì đây là một cơ sở dữ liệu gọn nhẹ nhung sẵn trong Android. SQLite sử dụng mô hình quan hệ, do đó thiết kế bảng cần đảm bảo được chuẩn hóa (normalization).

Do không cần quản lý người dùng, cơ sở dữ liệu của yacv chỉ dùng để *lưu thông tin metadata*, cho phép ứng dụng quét dữ liệu ít lần hơn và tìm kiếm truyện. Theo yêu cầu về metadata ở hai Phụ lục, và sau khi chuẩn hóa, ta có lược đồ cơ sở dữ liệu như sau:

Các bảng thực thể gồm:

- **Comic**: lưu thông tin *tập truyện* là bảng trung tâm
- **Series**: lưu thông tin *bộ truyện*

- Author: lưu tên tác giả
- Role: lưu vai trò của tác giả trong một tập truyện
- Character: lưu tên nhân vật
- Genre: lưu tên thể loại truyện

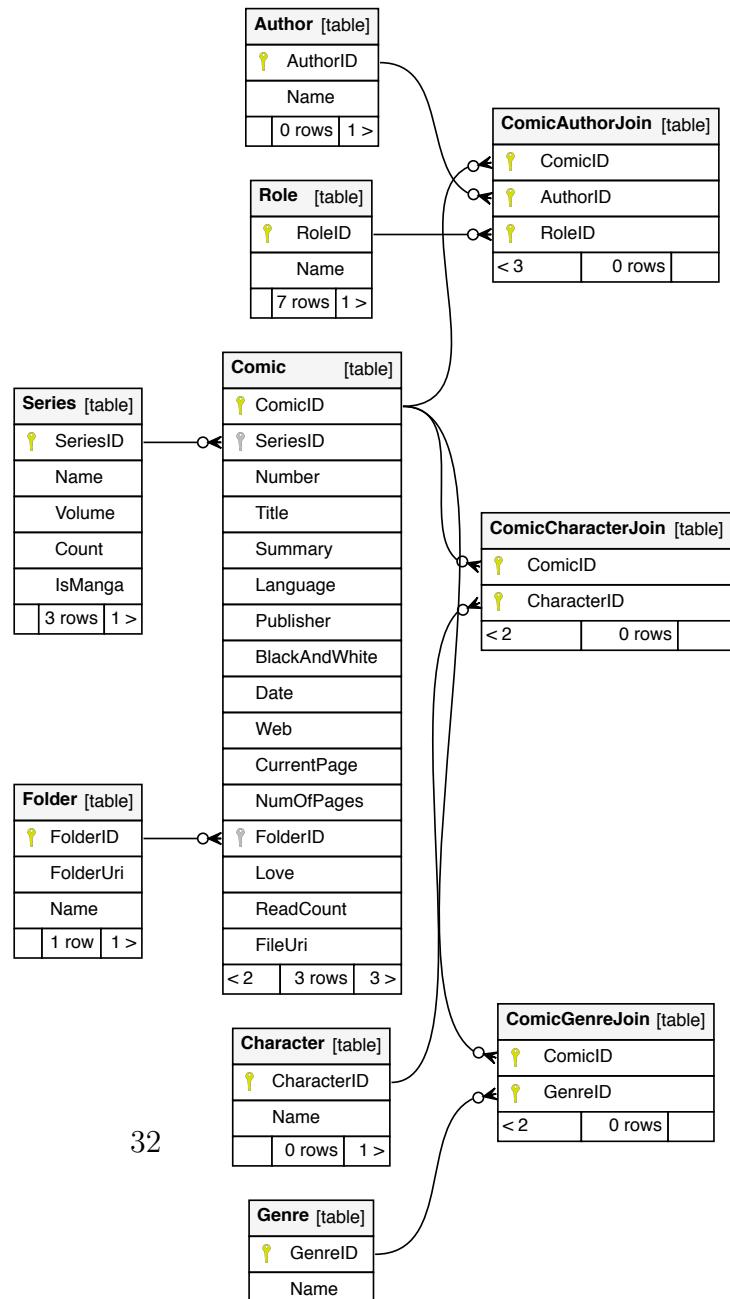
Một hạn chế quan trọng của các bảng **Character** và **Author** là chúng chỉ lưu thông tin tên, và chỉ phân biệt với nhau bằng tên. Nếu có hai tác giả/nhân vật trùng tên, yacy không thể phát hiện và hiển thị riêng.

Xét bảng trung tâm **Comic**. Bảng này có một số trường không phải metadata mà dùng để lưu thông tin của riêng ứng dụng, gồm:

- CurrentPage: lưu trang đang đọc
- Love: lưu trạng thái Yêu thích
- ReadCount: lưu số lần đọc

Trong lược đồ, có nhiều trường nhìn qua không cần thiết nhưng thực tế có ích, do thư viện SAF đã mô tả ở Chương 2:

- Trường **FileUri** trong **Comic**: Lưu đường dẫn của tệp truyện ở dạng URI.
- Trường **FolderPath** trong **Folder**: Lưu đường dẫn của thư mục ở dạng URI.
- Trường **Name** trong **Folder**: Tên thư mục. Thông thường nếu có đường dẫn, có thể tìm ra tên thư mục rất nhanh, tuy nhiên cũng do SAF mà việc này trở nên khó khăn, nên cần lưu riêng trường này.



Các trường URI đều cần có ràng buộc UNIQUE, do mỗi URI trả đích danh đến một đối tượng.

Ta xem xét đến các bảng nối:

- **ComicCharacterJoin:**

- Mỗi tập truyện có thể có nhiều nhân vật và ngược lại, do đó **Comic** và **Character** có quan hệ Nhiều - Nhiều.
- Chú ý rằng các nhân vật có quan hệ với tập truyện chứ không phải bộ truyện, vì có nhân vật phụ (không xuất hiện trong mọi tập truyện).

- **ComicAuthorJoin:**

- Mỗi tập truyện có thể có nhiều tác giả và ngược lại, do đó **Comic** và **Author** có quan hệ Nhiều - Nhiều.
- Chú ý rằng các tác giả có quan hệ với tập truyện chứ không phải bộ truyện, vì mô hình xuất bản nhiều truyện tranh là nhà xuất bản sở hữu nhân vật và thuê người viết.
- Đồng thời, một tác giả có thể giữ vai trò khác nhau trong các bộ truyện khác nhau, do đó bảng này còn nối với bảng **Role**.

- **ComicGenreJoin:** Mỗi tập truyện có thể có nhiều thể loại khác nhau và ngược lại, do đó **Comic** và **Genre** có quan hệ Nhiều - Nhiều.

Do dùng Room, mỗi bảng ứng với một lớp. Các truy vấn với bảng cần đóng gói dữ liệu vào các lớp này, trước khi gửi đến hoặc nhận về từ *DAO* (Data Access Object). Mỗi câu lệnh lại được chuyển thành một hàm trong DAO.

4.2 Module View

Phần này tập trung vào các Màn hình, và phân tích chúng theo hướng MVVM.

Trong phần này có dùng nhiều biểu đồ tuần tự (sequence diagram) để minh họa tương tác của ba thành phần MVVM (cùng với một số thành phần liên quan) trong các ca sử dụng. Có một số điểm chung về các biểu đồ này:

- Trừ khi cần thiết, thành phần View sẽ được lược bỏ cho ngắn gọn.

- Đường thẳng nét đứt thể hiện tính năng data binding (tự động cập nhật View), và thường trả về ViewModel. Đáng ra, mũi tên này phải trả về View, nhưng do View bị ẩn đi, nên nó trả về ViewModel. Mặc dù không được đề cập đến trong phần giới thiệu về MVVM, đây thực ra là một chi tiết đúng về mặt kỹ thuật: ViewModel hoàn toàn đọc được luồng dữ liệu gửi đến View (hoặc ít nhất là đúng trong cách viết ứng dụng Android thông thường).

Các biểu đồ trạng thái cũng có một số chi tiết chung:

- Trừ khi nêu rõ, mọi trạng thái đều có thể là trạng thái bắt đầu (trạng thái khi mở ứng dụng) hoặc kết thúc (khi đóng ứng dụng).
- Mũi tên chuyển trạng thái tương ứng với *tương tác của người dùng*, do vậy thường được ánh xạ đến một phương thức trong View.
- Kí hiệu hình tròn đen chỉ dùng để tả trạng thái đầu *khi lần đầu dùng* yacy.

Biểu đồ lớp thường có một phương thức chung là “Get InputStream from ID”. Cách truy cập để lấy `InputStream` của ảnh từ `ComicID` có thể tham khảo từ Màn hình Đọc truyện.

4.2.1 Nguồn dữ liệu - Repository - DAO - ComicParser

Như đã đề cập ở Chương 2, yacy sử dụng Kiến trúc Google khuyên dùng, vốn dựa trên MVVM. Phần này nêu rõ hơn cách triển khai MVVM của yacy trong phần nguồn dữ liệu (Model/Repository).

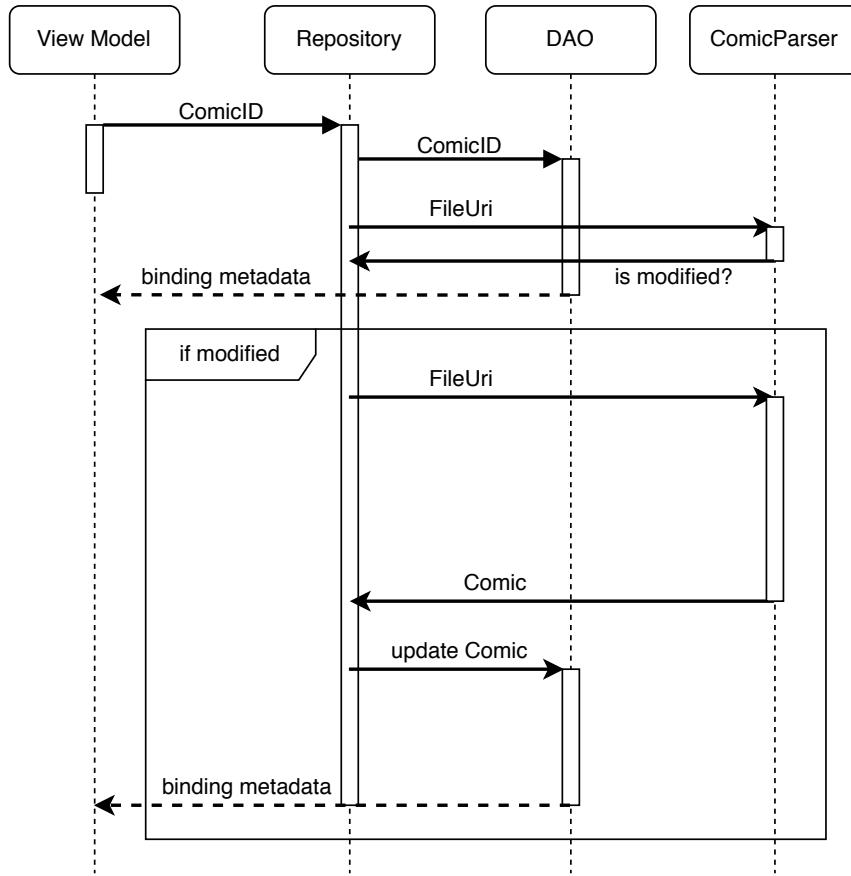
Dựa vào Hình 2.9, ta thiết kế được ba nguồn dữ liệu (model) sau:

Bảng 4.1: Ba nguồn dữ liệu tương đương với Hình 2.9

Tên	Tương đương với	Mục đích
<code>ComicParser</code>	Remote Data Source	Quét tệp để lấy metadata cập nhật nhất
<code>DAO</code>	Model	Lấy metadata từ cơ sở dữ liệu, tránh quét đi quét lại
<code>Repository</code>	Repository	Tổng hợp hai nguồn trên

Cụ thể hơn, `ComicParser` là bộ quét metadata tệp truyện (thuộc module Parser & Scanner, sẽ được mô tả sau). Lớp này nhận vào URI rồi trả về metadata của tệp truyện tương ứng dưới dạng đối tượng `Comic`.

Khi cần đọc dữ liệu metadata từ tệp truyện, ba thành phần này tương tác như sau:



Hình 4.3: Tương tác của ba nguồn dữ liệu, mũi tên gạch đứt thể hiện tính năng data binding

Mấu chốt ở đây là **ComicParser** dù có dữ liệu chính xác (trong trường hợp một ứng dụng khác sửa metadata tệp truyện) nhưng tốc độ rất chậm, còn cơ sở dữ liệu không chính xác nhưng rất nhanh, do đó *cơ sở dữ liệu làm bộ đệm cho parser*.

Repository làm nhiệm vụ gọi cả hai nguồn dữ liệu trên và cập nhật cơ sở dữ liệu (nếu cần) thay cho View/ViewModel. Hiện tại, Repository có thể không làm được nhiều, tuy nhiên nó giúp ích cho *khả năng mở rộng* của ứng dụng. Ví dụ, trong tương lai yacy có thể liên kết với một bên thứ ba cung cấp metadata cho truyện, khi đó để tích hợp API thì chỉ cần sửa phần Repository.

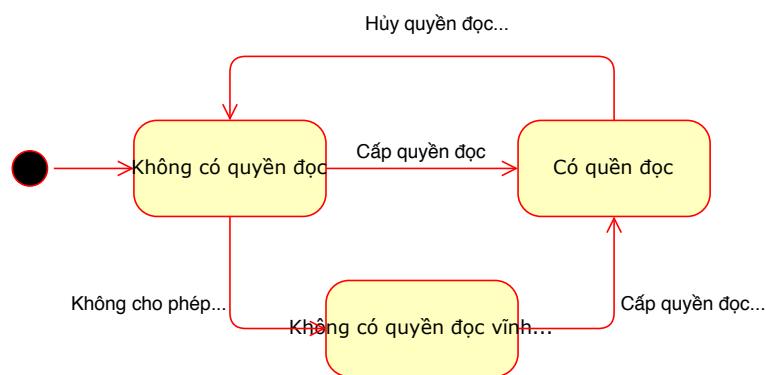
Cũng cần chú ý rằng việc đọc metadata từ tệp tin không phải là yêu cầu của mọi màn hình (cụ thể chỉ Màn hình Metadata cần), do đó trong đa số các ca sử dụng, *DAO đóng vai trò Model*, thay cho Repository. Tương tác trong Hình 4.3 vẫn được duy trì, tuy không có cả Repository lẫn ComicParser.

4.2.2 Màn hình Quyền đọc

Do sự phức tạp trong việc xin quyền của Android, một màn hình riêng để xin quyền đọc dữ liệu được tách ra khỏi Màn hình Thư viện, gọi là *Màn hình Quyền đọc*. Màn hình này sẽ là *màn hình đầu tiên hiển thị* khi dùng ứng dụng.

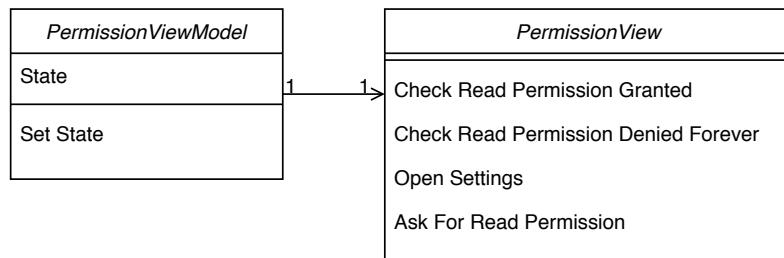
- Nếu có quyền đọc: chuyển ngay sang Màn hình Thư viện
- Nếu không: nêu lí do cần quyền, gợi ý người dùng cấp quyền

Hình sau mô tả trạng thái cấp quyền đọc của yacy (cũng như mọi quyền của một ứng dụng Android cơ bản nói chung).



Hình 4.4: Trạng thái cấp quyền của một ứng dụng Android

Dựa theo Hình 4.4, ta có biểu đồ lớp của ViewModel và View như sau:



Hình 4.5: Biểu đồ lớp của Màn hình Quyền đọc

4.2.3 Màn hình Thư viện

Màn hình Thư viện là một trong hai màn hình của ca sử dụng Duyệt truyện, bên cạnh Màn hình Thư mục.

Như đã phân tích ở Mục 3.3.1, Màn hình Thư viện cần hiển thị cả lối và gợi ý, bên cạnh việc hiển thị danh sách thư mục và chọn thư mục gốc. Do đó, phần này chia ra làm hai phần con tương ứng.

Chọn thư mục gốc và hiển thị danh sách thư mục

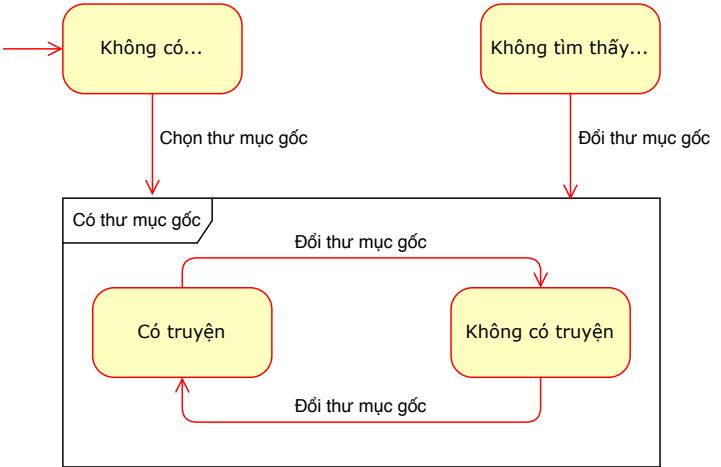
Để chọn thư mục gốc, người dùng ấn nút **Đổi thư mục gốc** để kích hoạt hộp thoại Chọn thư mục (picker), rồi chọn một thư mục trong đó. Luồng chạy của yacy sẽ được nêu sau, ở mục riêng về Scanner.

Ngoại lệ trong Màn hình Thư viện

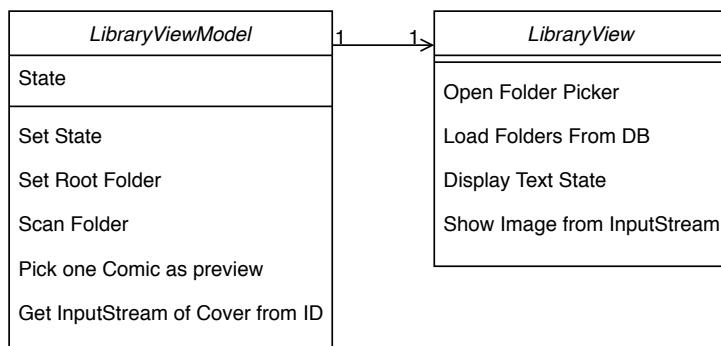
Ngoại lệ ở đây chỉ cả trường hợp không tìm thấy thư mục, lẫn trường hợp không quét được thư mục vì các lí do đã nêu trong Mục 3.3.1. Khi này, ứng dụng hiển thị một hàng chữ để gợi ý về việc nên làm.

Hình sau là biểu đồ trạng thái, cũng là mô tả về nội dung gợi ý. Riêng trạng thái “Có truyện” là trạng thái hiển thị danh sách thư mục trong luồng cơ bản đã nêu trên, tức thư mục được hiển thị đầy đủ thay vì chỉ hiện thông báo lỗi.

Tổng hợp lại, ta có biểu đồ lớp của Màn hình Thư viện như sau:



Hình 4.6: Trạng thái của Màn hình Thư viện



Hình 4.7: Biểu đồ lớp của Màn hình Thư viện

Nhắc lại, cách truy cập để lấy `InputStream` của ảnh từ `ComicID` có thể tham khảo từ Màn hình Đọc truyện.

4.2.4 Màn hình Thư mục

Màn hình Thư mục là một trong hai màn hình của ca sử dụng Duyệt truyện, bên cạnh Màn hình Thư viện.

Trong khi phân tích yêu cầu, ta đã phân tích được rằng màn hình hiển thị danh sách truyện - một phần trong ca sử dụng tìm kiếm - phải có giao diện giống Màn hình Thư mục, vì đều hiển thị danh sách truyện. Do đó, hai màn hình này được gộp lại, gọi chung là *Màn hình Danh sách truyện*, và sẽ được mô tả sau.

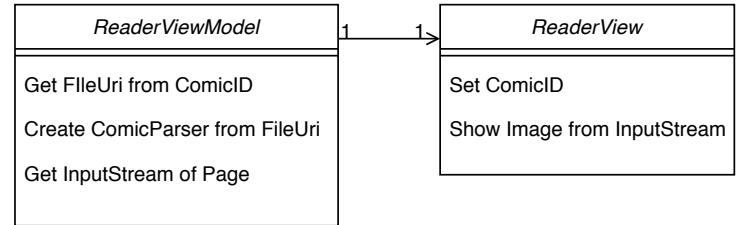
4.2.5 Màn hình Đọc truyện

Để hiển thị các trang truyện,

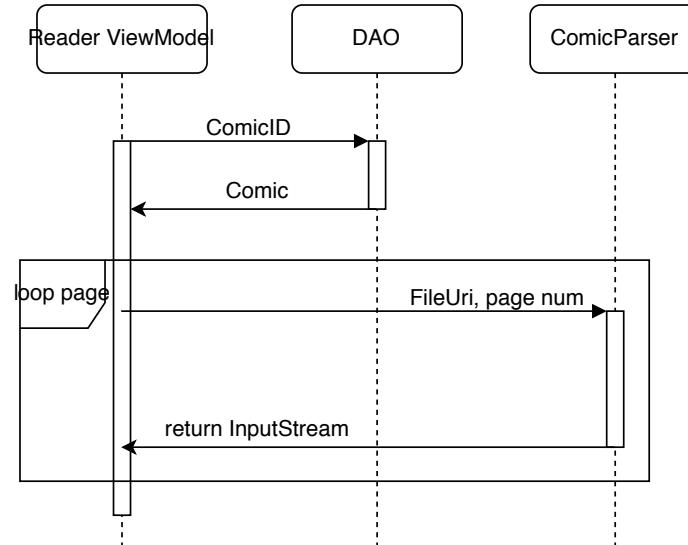
Màn hình Đọc truyện cần nhận ComicID (hoặc một đối tượng Comic hoàn chỉnh, tuy nhiên cốt yếu vẫn là thông tin ComicID) của một tệp truyện, sau đó đưa cho ComicParser để lấy luồng đọc cho từng trang truyện.

Hình 4.9 là biểu đồ tuần tự của màn hình này.

Ta cũng có biểu đồ lớp tương ứng trong Hình 4.8.



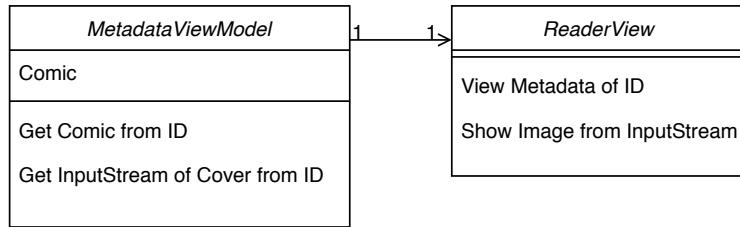
Hình 4.8: Biểu đồ lớp của Màn hình Đọc truyện



Hình 4.9: Biểu đồ tuần tự của Màn hình Đọc truyện

4.2.6 Màn hình Metadata

Màn hình Metadata tuân theo biểu đồ tuần tự đã nêu ở Hình 4.3. Biểu đồ lớp tương ứng của màn hình này như sau:



Hình 4.10: Biểu đồ lớp của Màn hình Metadata

4.2.7 Màn hình Tìm kiếm

Màn hình Tìm kiếm là hai trong ba màn hình của ca sử dụng tìm kiếm truyện, bên cạnh Màn hình Danh sách truyện (là tổng quát hóa của Màn hình Thư mục, đã nhắc ở trên). Màn hình Danh sách truyện sẽ được thiết kế ở ngay mục sau, còn mục này tập trung vào Màn hình Tìm kiếm.

Không khó để thấy thực ra Màn hình Tìm kiếm Tổng quan và Màn hình Tìm kiếm Chi tiết thực ra là một màn hình, về mặt thị giác:

- Điểm giống:
 - Cả hai cùng hiển thị danh sách.
 - Các phần tử cùng loại trong hai màn hình có cách hiển thị giống nhau, chuyển đến các màn hình giống nhau.
- Điểm khác: Danh sách trong Màn hình Tìm kiếm Tổng quan có *thêm*:
 - Hiển thị bìa với một số kết quả
 - Có thanh ngăn cách
 - Có nút “Xem thêm”

Do vậy, nếu thiết kế phù hợp, hoàn toàn có thể gộp hai màn hình này. Thiết kế sau giúp thỏa mãn việc này:

- Màn hình nhận vào một tham số chứa *câu truy vấn*. Tham số này thuộc một trong hai kiểu:
 - *QuerySingleType*: chứa câu truy vấn và *một* bảng để tìm kiếm
 - *QueryMultipleTypes*: chứa câu truy vấn và *một danh sách* bảng để tìm kiếm

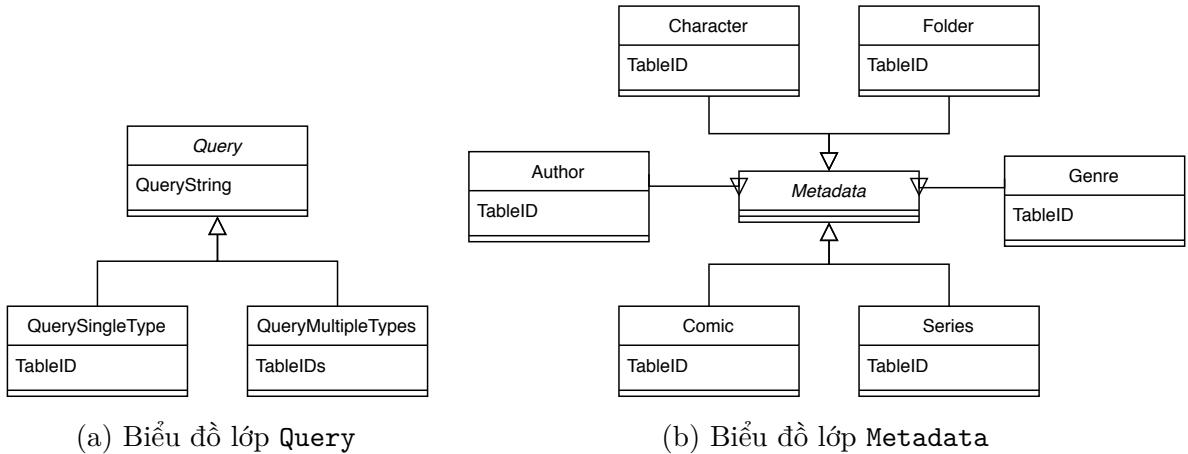
“Bảng để tìm kiếm” thực ra là một số quy định trước, ví dụ nếu là số 0 thì bảng được tìm là *Comic*,...

- ViewModel tìm kiếm dựa vào tham số truy vấn
 - Nếu tham số là `QuerySingleType`: truy vấn và hiển thị kết quả như thông thường
 - Nếu tham số là `QueryMultipleTypes`: truy vấn các bảng, gộp kết quả lại và thêm kiểu kết quả đặc biệt là `Placeholder` và `SeeMore` vào vị trí phù hợp để hiển thị lần lượt nhóm kết quả và nút “Xem thêm”
- Các kết quả, bao gồm hai dạng kết quả đặc biệt ở trên, cài đặt chung giao diện `Metadata`, để có thể được gộp thành một danh sách

Nói ngắn gọn, hai màn hình cùng hiển thị một danh sách, danh sách này có một số phần tử đánh dấu đặc biệt. Ta dùng lại ví dụ về truy vấn `Watchmen` ở Chương 3 để minh họa:

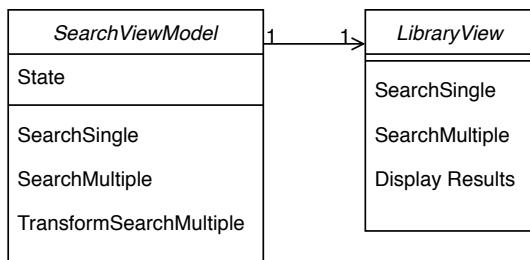
- Truy vấn `QueryMultipleTypes` được gửi đến Màn hình Tìm kiếm. Câu truy vấn là `Watchmen`, các bảng cần tìm là mọi bảng.
- ViewModel tìm `Watchmen` trong mọi bảng, tìm được:
 - 3 tệp truyện trong bảng `Comic`
 - 2 bộ truyện trong bảng `Series`
- Màn hình hiển thị:
 1. Dòng `Truyện`, rồi 3 tệp truyện (cùng với ảnh bìa)
 2. Dòng `Bộ truyện`, 1 bộ truyện, rồi dòng `Xem thêm`
- Khi ấn vào:
 - Một trong ba tệp truyện: Đưa đến Màn hình `Đọc truyện` tương ứng.
 - Một bộ truyện: Chuyển đến Màn hình `Danh sách truyện`, chứa các truyện trong bộ đó.
 - Nút `Xem thêm`: Chuyển đến Màn hình `Tìm kiếm`, lần này tham số là một `QuerySingleType`, với câu truy vấn là `Watchmen`, còn bảng để tìm là `Series`. Hai bộ truyện kết quả được hiển thị đầy đủ. Chọn một bộ truyện lúc này giống với chọn bộ truyện ở trên (sang Màn hình `Danh sách truyện`).

Biểu đồ lớp của các đối tượng liên quan như sau:



Hình 4.11: Biểu đồ các lớp liên quan đến Màn hình Tìm kiếm

Biểu đồ lớp của bản thân Màn hình Tìm kiếm như sau:



Hình 4.12: Biểu đồ lớp của Màn hình Tìm kiếm

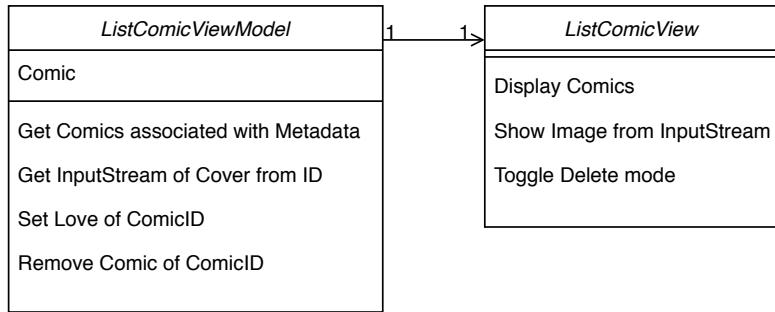
Do mỗi DAO trả kết quả của bảng tương ứng về ở dạng danh sách, nên khi dùng **QueryMultipleTypes**, các danh sách kết quả lẻ này tổng hợp, và thêm hai kiểu kết quả đặc biệt. Hàm **TransformSearchMultiple()** là để “làm phẳng” mảng kết quả hai chiều như trên.

4.2.8 Màn hình Danh sách truyện

Màn hình này là màn hình thứ ba trong chuỗi các màn hình liên quan đến ca sử dụng tìm kiếm, đồng thời đóng vai trò của Màn hình Thư mục (do là phiên bản tổng quát hơn của nó).

Màn hình này nhận vào một tham số kiểu **Metadata** thay vì một **Query**, và trả về danh sách các *tập truyện* - **Comic** - có liên kết với tham số đầu vào.

Biểu đồ lớp của Màn hình Danh sách truyện như sau:



Hình 4.13: Biểu đồ lớp của Màn hình Danh sách truyện

4.3 Module Parser & Scanner

4.3.1 ComicParser

`ComicParser` là một trong các thành phần trung tâm của yacv. Lớp này nhận vào URI trả về một tệp truyện, và đọc nội dung tệp truyện đó ra. Cần gọi đủ tên là `ComicParser`, vì phải phân biệt với hai parser (bộ đọc/giải mã) khác nhưng tích hợp trong nó:

Bảng 4.2: Hai kiểu parser trong `ComicParser`

	Parser cho tệp nén	Parser cho metadata
Đầu vào	Tệp nén	Tệp metadata
Đầu ra	Các tệp con hoặc tương đương (<code>InputStream</code> ,...)	Đối tượng <code>Comic</code>

`ComicParser` được thiết kế theo kiểu “lười”, có nghĩa là không có thông tin nào được đọc ra cho đến khi thực sự cần. Lý do vẫn là vấn đề về hiệu năng, vì gần như mọi thao tác đọc trong SAF - hệ thống đọc ghi tệp của Android - đều rất chậm.

Parser cho tệp nén

Parser cho tệp nén hiện gồm một giao diện và hai lớp:

- `ArchiveParser`: giao diện chung cho mọi parser tệp nén
- `CBZParser`: parser riêng cho tệp CBZ
- `ArchiveParserFactory`: giúp khởi tạo các parser

`ArchiveParser` là một giao diện (interface), định nghĩa một số phương thức chung mọi parser cho tệp nén đều phải có. Do hiện tại yacy mới hỗ trợ định dạng CBZ, chỉ có lớp `CBZParser` cài đặt giao diện này.

Trong `ArchiveParser`, có hai phương thức quan trọng:

- `getEntryOffsets()`: Phương thức này trả về một từ điển như sau:
 - Khóa: tên tệp lẻ
 - Giá trị: offset tệp lẻ, tức vị trí tệp lẻ trong tệp nén
- `readEntryAtOffset()`: Phương thức này nhận vào một offset, và trả về `InputStream` tương ứng với tệp lẻ ở offset đó bằng cách “nhảy cóc” đến đúng chỗ và đọc.

`ArchiveParserFactory` là một lớp theo mẫu thiết kế factory, nhận vào URI của tệp truyện và trả về `ArchiveParser` để đọc loại tệp truyện đó (ví dụ, nếu URI có đuôi CBZ thì trả về một đối tượng `CBZParser`). Do `ArchiveParser` cần một số cài đặt khởi tạo riêng, nên mới cần một lớp riêng để tạo parser. Chữ “Factory” thể hiện lớp này sử dụng mẫu thiết kế factory.

Parser cho metadata

yacy hiện hỗ trợ định dạng ComicRack, được giới thiệu chi tiết trong Phụ lục 2. Định dạng này là một tệp tin XML, do đó được đọc đơn giản bằng các thư viện XML sẵn có.

Để mở rộng định dạng tệp đọc, có thể dùng mẫu thiết kế factory như đã dùng với parser cho tệp. Theo cách này, các parser cần có hàm `parse()` trả về một đối tượng `Comic` và nhận hai tham số:

- Nội dung tệp metadata: ở dạng chuỗi thông thường
- Tên tệp metadata: tên tệp giúp phân biệt các định dạng tệp với nhau

`CBZParser` là lớp cài đặt giao diện `ArchiveParser`. Như đã phân tích ở Chương 2, hệ thống đọc ghi tệp SAF của Android chỉ cho phép đọc ghi tuần tự. Việc tạo ra mảng offset không đơn giản, do phần mục lục của tệp ZIP nằm ở cuối, và có nhiều thao tác cần dò ngược từ cuối lên.

Để giải quyết vấn đề danh sách offset, có hai cách đơn giản nhất:

- Chép toàn bộ tệp truyện vào phần bộ nhớ riêng của ứng dụng
 - Ưu: Phần bộ nhớ này vẫn được dùng API File của Java, do đó có thể đọc ghi ngẫu nhiên, cho phép đọc mục rất nhanh.

- Nhược: Tệp truyện rất nặng (vài chục đến vài trăm MB) dẫn đến tốn cả dung lượng đĩa lẫn băng thông đọc/ghi. Ghi xóa liên tục cũng có hại cho bộ nhớ thể rắn của điện thoại.
- Đọc tệp ZIP ở chế độ đọc tuần tự
 - Ưu: Dùng ngay được với cơ chế đọc qua `InputStream` của SAF
 - Nhược: Do không có mục lục, dữ liệu phải được “dò” từ từ để đọc từng tệp lẻ một. Hậu quả là phương pháp này vừa tốn băng thông đọc, vừa tốn CPU để giải nén những tệp không cần thiết.

`CBZParse` giải quyết vấn đề này bằng cách làm giả một luồng nhập ngẫu nhiên, được miêu tả rõ hơn trong Phụ lục 3. Cách làm đó có thể được tóm tắt như sau:

- Hai phần đầu tệp nén được lưu đệm trong RAM, do là hai phần có nhiều truy cập nhất trong khi đọc mục lục
- Các phần còn lại được đọc xuôi khi cần theo luồng nhập `InputStream`, nếu đọc ngược sẽ phải tạo mới luồng nhập

Kết quả là mục lục đọc được mà chỉ cần:

- Trung bình hai lần đọc tuần tự theo `InputStream`
- Không phải ghi ra đĩa
- Không phải giải nén những tệp không cần thiết

Tổng hợp lại `ComicParser`

Tương tác trong một ca sử dụng hiển thị của `ComicParser` được mô tả trong Hình 4.14.

Ở đây cần làm rõ chi tiết về việc sắp xếp tệp theo tên. Không có quy chuẩn cho tên trang truyện, tuy nhiên đa số các tệp truyện đặt tên theo định dạng sau:

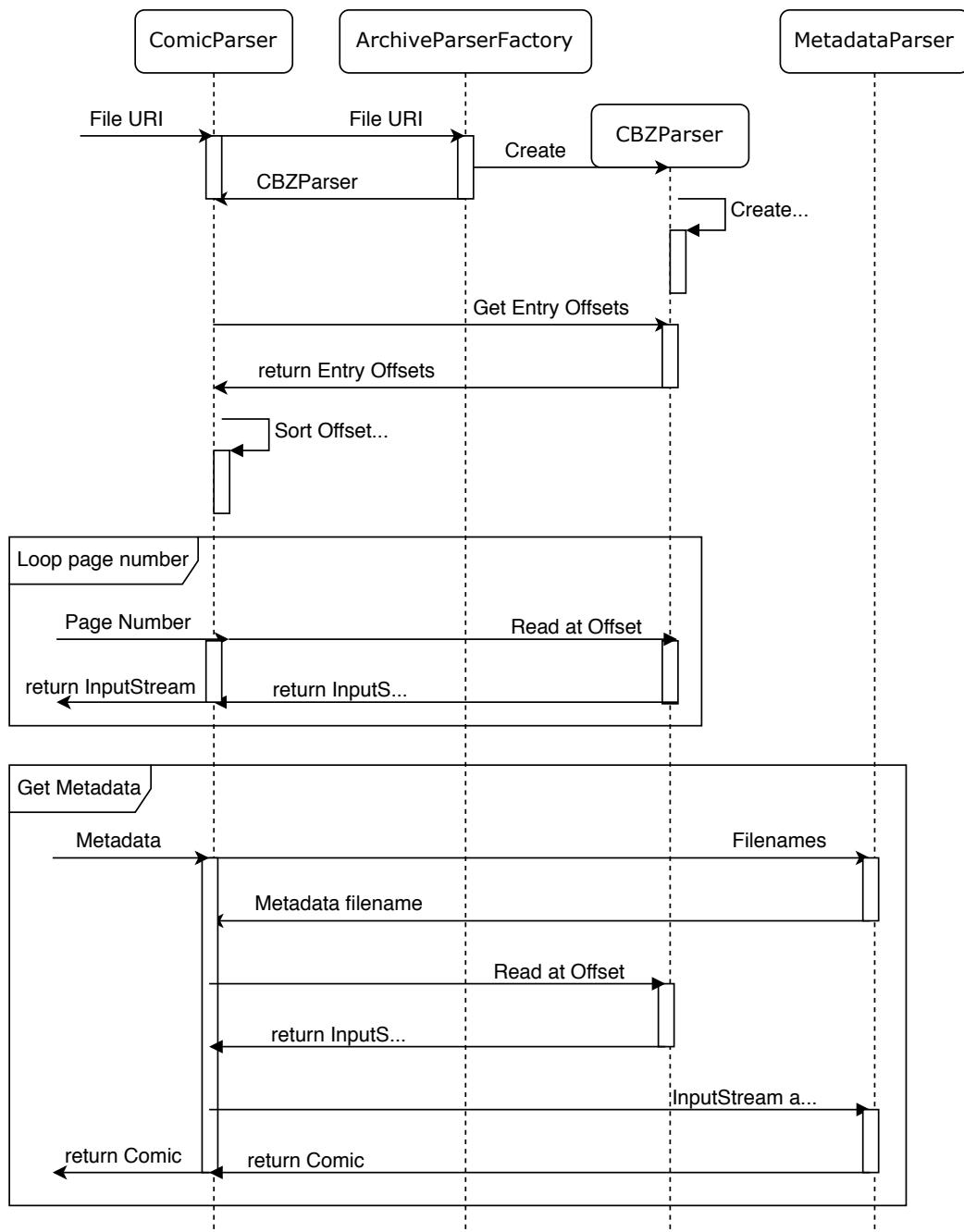
X-Men Vol 40 1.jpg

			Trang truyện số
			Số Volume, Number,...

Tên tệp truyện

Vấn đề với định dạng này xuất hiện khi truyện có nhiều hơn 10 trang. Khi sắp xếp tệp ảnh theo ABC, các trang sẽ có thứ tự như sau:

X-Men Vol 40 1.jpg
 X-Men Vol 40 10.jpg
 X-Men Vol 40 11.jpg
 ...
 X-Men Vol 40 19.jpg
 X-Men Vol 40 2.jpg
 ...



Hình 4.14: ComicParser và các thành phần của nó

Ta thấy ngay rằng thứ tự tệp ảnh bị đảo lộn. Để giải quyết vấn đề này, cần viết hàm so sánh riêng cho tên tệp ảnh. Ý tưởng ở đây là gom những kí tự số liên tiếp với nhau thành một “kí tự” rồi mới so sánh. Đoạn mã giả ở trang sau trình bày thuật toán:

```
def compare(str1, str2):
    arrs = []

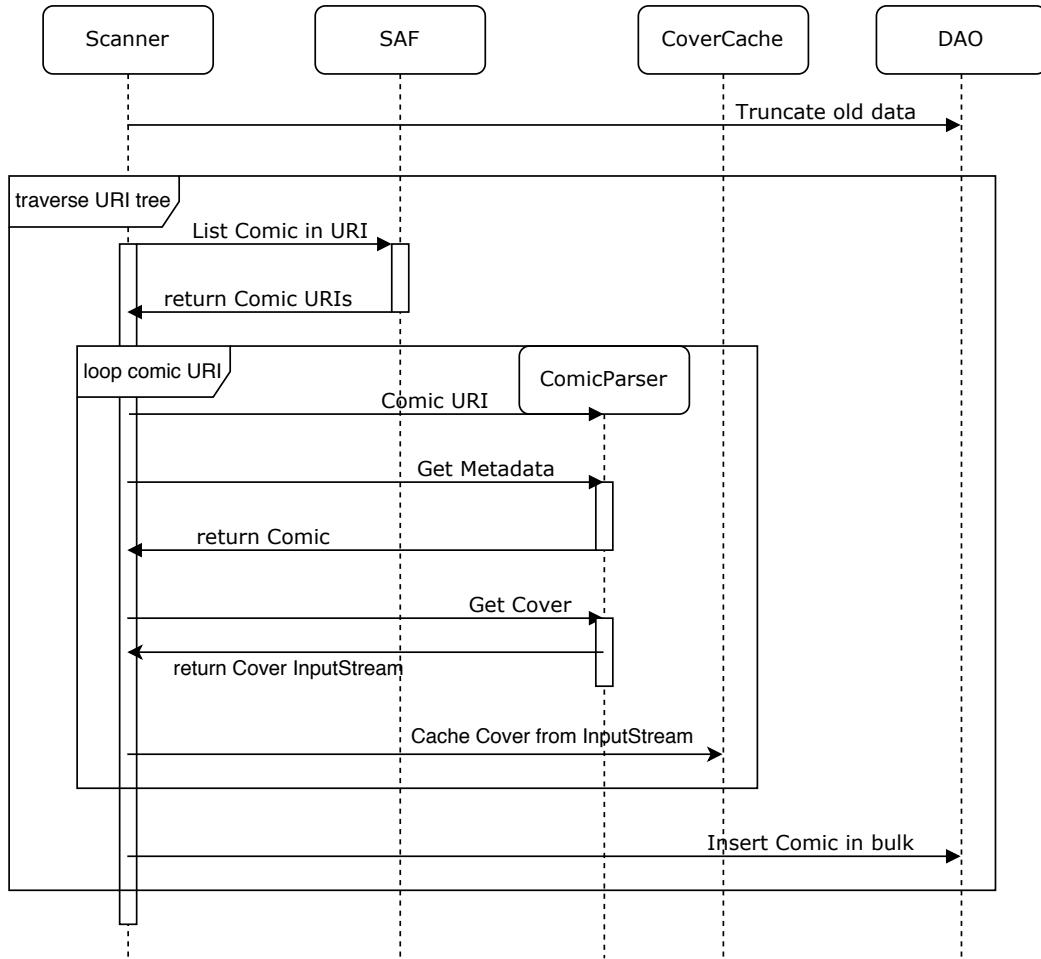
    for str in [str1, str2]:
        arrtmp = []
        acc = []

        for char in str:
            if is_number(char):
                acc.append(char)
            else:
                if len(acc) != 0:
                    acc = ''.join(acc)
                    acc = to_num(acc)
                    arrtmp.append(acc)
                    acc = 0
            arrtmp.append(to_codepoint(char))

    return compare_left_to_right(arrs[0], arrs[1])
```

Scanner

Đây là lớp phục vụ cho tính năng quét truyện trong yacy. Biểu đồ luồng của ca sử dụng *quét mới* được thể hiện trong Hình 4.15.



Hình 4.15: Biểu đồ tuần tự của ca sử dụng quét mới tệp truyện

Một chi tiết mới trong biểu đồ này là lớp `ImageCache`, sẽ được giới thiệu ở mục về cache ảnh.

Ca sử dụng quét lại cũng có thiết kế gần tương tự, trong đó bỏ bước xóa dữ liệu, và thêm một bước quét cơ sở dữ liệu sau cùng để xóa truyện không còn trong bộ nhớ. Việc cập nhật và thêm truyện mới được thực hiện trong vòng lặp lớn bình thường.

Scanner nhận vào URI của thư mục gốc, rồi lặp qua từng tệp con, cháu,... Nếu đó là tệp truyện, nó gọi ‘ComicParser’ để lấy metadata, rồi lưu vào cơ sở dữ liệu qua DAO.

Quá trình quét tệp này giống như duyệt cây, do đó có hai cách cơ bản:

- Duyệt theo độ sâu (depth-first search, gọi tắt là DFS)
- Duyệt theo độ rộng (breadth-first search, gọi tắt là BFS)

Trong trường hợp cụ thể này, DFS được chọn. Lý do cho lựa chọn này là DFS có thể phát hiện *thư mục* nhanh hơn nhiều so với BFS. Mỗi khi gặp thư mục, DFS xử lý (đi xuống các thư mục con) ngay, thay vì thêm vào hàng đợi. Do phát hiện được thư mục

nhanh hơn BFS, Màn hình Thư viện, vốn hiển thị danh sách các *thư mục*, cũng hiển thị sớm hơn. Dù thời gian quét tổng thể không thay đổi, người dùng được thấy thư mục sớm hơn giúp tạo cảm giác ứng dụng khá nhanh.

4.4 Module Image Loader

Module Image Loader chịu trách nhiệm trích xuất và lưu đệm (cache) tệp ảnh. Module này gồm hai phần như sau:

4.4.1 Image Extractor

Đây thực chất là một lớp đóng gói quanh `ComicParser`. Bản thân chức năng trích xuất được thực hiện trong `ComicParser` (qua các đối tượng `ArchiveParser`), tuy nhiên Image Extractor thực hiện một số tối ưu giúp việc hiển thị ảnh nhanh chóng hơn.

Ở các mục trước, ta đã tệp lẻ trang truyện không được lưu theo thứ tự đọc. Do đó, cần mục lục tệp nén để có thể nhảy cóc đến trang truyện theo yêu cầu. Tuy nhiên, việc tải ảnh còn có thể tối ưu hơn nữa. Ta xét ví dụ trong Bảng 4.3:

Bảng 4.3: Danh sách các tệp ảnh trong một tệp truyện nén

Thứ tự trong tệp nén	Tên tệp ảnh	Dung lượng
1	7.jpg	100KB
2	6.jpg	100KB
3	8.jpg	100KB
4	1.jpg	100KB
5	3.jpg	100KB
6	4.jpg	100KB
7	2.jpg	100KB
8	5.jpg	100KB

Hiển nhiên, thứ tự ảnh cần xem là từ tệp 1.jpg đến tệp 8.jpg. Ta xem xét và cải tiến các chiến lược tải ảnh qua các tiểu mục tiếp theo.

Tải theo yêu cầu

Ảnh được tải theo đúng yêu cầu ngay lúc đó. Quá trình đọc tệp tin như sau:

- Đọc 1.jpg: 100KB (bản thân ảnh) + 300KB (do trước khi đọc được 1.jpg cần đi qua 3 ảnh 7.jpg, 6.jpg, 8.jpg)

- Đọc 2.jpg: 100KB + 600KB
- ...

Vậy để đọc hết truyện, cần đọc 4500KB. Chưa hết, luồng nhập phải được tạo mới mỗi lần đọc (tức bằng số trang truyện), gây ra nhiều overhead. Lý do là bản thân SAF là một Content Provider, do đó nó nằm ở một tiến trình (process) riêng, và cần cơ chế liên lạc xuyên tiến trình (IPC) - vốn đắt đỏ về mặt tính toán trên mọi hệ điều hành - để gọi.

Nguyên nhân của cả hai điểm yếu trên là việc không sử dụng lại luồng nhập (mỗi `InputStream` chỉ đọc một ảnh). Phương án tiếp theo cần xử lý được điểm yếu này.

Tối thiểu hóa số luồng đọc

Để giảm số luồng đọc, ta cần kiểm soát một vài luồng đọc, và phân mỗi trang truyện cho một luồng đọc cụ thể. Để tối thiểu hóa số luồng đọc, ta cần dùng thêm thuật toán *chuỗi con tăng dài nhất* (longest increasing subsequence). Thuật toán cuối cùng thể hiện bằng mã giả như sau:

```
def minStream(pages):
    stream_count = 0
    map_idx_to_stream = []      # Từ điển ánh xạ số trang - số luồng

    while len(pages) != 0:
        # Dây trang tiến lên
        lis = longest_increasing_subsequence(pages)

        for page in lis:
            pages.remove_at(page)    # Bỏ trang trong dây khỏi danh sách
            map_idx_to_stream[page] = stream_count    # Gán số luồng hiện tại

        stream_count += 1

    return map_idx_to_stream
```

Thuật toán nhận vào một mảng `pages` là *thứ tự trong tệp nén* của từng trang truyện. Thuật toán trả về một từ điển như sau:

- Khóa: trang truyện số (bắt đầu từ trang 1)
- Giá trị: số luồng

Ta nhận thấy thuật toán hiển nhiên cho (xấp xỉ) số luồng ít nhất có thể, vì mỗi lần chia trang cho các luồng, ta chọn một bộ trang tăng dần dài nhất lúc đó.

Áp dụng vào ví dụ đang dùng, ta có:

- Luồng 0: đọc trang 1, 3, 4, 5
- Luồng 1: đọc trang 7, 8
- Luồng 2: đọc trang 6
- Luồng 3: đọc trang 2

Vậy để đọc hết truyện, cần đọc 2000KB, và 4 lần tạo mới luồng nhập, khá tốt so với phương pháp đầu.

Tới đây chỉ cần một số chỉnh sửa nhỏ: Giới hạn số luồng nhập. Trường hợp xấu nhất là thứ tự trong tệp ZIP ngược với thứ tự đọc, do đó có nhiều luồng mà mỗi luồng chỉ để đọc một trang truyện. yacv tránh điều này bằng cách giới hạn chỉ có 4 luồng nhập cùng lúc. Những trang không ở trong phạm vi của các luồng này quay về cách đọc nhảy cóc thông thường, không dùng lại luồng nhập chờ sẵn.

4.4.2 Image Cache

Bản thân Image Cache *không* phải là một đoạn mã, đối tượng, mà chỉ là một thư mục. yacv có 2 loại/thư mục cache, cho hai trường hợp hiển thị:

- Cache ảnh bìa
- Cache trang truyện

Lý do cần đến hai bộ cache khác nhau là vì dung lượng lớn của truyện. Các thư viện cache ảnh chọn mốc 100-200MB cho thư mục cache ảnh, đồng thời dùng cố định phương pháp thay thế LRU (nếu cache đầy sẽ xóa ảnh lâu nhất không được dùng). Khi đọc một bộ truyện dung lượng lớn hơn mốc này, toàn bộ ảnh trong cache sẽ sớm bị thay thế bởi ảnh của trang truyện. Sau khi đọc, quay về các màn hình, ảnh bìa dùng để hiển thị trong hai màn hình duyệt truyện bị mất, gây suy giảm trải nghiệm người dùng. Do đó, cần phải tách hai thư mục cache này ra để tránh ảnh hưởng đến cache trang bìa.

Cache trang truyện

Khác với cache trang truyện, cache trang bìa *không* có liên quan tới Image Loader. Thực ra ảnh bìa thì cũng được trích xuất bởi ComicParser, tuy nhiên với mỗi tệp truyện chỉ

cần trích ra một ảnh bìa, do đó không cần cơ chế tái sử dụng luồng đọc phức tạp của Image Loader.

Cache trang bìa lại gồm 2 thư mục cache:

1. Cache ảnh hiển thị thực tế

Cache ảnh được hiển thị bởi `ImageView`. Ảnh này là ảnh bìa đã được cắt (xem phần thiết kế màn hình duyệt truyện) và được thu phóng về chính xác kích cỡ khung nhìn. Dung lượng cache là 100MB.

Cùng một bìa có hai kiểu hiển thị

Thư mục cache này được quản lý bởi thư viện hiển thị ảnh. Thư viện đó nhận luồng đọc ảnh (từ `ComicParser`), ghi vào thư mục cache này, và xóa ảnh để giải phóng dung lượng khi cần.

2. Cache ảnh bìa thu nhỏ

Cache ảnh bìa thu nhỏ, khoảng 50KB mỗi ảnh, chưa bị cắt. Dung lượng cache là 10MB.

Lí do riêng phần bìa cần hai thư mục cache là vì cache ảnh hiển thị thực tế, giống với cache trang truyện, có thể bị xóa bất cứ lúc nào. Do đó cần có một cache rất nhỏ gọn, nằm ở thư mục riêng mà Android không xóa được, chứa ảnh bìa chất lượng thấp, để khi ảnh bìa bị xóa vẫn có một bản bìa nhỏ để hiển thị trong khi chờ ảnh bìa chất lượng cao, có cắt cúp phù hợp được sinh lại.

Trong Hình 4.15, có một lớp `ImageCache` nhận `InputStream` của ảnh bìa và cache ảnh, đó chính là lớp quản lý và sinh ảnh bìa thu nhỏ.

Chương 5

Lập trình & Kiểm thử

Chương này nêu một số đoạn mã trong quá trình lập trình và kiểm thử, cùng với ảnh chụp màn hình của kết quả cuối cùng.

5.1 Lập trình

5.1.1 Coroutine

Về mặt lập trình, trong quá trình viết ứng dụng, một trong các khó khăn chủ yếu là tăng tốc ứng dụng bằng coroutine. Sau đây trình bày một trong những đoạn mã dùng coroutine trong ca sử dụng tìm kiếm, có sử dụng cả LiveData:

```
suspend fun search(query: QueryMultipleTypes, limit: Int):  
    LiveData<List<List<Metadata>>> = liveData(timeoutInMs = 3000) {  
    val results2D = mutableListOf<List<Metadata>>()  
    val latch = CountDownLatch(daos.size)  
  
    withContext(Dispatchers.IO) {  
        daos.parallelForEach { dao ->  
            val results = dao.search(query.query, limit)  
            var shouldEmit = false  
  
            // TODO: How about making use of the other search function...  
            synchronized(results2D) {  
                if (results.isNotEmpty()) {  
                    results2D.add(results)  
                    shouldEmit = true  
                }  
            }  
        }  
    }  
    results2D  
}  
}
```

```

        }

        latch.countDown()
        if (latch.count == 0 && results2D.size == 0) {
            shouldEmit = true
        }
    }

    if (shouldEmit) emit(results2D)
}
}
}

```

Đoạn mã này trả về một đối tượng LiveData<List<List<Metadata>>>:

- Metadata: Đối tượng chứa một kết quả tìm kiếm, có thể là Comic, Series,...
- List<Metadata>: Chứa kết quả tìm kiếm của một DAO, các Metadata trong danh sách này thuộc cùng một kiểu.
- List<List<Metadata>>: Danh sách kết quả của một DAO, nếu có kết quả (không rỗng) sẽ được thêm vào danh sách tổng hợp. Danh sách tổng hợp chính là danh sách kết quả tìm kiếm chưa làm phẳng (xem lại ca sử dụng tìm kiếm về cách làm phẳng).
- LiveData: Dùng cho tính năng data binding với View.

Đoạn mã làm những việc sau:

- Tạo ra 6 coroutine cho 6 DAO, tương ứng với 6 bảng, để tìm kiếm dữ liệu.

```
daos.parallelForEach
```

Cụ thể hơn, đây là một hàm tự viết, có chữ kí như sau:

```
suspend fun <A> Iterable<A>.parallelForEach
    (f: suspend (A) -> Unit): Unit =
        coroutineScope { forEach { launch { f(it) } } }
```

Hàm này “mở rộng” giao diện Iterable<A> (trong trường hợp này là daos, có kiểu là List<Dao>). Nó nhận một hàm f, tạo một coroutine (launch) ứng với mỗi phần tử trong Iterable<A>, và chạy hàm f trong coroutine riêng đó.

Bản thân hàm `f` nhận vào một phần tử A (trong trường hợp này là Dao), và thực hiện tính toán trên coroutine riêng.

- Mỗi khi có kết quả (danh sách kết quả không rỗng), thêm vào danh sách tổng hợp, và thông báo cho View để hiển thị.

Biến `shouldEmit` kiểm soát xem có cần thông báo (emit) cho View không. Nếu danh sách kết quả từ một Dao không rỗng, danh sách đó được đưa vào danh sách tổng hợp.

6 Dao sẽ có tối đa 6 lần thông báo khác nhau, giúp Màn hình Tìm kiếm được hiển thị từ từ theo quá trình tìm kiếm, đúng với yêu cầu phi chức năng về việc liên tục cập nhật.

- Thông báo cho view ít nhất một lần để báo kết thúc tìm mà không có kết quả.

Trong trường hợp không có kết quả từ cả 6 Dao, thì cần thông báo danh sách tổng hợp rỗng cho View. Việc này được thực hiện bằng một `CountDownLatch`.

Mỗi khi Dao tìm xong, latch sẽ giảm đi 1 (`countDown()`). Khi latch bằng không, tức đã xong cả 6 Dao, mà danh sách tổng hợp vẫn trống (không tìm thấy gì), thì đặt `shouldEmit` để báo một lần cho View.

Do coroutine có thể chạy trên nhiều luồng khác nhau, nên cần một `CountDownLatch` để đếm (chứ không thể dùng một biến đếm thông thường) và môi trường `synchronized` để chạy. Nói cách khác, đoạn mã này là một “critical section” trong lập trình đa luồng.

- Toàn bộ 6 coroutine được chạy trên (các) luồng IO.

`withContext(Dispatchers.IO)`

Đây là mấu chốt để ứng dụng có cảm giác mượt mà khi tìm kiếm.

Việc tạo ra 6 coroutine có thể so sánh với cùng lúc tìm kiếm song song trên 6 luồng (thread) khác nhau, giúp giảm thời gian tìm kiếm. Tuy nhiên, nếu trong 6 luồng này lại có một luồng là *luồng giao diện* (UI thread), thì ứng dụng sẽ bị treo, không phản ứng cho đến khi Dao trong luồng đó tìm xong.

Để tránh việc này, coroutine cần được đảm bảo không chạy trên luồng giao diện. Trong trường hợp này, 6 coroutine được chạy trên các luồng IO - một thread pool dành riêng cho nhập/xuất - phù hợp với bản chất công việc là đọc cơ sở dữ liệu.

Đoạn mã trên thể hiện hai tác dụng quan trọng của coroutine:

Tính năng	Vấn đề giải quyết
Chạy song song	Tăng tốc ứng dụng
Không chạy trên luồng giao diện	Giúp ứng dụng luôn phản hồi (responsive)

Mỗi khi ứng dụng cần đọc ảnh từ tệp nén, và khi truy cập cơ sở dữ liệu, hai vấn đề trên xảy ra và cần coroutine để giải quyết.

5.1.2 CBZParser

Chương 4 - Thiết kế đã giới thiệu sơ lược về cách **CBZParser** hoạt động, là lưu ý hai phần đầu-cuối tệp tin. Cụ thể hơn, mã nguồn của thư viện Apache Commons Compress được phân tích để tìm mẫu đọc (read pattern), và cache những phần được đọc nhiều. Sau đó, **CBZParser** tạo ra một luồng đọc ngẫu nhiên - là đầu vào yêu cầu của thư viện - nhưng bên dưới là **InputStream** thông thường và dữ liệu cache.

Thư viện Apache Commons Compress được chọn vì những lí do sau:

- Viết bằng Java: tiêu chí này loại được zlib, dù là thư viện mạnh nhất nhưng viết bằng C++ và có API phức tạp; hơn nữa đâu nào mọi thư viện đọc tệp ZIP đều chỉ là “vỏ”, “lõi” thực chất đều là zlib
- Mở rộng: tiêu chí này loại được thư viện ZIP tích hợp trong Java, vì nếu hỗ trợ Apache Commons Compress thì sau này có thể thêm một số định dạng nén được thư viện này hỗ trợ.

Mẫu đọc *tệp ZIP*, không phải chế độ luồng (phù hợp với **InputStream** nhưng chậm và không thể nhảy cóc), có thể tóm tắt như sau:

1. Tìm “mục lục” - Central Directory

- (a) Tìm đuôi mục lục

“Đuôi” mục lục được đánh dấu bằng chuỗi 0x06054b50. Chuỗi này được dò ngược từ cuối tệp ZIP, và có thể dò tối đa 65537 byte.

- (b) Đặt vị trí đọc ở đầu mục lục

Sau khi tìm được đuôi mục lục, ta tìm được vị trí trong tệp (offset) của mục lục, và cần đặt vị trí đọc ở đó.

2. Đọc mục lục

- (a) Đọc một đoạn nhỏ 4 byte ở đầu để kiểm tra có lỗi không

(b) Đọc mục lục để có offset của từng “đề mục” - File Entry.

3. Đọc đề mục

Đọc đề mục từ đầu đến cuối để kiểm tra một số thông tin.

Dựa theo mẫu đọc trên, `CBZParser` cache - hay trong trường hợp này từ chính xác hơn là `buffer` - và đọc dữ liệu như sau:

- Một luồng đọc `InputStream` đọc đến hết tệp, buffer lại 128KB đầu tiên và 1MB cuối cùng.

Theo kinh nghiệm, 1MB là đủ để chứa mục lục. 1MB này sẽ giúp cho việc dò ngược và đọc mục lục không cần tạo mới luồng nhập, miễn là truy cập trong vòng 1MB cuối.

128KB ở đầu giúp cho một số kiểm tra chạy được.

Đến đây, luồng đọc đầu tiên có thể được bỏ đi. Nếu truy cập ra ngoài phạm vi đã lưu, ném ngoại lệ và ứng dụng thử lại với bộ đệm lớn hơn.

- Tạo mới luồng đọc thứ hai: chuẩn bị cho một loạt hoạt động đọc tuần tự các File Entry.

Quá trình sử dụng các thao tác trên trong `CBZParser` như sau:

- Toàn bộ các thao tác trên được cài đặt trong `ZipBuffer` là một lớp cài đặt giao diện `SeekableByteChannel` - một luồng đọc ngẫu nhiên.
- `ZipBuffer` được truyền vào bộ đọc `ZipFile` của thư viện.
- Sau khi `ZipFile` khởi tạo xong, tức đọc xong mục lục, `CBZParser` lấy ra từ `ZipFile` danh sách tệp lẻ (File Entry) cùng với offset của nó.

5.2 Kiểm thử

5.2.1 ZipBuffer

Lớp này được kiểm thử riêng, do có quan hệ chặt chẽ với thư viện Apache Commons Compress. Cách kiểm thử là dùng lại nguyên các unit test cho `ZipFile` (đi kèm mã nguồn), nhưng thay vì truyền vào `SeekableByteChannel` thì truyền vào một `ZipBuffer`.

Các unit test này đều chạy thành công, và thống kê được một số trường hợp ngách (chủ yếu là khi dung lượng tệp nén lớn nhưng nhiều tệp nhỏ) cần 3 lần tạo mới luồng đọc, thay vì 2 như dự kiến.

5.2.2 Unit test

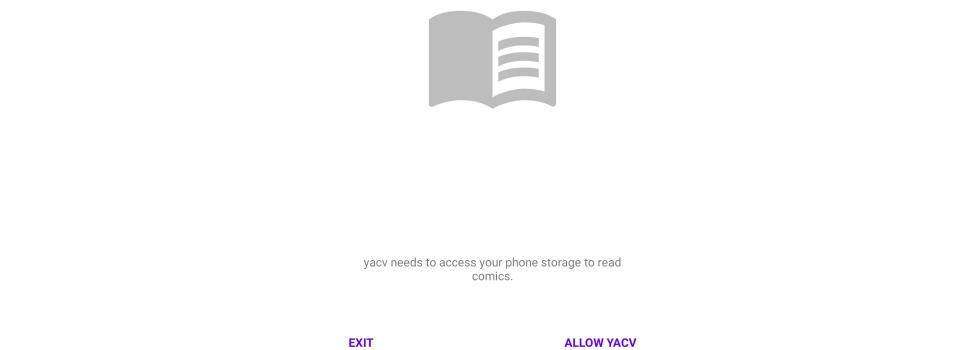
Các unit test của yacy tập trung vào phần cơ sở dữ liệu Room. Việc test cơ sở dữ liệu diễn ra theo bản mẫu sau:

```
/**  
 * Tao moi va dua du lieu gia  
 */  
  
@Before fun createDb() = runBlocking {  
    val context = InstrumentationRegistry.getInstrumentation().targetContext  
    database = Room.inMemoryDatabaseBuilder(context, AppDatabase::class.java).build()  
  
    database.comicDao().insertAll(testComics)  
}  
  
/**  
 * Test them truyen trong folder moi hoan toan  
 */  
  
@Test fun test_addComic_newFolder() {  
    database.comicDao().insert(testComic)  
  
    assertEquals(database.folderDao().last().Uri, testComic.FolderUri)  
}
```

5.3 Sản phẩm

5.3.1 Màn hình Quyền đọc

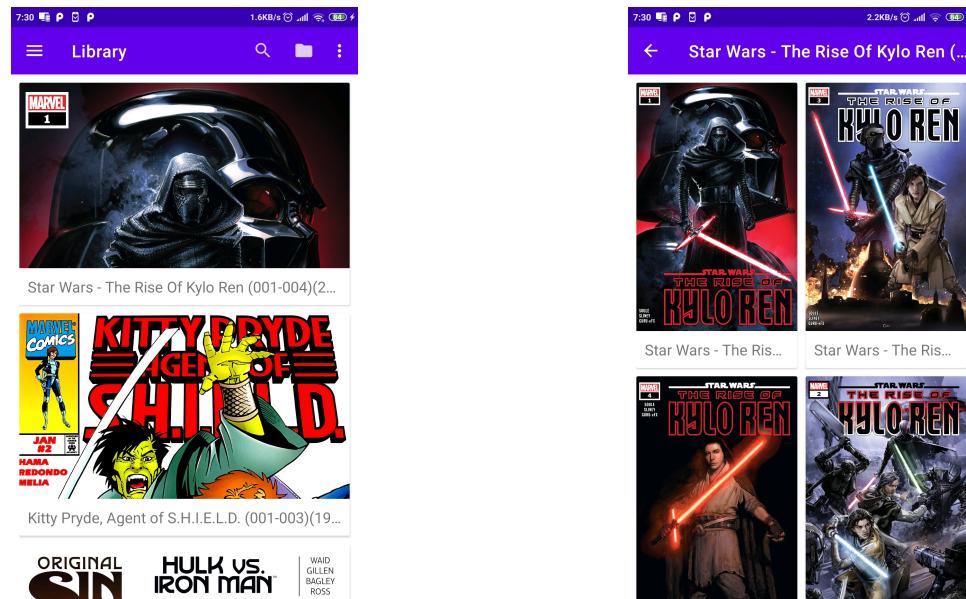
Màn hình xuất hiện khi mở ứng dụng mà không có quyền đọc dữ liệu (lần đầu dùng, người dùng chủ động bỏ quyền đọc trong Settings,...)



Hình 5.1: Màn hình Quyền đọc

5.3.2 Màn hình Duyệt truyện

Hai màn hình - Màn hình Thư viện và Màn hình Thư mục dùng để duyệt danh sách truyện. Màn hình Thư viện là màn hiển thị mặc định. Từ nó, khi ấn vào một thư mục, ta truy cập được Màn hình Thư mục tương ứng.



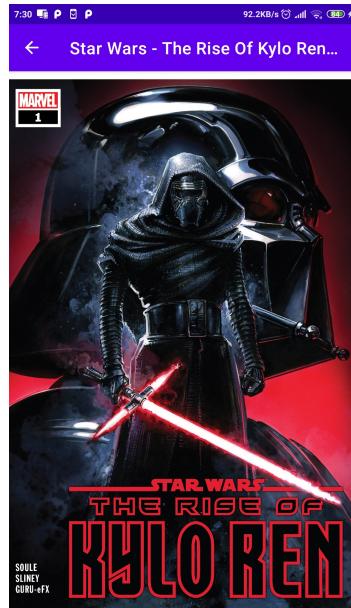
(a) Màn hình Thư viện

(b) Màn hình Thư mục

Hình 5.2: Hai Màn hình Duyệt truyện

5.3.3 Màn hình Đọc truyện

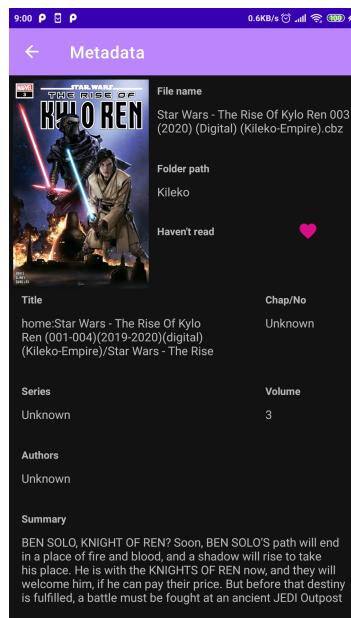
Màn hình Đọc truyện dùng để xem các trang truyện, được truy cập bằng cách ấn vào một truyện trong Màn hình Thư mục.



Hình 5.3: Màn hình Đọc truyện

5.3.4 Màn hình Metadata

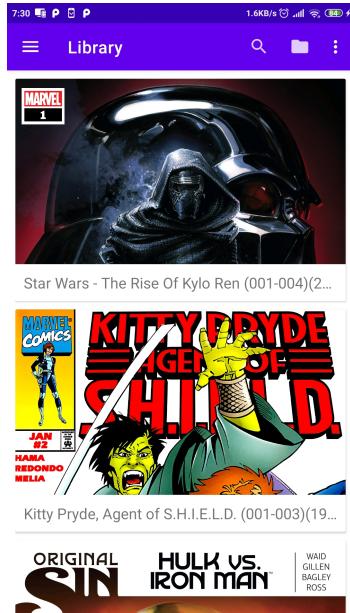
Màn hình này hiển thị metadata đi kèm, được truy cập từ Màn hình Đọc truyện.



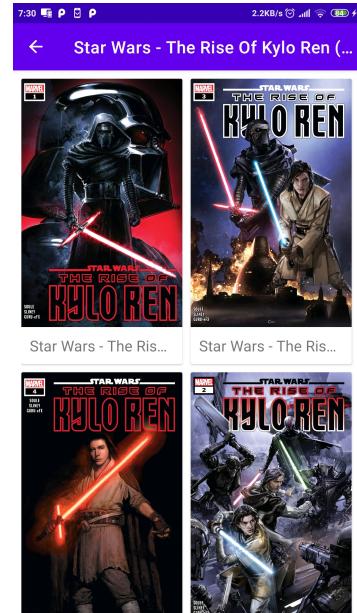
Hình 5.4: Màn hình Metadata

5.3.5 Màn hình Tìm kiếm

Màn hình Tìm kiếm gồm hai màn hình con gần giống nhau, dùng để rút gọn số lượng kết quả hiển thị. Cả hai màn hình đều có đích đến là Màn hình Đọc truyện.



(a) Màn hình Tìm kiếm Tổng quan



(b) Màn hình Tìm kiếm Chi tiết

Hình 5.5: Hai Màn hình Tìm kiếm

Tài liệu tham khảo

- [1] *.ZIP File Format Specification*. URL: <https://web.archive.org/web/20210502033210/> <https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT> (**urlseen 02/05/2021**).
- [2] *Guide to app architecture*. URL: <https://web.archive.org/web/20210421063755/> <https://developer.android.com/jetpack/guide> (**urlseen 21/04/2021**).
- [3] loocool2. *Feature Request: Add tachiyomi to play store*. 11/2018. URL: <https://github.com/tachiyomio/tachiyomi/issues/1745#issuecomment-441163306> (**urlseen 01/05/2021**).
- [4] *Model - view - presenter*. URL: <https://web.archive.org/web/20210323022152/> <http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93presenter> (**urlseen 23/03/2021**).
- [5] Florina Muntenescu. *Android Architecture Patterns Part 3: Model-View-ViewModel*. 11/2016. URL: <https://web.archive.org/web/20210218163524/> <https://medium.com/upday-devs/android-architecture-patterns-part-3-model-view-viewmodel-e7eeee76b73b> (**urlseen 18/02/2021**).
- [6] *Open files using storage access framework*. URL: <http://web-old.archive.org/web/20210501061438/> <https://developer.android.com/guide/topics/providers/document-provider> (**urlseen 01/05/2021**).
- [7] *Platform Architecture*. URL: <https://web.archive.org/web/20210126104437/> <https://developer.android.com/guide/platform> (**urlseen 26/01/2021**).
- [8] Nathaniel J. Smith. *Notes on structured concurrency, or: Go statement considered harmful*. 04/2018. URL: <https://vorpus.org/blog/notes-on-structured-concurrency-or-go-statement-considered-harmful> (**urlseen 01/05/2021**).
- [9] *ZIP (file format)*. URL: [https://web.archive.org/web/20210427132143if_/https://en.wikipedia.org/wiki/ZIP_\(file_format\)](https://web.archive.org/web/20210427132143if_/https://en.wikipedia.org/wiki/ZIP_(file_format)) (**urlseen 27/04/2021**).

Phụ lục A

Giải thích các trường metadata

Mô hình xuất bản của truyện tranh siêu anh hùng phương Tây là phức tạp nhất. Lý do là các nhân vật không đổi trong hàng chục năm xuất bản nhưng cốt truyện không ngừng được thêm mới, hoặc thậm chí viết lại; khác với các truyện tranh khác luôn đi đến hồi kết. Do đó, các thông tin của kiểu truyện tranh này được chọn để thiết kế các định dạng metadata.

Ta xét một tập truyện *Wolverine 1982(1) #1*:

Wolverine 1982(1) #1		
		Tập truyện số (Number)
		Volume
Bộ truyện (Series)		

Chú ý ở đây không có *tiêu đề* (ta sẽ quay lại điểm này sau). Trước hết các thành phần trong tên tập truyện trên gồm:

- *Bộ truyện*: Tên bộ truyện. Một bộ truyện gồm nhiều tập truyện.
- *Tập truyện số*: Thể hiện số thứ tự xuất bản của tập truyện, tương tự như chương trong manga. Từng tập truyện lẻ còn có thể có tên riêng.
- *Volume*: Các bộ truyện có thể trùng tên, do đó cần con số này để phân biệt. Số này có thể là năm xuất bản hoặc lần xuất bản.

Số Volume cần thiết vì có rất nhiều bộ truyện cùng tên như sau:

- Có một bộ Wolverine ngắn gồm 4 tập, xuất bản năm 1982
- Có một bộ Wolverine gồm nhiều tập, xuất bản từ 1989 đến 2003

- Có một bộ Wolverine gồm nhiều tập, xuất bản từ 2003 đến 2010

Các bộ Wolverine trên đều có nội dung khác nhau, thậm chí cũng không cùng dòng thời gian, không cùng tác giả để có thể gom lại. Nhưng chúng cùng dùng một tên bộ truyện (là tên nhân vật chính), đều có những tập truyện số 1, 2, 3, 4. Số Volume là cách duy nhất để phân biệt ba bộ truyện này.

Số Volume thường là số của lần xuất bản, nhưng cũng có thể là số của năm xuất bản trong trường hợp bộ truyện được xuất bản trong một năm.

Ngoài ra, một số định dạng metadata còn có số Count, chỉ số tập truyện trong một bộ truyện:

- Bộ Wolverine đầu tiên được gọi là “ngắn” (miniseries) vì nhà xuất bản xác định và thông báo trước rằng chỉ có bốn tập truyện. Do đó, số Count là 4.
- Hai bộ còn lại được coi là “dài” (on-going), do không xác định số tập truyện từ đầu. Do đó, số Count có thể để trống.

Ta quay lại vấn đề về *tiêu đề*. Tiêu đề có thể nói giống như tên của chương truyện, lại chỉ phổ biến trong truyện tranh Nhật, hoặc ít nhất là không phổ biến trong đặt tên truyện tranh siêu anh hùng phương Tây.

Phụ lục B

Lược đồ XSD ComicInfo

Phụ lục này trình bày phiên bản rút gọn của lược đồ XSD của định dạng metadata ComicInfo. Định dạng này được lưu trong một tệp có tên `ComicInfo.xml`. Các trường nêu trong Phụ lục A đều được chứa trong định dạng này, ngoài ra còn có các trường về tác giả và nhân vật.

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema elementFormDefault="qualified"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="ComicInfo" nullable="true" type="ComicInfo" />
  <xs:complexType name="ComicInfo">
    <xs:sequence>
      <xs:element name="Title" type="xs:string" />
      <xs:element name="Series" type="xs:string" />
      <xs:element name="Number" type="xs:string" />
      <xs:element name="Count" type="xs:int" default="-1" />
      <xs:element name="Volume" type="xs:int" default="-1" />
      <xs:element name="Summary" type="xs:string" />
      <xs:element name="Year" type="xs:int" default="-1" />
      <xs:element name="Month" type="xs:int" default="-1" />
      <xs:element name="Writer" type="xs:string" />
      <xs:element name="Publisher" type="xs:string" />
      <xs:element name="Genre" type="xs:string" />
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```