

APP REPORT – PART 3

TACHIYOMI

Last Update: 2023-11-25

Corrections:

- Caching Strategies: Include snippets of code of the first strategy.
- Caching Strategies: Include the average amount of cache storage used by Tachiyomi
- Memory management strategies: Mentions the data structures used, as well management of weak references, garbage collector, image caching, general strategy of caching and improves the explanation in each snippet of code.

Link to the repository: <https://github.com/tachiyomior/tachiyomi>

2. Identify good/bad Eventual connectivity (ECn) strategies in the third-party app

Tachiyomi is a primary offline manga reader that utilizes connectivity for several purposes, including downloading content from third-party providers, updating information in manga trackers, and checking for app updates. The eventual connectivity within the application involves caching, retrieval, and fetching strategies. The following section discusses caching strategies. In this part, we will review best practices in the use of fetching strategies and identify bad practices in the form of anti-patterns.

Fetching Strategies:

- *Checking & Downloading App Updates:*

All the code related to app updates is available on the folder `app/src/main/java/eu/kanade/tachiyomi/data/updater`. In this folder there are three files that focus on two activities: checking for updates, downloading and initiating installation of the update.

When the application is launched, it initiates the `checkForUpdate()` function. As observed from the code presented (Figure 1), this function is a suspend function, thus requiring asynchronous invocation, either within another suspend function or within a coroutine . This fetching strategy retrieves data **at launch time (background fetch)**.

```

suspend fun checkForUpdate(context: Context, forceCheck: Boolean = false): GetApplicationRelease.Result {
    return withIOContext {
        val result = getApplicationRelease.await(
            GetApplicationRelease.Arguments(
                BuildConfig.PREVIEW,
                context.isInstalledFromFroid(),
                BuildConfig.COMMIT_COUNT.toInt(),
                BuildConfig.VERSION_NAME,
                GITHUB_REPO,
                forceCheck,
            ),
        )
        when (result) {
            is GetApplicationRelease.Result.NewUpdate -> AppUpdateNotifier(context).promptUpdate(result.release)
            is GetApplicationRelease.Result.ThirdPartyInstallation -> AppUpdateNotifier(context).promptFroidUpdate()
            else -> {}
        }
        result
    }
}
return go(f, seed, [])
}

```

Figure 1. Code for Checking App Updates Summary

When a new version is available, the application sends a push notification to the system panel (Figure 3). It also displays a screen with the choice to either download the APK immediately (Download) or postpone it (Not now) (Figure 2).

If the user decides to download the new version, Tachiyomi keeps the download process in the foreground, by generating a notification that provides real-time updates on the download progress (Figure 4). Please take notice that in the download, another fetching strategy is employed, which is: pull (on user demand).

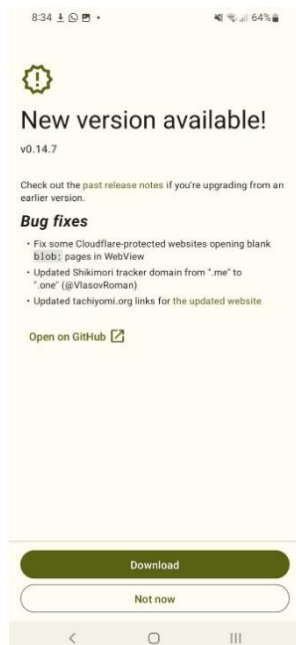


Figure 2. Screen within the app showing options when new version is available.

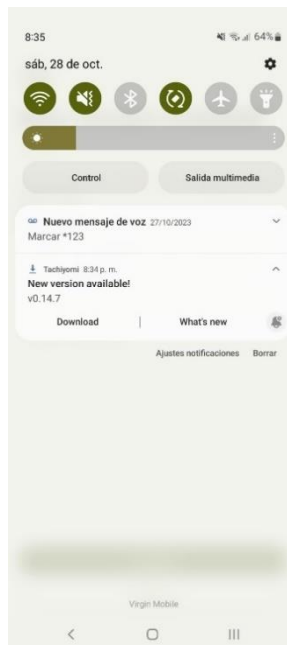


Figure 3. Notification when new version is available.

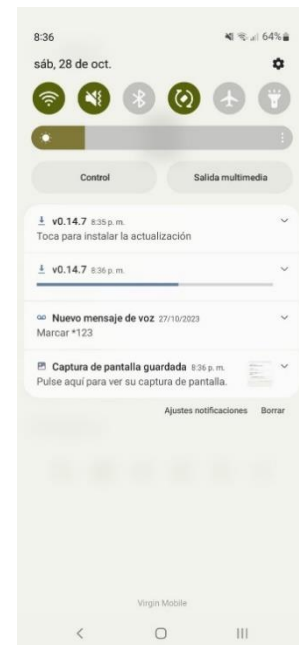


Figure 4. Notification with progress of the background service downloading APK.

- Update read status on mange trackers:

Tachiyomi also utilizes connectivity to update the reader's status on linked manga trackers. As shown in Figure 5, some manga is linked to Anilist, which serves as a tracker. The data fetching strategy, in this case, is determined by an expiration policy. At regular intervals, the application sends data to the tracker. As observed from the code (Figure 6), the application asynchronously (suspend function) retrieves and manages this data in the background using a service.

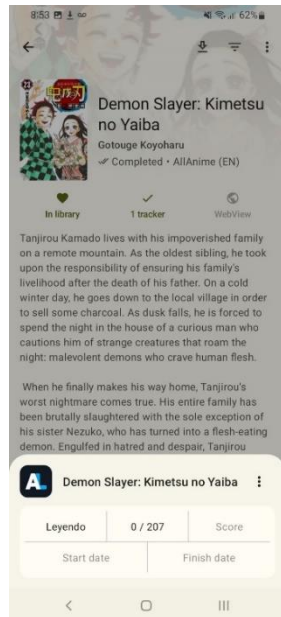


Figure 5. Tracked Manga

```
class AnilistApi(val client: OkHttpClient, interceptor: AnilistInterceptor) {
    ...
    suspend fun updateLibManga(track: Track): Track {
        return withIOContext {
            val query = """
            |mutation UpdateManga(
            |  ${'$'}listId: Int, ${'$'}progress: Int, ${'$'}status: MediaListStatus,
            |  ${'$'}score: Int, ${'$'}startedAt: FuzzyDateInput, ${'$'}completedAt: FuzzyDateInput
            |) {
            |  SaveMediaListEntry(
            |    id: ${'$'}listId, progress: ${'$'}progress, status: ${'$'}status,
            |    scoreRaw: ${'$'}score, startedAt: ${'$'}startedAt, completedAt: ${'$'}completedAt
            |  ) {
            |    id
            |    status
            |    progress
            |  }
            |}
            |"""
            val payload = buildJsonObject {
                put("query", query)
                putJsonObject("variables") {
                    put("listId", track.library_id)
                    put("progress", track.last_chapter_read.toInt())
                    put("status", track.toAnilistStatus())
                    put("score", track.score.toInt())
                    put("startedAt", createDate(track.started_reading_date))
                    put("completedAt", createDate(track.finished_reading_date))
                }
            }
            authClient.newCall(POST(apiUrl, body = payload.toString().toRequestBody(jsonMime)))
                .awaitSuccess()
            track
        }
    }
    ...
}
```

Figure 6. Code for update state in external manga tracker

- **Downloading content from providers**

The main classes in charge of download the main content in the app are located on `app/src/main/java/eu/kanade/tachiyomi/data/download` folder. In order to discover the fetching strategy, we analyze the code of the principal class.

```

class Downloader {
    ...
} {
    private fun CoroutineScope.launchDownloadJob(download: Download) = launchIO {
        try {
            downloadChapter(download)

            // Remove successful download from queue
            if (download.status == Download.State.DOWNLOADED) {
                removeFromQueue(download)
            }
            if (areAllDownloadsFinished()) {
                stop()
            }
        } catch (e: Throwable) {
            if (e is CancellationException) throw e
            logcat(LogPriority.ERROR, e)
            notifier.onError(e.message)
            stop()
        }

        private suspend fun downloadChapter(download: Download) {
            val mangaDir = provider.getMangaDir(download.manga.title, download.source)

            val availSpace = DiskUtil.getAvailableStorageSpace(mangaDir)
            if (availSpace <= -1L || availSpace < MIN_DISK_SPACE) {
                download.status = Download.State.ERROR
                notifier.onError(context.getString(R.string.download_insufficient_space), download.chapter.name,
                    download.manga.title)
                return
            }

            val chapterDirName = provider.getChapterDirName(download.chapter.name, download.chapter.scanlator)
            val tmpDir = mangaDir.createDirectory(chapterDirName + TMP_DIR_SUFFIX)

            try {
                val pageList = download.pages ?: run {
                    // Otherwise, pull page list from network and add that to download object
                    val pages = download.source.getPageList(download.chapter.toSChapter())

                    if (pages.isEmpty()) {
                        throw Exception(context.getString(R.string.page_list_empty_error))
                    }
                }
                // Don't trust index from source
                val reIndexedPages = pages.mapIndexed { index, page -> Page(index, page.url, page.imageUrl, page.url) }

                download.pages = reIndexedPages
            }

            // Delete all temporary (unfinished) files
            tmpDir.listFiles()
                ?.filter { it.name.endsWith(".tmp") }
                ?.forEach { it.delete() }

            download.status = Download.State.DOWNLOADING
        }
    }
    ...
}

```

Figure 7.

Antipatterns: The next are bad handling of connectivity examples in Tachiyomi. For each antipatter is the number according to the slides of the course.

- *Anti-pattern #2: Stuck Progress Notification*

When the app is initially launched with an internet connection, it assumes connectivity is constant. However, if the internet is deactivated and a download is initiated, the app gets stuck on a progress notification, as it falsely believes it still has an active connection. This issue can be easily resolved by implementing a listener for connectivity changes. By doing so, the app can respond more effectively to real-time events, ensuring a smoother user experience (Figure 8).

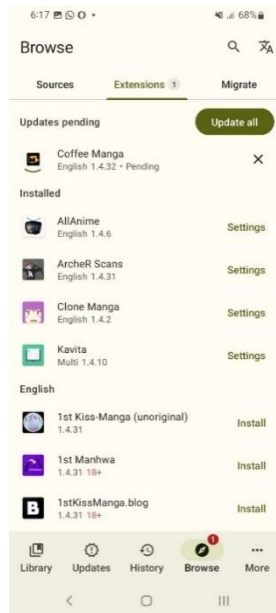


Figure 8.

- Anti-pattern #3: Non-informative Message**
 This anti-pattern is encountered under similar circumstances as the previous one. When the app lacks network connectivity at startup, it can display a non-informative error message in response to a connection attempt (Figure 9 and Figure 10). Most users may find this message unhelpful as it does not clearly communicate the issue. The solution, as mentioned before, is to include a connectivity listener to keep the app informed of the network status in real-time. This way, the app can provide informative messages that users can understand

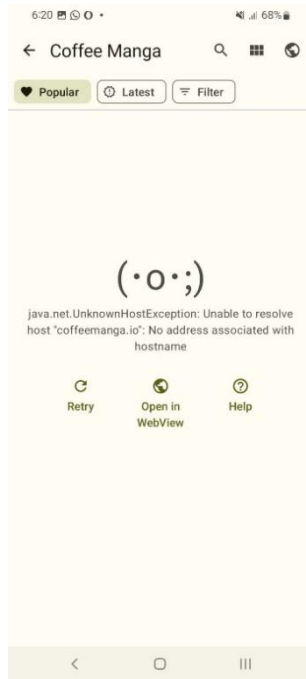


Figure 9.



Figure 10



Figure 11

- *Anti-Pattern 4: Lost Content*
- The final anti-pattern occurs in situations with poor network connectivity, as depicted in Figure 12. Ideally, each container should display an image, but due to the unreliable connection, they remain empty. Moreover, there are no placeholder elements, such as loading skeletons, to indicate to the user that the images are in the process of loading. The solution is straightforward: introduce images or animations in each container to signal that content is loading. This will improve the user experience and prevent the loss of content (Figure 11).

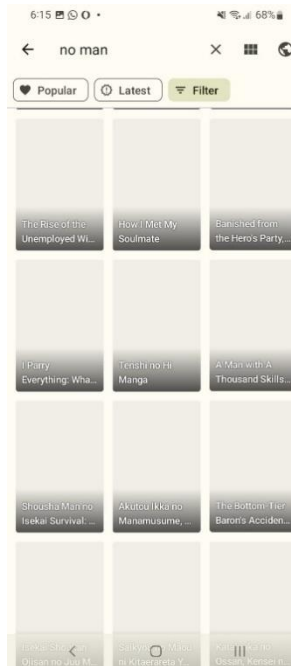


Figure 12

3. Identify and evaluate Caching Strategies of the third-party app that you are analyzing.

Tachiyomi is an Android application that prioritizes storage data and can work completely without an internet connection. It's possible to import manga from local storage, which makes it convenient for users who want to read manga offline. For this reason, Tachiyomi's caching strategy is **offline first**. The app uses a simple caching mechanism to store manga in memory, which is cleared when the app is closed or when the device runs low on memory.

Although Tachiyomi can work completely offline, the normal use of the app involves downloading manga from third-party sources available within the app. These sources are community-driven and provide a wide range of manga titles for users to choose from.

Depending on the type of data Tachiyomi use to different strategies for caching:

- *Cache, falling back to network*
When it comes to manga images, the following strategy is employed: if the image is already present in the cache or stored in permanent local storage, the app refrains from making a download request (Figure 13, 14). Only when the image hasn't been downloaded does the app request the data from the network (Figure 15).

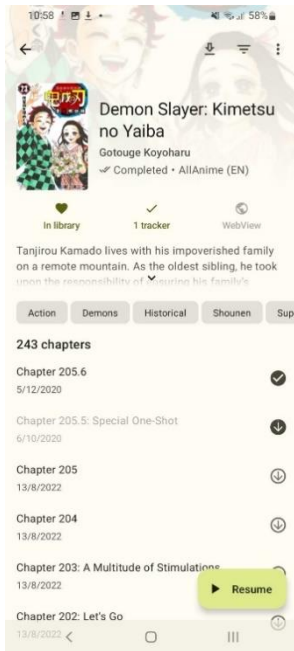


Figure 13

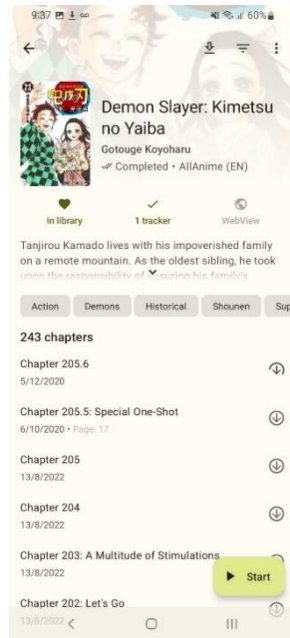


Figure 14



Figure 15

In this scenario, the file responsible for defining the cache can be found at `app/src/main/java/eu/kanade/tachiyomi/data/cache/ChapterCache.kt`. This file retrieves information from the JSON files created for each image in a manga. Within this class, you can find the implementation of the function `getPageListFromCache()` (Figure 16). When a page is required, the application employs the `HttpPageLoader` class. However, as illustrated in Figure 17, this class first checks if the image is already cached and only retrieves it if it is not present (through the condition `if (!chapterCache.isImageInCache(imageUrl))`).

```
/**
 * Class used to create chapter cache
 * For each image in a chapter a file is created
 * For each chapter a Json list is created and converted to a file.
 * The files are in format *md5key*.0
 *
 * @param context the application context.
 */
class ChapterCache(private val context: Context) {

    /**
     * Get page list from cache.
     *
     * @param chapter the chapter.
     * @return the list of pages.
     */
    fun getPageListFromCache(chapter: Chapter): List<Page> {
        // Get the key for the chapter.
        val key = DiskUtil.hashKeyForDisk(getKey(chapter))

        // Convert JSON string to list of objects. Throws an exception if snapshot is null
        return diskCache.get(key).use {
            json.decodeFromJsonString(it.getString(0))
        }
    }
}
```

Figure 16

```
/**
 * Loader used to load chapters from an online source.
 */
internal class HttpPageLoader(
    private val chapter: ReaderChapter,
    private val source: HttpSource,
    private val chapterCache: ChapterCache = Injekt.get(),
) : PageLoader() {

    /**
     * Loads the page, retrieving the image URL and downloading the image if necessary.
     * Downloaded images are stored in the chapter cache.
     *
     * @param page the page whose source image has to be downloaded.
     */
    private suspend fun internalLoadPage(page: ReaderPage) {
        try {
            if (page.imageUrl.isNullOrEmpty()) {
                page.status = Page.State.LOAD_PAGE
                page.imageUrl = source.getImageUrl(page)
            }
            val imageUrl = page.imageUrl!!

            if (!chapterCache.isImageInCache(imageUrl)) {
                page.status = Page.State.DOWNLOAD_IMAGE
                val imageResponse = source.getImage(page)
                chapterCache.putImageToCache(imageUrl, imageResponse)
            }

            page.stream = { chapterCache.getImageFile(imageUrl).InputStream() }
            page.status = Page.State.READY
        } catch (e: Throwable) {
            page.status = Page.State.ERROR
            if (e is CancellationException) {
                throw e
            }
        }
    }
}
```

Figure 17

- *Cache then network:*

A distinct strategy is applied when handling cover images. The application initially checks for the existence of cached cover images. If cached images are found, they are immediately displayed to provide a swift user experience. Simultaneously, the app downloads the cover image in the background. This approach ensures that users receive images quickly, while also enabling the application to reflect any server-side changes. Figure 18 illustrates this process, showing the initial image being replaced with an updated cover image to reflect changes on the server.

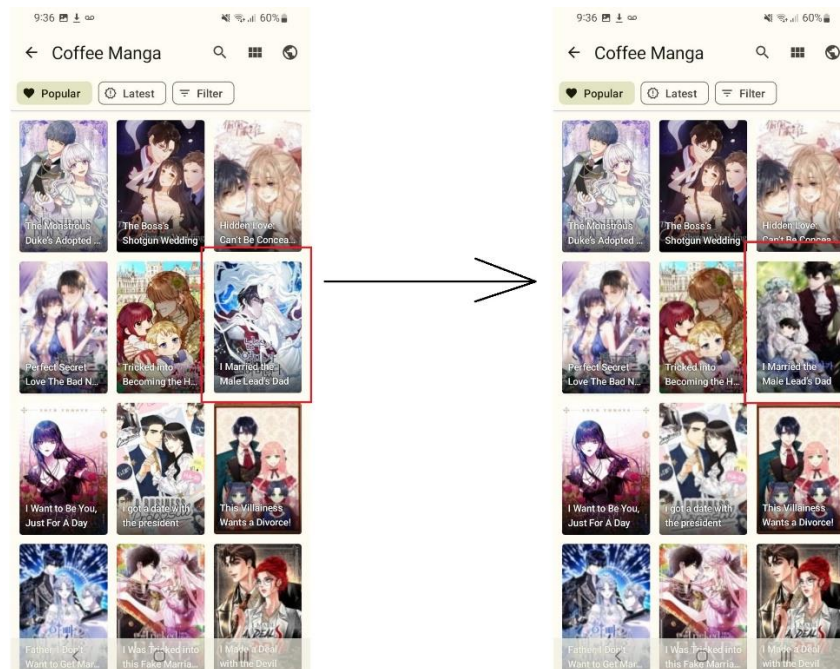


Figure 18

If we examine the file responsible for caching cover images, it is located at `app/src/main/java/eu/kanade/tachiyomi/data/cache/CoverCache.kt`. Within this file, we can find the method for removing cover images, which are stored in a dedicated folder within the cache storage directory. Figure 20 highlights that the cover image cache utilizes an LRU (Last Recently Used) protocol to manage cache when the allocated memory for caching is full. This cache management method is called from `MangaCoverFetcher.kt` on `CoverCache.kt`. The dependency used for this purpose is `com.jakewharton:disklrucache:2.0.2`.

```
fun Manga.removeCovers(coverCache: CoverCache = Injekt.get()): Manga {
    if (isLocal()) return this
    return if (coverCache.deleteFromCache(this, true) > 0) {
        return copy(coverLastModified = Date().time)
    } else {
        this
    }
}
```

Figure 19

```

class CoverCache(private val context: Context) {
    ...
    /**
     * Delete the cover files of the manga from the cache.
     *
     * @param manga the manga.
     * @param deleteCustomCover whether the custom cover should be deleted.
     * @return number of files that were deleted.
     */
    fun deleteFromCache(manga: Manga, deleteCustomCover: Boolean = false): Int {
        var deleted = 0

        getCoverFile(manga.thumbnailUrl)?.let {
            if (it.exists() && it.delete()) ++deleted
        }

        if (deleteCustomCover) {
            if (deleteCustomCover(manga.id)) ++deleted
        }

        return deleted
    }
    ...
}

```

Figure 20

Tachiyomi sets a fixed maximum storage cache size. We can find this policy on ChapterCache.kt. The maximum amount is 100MB (100*1024*1024bytes).

```

/** Application cache version. */
private const val PARAMETER_APP_VERSION = 1

/** The number of values per cache entry. Must be positive. */
private const val PARAMETER_VALUE_COUNT = 1

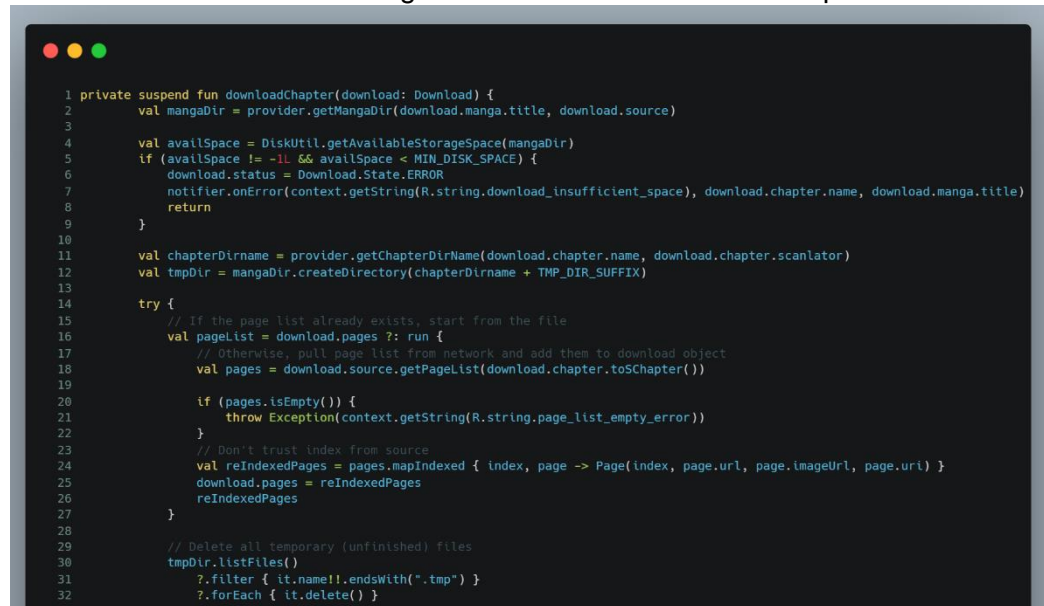
/** The maximum number of bytes this cache should use to store. */
private const val PARAMETER_CACHE_SIZE = 100L * 1024 * 1024

```

4. Identify and evaluate Memory Management Strategies of the third-party app you are analyzing

- **App-specific files:** Tachiyomi's approach to storing data involves creating dedicated directories for efficient organization. When users download manga chapters, the app checks the available disk space. If it's insufficient, an error message is triggered, halting the process. Conversely, if there's enough space, the app defines a directory for storage and initiates the download. To prevent redundant downloads, it checks for existing files. If a chapter is missing on disk, the app fetches it from the network and ensures the retrieved file isn't empty before saving it. Any temporary or incomplete files generated during this process are promptly removed.

In the code snippet (Tachiyomi/data/download/Downloader.kt), the process of checking disk space availability and handling insufficient space is demonstrated, offering insights into alternate strategies when disk space is lacking.



```
1 private suspend fun downloadChapter(download: Download) {
2     val mangaDir = provider.getMangaDir(download.manga.title, download.source)
3
4     val availSpace = DiskUtil.getAvailableStorageSpace(mangaDir)
5     if (availSpace != -1L && availSpace < MIN_DISK_SPACE) {
6         download.status = Download.State.ERROR
7         notifier.onError(context.getString(R.string.download_insufficient_space), download.chapter.name, download.manga.title)
8         return
9     }
10
11     val chapterDirname = provider.getChapterDirName(download.chapter.name, download.chapter.scanlator)
12     val tmpDir = mangaDir.createDirectory(chapterDirname + TMP_DIR_SUFFIX)
13
14     try {
15         // If the page list already exists, start from the file
16         val pageList = download.pages ?: run {
17             // Otherwise, pull page list from network and add them to download object
18             val pages = download.source.getPageList(download.chapter.toSChapter())
19
20             if (pages.isEmpty()) {
21                 throw Exception(context.getString(R.string.page_list_empty_error))
22             }
23             // Don't trust index from source
24             val reIndexedPages = pages.mapIndexed { index, page -> Page(index, page.url, page.imageUrl, page.url) }
25             download.pages = reIndexedPages
26             reIndexedPages
27         }
28
29         // Delete all temporary (unfinished) files
30         tmpDir.listFiles()
31             ?.filter { it.name!!.endsWith(".tmp") }
32             ?.forEach { it.delete() }
```

Code snippet of manga chapters download process (*tachiyomi/data/download/Downloader.kt*)

The code snippet, specifically lines 4-9, illustrates Tachiyomi's initial step before downloading a chapter: checking available disk space. This crucial check determines whether the user can proceed with downloading the chapter. This disk space management has a direct impact on memory usage. When it's impossible to save information to the disk due to space constraints, Tachiyomi implements a strategy to load the content directly into memory via the internet, bypassing the disk storage altogether.

- **Caching Images:** While Tachiyomi incorporates the Coil library for image caching, it deliberately opts to manage caching policies directly. This involves disabling Coil's built-in caching policies within specific classes such as BaseUpdatesGridGlanceWidget.kt,

SaveImageNotifier.kt, MangaCoverDialog.kt, and ReaderPageImageView.kt.

```
val request = ImageRequest.Builder(context)
    .data(data)
    .memoryCachePolicy(CachePolicy.DISABLED)
    .diskCachePolicy(CachePolicy.DISABLED)
    .target(
        onSuccess = { result ->
            setImageDrawable(result)
            (result as? Animatable)?.start()
            isVisible = true
            this@ReaderPageImageView.onImageLoaded()
        },
        onError = {
            this@ReaderPageImageView.onImageLoadError()
        },
    )
    .crossfade(false)
    .build()
```

Code snippet of request in BaseUpdatesGridGlanceWidget.kt

These snippets serve as a demonstration of how Tachiyomi chooses to utilize the Coil API while intentionally deactivating Coil's memory and disk caching policies to have greater control over image caching mechanisms. The approach to configuring Coil can be observed in several classes: BaseUpdatesGridGlanceWidget.kt, SaveImageNotifier.kt, MangaCoverDialog.kt, and ReaderPageImageView.kt. These classes demonstrate how Tachiyomi customizes Coil's settings for image handling and caching.

- **Management of Weak References:** Tachiyomi does not implement in any files the WeakReference class of Kotlin introduced with the library kotlin.native.ref. Coil allows the use of weak reference but as seen in the caching images section they deactivate the memory management policies of coil, restricting as well the option for using Weak References in the images.

This absence of explicit utilization of Kotlin's WeakReference class is interesting, given its capacity to optimize memory usage by allowing the garbage collector to free memory when necessary without destroying the entire object.

- **General Caching Strategy:** Tachiyomi employs a unified caching strategy that integrates both memory and disk caching for various elements within chapters. For example, in ChapterCache.kt, the app implements an LRU (Least Recently Used) memory cache known as DiskLruCache. This strategy ensures that in cases where a requested element is not available in memory, Tachiyomi can retrieve it from the disk cache.

This approach, demonstrated in code housed within app/src/main/java/eu/kanade/tachiyomi/data/cache, showcases the app's sophisticated

caching strategy aiming to optimize performance by leveraging both memory and disk-based caching mechanisms.

```
private val diskCache = DiskLruCache.open(
    File(context.cacheDir, "chapter_disk_cache"),
    PARAMETER_APP_VERSION,
    PARAMETER_VALUE_COUNT,
    PARAMETER_CACHE_SIZE,
)

...
fun getPageListFromCache(chapter: Chapter): List<Page> {
    // Get the key for the chapter.
    val key = DiskUtil.hashKeyForDisk(getKey(chapter))

    // Convert JSON string to list of objects. Throws an exception if snapshot is null
    return diskCache.get(key).use {
        json.decodeFromString(it.getString(0))
    }
}
...
fun putPageListToCache(chapter: Chapter, pages: List<Page>) {
    // Convert list of pages to json string.
    val cachedValue = json.encodeToString(pages)

    // Initialize the editor (edits the values for an entry).
    var editor: DiskLruCache.Editor? = null

    try {
        // Get editor from md5 key.
        val key = DiskUtil.hashKeyForDisk(getKey(chapter))
        editor = diskCache.edit(key) ?: return

        // Write chapter urls to cache.
        editor.newOutputStream(0).sink().buffer().use {
            it.write(cachedValue.toByteArray())
            it.flush()
        }

        diskCache.flush()
        editor.commit()
        editor.abortUnlessCommitted()
    } catch (e: Exception) {
        logcat(LogPriority.WARN, e) { "Failed to put page list to cache" }
        // Ignore.
    } finally {
        editor?.abortUnlessCommitted()
    }
}
```

- **External storage:** Tachiyomi app requests permission for writing data in external storage, specifically to the Firebase service that the app uses. The `rememberPermissionState` method is imported from the Google permissions API to establish if the app has the required permissions to write on Firebase DB. This is used when the app saves the user's downloaded data (like manga chapters or settings) as a backup to be able to restore the information later.

```

1 object PermissionRequestHelper {
2
3     @Composable
4     fun requestStoragePermission() {
5         val permissionState = rememberPermissionState(permission = Manifest.permission.WRITE_EXTERNAL_STORAGE)
6         LaunchedEffect(Unit) {
7             permissionState.launchPermissionRequest()
8         }
9     }
10 }

```

Code snippet of writing in external storage request (presentation/permissions/PermissionRequestHelper.kt)

- Preferences:** The Preferences storage mechanism is used to restore the data to which the backup was made. In fact, this mechanism is used along with the External Storage in the same file (SettingsDataScreen.kt). First, the app requests permissions using the requestStoragePermission imported method from the PermissionRequestHelper.kt file. Then, it saves the backed-up information using the other methods in the file, one example is when it retrieves the created backup and returns it as a Preference object using the Preferences mechanism in line 14 of the below image.

```

1 @Composable
2 override fun getPreferences(): List<Preference> {
3     val backupPreferences = Injekt.get<BackupPreferences>()
4
5     PermissionRequestHelper.requestStoragePermission()
6
7     return listOf(
8         getBackupAndRestoreGroup(backupPreferences = backupPreferences),
9         getDataGroup(),
10    )
11 }
12
13 @Composable
14 private fun getCreateBackupPref(): Preference.PreferenceItem.TextPreference {
15     val scope = rememberCoroutineScope()
16     val context = LocalContext.current
17
18     var flag by rememberSaveable { mutableIntStateOf(0) }
19     val chooseBackupDir = rememberLauncherForActivityResult(
20         contract = ActivityResultContracts.CreateDocument("application/*"),
21     ) {
22         if (it != null) {
23             context.contentResolver.takePersistableUriPermission(
24                 it,
25                 Intent.FLAG_GRANT_READ_URI_PERMISSION or
26                 Intent.FLAG_GRANT_WRITE_URI_PERMISSION,
27             )
28             BackupCreateJob.startNow(context, it, flag)
29         }
30         flag = 0
31     }
32 }

```

Code snippet of user's data backup (presentation/more/settings/screen/SettingsDataScreen.kt)

Persistent data

- **App-specific files:** In the example of the manga chapters download, the data must be persisted since the app will ask for it later when one or many users want to download or have access to it from the app. Also, the information must be in a private space on disk, since other apps shouldn't have access to the data downloaded and saved by the Tachiyomi app.
- **External storage:** In the case of the external storage use for saving the download information or settings of the user, it would be better if that kind of data is transient, since asking permissions to write on external storage some information from the user might be insecure or even inefficient if the data that is retrieved from the user is too much to be saved.
- **Preferences:** As said earlier, in the case of the Preferences use for saving the download information or settings of the user, it would be better if that kind of data is transient, since writing and moving the user's data to external storage can be an insecure feature or even an inefficient one in cases when the data saved by the user is too much.

5. Identify and Evaluate threading/concurrency strategies of the third-party you are analyzing

Tachiyomi utilizes various threading and concurrency strategies to ensure user experience and smooth performance. Alike, the main strategies and practices employed by Tachiyomi are the following:

- **Asynchronous Operations:** Tachiyomi uses asynchronous programming techniques to handle background tasks. By using Kotlin Coroutines, Tachiyomi performs non-blocking operations, such as fetching data from external sources without freezing the user interface. This ensures that the user can navigate along the app and perform actions while the app fetches data in the background.
- **Multithreading:** Tachiyomi uses multithreading to manage concurrent tasks and processes. Allowing the application to execute multiple operations simultaneously, enhancing the apps performance and responsiveness. In addition, Tachiyomi employs multithreading to handle the loading of multiple images and for the fetching of the data in multiple sources.
- **Background Service Utilization:** Tachiyomi utilizes background services to handle tasks that require continuous processing, these are updating the feeds, checking for new chapters and to perform sync operations. By using background services Tachiyomi can do these tasks without affecting the users current activities or consuming a lot of resources.

Thread Pool:

- A. Tachiyomi makes use of various multi-threading and concurrency strategies to ensure smooth and efficient execution of tasks, particularly those related to network operations and search functionalities. One example of these are in the SearchScreenModel.kt file. This class primarily manages the search functionality within the app, encapsulating various operations used for the searching process of manga titles across different sources.

B. The following code figures highlight the implementation of these strategies in Tachiyomi:

```
private val coroutineDispatcher = Executors.newFixedThreadPool(5).asCoroutineDispatcher()
return go(f, seed, [])
}
```

Figure 21 - Thread Pool Creation

In the Figure 1, Tachiyomi creates a thread pool with a fixed capacity of 5 threads. The `coroutineDispatcher` facilitates the management of coroutines within the application, enabling concurrent processing of tasks within the specified thread pool.

```
searchJob = ioCoroutineScope.launch {
    sources.map { source ->
        async {
            if (state.value.items[source] != SearchItemResult.Loading) {
                return@async
            }

            try {
                val page = withContext(coroutineDispatcher) {
                    source.getSearchManga(1, query, source.getFilterList())
                }

                val titles = page.mangas.map {
                    networkToLocalManga.await(it.toDomainManga(source.id))
                }

                if (isActive) {
                    updateItem(source, SearchItemResult.Success(titles))
                }
            } catch (e: Exception) {
                if (isActive) {
                    updateItem(source, SearchItemResult.Error(e))
                }
            }
        }
    }
    .awaitAll()
}
```

Figure 22 - Search Function

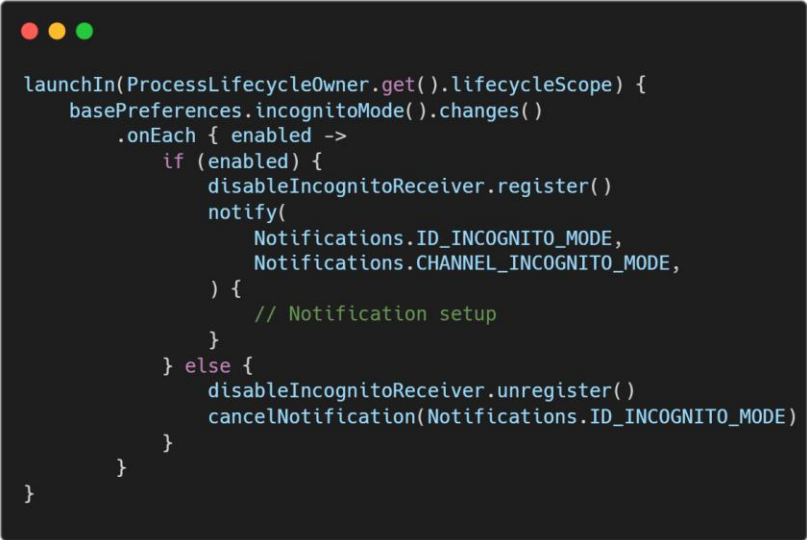
Taking into account the Figure 2. The `searchJob` coroutine in Tachiyomi's search function utilizes the thread pool to perform network operations asynchronously. The `async` function manages concurrent tasks within the thread pool, allowing the application to fetch search results from different sources concurrently. Additionally, it facilitates the conversion of

network results to local manga objects and updates the application state with the obtained results.

- C. The thread pool and concurrency strategies are primarily associated with the search functionality in Tachiyomi. These strategies enable the application to handle multiple network requests and processing tasks concurrently, ensuring a responsive user interface while fetching and managing data from various sources.

Asynchronous Processing:

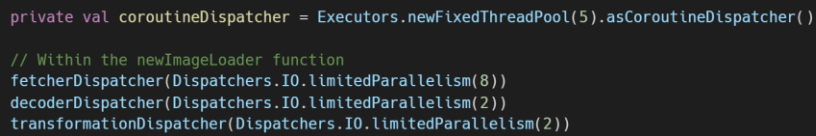
- A. Tachiyomi uses asynchronous programming models like Kotlin Coroutines to handle non-blocking operations. In the case of the app.kt file, the primary operations used are data fetching, image loading, and handling of notifications without freezing the UI.
- B. The following code figures highlight the implementation of these strategies in Tachiyomi:



```
launchIn(ProcessLifecycleOwner.get().lifecycleScope) {
    basePreferences.incognitoMode().changes()
        .onEach { enabled ->
            if (enabled) {
                disableIncognitoReceiver.register()
                notify(
                    Notifications.ID_INCOGNITO_MODE,
                    Notifications.CHANNEL_INCOGNITO_MODE,
                ) {
                    // Notification setup
                }
            } else {
                disableIncognitoReceiver.unregister()
                cancelNotification(Notifications.ID_INCOGNITO_MODE)
            }
        }
}
```

Figure 23 - Coroutine-based Concurrency

This figure demonstrates the use of coroutines to manage the incognito mode changes and corresponding notifications. It observes changes in the incognito mode preference and performs actions accordingly. When the incognito mode is enabled, it registers the 'disableIncognitoReceiver' and displays a notification to the user. When the incognito mode is disabled, it unregisters the receiver and cancels the notification.



```

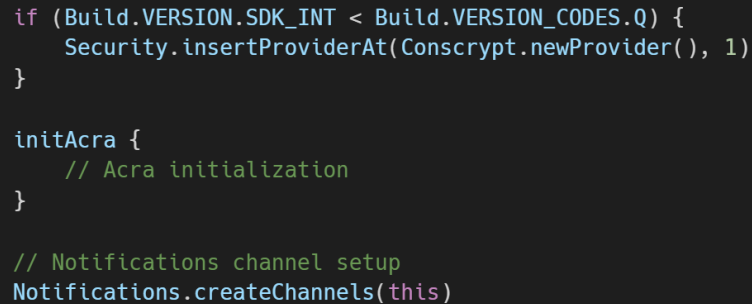
private val coroutineDispatcher = Executors.newFixedThreadPool(5).asCoroutineDispatcher()

// Within the newImageLoader function
fetcherDispatcher(Dispatchers.IO.limitedParallelism(8))
decoderDispatcher(Dispatchers.IO.limitedParallelism(2))
transformationDispatcher(Dispatchers.IO.limitedParallelism(2))

```

Figure 24 - Thread Pool Management

This figure establishes a fixed-size thread pool through the 'Executors.newFixedThreadPool' function, ensuring efficient management of concurrent tasks. Within the 'newImageLoader' function, it configures various dispatchers for tasks such as image fetching, decoding, and transformation. The limited parallelism ensures that these tasks are executed concurrently without overwhelming the system.



```

if (Build.VERSION.SDK_INT < Build.VERSION_CODES.Q) {
    Security.insertProviderAt(Conscrypt.newProvider(), 1)
}

initAcra {
    // Acra initialization
}

// Notifications channel setup
Notifications.createChannels(this)

```

Figure 25 - Asynchronous Network Operations

The Figure above involves setting up TLS 1.3 support for devices with Android versions earlier than Android Q. It ensures the insertion of the Conscrypt provider to enable TLS 1.3 support. Additionally, it initializes the crash reporting feature through the 'initAcra' function and configures the notification channels using the 'Notifications.createChannels' method.



```
setContentIntent(pendingIntent) // within the notification setup

// Setting up Coil ImageLoader
ImageLoader.Builder(this).apply {
    // ImageLoader setup
}
```

Figure 26 - Asynchronous UI Updates

This figure is responsible for setting the content intent within the notification setup, allowing users to interact with the notification content. It also sets up the 'ImageLoader' using Coil's 'ImageLoader.Builder', configuring the necessary parameters for efficient loading and displaying of images. The setup ensures smooth and responsive UI updates when handling image loading operations.

C. Functionalities associated with each strategy:

- **Coroutine-based Concurrency:** This strategy is associated with managing the application's lifecycle and handling asynchronous tasks related to user preferences, such as enabling/disabling the incognito mode and managing corresponding notifications.
- **Thread Pool Management:** The thread pool management strategy is primarily linked with image loading and handling tasks related to image fetchers and decoders. It ensures efficient and parallel processing of image loading operations without causing UI freezes.
- **Asynchronous Network Operations:** This strategy is associated with setting up TLS 1.3 support, initializing Acra for crash reporting, and configuring notification channels. These operations are crucial for maintaining the application's network connectivity and ensuring secure and reliable communication.
- **Asynchronous UI Updates:** The asynchronous UI updates strategy is closely tied to the loading and displaying of images using Coil's 'ImageLoader'. It enables the application to load and present images efficiently, contributing to a seamless and responsive user interface experience.

Audit Report

We, Daniel Bernal, Daniel Gómez and Juan Pablo Martinez, are part of a company in charge of analyzing Android mobile applications reporting any technical issues found for the source code. For this audit, we are going to give a free analysis of Tachiyomi. Tachiyomi is a comprehensive manga reading application that offers a rich set of features, extensive customization options, and a vibrant community, making it a popular choice for manga enthusiasts seeking a versatile and user-friendly reading experience. For the following analysis we are going to focus on four main aspects: Eventual Connectivity Strategies, Caching Strategies, Memory Management Strategies and Multi-threading Strategies.

Tachiyomi effectively leverages Kotlin coroutines to manage asynchronous tasks, prioritizing user preferences and lifecycle management. The application also implements a fixed-size thread pool to handle concurrent tasks, ensuring seamless execution of network operations and image loading without compromising the user interface's responsiveness. Additionally, Tachiyomi integrates various strategies to support asynchronous network operations, crash reporting initialization, and notification channel configuration. With a focus on delivering smooth UI updates, the application uses asynchronous processes for image loading and display, ultimately enhancing the overall user experience. To further enhance Tachiyomi's threading and concurrency strategies, it is recommended to conduct regular performance testing and profiling, implement comprehensive error handling and exception management, introduce additional monitoring and logging mechanisms, and stay updated with evolving best practices and technological advancements in the field.

Tachiyomi exhibits a robust overall connectivity strategy, effectively balancing offline functionality and online content access. The application's use of caching and fetching strategies is commendable, ensuring efficient data retrieval and a smooth user experience. In the case of manga images, Tachiyomi wisely adopts a cache-first approach, minimizing unnecessary network requests and providing users with quick access to locally stored data. This is a prudent choice, as it reduces data usage and enhances the application's performance. Furthermore, the application demonstrates a "cache then network" approach when handling cover images, ensuring users can access cached images while allowing for background downloads when cover images are updated. These strategies enhance the user experience by maintaining a delicate balance between data efficiency and content access speed.

Despite its overall strong connectivity design, Tachiyomi does suffer from certain connectivity anti-patterns that warrant attention. The most prominent issue is the occurrence of "Stuck Progress Notifications" (Anti-pattern #2). In cases where the application was initially launched with an internet connection and then the internet is deactivated, Tachiyomi wrongly assumes a continuous internet connection. As a result, the app gets stuck on a progress notification, incorrectly believing that it still has an active connection. This can lead to user frustration and a less-than-optimal experience. Furthermore, the "Non-informative Message" anti-pattern (Anti-pattern #3) occurs under similar circumstances. The application, in the absence of network connectivity at startup, displays uninformative error messages. These messages can confuse users and hinder their ability to understand and resolve the issue.

To improve Tachiyomi's connectivity and rectify these anti-patterns, the application can implement a listener to monitor real-time connectivity changes. This solution will allow Tachiyomi to respond effectively to shifts in connectivity status. By monitoring network status and providing informative messages when necessary, the app can offer a smoother and more intuitive user experience. Implementing such a feature would not only enhance user satisfaction but also align with best practices in connectivity handling.

In summary, while Tachiyomi's overall use of connectivity patterns is strong, there is room for improvement in addressing the identified anti-patterns related to connectivity handling. By adopting the suggested solution of implementing connectivity listeners and providing informative messages, Tachiyomi can further enhance its reliability and responsiveness to connectivity changes, ensuring a more seamless user experience.