Firefox Focus for Android

Team UTSGSC - Deliverable #1

Summary of Firefox Focus

Firefox Focus is an open source privacy based mobile browser for Android. The project was initially started by Mozilla and utilizes many of Mozilla's open source tools. While other popular mobile browsers such as Google Chrome do not allow you to block ads at all or require a plugin to enable this feature (similar to how the desktop version of Chrome works), Focus defaults to privacy and provides unique privacy features such as the default blocking of web trackers, ad blocking, as well as an easy way to erase a user's browsing history. In a world where we are constantly barraged by trackers and ads, Focus is a reliable alternative for users that want to browse the internet safely.

Overall Architecture

1. Architecture Overview

From the perspective of a developer, Focus is an application that ties two important mozilla libraries together:

a) **GeckoView library**: the Android GeckoView view component for rendering a web page
b) **Components library**: a set of components that form the basic building blocks of a browser such as SessionManager and SearchEngineManager

Focus is heavily reliant on both of these libraries as they abstract the majority of the difficulty associated with browsing. Focus wraps these two libraries to provide the browser end user experience that we're used to such as Tabs, Settings control, Privacy Control, etc.

2. Technical architecture - how does browsing on Focus work?

To start, let's discuss the application itself. The **FocusApplication** class acts as the host for all other components and is in charge of initializing a Components instance when the app launches. The Components class is an aggregate of two classes (SessionManager and SearchEngineManager) from the Components library discussed above.

a) SearchEngineManager

Not super important to the architecture of the app, but provides a clean interface for interacting with search engines as a developer such as setting the default one and getting a list of search engines.

b) **SessionManager**:
A session instance represents that state of a single browser session (think of each tab as a session). The SessionManager manages sessions for the Focus application by providing methods to add and remove sessions. It also leverages the observer pattern by defining an Observer interface with hooks for specific events:
    1. Session selection
    2. Session added
    3. Session restored
    4. Session removed,
    5. All sessions removed

Several classes need to be notified of changes to sessions. These include:

a) **BrowserFragment**: the fragment that displays the entire focus UI. Since focus has a view displaying the number of tabs, this class has to be notified when the session count changes, and will handle updating the UI appropriately.
b) **MainActivity**: the activity that hosts all the BrowserFragments. Whenever a session is selected (think tab switching), this activity will search through the list of BrowserFragments it hosts and find the one associated with a session that is selected or create a new one if it can't. It'll then place the fragment on tap of the view stack so it is the session that is currently being displayed

So what happens when a new tab is opened?
There are multiple ways to open an url in Focus (either from an external link in another application, a new tab, clicking a link that opens in a tab, etc) but they all involve creating a new session instance and adding it to the session manager.

Once added, the Session Manager will notify all its observers. The MainActivity class will look for the BrowserFragment associated with the session. Since this is a completely new session, it'll create a new BrowserFragment for it and place it on top of the view stack. The BrowserFragment class will then update the view to display the proper number of tabs.

Now what happens when the tab is closed?
The BrowserFragment will call its finish function to remove itself and remove the session from the SessionManager which will notify its observers. The MainActivity will only have to handle a

bit of cleanup as the next BrowserFragment on the viewstack will automatically be displayed when the current one is removed.

3. Concurrency and asynchronous design

Focus makes heavy use of Kotlin coroutines for asynchronous programming. To integrate coroutines with Android development, the FocusApplication class implements a CoroutineScope to create a coroutine scope tied directly to the lifetime of the Application. This ensures that any coroutines created in the scope of the application scope will never be orphaned (as is the case with threads). Similarly, other specific components such as the SearchSuggestionsFragment also implement the CoroutineScope interface so that any asynchronous operations are cancelled when the fragment is destroyed or paused.

4. Comments on Architecture

The app is packaged by feature so all the classes directly associated with a feature will belong to the specific feature package. This is generally a good idea as it ensures classes and packages have high cohesion.

The Components instance is a member of the FocusApplication class which allows any descendant activity and fragment to access it (and by extension, the session manager and search engine manager) through the context. Although the components class is not a singleton, it behaves like one because all other components obtain their context from the application.

5. Possible Improvements

Focus wraps the base Android classes (i.e., Application, AppCompatActivity, Fragment) in a custom locale layer (LocaleAwareApplication, LocaleAwareAppCompatActivity, LocaleAwareFragment) to enable localization features across these classes.

Naming schemes as well as the separation of concerns between classes could be improved. For example, although the classes are located within the “locale” layer, they are essentially base classes with additional functionality that isn’t properly represented by just the “locale” name. Instead, this package should be renamed to “base” and classes should be prefixed with “Focus”.

Also, it might not even make sense to have the base class for Application because there should only be a single Application class.

Team Software Development Process

The Software Development process we chose is the Incremental development model that is plan-driven and uses Kanban to manage our workflow.

Reasons why we chose this process
The reason for choosing this process is due to our various needs as it relates to the codebase, the type of work we will take on, and the team.

The following was our thought process when deciding on the appropriate process:

To start, we evaluated a bug for Mozilla Firefox Focus and we concluded that the bugs are well documented with reproduction steps, provided a good description, and acceptance criteria. We also do not expect these requirements to change much since the acceptance criteria is to restore previously defined functionality. In addition, these are detailed outputs of the Requirements Engineering phase which are required before moving onto the design and implementation activity. Once the requirements engineering phase is complete, we move onto the "Design and Implementation and Testing" activity where iterations occur. With this in mind, we started thinking about plan-driven.

Since we are fixing bugs, we wanted to also ensure rapid delivery and deployment of an implemented and tested fix as soon as it's ready. As a result of this, we also started to consider incremental development.

Lastly, since the codebase is very large, this is the first time this team is working together and we want high quality code, we need something that can provide a high level of transparency and ensure that we can identify bottlenecks. As a result, we chose Kanban due to its features like limited WIP, feedback loops, and Kanban board. The limited WIP is useful for ensuring that we don't take on too many tasks for a specific column so the team's focus is not divided and can produce high quality code. The feedback loops allow consistent and frequent short sync meetings between the team members to provide transparency and understanding into the current progress and work that each member has. Lasly, the Kanban board provides visual transparency into the team's progress and easily allows us to identify any bottlenecks and address them as soon as possible so we can work efficiently as a team.

Based on all of the decisions above, we decided to go with the incremental development model that is plan-driven and uses Kanban to manage our workflow.

Pros and Cons of Other Processes

Waterfall

- **Pro:** The idea of a plan-driven model of ensuring detailed outputs are at each stage such as requirements specification for the requirements engineering phase and a completed bug fix with an automated test suite is something we have adopted into our own process.
- **Con:** Waterfall is plan-driven with the process of having separate phases of "Implementation and Unit Testing" and "Systems and Integration Testing" which wasn't appropriate for us if we wanted to deliver a bug as soon as it was completed. This rigidness prevented us from adopting this model.

Extreme Programming

- **Pro:** Practices from XP such as Test-Driven Development (TDD) are important for ensuring there is good test coverage and that developers don't forget or take shortcuts after implementation in testing code. As a result, we will adopt TDD when fixing new bugs by first writing a test to reproduce the bug before implementing the fix.
- **Con:** XP aims to produce high quality code with very strict engineering practices. This includes TDD and pair programming. Consistently doing pair programming is realistically too rigid and based on the nature of the team member's schedules and we cannot effectively work in this manner.

As we can see, there are both positives and negatives to Waterfall and XP as it pertains to our project. We have customised our own process by taking the pros for the Waterfall and XP.