Deliverable #3

Team 10 - UTSGSC

Potential Features

1. **Consider implementing long press on back button to open back stack**
    - **Link:** https://github.com/mozilla-mobile/focus-android/issues/3845
    - **UML:** The UML diagram that highlights the changes needed to implement this feature is **deliverable3/issue_3845.png** in our team repo. The blue components are the relevant ones

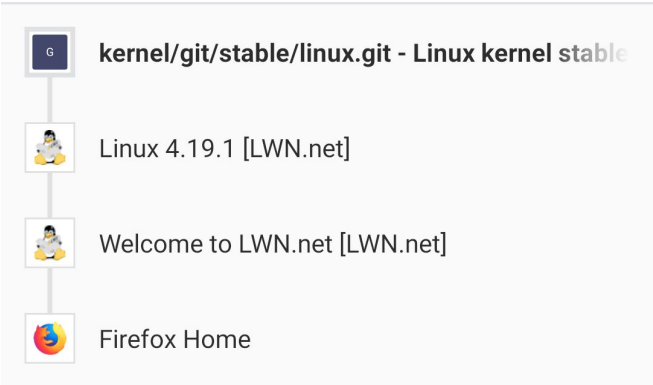
DESCRIPTION:

Problem: Sometimes users want to go back to a page they visited that is **not** the last page, which requires them to hit the back button multiple times until they arrive at their desired page. Some devices are slow and have limited bandwidth which makes this process slow and costly.

Solution: When the user long presses the back button, this will open up a UI fragment that displays their recent history. The user can then tap on any of those links to bring them back to a previous page, without having to load every page in between.

Design for feature

This is currently how Fennec does this feature which should be similar to Focus to maintain consistency. For each row, the left shows an icon for the website and the right is the title of the website.



SPIKE:

Preliminary research

This includes our findings from the Spike investigations.

Since android 2.0, there has been a listener for the event of a long hold of buttons. The method is called onKeyLongPress, and is part of the Activity class.

Therefore for our implementation, we can override this method in the MainActivity and add our desired functionality to it, by opening a RecyclerView that will display the past histories in a list-style format.

As explained earlier, each Geckoview holds a reference to the GeckoSession it's associated with. The GeckoViewProvider (a class defined within Focus) uses the factory pattern to generate new instances of GeckoSessions. GeckoSessions uses the delegation pattern to allow various classes to access different, specific functionality. For example, to access history data and hook into its functionality for a GeckoSession instance, a History delegate has to be supplied by the GeckoViewProvider. Delegation also relies on abstraction by communicating with delegates through the delegate interfaces.

Implementation research

Since we will need to display a RecyclerView as well as display the past history of viewed sites in the correct order, we will have to get the session history from the GeckoSession inside Firefox's GeckoView. Specifically, there is a component inside GeckoSession that is called HistoryList, as well as HistoryItem that will prove useful mapping the history tree to our RecyclerView display.

To keep this functionality consistent with what is already available on other Firefox browsers, when clicking an item in the RecyclerView, it will go to the correct website **without** updating the history, keeping the history still in the order of how it was before going back to one of the past sites visited. When clicking on an item in the intermediate sites, or entering a new URL, the new directed site will take the top position of the RecyclerView, removing the previous future history from the current site.

Acceptance criteria:

Ensure the existing MainActivity is used to handle our desired functionality for this feature. This includes a RecyclerView displayed on a popup box on a long hold of the back button, and when clicking those individual items it will go to the correct chosen site. Order of the history will be as explained previously in the implementation research section.

Exploration of solution:

We currently have 3 different WebView providers that implement the WebViewProvider interface

1. WebViewProvider
    - Just wraps the **GeckoWebViewProvider**
2. GeckoWebViewProvider
    - provides **GeckoWebView**
3. ClassicWebViewProvider
    - provides **SystemWebView** instances
    - Only used for the **InfoActivity**

From this, we can see that focus uses the GeckoViewProvider to create GeckoWebViews for the browser. The tricky part is figuring out how to expose the history information for the session associated with each WebView.

Back press actions are propagated downwards through:

1. MainActivity:
    a. **onBackPressed** will check if a BrowserFragment is visible
2. BrowserFragment

a. **onBackPressed** access the current IWebView instance and calls its canGoBack method to see if we can go back in history
b. **goBack** access the current IWebView and calls its goBack method

Specifically, since the browser uses GeckoViews, the goBackMethod will use the go back method for the existing geckoview which will use the goBack method of the GeckoSession tied to it.

The issue is that IWebView doesn't currently expose the history for a session. Although the GeckoWebView has a GeckoSession which exposes the history, we currently operate on the IWebView interface which doesn't.

Solutions:

1. **Expose the history in the IWebView interface**
   We would also have to create a history class and a method in the interface for accessing the history as a list. Both WebViews would have to adapt their own history data to history class instances in their implementation of the class

2. **Check specifically for the GeckoWebView**
   We can cast the current IWebView as a GeckoWebView in and create a GeckoSession bundle instance to pass to the HistoryFragment when we create it. Then we can access the history from within the History fragment and render it appropriately

   I definitely prefer this approach as the BrowserFragment will only be using GeckoWebViews and the implementation of the history fragment should only be available to fragments that use GeckoViews. Moreover, it means we don't have to introduce unnecessary complexity and adapters to the WebView class which will cause it to get bloated and force us to implement the feature for the **SystemWebView** class which will never get used.

   It's important to note that we can't access the session from the session manager because it gets the selected session from its delegate which is a LegacySessionManager providing instances of the session class defined in Focus, not instances of GeckoSession.

   Our UML diagram in our repository (deliverable3/issue_8345.png) illustrates this solution.

2. Consider search shortcuts in the URL bar

- **Link:** https://github.com/mozilla-mobile/focus-android/issues/3808
- **UML:** Diagram is in the same folder, blue components are the relevant ones

DESCRIPTION: Allow users to use built-in shortcuts for search similar to desktop Firefox, where prefixing the search with "@" will show the website shortcut options. After selecting the option, searching will take the user directly to the website's search page showing the results of the query.

SPIKE:

Preliminary research

Looking at the current primary firefox browser, it looks like it already implements this functionality, albeit a bit differently. When the query field is non-empty, the bottom of the search options will show the websites that when clicked, will produce the behaviour we're looking for.

Looking at it further, this might be preferable to the implementation described by the issue

- Allows the user to make use of this feature without being forced to type the "@" prefix
- Shows search hints for normal search

Implementation research:

Ideally, we should look to split this feature into 2 PR's for implementing the base functionality:

1. Add functionality for searching each extension

This would involve individually looking at the search page for each website and analyzing how we can perform the search by constructing the url manually with the query. On Firefox, the amazon shortcut for query "ipad mini" takes us to this url:

https://www.amazon.com/s?k=iPad+mini&sourceid=Mozilla-search&tag=mozilla-20

We can break this url down into parts

- "https://www.amazon.com/s?": this is the base url for the search page
- "k=iPad+mini": this is the part we need to care about since we need to modify it based on our query. Notice we might need to worry about special characters like "+" since it is used as a space. Needs further investigation
- Looks like "+" gets converted into %2B, so we may need to investigate how to do the conversion by looking at Firefox since it already handles this
- "sourceid=Mozilla-search&tag=mozilla-20": we shouldn't have to worry about this but it would be nice to get some feedback on the feature page to see whether we should have a different tag

We should implement this for AMAZON first as a proof of concept for this PR, since this feature should represent the foundation needed to create additional search shortcuts.

Acceptance criteria:

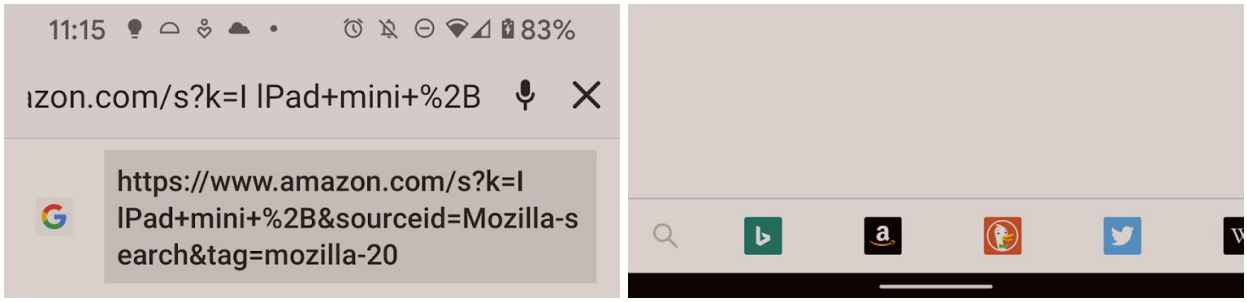
We would need unit tests to ensure:

a. Sessions with the correct attributes are created
b. Urls are constructed properly

2. Add the UI for the shortcut

The UI for the shortcut should be implemented after the functionality for searching each extension is implemented. This PR should simply add the shortcut bar at the bottom of the search bar and hook into that functionality when the user presses the shortcut.

Top of screen with search query (left picture) and bottom of screen with search shortcuts (right picture)



Since we're taking the UI from the Firefox browser, we can see if we can reuse its components.

For now, I'm unable to understand the main Firefox code in the mozilla central repo as it uses a mix of JS and rust to compile the android code. So we should first focus on how we should implement the UI for this component natively in Android.

This shortcut bar component is a horizontal view that contains multiple square ImageViews which can. The search shortcuts aren't selected by the user, so this order should be maintained for the search shortcut icons (ie. Search icon, Bing, Amazon, etc).

Since we don't expect there to be too many shortcut items, the scrolling part of the shortcut bar can be implemented using a ScrollView with an adapter that renders each shortcut item as a view and registers an onclick for them.

The first button in Firefox currently takes users to the settings associated with the search shortcuts. We should not implement this yet as this would be considered another feature! So for now, our search bar should only show the search shortcut icons.

Chosen Feature

Consider search shortcuts in the URL bar

Issue we ultimately decided on:

We chose to implement feature #2 - "Consider search shortcuts in the URL bar". The reason for this feature was because it is a helpful feature to many users, and also a common feature already available in most other web browsers. With this implementation users will be able to search quickly on Firefox Focus in a common search engine of their choice.

We ultimately decided on the feature #2 - "Consider implementing long press on back button to open the back stack" - because of difficulties encountered when prototyping the first issue. Focus currently uses GeckoSessions that are special session instances tied to each GeckoView. Each session delegates the responsibility of keeping track of its history to a HistoryDelegate which needs to be provided upon creation.

Although it is feasible to create a custom history delegate within the GeckoWebViewProvider and add it to the sessions it creates, we'd have to create a way to associate the history list with each session. This in itself is fairly easy, but creates additional scenarios where we'd have to also register another handler for checking when sessions are removed so that we don't hold the history in memory any longer than we need. Moreover, the act of adding another layer specifically for managing history is something we're hesitant to do as a team because it conflicts with the goal of Focus which is to be a minimal and privacy first browser.

We also felt that this feature was narrower in scope to feature #2. We wanted to make a contribution that would benefit more users.

Detailed Implementation Plans

1. Logic

For the logic, we need to leverage the existing SearchEngine and SearchEngineManager classes which are currently being used to set and manage the default Search Engine in the Settings menu. With this we should be able to provide a search shortcut for search engines such as, duckduckgo, google, amazon, twitter, wikipedia, and any additional search engine that the user adds in the search engine settings. The SearchEngine class exposes functions that will help build a search url using a specific search engine. This could be used in combination of some search query manipulation logic that would help decide if the user has searched using a search engine shortcut, and if so which search engine the user desires.

We also need to check when the user types an "@" symbol in the search bar so that they could immediately see all the possible search engines that they can use. To do this we would need to add to the current logic present in the UrlInputFragment. UrlInputFragment currently has a listener for the search bar that renders all the possible suggestions based on the user's current input. Logic to check for an "@" symbol can be added here.

When the users completes a search by pressing Done on the keyboard, the input can be parsed to check if it is a search shortcut query with an "@" symbol, and if so the query can be mapped to one of the appropriate search engines that is currently supported by the SearchEngineManager.

2. UI

Currently, search suggestions are rendered in the SearchSuggestionsFragment using a RecyclerView with a custom viewholder (SuggestionViewHolder) which is responsible for rendering the view for a search suggestion. Each search suggestion already has a "search" icon on the left side of the view followed by the actual text of the suggestion.

Since we plan to add the search engine shortcuts as search suggestions, we just need to render each the correct icon associated with each search engine instead of the "search" icon.

a. **Check which search suggestions are search shortcuts**: We can add a static function in the URIUtils class to check if the suggestion text matches a search engine and
b. **Figure out how to render an icon for each search shortcut**
Looking at the existing SearchEngine class used in URIUtils, we see that each instance contains a bitmap, so we can add another static function in URIUtils that the SuggestionViewHolder can use to get the search engine instance for a suggestion

When we have both of these implemented, we can simply get the bitmap icon of the icon for any search shortcut being rendered and create a BitmapDrawable from it to replace the search shortcut image to obtain the UI we're looking for

Acceptance Tests For Selected Feature

Complete these steps before any running acceptance test case:

1. Clone the repo https://github.com/CSCD01-team10/focus-android and follow the build instructions
2. Open Firefox Focus application
3. Skip all onboarding menus (if this is a fresh install)

Test 1: Check '@' sign displays all search engines available when search suggestions is turned on
Steps:

1. Tap the 3 vertical dots at top right corner
2. Tap 'Settings'
3. Tap 'Search'
4. Turn on 'Get search suggestions'
5. Go back to main screen by tapping back twice
6. In search box enter '@'

Expected Result: A list of all available search engines should be shown, which are amazon, google, duckduckgo, twitter, and wikipedia. Each item should have an icon correlating with the search engine.

Test 2: Check choosing search engine suggestion shows the user that they are going to search with the specified engine.
Steps:

1. Tap the 3 vertical dots at top right corner
2. Tap 'Settings'
3. Tap 'Search'
4. Turn on 'Get search suggestions'
5. Go back to main screen by tapping back twice
6. In search box enter '@amazon books'

Expected Result: User should be shown a bar that says “Books - Search with Amazon.ca”

Test 3: Check ‘@’ sign does not display all search engines available when search suggestions is turned off

Steps:

1. Tap the 3 vertical dots at top right corner
2. Tap ‘Settings’
3. Tap ‘Search’
4. Turn off ‘Get search suggestions’
5. Go back to main screen by tapping back twice
6. In search box enter ‘@’

Expected Result: A list of search engines should not be shown as suggestions.

Test 4: Check using search shortcut searches the correct search engine correctly

Steps:

1. In search box enter ‘@duckduckgo test search’
2. Tap enter button

Expected Result: Browser should direct to duckduckgo searching for ‘test search’.

Test 5: Check searching without shortcut still searches default search engine

Steps:

1. In search box enter ‘covid-19’
2. Tap enter button

Expected Result: Browser should direct to google searching for ‘covid-19’.

Test 6: Check entering search engine shortcut without search query should search shortcut in default search engine

Steps:

1. In search box enter ‘@twitter’
2. Tap enter button

Expected Result: Browser should direct to google searching for ‘@twitter’.

Architecture Document

A UML representing the new overall architecture is **deliverable3/overallArchitecture.png** in our team repo.

Since we’ve spent a lot of time researching history in depth for the feature spikes, there’s a lot more specific pieces of architecture information we can discuss.

Focus uses an interesting layered architectural pattern with a heavy emphasis on reuse. Firefox has created a clear separation between a lot of the components involved in the creation of a web browser, the most important of which is the Geckoview. It contains:

1. GeckoSession
2. GeckoWebView
3. GeckoView

Focus has an open layered architecture with two distinct layers where the higher layers can access any layers below it (where the first item in the list is the lowest).

1. Persistence
2. Business
3. Presentation

All three classes from the GeckoView package represent the core of the business layer of Focus and abstract the bulk of the complexity for rendering and sessions away. Because of the unique approach of this architecture, the rest of the business layer acts as a glue layer to allow the presentation layer to leverage the existing packages.

The presentation layer for Focus consists of Activities, Fragments, and ViewModels and uses Model View ViewModel for its presentation architecture. It's a more decoupled version of MVC that maintains the separation between the Model, View, and ViewModel. The View communicates and subscribes to events from the ViewModel whereas the ViewModel manipulates the Model to notify the View. This allows the ViewModel which contains the logic used for presentation to be entirely independent of the view. Not only does this make it easier to work with, it also makes it much easier to test! It doesn't need to be aware of Android components which means it can be tested with JVM tests instead of Android tests. This is a standard architecture for Android applications and it's no surprise that Focus also follows it.

The presentation layer also relies heavily on components from lower layers such as the GeckoWebView and GeckoView for rendering content.

It should be noted that focus doesn't have a relevant persistence layer because it doesn't keep track of history outside of each individual session. Fenix (which is the other mobile browser leveraging the same components - albeit with different goals - actually has a persistence layer. It uses a custom HistoryHandlerDelegate and HistoryDelegate which is supplied to the EngineProvider and allows the history to be recorded and persisted in storage. Focus only keeps track of a few settings and preferences in its persistence layer.

More interesting architecture can be found in the spikes for each issue because we invested a lot of time into investigating each one