TabX

Khayyon Parker, Miguel Guerrero,
Mishig Davaadorj, Nicole Woch, & Russell Skorina

Overview

TabX is a writing assistant program that makes typing in a browser faster and easier. It is an accessibility tool that functions as a Google Chrome extension. The client is a public library which chooses to invest in the project so that the application can serve as an accessibility tool for its patrons. Given that the library is publically funded, its priority is not pulling in the greatest profit, but rather improving the overall experience of its patrons, many of whom are not very familiar with technology and are slow typers.

The people who often frequent the library either do not have regular access to a computer at home or are seeking information online, but are not sure where or how to access it -- potentially due to lack of experience or because they are not native English speakers. The library wants a product which will help to more efficiently fulfill such people's needs.

The library is also interested in minimizing the amount the amount of time librarians need to invest in assisting patrons with navigating web pages and finding resources, as some of the most frequently asked questions are related to these technological concerns. In addition, the library wants the finished product to be as close to universal as possible; it should be able to function on almost any web page, and especially the most popular ones frequented by its patrons.

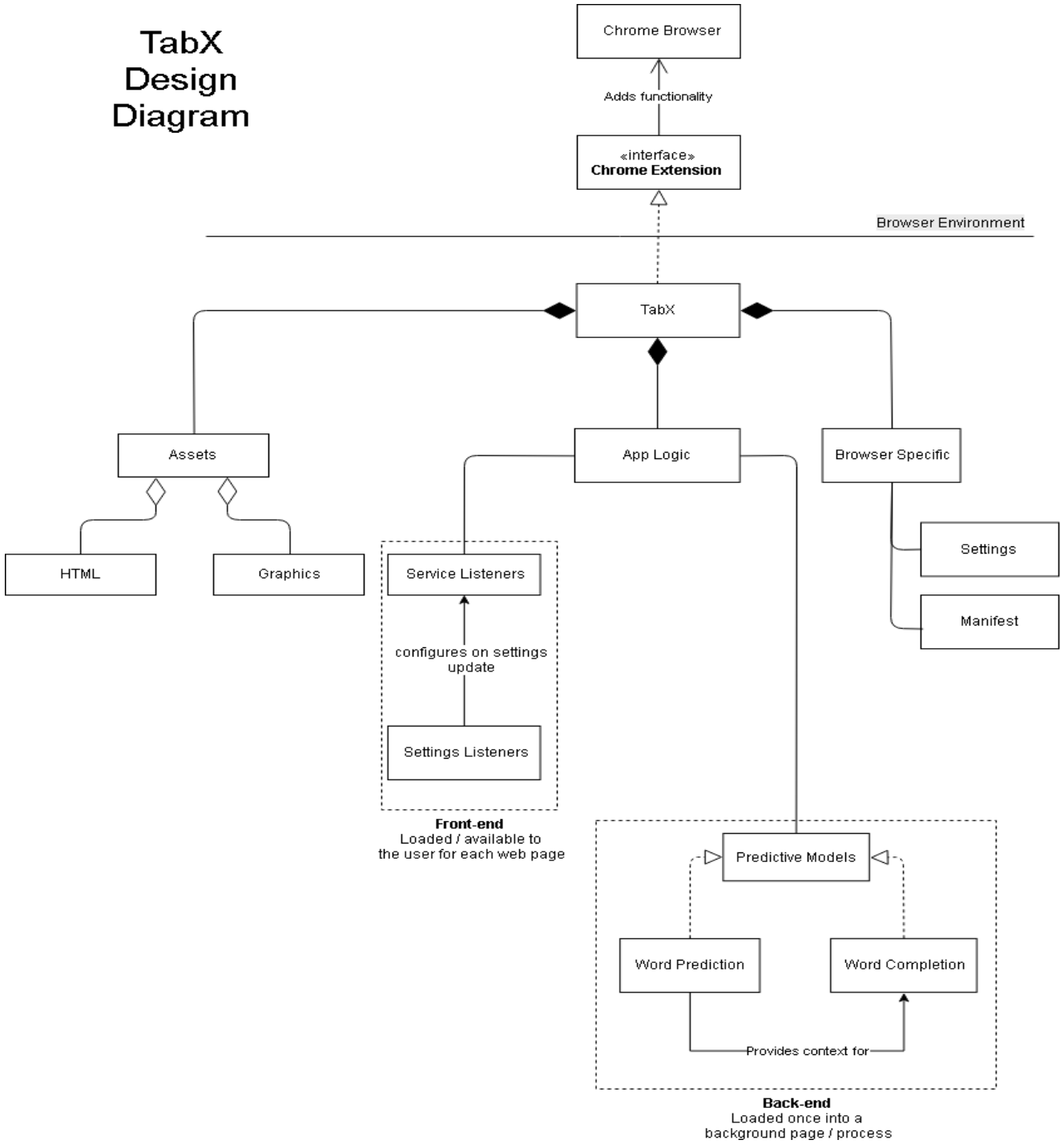
Given the intent of the product to be an accessibility tool for patrons, user-friendliness is a priority for the library over speed and efficiency; so long as the application is functional, provides useful suggestions, and is efficient to the extent to which users do not experience noticeable delays in receiving suggestions when typing, seamless functionality of the user interface is most important.

System Architecture

There are two major considerations that went into TabX's design:

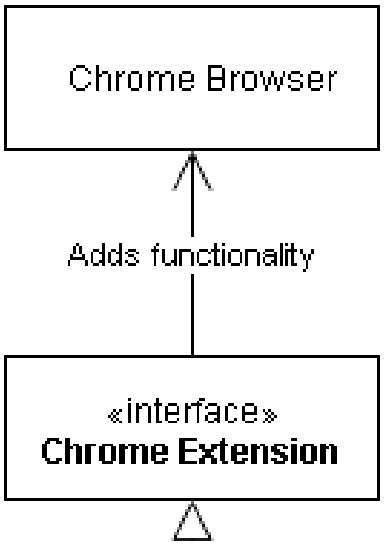
1. Meeting the client's requirements in providing several writing assistant features
2. Abiding by Chrome's extension interfaces and API

As a result, TabX is being designed such that it allows developers to add new features such was word completion and word prediction without too much modification to the code and allows TabX to operate within a browser environment. This was done by separating the concerns of the application into small subsystems that are responsible for one task.



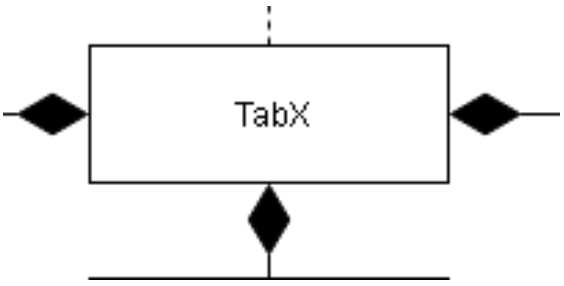
Subsystems

Chrome Extension Interface



By abiding by this interface, TabX "contractually" agreed to follow Chrome's specifications. This puts technical constraints on the rest of TabX's subsystem. This mainly includes that code must run in a browser environment and follows standard security protocols (e.g. CORS).

TabX



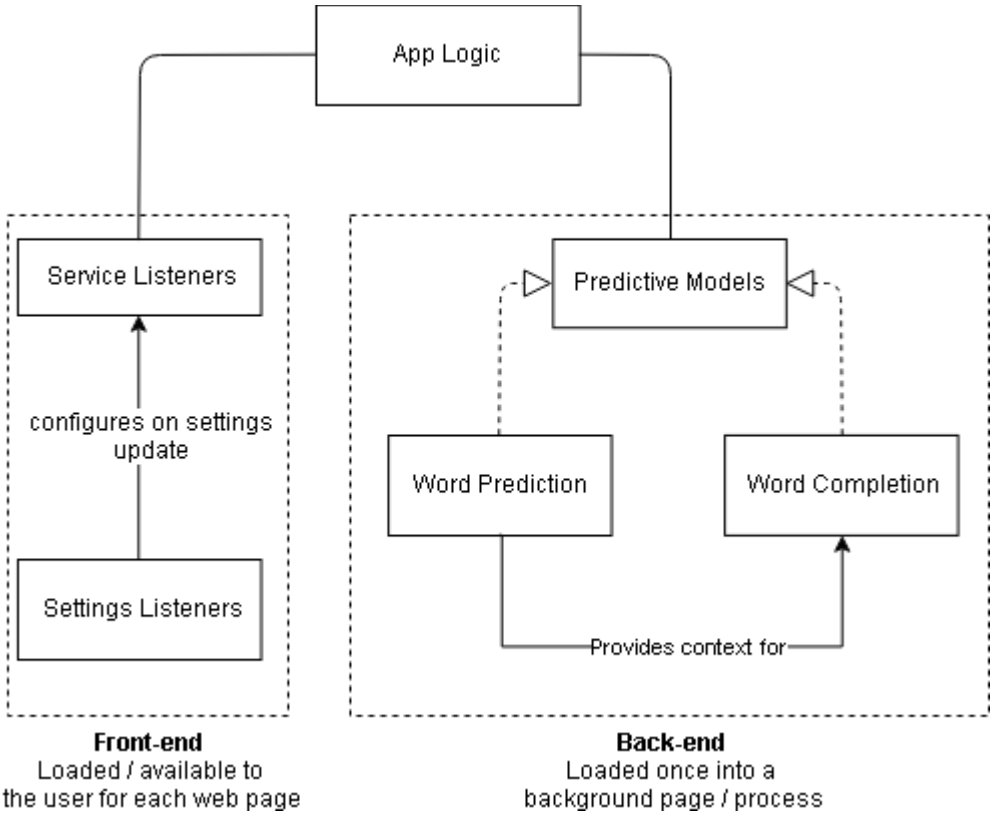
TabX is the point-of-contact to Chrome. It is composed of three main subsystems:

1. The application logic
2. Assets and static resources
3. Browser specs

Source code: TabX/ext/src/main.js

This composition is found in *main.js,* where the subsystems are explicitly required and configured.

Application Logic



This layer of the TabX system is responsible for scripts that can run webpages and allows users to access writing assisting services (the back-end). These subsystems are deployed separately, since they work in different processes.

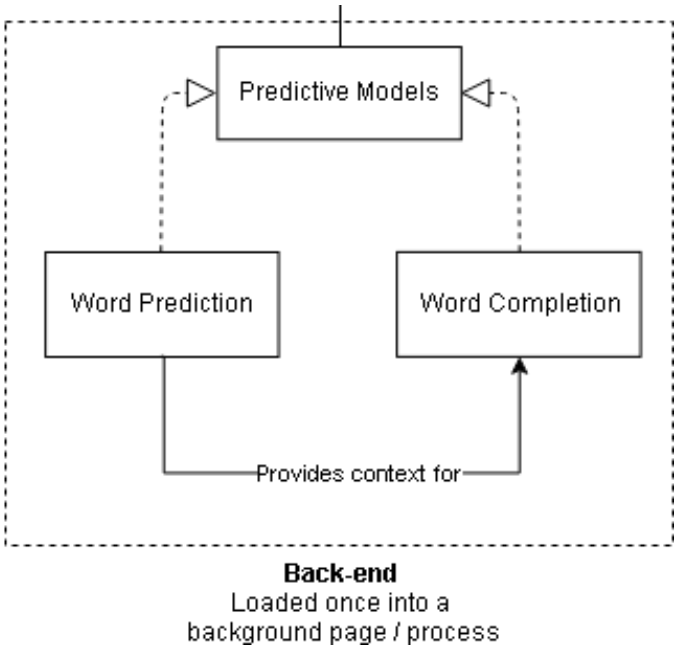
Source code:

- *Front end: TabX/ext/src/tabx.js*
- *Back end: TabX/ext/src/models.js*

tabx.js represents the UI logic and "intermediary" models that message the back-end. These intermediaries adapt TabX to use flavor of client-server architecture according to to the interface between the front-end and back-end.

models.js requires the models that will be used upon deployment, as well as install message listeners into Chrome so that the front-end can send requests to the models.

Application Logic: Predictive Models



The application logic includes a backend component that is responsible for processing user input and providing suggestions on what words the user might find helpful. Each model is given an interface by which to interact with the front-end.

Word completion models "agree" to receive a string that represents a potentially incomplete word (e.g. "hel", "worl") and output is a list of suggestions, represented as strings with which to complete the input string.

Word prediction models "agree" to receive a string that represents a string of words (eg. "hello world", "this an example"). The output is a list of suggestions, each representing a single-word string with which to append to the input string.

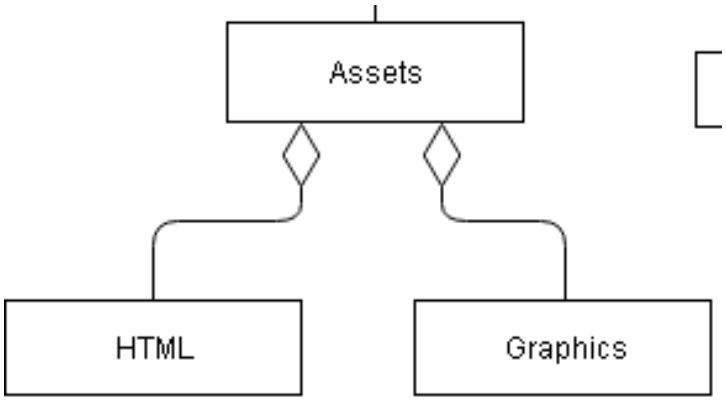
The word completion/correction model makes suggestions by traversing a trie and retrieving the words nearest to the input word with respect to matching characters. The word prediction model makes suggestions by using a Markov chain algorithm and suggesting the top three words with the highest probabilities relative to the input word. These models contribute to meeting the client's expectation of a writing assistant that is both fast and accurate in its predictions.

Source code:

- *Front end: TabX/ext/src/lib/wordcompletion/wordCompletionModel.js*
- *Back end: TabX/ext/src/lib/wordprediction/markov/word-prediction.js*

These are modules are required by *models.js*, which is loaded in the extension's background page. Each module exports an object that has methods that *tabx.js* expects. Accordingly, word prediction models must have a predictNextWord(str) method and returns a list of words, while word completion models must have a predictCurrentWord(str) method and also return a list of words.

Assets

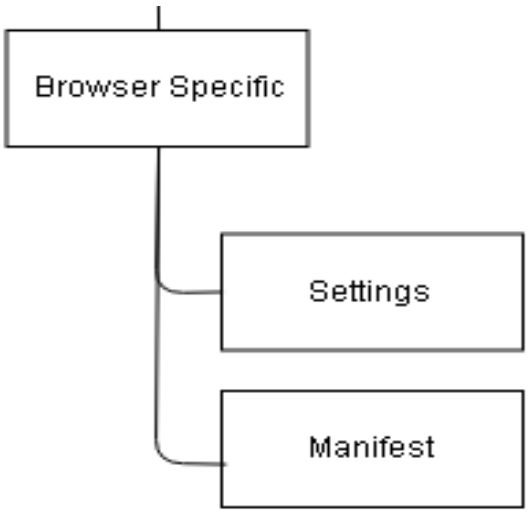


The assets subsystem contains resources that are used in displaying information to the user and separating the presentation aspect of TabX from the application logic subsystem.

Source: Tabx/ext/chrome/assets

Most assets are not required by *main.js,* but are used for the TabX's settings page and popup page for when users click the TabX browser icon. This logic is contained in chrome's manifest.

Browser Spec Subsystem



Finally, this subsystem is responsible for configuring TabX for use as a Chrome extension and to reflect user settings to TabX upon its initialization.

Source: Tabx/ext/browserspec/settings.js
Tabx/ext/chrome/manifest.json

Settings contains the necessary calls to chrome's storage API to create a TabX instance that reflects the current user's settings. Settings is required by *main.js,* which delegates the responsibility of configuration to settings.js. Eventually, Browser Spec will provide TabX an abstraction of how it can make API calls to a browser and persist data without explicit calls to Chrome's API in the app logic or the assets.

Set-up for Development Environment

To set up the development environment, clone the TabX repository.

TabX was developed using the following technologies:

1. HTML5
2. CSS3
3. ECMAScript 7
4. Node.js v10.13.0
5. Npm/npx v6.4.1
6. Chrome 70

Jasmine
Currently TabX has depends on the Jasmine Testing Framework for front-end development. It is not locally installed on the repository, so installing jasmine is necessary.

```
> npm install jasmine
```

TabX's specs were developed using jasmine v3.3.0 and jasmine-core v3.3.0.

Webpack
In addition, TabX requires webpack for deployment. This is used to bundle its dependencies for use in the browser. Version 4.26.1 is currently being used.

```
> npm install webpack
```

Webpack is executed from the TabX/ext folder using Node.js package runner npx.

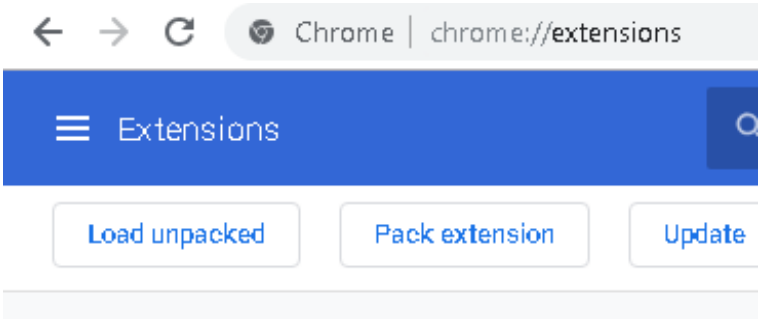
```
> npx webpack
```

Note: Lastly, part of TabX development relies on installing it to Chrome. Installing TabX into Chrome is needed to test how it is operating on the browser and web applications.

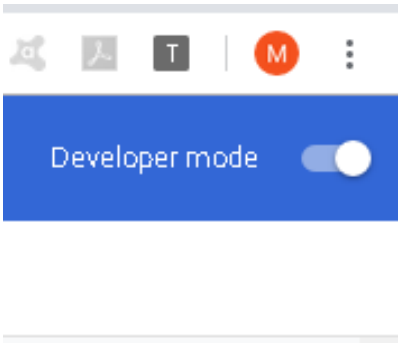
Installation

TabX is currently not yet published on the chrome store. TabX has no production form yet, so installation is for development purposes as of now. To install TabX in Chrome:

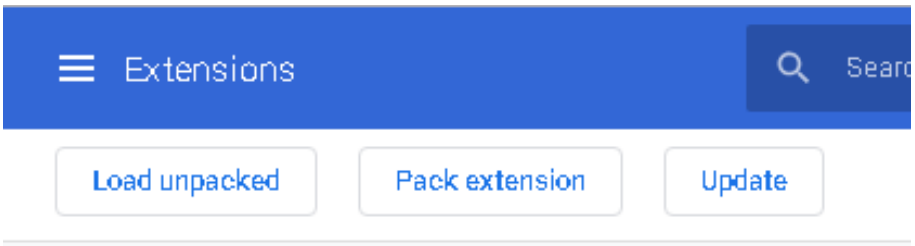
1. Open Chrome
2. Enter ‘chrome://extensions’ in the search bar



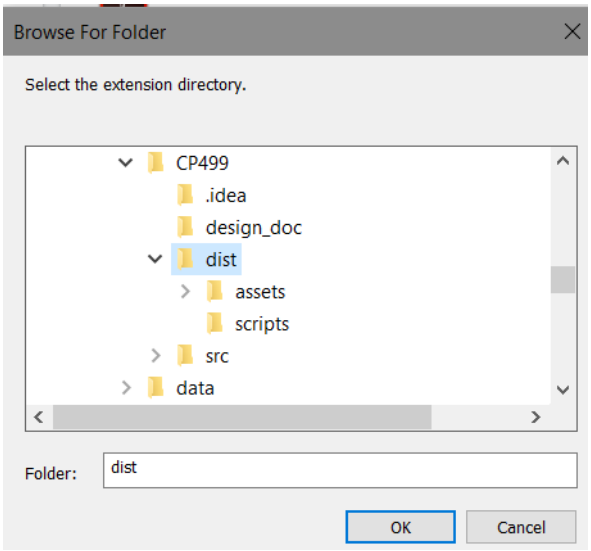
3. Enable developer mode (on top right of page)



4. Click on ‘Load Unpacked’ button



5. Navigate to local copy of TabX and select the ‘dist’ directory



6. Click ‘OK’

Note: Once TabX is installed in Chrome, any modifications in the source folder will reflected in the browser. This occurs because Chrome uses TabX in the development environment.

Configuration & Maintenance

Chrome Extension Interface

TabX, being a chrome extension, has to conform to Chrome's extension interface. If the interface were to be broken because of changes to this interface, Chrome will throw errors on the extension's page or upon the extension's installation. These errors include detailed information on what's wrong with the directory. Currently, Chrome relies on a manifest.json file to provide metadata on extensions, and it is what tell's Chrome that a directory should be treated as an extension. Manifest.json must be kept up-to-date to reflect Chrome's specifications.

Chrome API Calls and Deprecation

Also, TabX has explicit calls to Chrome's APIs throughout the source code. Namely, TabX relies on Chrome's tabs, storage, and runtime APIs. These may become deprecated and expose TabX to technical difficulties in the future. To keep TabX more flexible, abstractions on how to interact with Chrome for services (eg. storage, messaging, event listening) should be developed so that there is only one place to modify these calls.

Development Dependencies

Outside of Chrome interfacing, maintaining TabX is factored by the technologies used in its initial deployment.

1. HTML5
2. CSS3
3. ECMAScript 7
4. Node.js v10.13.0
5. npm/npx v6.4.1

Compatibility for future versions have to be considered is for TabX continued development. Maintainers should consistently update the source code to reflect the newest versions, or ensure means for older technology (esp. HTML5, CSS3, ES7) to work with newer environments.

Errors arising from lack of backwards-compatibility are expected to come up as developers work in environments with newer versions of the above dependencies.

Progress Summary

Feature	Description	Expected Date of Completion	Assigned
Word completion	Trie	Complete	Russell
Word correction	Trie - if word is misspelled	Complete	Russell
Word prediction	Modified Markov chain (takes up to 3 preceding words into context)	Complete	Nicole & Mishig
Customizable word/text training sets	For word prediction - results in interchangeable trained backend models	Complete	Nicole
Register user input from text fields	Grab user input from text fields to be passed to backend prediction models	Complete	Khayyon & Miguel
Hotkey to select next word	Press 1 to complete with first suggestion, 2 for second suggestions, and 3 for third suggestion	Complete	Khayyon & Miguel
Chrome extension	Function in the browser as a Chrome extension	Complete	Khayyon & Miguel
Consistent display of suggestion table	Suggestion table is already displayed, but needs to be made to persist across all webpages consistently (or at least the most frequented ones at the library). Currently, HTML on certain webpages overrides the display of the suggestion table.	Monday	Mishig
Abbreviation expansion	Backend model to expand common abbreviations: most likely in the form of a dictionary of abbreviations as keys and expansions as values	Icebox (if extra time)	Nicole
Profanity filter	Do not provide suggestions for profanity if typed by user due to library's concerns about usage of its public computers	Monday	Nicole
Ignore form/sensitive info text fields	Do not register input from/provide suggestions for password fields or other fields that may contain sensitive information, such as in forms (currently ignores only passwords)	Complete	Khayyon
Horizontal & vertical suggestion display table	Suggestions can be displayed in a vertical table or a horizontal bar (but no setting to switch between them yet)	Complete	Khayyon, Miguel, & Mishig
Backend validation tests	Unit tests to make sure backend models are functioning as expected	Complete	Russell

User testing metrics	Metrics to quantify feedback from user testing: measuring typing speed, how many suggestions are actually used by the user, etc. (check validation testing section from Report 2 for details)	Monday	Russell
Settings tab display	Ability to toggle on/off writing assistant features and services in the display, but not functional with the application yet	Complete	Khayyon & Miguel
Integration of settings tab with TabX functionality	Settings display tab already exists, but needs to be integrated with the corresponding functioning features in the application	Complete	Miguel
Setting: toggle between horizontal suggestion display bar and vertical display table	Setting to allow users to choose a suggestion display method: vertical table or horizontal bar	Monday	Khayyon
Setting: configure display options such as font size, color, etc.	Setting to configure user displays options to improve readability, accessibility, and overall user experience	Complete	Mishig
Setting: toggle between models/word sets used for training	Setting to switch between different types of the same prediction model so that the training word set used by the application better fits the needs of library patrons with respect to usefulness of word suggestions (e.g. library-specific word set for word prediction vs. generic word set)	Icebox (if extra time)	Mishig
Setting: choose how many suggestions to display	Setting to choose how many suggestions to display for a particular word: 2, 3, 5, etc.	Monday	Miguel