TEAM DATA - DELIVERABLE 4

07.04.2020

—

DHARMIK SHAH
YUN JIE (JEFFREY) LI
JEFFREY SO
ALVIN TANG
JIMMY PANG

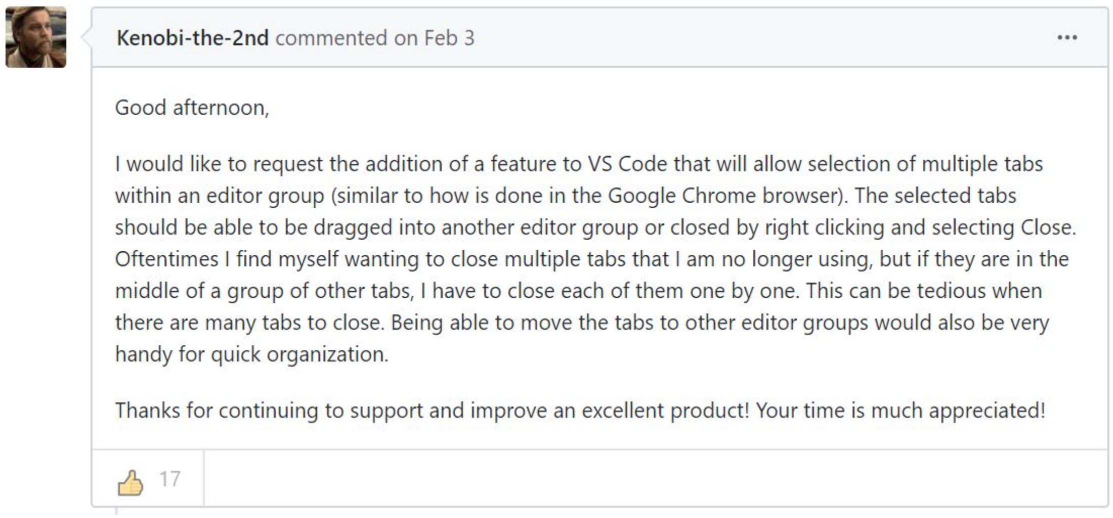
Table of Contents

Introduction

For this deliverable, Team DATA has decided to work on a feature for Visual Studio Code, that allows the user to select and interact with multiple tabs in the VS Code editor.

Background

GitHub issue #89958: Allow Selection of Multiple Editor Tabs



Kenobi-the-2nd commented on Feb 3

Good afternoon,

I would like to request the addition of a feature to VS Code that will allow selection of multiple tabs within an editor group (similar to how is done in the Google Chrome browser). The selected tabs should be able to be dragged into another editor group or closed by right clicking and selecting Close. Oftentimes I find myself wanting to close multiple tabs that I am no longer using, but if they are in the middle of a group of other tabs, I have to close each of them one by one. This can be tedious when there are many tabs to close. Being able to move the tabs to other editor groups would also be very handy for quick organization.

Thanks for continuing to support and improve an excellent product! Your time is much appreciated!

The feature was requested in a ticket for the official vscode repository on February 3. The user requested to allow the selection of multiple tabs within an editor group. To be more specific, the user should be allowed to select and interact with two or more multiple tabs, be it in a consecutive or non-consecutive manner. This is similar to the process of selecting multiple tabs in Google Chrome or Firefox. As of this deliverable, there are 17 upvotes for the ticket, but the lack of work on it has made the ticket a candidate for the backlog.

Team DATA decided to implement this feature for this deliverable because it gives us an opportunity to work with more layers of the Visual Studio Code codebase and better understand how multiple components interact with each other.

As well, implementing this feature would improve the productivity of VS Code users because it would help them in better organizing the editor tabs into specific groups and saves time for when they want to move or close multiple tabs to another view.

User Guide

How to Use

Installation

To start using this feature, be sure to pull the latest code from the team19/89958 branch. You can follow the steps to build and run the project using the Build and Run instructions provided in the wiki of the Github repository. However, in summary, you can run **`yarn`** to install all dependencies, **`yarn watch`** to build and then finally *in another terminal,* run **`yarn web`** to run VSCode in your browser. Once that is done, you will be able to start using VSCode with the feature.

Visualization

As the feature we built deals with the user interface, we have added settings to VSCode to actually visualize the selection. By default, we added some UI visualization changes but they are very hard to see with the current themes available on VSCode. We recommend you follow these steps.

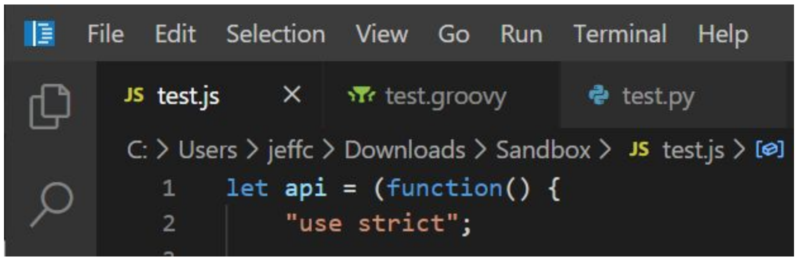
1. Open up the command palette (Ctrl + Shift + P on Windows)
2. Execute this command: *Preferences: Open Settings (JSON)*
3. Paste the following into the **settings.json** file opened up for you, which will allow you to visually see when tabs are selected:

```
{
    "workbench.colorCustomizations": {
        // Green
        "tab.activeSelectedBackground": "#008000",
        // Red
        "tab.unfocusedActiveSelectedBackground": "#ff0000",
        // Blue
        "tab.inactiveSelectedBackground": "#0000ff"
    }
}
```

4. Save the file

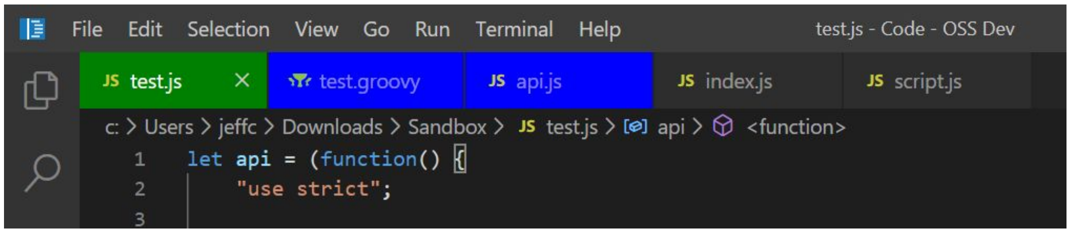
Selecting Multiple Tabs

Before continuing on to doing the wonderful things this new feature can do, one must first select multiple tabs. To do so, select Ctrl/Cmd and then start clicking tabs you want selected. If you have followed the previous steps and updated settings.json, then you can see any tab you click becomes green and any tabs that are also selected are blue.



```
let api = (function() {
    "use strict";
```

The above image is an example of two tabs, test.js and test.groovy, being selected. Both tabs are in a single selected group, but the current viewed tab will have its closed button visible.

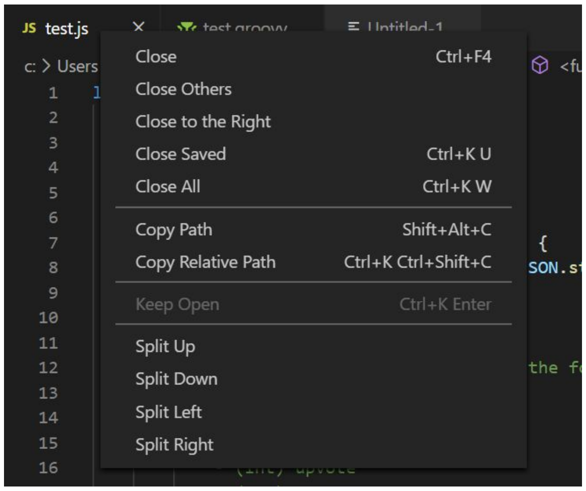


```
let api = (function() {
    "use strict";
```

If you had modified the settings.json file to visually see which tabs are selected. In the image above with colorCustomizations set, test.js, test.groovy, and api.js are selected. The tab for test.js is colored green as it is the current tab in the group being viewed in the editor, while test.groovy and api.js are not focused on are the moment.

Context Menu

The context menu for multiple selected tabs behaves slightly differently from its single-selected counterpart. When the selected tab group is right-clicked, the context menu is shown:



As of this deliverable, only the close and split options are implemented for the selected tab groups. The copy path and reveal options will behave as normal for the selected tab that was right clicked on.

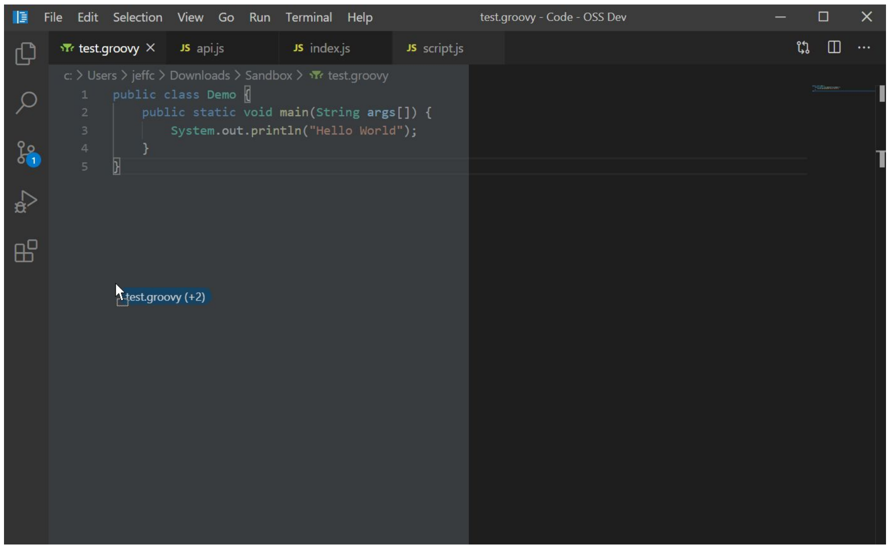
The following options in this context menu will behave as described:

- **Close**: all tabs that are selected are closed.
- **Close Others**: all tabs other than the ones selected are closed.
- **Close to the Right**: all tabs right of the rightmost selected tab are closed. Note that tabs to the left of the rightmost selected tab are still open.
- **Closed Saved**: behaves the same as with single tabs.
- **Close All**: all tabs are closed.
- **Split Up**: the selected tabs are moved into their own editor group and split to the top of the parent group.
- **Split Down**: the selected tabs are moved into their own editor group and split to the bottom of the parent group.
- **Split Left**: the selected tabs are moved into their own editor group and split to the left of the parent group.
- **Split Right**: the selected tabs are moved into their own editor group and split right of the parent group.

Dragging and Dropping

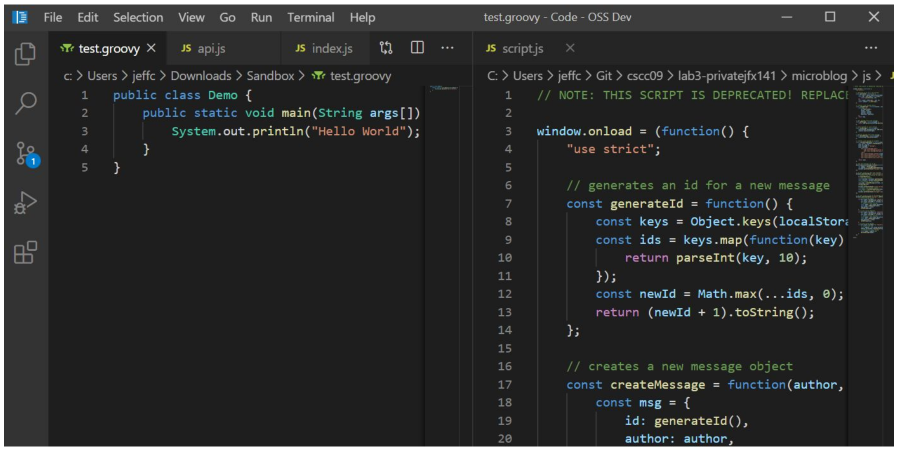
Dragging and dropping the selected tab group is also implemented. Dragged selected tabs can be dropped in their own editor group, depending on which area of the screen they are dropped on; a section of the editor will be highlighted showing where the new editor group will end up.

To drag-and-drop, simply hold down the left mouse button (drag) and then release the mouse button (drop) on an area that is highlighted.


File Edit Selection View Go Run Terminal Help
test.groovy - Code - OSS Dev
test.groovy
api.js
index.js
script.js
c: > Users > jeffc > Downloads > Sandbox > test.groovy
test.groovy (+2)

```
public class Demo {
    public static void main(String args[]) {
        System.out.println("Hello World");
    }
}
```

In the screenshot above, the selected tab group consisting of test.groovy, api.js, and index.js are being dragged. The left section of the parent editor is highlighted, so if the user were to drop in that section, a new editor group will be created, split left from the parent editor group. This is equivalent to using the context menu and selecting Split Left.


File Edit Selection View Go Run Terminal Help
test.groovy - Code - OSS Dev
test.groovy
api.js
index.js
script.js
c: > Users > jeffc > Downloads > Sandbox > test.groovy
C: > Users > jeffc > Git > cscc09 > lab3-privatejfx141 > microblog > js >

```
public class Demo {
    public static void main(String args[])
        System.out.println("Hello World");
    }
}
```

```
// NOTE: THIS SCRIPT IS DEPRECATED! REPLACE

window.onload = (function() {
    "use strict";

    // generates an id for a new message
    const generateId = function() {
        const keys = Object.keys(localStora
        const ids = keys.map(function(key)
            return parseInt(key, 10);
        });
        const newId = Math.max(...ids, 0);
        return (newId + 1).toString();
    };

    // creates a new message object
    const createMessage = function(author,
        const msg = {
            id: generateId(),
            author: author,
```

After the editor group is created, all tabs in the group will be unselected.

Feature Design

Recall that the workbench layer implements the elements surrounding the Monaco Editor. As the feature request involves modifying the tabs and actions that occur based on the tabs, the majority of our changes were centered around the workbench layer which surrounded each editor.

Selecting Multiple Tabs State

Before anything, we had to figure out how to store the state of the tabs to be selected. This led us to find **`TabsTitleControl`** (in **`tabsTitleControl.ts`**) which is a class in which when it is instantiated, holds the state of the tabs, registers specific DOM event listeners, and provides methods such as tab deletion in each editor group. An editor group is a group of editors and more than one editor group can exist if you had a split view of editors.

In our implementation we created an instance variable called **`selectedTabsElements`** which was an ES6 Map object which held the index of a selected tab as a key and its associated HTML element of the container that held the tab as the value. Then we modified **`registerTabListeners`** which registered an event listener for each tab. From this, we updated the mouse down event listener such that when the user left clicked while the ctrl key was down, we added the tab to the **`selectedTabsElements`** map. If the tab existed in the map already, we deleted the tab. This simply allowed us to store the state when a user selects or deselects a tab. We would also clear the map if the user right clicked on a tab but didn't hold down the ctrl key.

Note that **`registerTabListeners`** is called for each tab that is created.

UI and Styling

To ensure that the user is aware of state changes, the UI has to provide some sort of feedback to the user whenever the state changes. As such in **`TabsTitleControl`**, we created a private method called **`doRedrawEditorSelected`** which sets the background colour of an editor's tab based on the editor's state. If it was inactive and selected, the background would be set to **`TAB_INACTIVE_SELECTED_BACKGROUND`**. If it was active in an active editor group and selected, it would be set to **`TAB_ACTIVE_SELECTED_BACKGROUND`**. If it was active in an unfocused editor group and selected, it would be set to **`TAB_UNFOCUSED_ACTIVE_SELECTED_BACKGROUND`**. These constants were then defined in **`theme.ts`** in workbench with default values and were registered to be part of Visual Studio Code's configurable theme. This allows theme extensions to set the values for these specific states or allows you as a user to set them yourself in **`settings.json`**.

This **`doRedrawEditorSelected`** method would then be called every time it is expected the state would change. For example, if a tab gets clicked, **`this.redraw()`** gets called which then in turn calls **`this.doRedrawEditorSelected()`** which checks the state by looking in the **`selectedTabsElements`** map or accessing other instance variables and services.

Context Menu

We needed to edit the commands so when right clicking after selecting multiple tabs and selecting an action would perform that specific action on all tabs selected, as well, we needed to not show some options since they were not applicable to the context of multiple things selected. For this, we edited these files.

- **`editorActions.ts`**: Defines each individual action.
- **`editor.contribution.ts`**: Registers actions with a description an associated ID, and Command
- **`editorCommands.ts`**: Creates handlers for each command within the editor.
- **`fileActions.contribution.ts`** Registers handlers for each command relating to file actions

In our implementation, we mostly edited editorCommands.ts and fileCommand.ts to allow for multiple files for being selected and adding specific behaviors for options. We edited fileActions.contributions.ts because it contained a few of the commands that were used within the context menu for tabs, and we also added conditions to enable hiding certain options when opening the context menu after selecting multiple tabs.

Dragging & Dropping Support

While implementing the feature to move multiple tabs at once, we had received further insight into what each file is responsible for.

- **`titleControl.ts`**: Calls context menu services, the base class for TabsTitleControl
- **`workbench/browser/dnd.ts`**: Drag and drop class containing identifiers and class prototypes. We added an identifier class prototype to use with multiple selected tabs, seperate from the single selected tabs.
- **`editorDropTarget.ts`**: Handles when you drag and drop a file onto the actual "editor" part, e.g, we added the ability for the action handlers to be able to split with multiple tabs selected.

Under the **`platform/actions/common/`** path:

- **`actions.ts`**: Defines EditorTitleContext and MenuRegistry. When the getMenuItems method of MenuRegistry is invoked, it gets a list of menu items by first calling this._menuItems.get(id).
- **`menuService.ts`**: Defines the service to actually create a Menu instance and also defines a Menu class. The Menu service takes in the MenuId and then creates a Menu which then in turn calls a private method _build(). That function then looks into the MenuRegistry to get the specific menu items based on the Id. In this case,

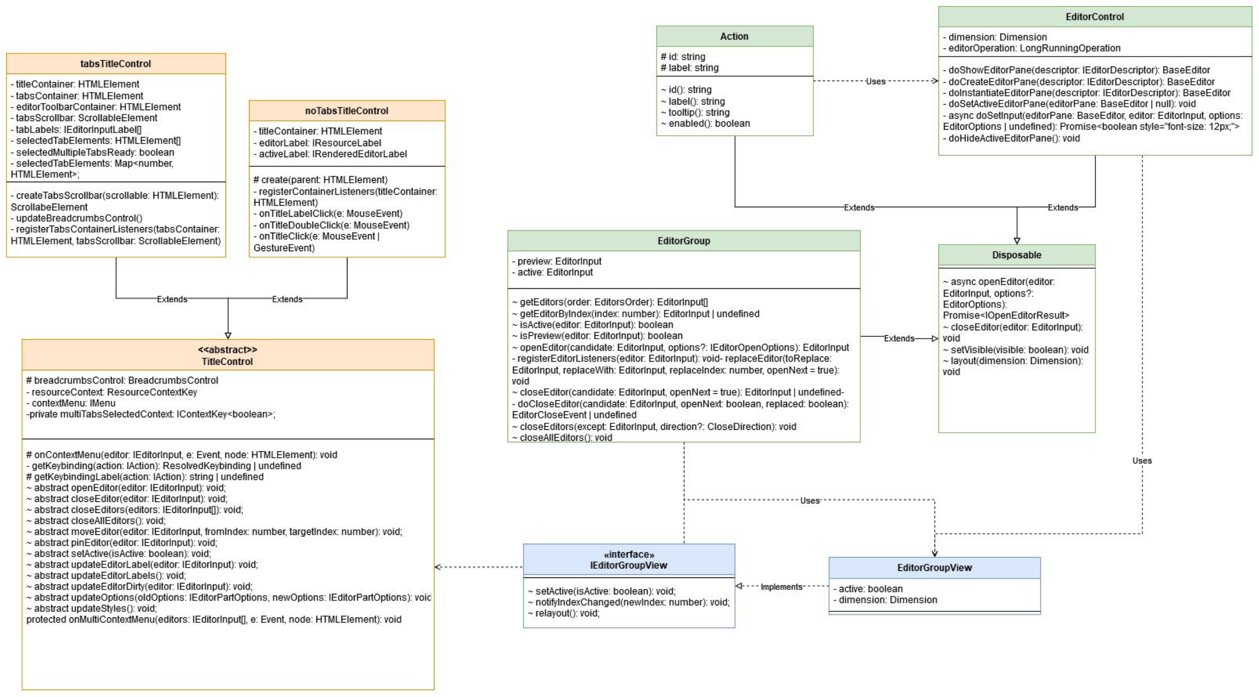
MenuId.EditorTitleContext in titleControl.ts is passed in to MenuRegistry.getMenuItems.

Lastly, we modified the file responsible for browser file actions:

- **`fileActions.contribution.ts`:** we needed to hide some context menu options when we select multiple tabs, as some actions are not possible or do not make sense with multiple files, for this, we added a new context service and a when statement when registering the command in the contextMenu to check if multiple files are selected

UML Diagram

The result UML diagram is similar to that of deliverable 3, albeit with the previously mentioned code modifications.



tabsTitleControl
- titleContainer: HTMLElement
- tabsContainer: HTMLElement
- editorToolbarContainer: HTMLElement
- tabsScrollbar: ScrollableElement
- tabLabels: IEditorInputLabel[]
- selectedTabElements: HTMLElement[]
- selectedMultipleTabsReady: boolean
- selectedTabElements: Map<number, HTMLElement>;
- createTabsScrollbar(scrollable: HTMLElement): ScrollabeElement
- updateBreadcrumbsControl()
- registerTabsContainerListeners(tabsContainer: HTMLElement, tabsScrollbar: ScrollableElement)
noTabsTitleControl
- titleContainer: HTMLElement
- editorLabel: IResourceLabel
- activeLabel: IRenderedEditorLabel
# create(parent: HTMLElement)
- registerContainerListeners(titleContainer: HTMLElement)
- onTitleLabelClick(e: MouseEvent)
- onTitleDoubleClick(e: MouseEvent)
- onTitleClick(e: MouseEvent | GestureEvent)
Extends
Extends
<<abstract>>
TitleControl
# breadcrumbsControl: BreadcrumbsControl
- resourceContext: ResourceContextKey
- contextMenu: IMenu
-private multiTabsSelectedContext: IContextKey<boolean>;
# onContextMenu(editor: IEditorInput, e: Event, node: HTMLElement): void
- getKeybinding(action: IAction): ResolvedKeybinding | undefined
# getKeybindingLabel(action: IAction): string | undefined
~ abstract openEditor(editor: IEditorInput): void;
~ abstract closeEditor(editor: IEditorInput): void;
~ abstract closeEditors(editors: IEditorInput[]): void;
~ abstract closeAllEditors(): void;
~ abstract moveEditor(editor: IEditorInput, fromIndex: number, targetIndex: number): void;
~ abstract pinEditor(editor: IEditorInput): void;
~ abstract setActive(isActive: boolean): void;
~ abstract updateEditorLabel(editor: IEditorInput): void;
~ abstract updateEditorLabels(): void;
~ abstract updateEditorDirty(editor: IEditorInput): void;
~ abstract updateOptions(oldOptions: IEditorPartOptions, newOptions: IEditorPartOptions): void
~ abstract updateStyles(): void;
protected onMultiContextMenu(editors: IEditorInput[], e: Event, node: HTMLElement): void
Action
# id: string
# label: string
~ id(): string
~ label(): string
~ tooltip(): string
~ enabled(): boolean
Uses
EditorControl
- dimension: Dimension
- editorOperation: LongRunningOperation
- doShowEditorPane(descriptor: IEditorDescriptor): BaseEditor
- doCreateEditorPane(descriptor: IEditorDescriptor): BaseEditor
- doInstantiateEditorPane(descriptor: IEditorDescriptor): BaseEditor
- doSetActiveEditorPane(editorPane: BaseEditor | null): void
- async doSetInput(editorPane: BaseEditor, editor: EditorInput, options: EditorOptions | undefined): Promise<boolean style="font-size: 12px;">
- doHideActiveEditorPane(): void
Extends
Extends
EditorGroup
- preview: EditorInput
- active: EditorInput
~ getEditors(order: EditorsOrder): EditorInput[]
~ getEditorByIndex(index: number): EditorInput | undefined
~ isActive(editor: EditorInput): boolean
~ isPreview(editor: EditorInput): boolean
~ openEditor(candidate: EditorInput, options?: IEditorOpenOptions): EditorInput
- registerEditorListeners(editor: EditorInput): void- replaceEditor(toReplace: EditorInput, replaceWith: EditorInput, replaceIndex: number, openNext = true): void
~ closeEditor(candidate: EditorInput, openNext = true): EditorInput | undefined-
- doCloseEditor(candidate: EditorInput, openNext: boolean, replaced: boolean): EditorCloseEvent | undefined
~ closeEditors(except: EditorInput, direction?: CloseDirection): void
~ closeAllEditors(): void
Extends
Disposable
~ async openEditor(editor: EditorInput, options?: EditorOptions): Promise<IOpenEditorResult>
~ closeEditor(editor: EditorInput): void
~ setVisible(visible: boolean): void
~ layout(dimension: Dimension): void
Uses
Uses
«interface»
IEditorGroupView
~ setActive(isActive: boolean): void;
~ notifyIndexChanged(newIndex: number): void;
~ relayout(): void;
Implements
EditorGroupView
- active: boolean
- dimension: Dimension

Testing Suite

The VSCode project stores 4 types of tests, and has runners for each of them. Since our feature is more user interactive, some types of tests do not apply:

Unit Tests

- These are tests that make sure that a specific functionality is working properly
- It makes sure that given a certain input, the result of the function is the correct output
- As we are doing more of a user interface feature, unit tests would not be appropriate. An example of appropriate unit testing would include, making sure a sorting function works, making sure that when keystrokes are pressed, a certain response is shown on the editor (for instance the commenting lines we did in the last deliverable)
- Unit tests for moving tabs, selecting multiple of them are not appropriate, and are not the best type of tests for our feature

Integration Tests

- These are tests in which individual software modules are combined and tested as a whole, these type of tests combine all the tabs logic (pre-existing and newly implemented), and test it all together
- However, we are simply extending based on the current features (like the ability to slide a tab, to close a tab, to select a tab), therefore these features are being tested individually rather than as a whole
- Thus it does not make sense to write separate integration test cases, because it needs to work as a whole rather than individual parts
- Although integration tests may be appropriate, the implemented feature requires more manual testing rather than code

Smoke Tests

- These are a set of non-exhaustive tests done whose goal is to make sure that the most important functions of the application work
- For instance, smoke tests would test that users are able to create tabs, delete tabs, and move them around
- Since our feature was to take the tab functionality to another level, the set of smoke tests which check for tab functionality should remain unaltered. There is actually no need to modify the smoke test because the main features of the application, like tab control is already being tested

UI Tests

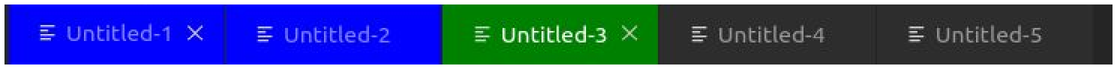
- These are the set of tests that deal if specific UI actions that are hard to replicate in code
- This includes our feature which is a user dependent process
- UI tests are the best option for our feature
- The best idea here is to use acceptance tests as the primary testing framework, since they can illustrate that the function performs as expected

Acceptance Tests

Before you begin these tests, make sure that you have the custom settings.json file as illustrated in the **User Guide** section. This will visually show you what is going on.

Selection of Multiple Tabs

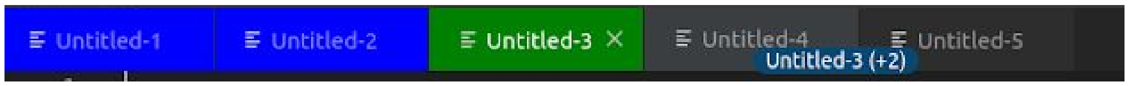
- Create 5 files by pressing **Ctrl + N**, 5 times. These files do not have to be saved.
- Click on **Untitled-1**.
- While holding the Ctrl key, click the first tab, the second, and the third.
- If done correctly, the tabs section should resemble closely to the following image:



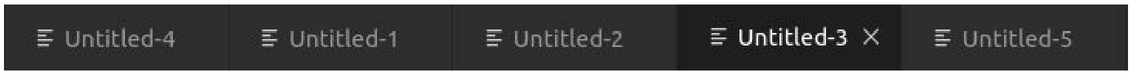
- This means that the first 3 tabs are selected, and the current tab is Untitled 3.

Drag and Drop Tab Groups

- Assuming that tabs 1-3 are still selected, you can let go of the Ctrl key and the tabs should still be coloured in.
- Click and drag Untitled-3 **on top** of Untitled-4, you should see the following:



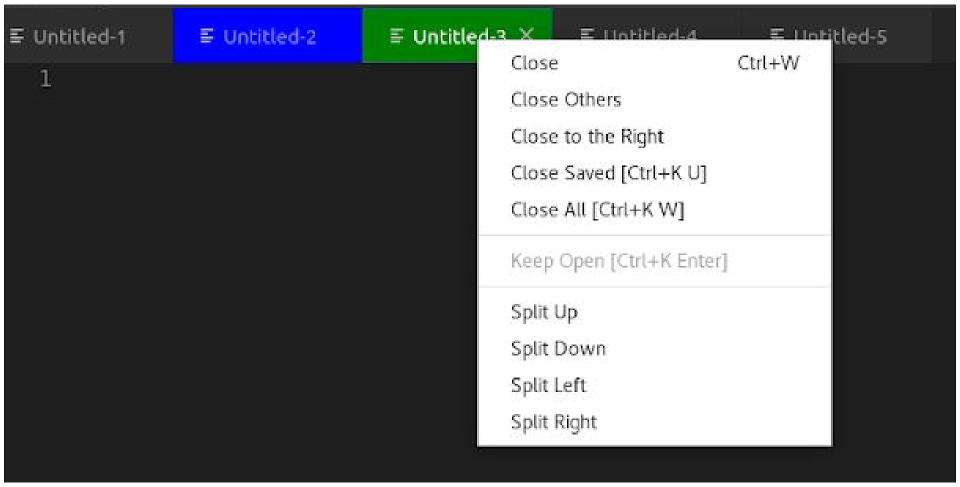
- This indicates that we are taking Untitled-3 plus an additional two tabs, and placing it **after** Untitled-4 and **before** Untitled-5.
- If you let go, you should see the following:



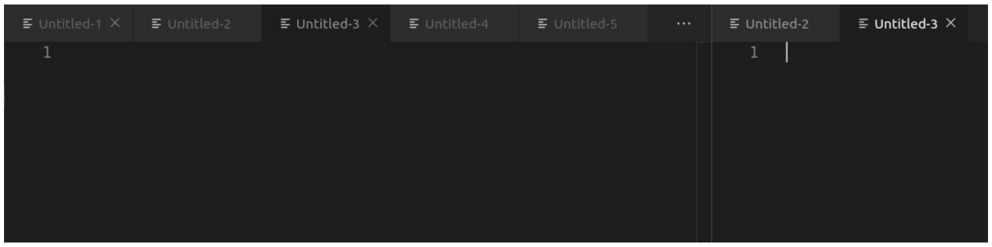
- Untitled-4 is in the first tab position, and the rest of the tabs have moved over.
- From here you can try selecting Untitled-1 and Untitled-2, and moving them on top of Untitled-3. If you did it correctly, the new order of the tabs should be 4-3-1-2-5.

Context Menu of Tab Groups

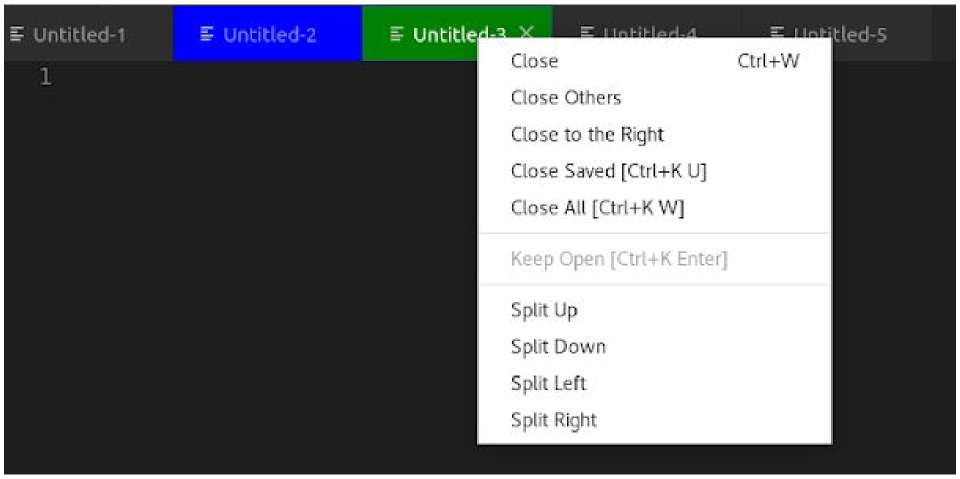
- After a user selects multiple tabs, they can right click any of the selected tabs to bring up the context menu.
- When a user selects an option from the context menu, the selected action is applied to all selected tabs.
- This time, select tabs 2 and 3, and right click either one of them



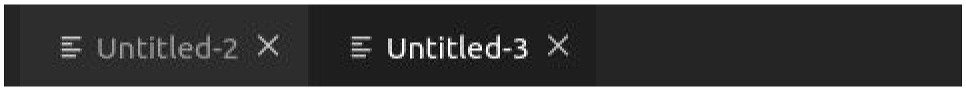
- You should see a context menu pop up
- Select **Split Right**



- You should see that tabs 2 and 3 are also displayed on the right hand side of the view panel
- Initially you could only split with one tab, and then you would have to drag the other tabs over, but with multiple tab select, this is not the case anymore
- Note that now that we have two "views", we can drag multiple tabs from one view to another as well
- Press the **x** button on the tabs on the right view, when all tabs are closed on the right view, the panels will return to normal with a single view.



- This time, select the tabs and press **Close Others**
- If you did this correctly, you should see that only these tabs remain



Conclusion

Should all of the above acceptance tests pass, then the feature is working as intended.

Software Development Process

Trello board: https://trello.com/b/JPvqOrCc/vs-code

As with our previous deliverables, we continued to use Kanban as our main software development process. As with what we did in deliverable 2, we used Trello to visualize our progress while working on the feature. Our columns are similar to what they were before: Backlog, To Do, In Progress and Complete.

The modification that we had made in our software development process in order to accommodate for developing this feature is setting a limit for the In Progress column. The Work In Progress limit during this deliverable was structured a bit differently from previous deliverables due to the requirements of the final report. Although we kept the amount of work delegated to each individual as even as possible, limiting one major task to any one individual, some work ended up overlapping with other major tasks. For example, implementation of the feature would have been considered a major task and therefore to stay within the WIP limit, anyone who was assigned to work on the implementation would not have to contribute much to other major tasks such as writing up the main components of the documentation report.

However, due to components such as “Comparison of New/Old code base” in the final report, it was unavoidable for the individual who coded the implementation to also take on partial responsibility to write up the aforementioned section of the report, since they would have the best insight as to what was changed as well as how it affected the rest of VS Code as a whole. However this meant surpassing our initial WIP limit plan. Therefore in order to compensate for this extra work load, we re-structured our WIP limit to better encompass which tasks are closely related to each other rather than strictly counting task story points. This allowed us to stack members on a number of overlapping tasks in order to appropriately distribute the workload without overburdening any one individual.

For example, throughout the implementation of this feature, we used VS Code liveshare in order to collaboratively work on the same set of files in order to synchronize information. Although the individual who coded the implementation of the feature may understand how the feature affects VS Code as a whole the most, others who were in charge of the write up could have learned alongside them via liveshare or having the implementer explain what they were thinking as they coded, allowing them to have equal knowledge about what needs to be written.