TEAM DATA - DELIVERABLE 1

26.02.2020

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

TABLE OF CONTENTS

Visual Studio Code Infrastructure

Much of the underlying structure of Visual Studio Code is located under the `src/vs` folder. Furthermore, inside `src/vs`, you can see that the codebase is divided into 5 main layers: `base`, `code`, `editor`, `platform`, and `workbench`.

Base

The `base` layer provides utilities and basic building blocks required to construct user interfaces. The code in the base layer defines the look (via CSS) and the basic behaviour of the UI elements.

Platform

The `platform` layer provides the base services of Visual Studio Code, and how they are interacted with via dependency injection. These base services are mostly just definitions; they do not have any implementation as most of them are exported interfaces.

The services described in this layer are used by the rest of the code. Even extensions will use some sort of dependency from the platform layer, such as dialogs.

The following code snippet shows how the model service is injected into a client:

```
class Client {
  constructor(
    @IModelService modelService: IModelService,
    @optional(IEditorService) editorService: IEditorService
  ) {
    // use services
  }
}
```

Editor

The `editor` layer contains the Monaco Editor, which is an independent component and serves as the "text editor" portion of Visual Studio Code. The layer also contains implementations of editor-specific commands like find & replace, shift selection, and more.

Workbench

The `workbench` layer implements the elements surrounding the Monaco Editor. This is different from the editor which itself implements editor-specific features. Workbench features that are contributed by developers are stored in the `workbench/contrib` folder,

which is then called via an internal API. This includes features such as emmet, themes, bulk editing, and debug view.

Code

The **`code`** layer is where all the components of the previous four layers are integrated together using the Electron framework.

Layer Structure

Each of these layers has a specific subfolder structure which is fully written in **TypeScript**.

The underlying folder structure is as follows:

- **`/common`** - any source code that only requires basic JavaScript APIs to run
- **`/browser`** - any source code that requires access to the DOM (document object model; we can have this since its an electron application)
    - Could use code from **`/common`**
- **`/node`** - any source code that requires Node.js APIs
    - Could use code from **`/common`**
- **`/electron-browser`** or **`/electron-main`** - any source code related to electron
    - Could use code from **`/common`**, **`/browser`**, **`/node`**

The following is some more information about the workbench folder, as it is a major component that we will likely be modifying:

`workbench/browser/workbench.ts`

- This is the instantiation of workbenches.

`workbench/browser/layout.ts`

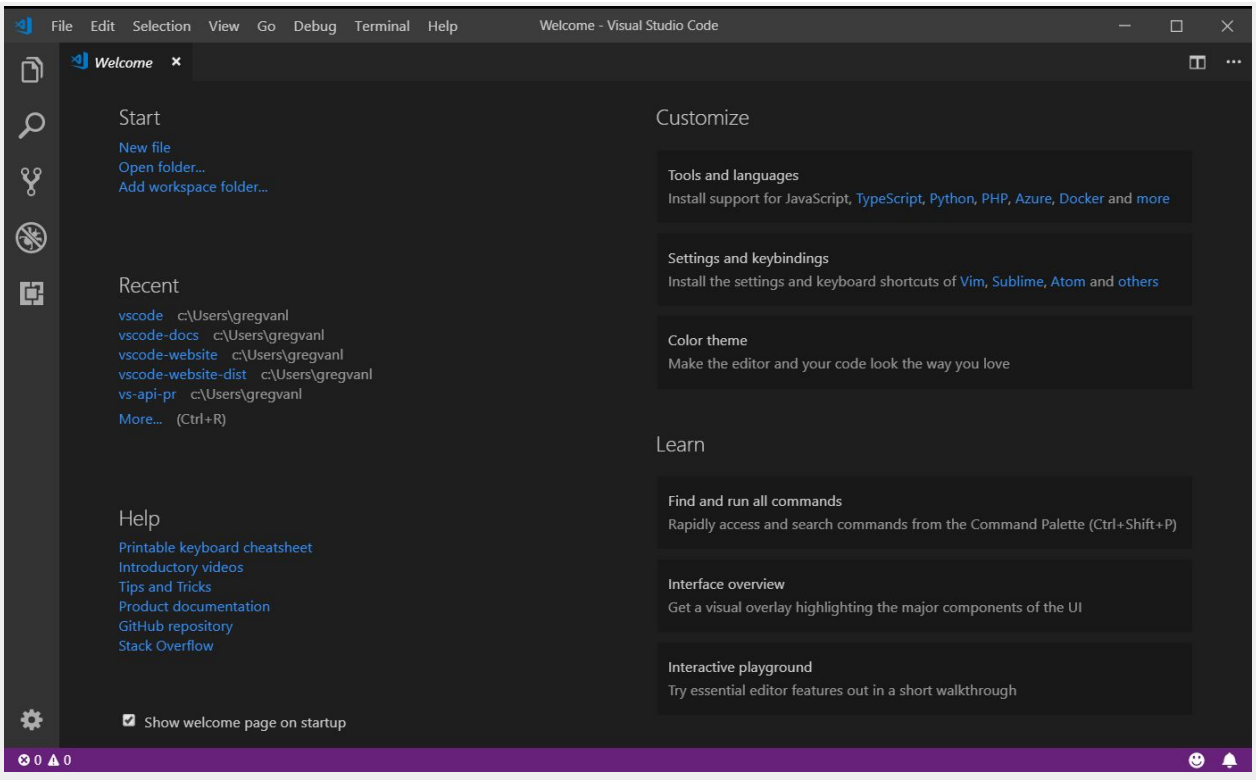
- Formats the session layout upon launching VSCode.
- Saves session states, such as active files and active electron-browsers which function as the editorial tabs that the user interacts with.
- For example, if you have three files (tabs) open, close VSCode, then open it up again, VSCode will re-open those three files you had opened earlier (unless you specifically tell VSCode to open a specific file) because the session layout is saved.
- Function - **`resolveEditorsToOpen`**

`editor/common/services/editorWorkerServiceImpl.ts`

- Takes in a new **`workerFactory`** and builds an **`EditorWorkerClient`** class which is used to register/create a new **`SimpleWorkerClient`**.

`workbench/contrib/remote/electron-browser/remote.contributions.ts`

- Has code that generates the **Welcome to VSCode** homepage (Figure Below) upon a fresh boot of VSCode.
- Class - **`RemoteEmptyWorkbenchPresentation`**



Interesting Aspects of Design

The app starts in `vs/code/electron-main/main.ts`. From there, a variety of services are instantiated and initialized which helps start up the application. We had the chance to look around with a debugger and found a variety of interesting aspects of design in the codebase which provides a lot of benefits, such as extendibility.

Dependency Injection

In `platform`, a variety of services are defined which rely on the architectural concept of dependency injection. As JavaScript is a non-statically typed language, Visual Studio Code utilized TypeScript to provide static type checking. With TypeScript comes interfaces that have overall made it easier to define dependencies provided through class constructors.

An example of dependency injection could be found in the constructor of `ExtensionGalleryService`, which is a service that contains logic to manage VSCode extensions such as querying the extensions gallery or downloading an extension. Specifically, you can find a dependency as a parameter in the constructor named `requestService` with the interface of `IRequestService`. This service in itself provides underlying logic to make HTTP requests. This dependency was utilized in the method for querying the gallery (`queryGallery`) via `this.requestService.request()`.

Factory Design Pattern

The factory design pattern is utilized throughout VSCode.

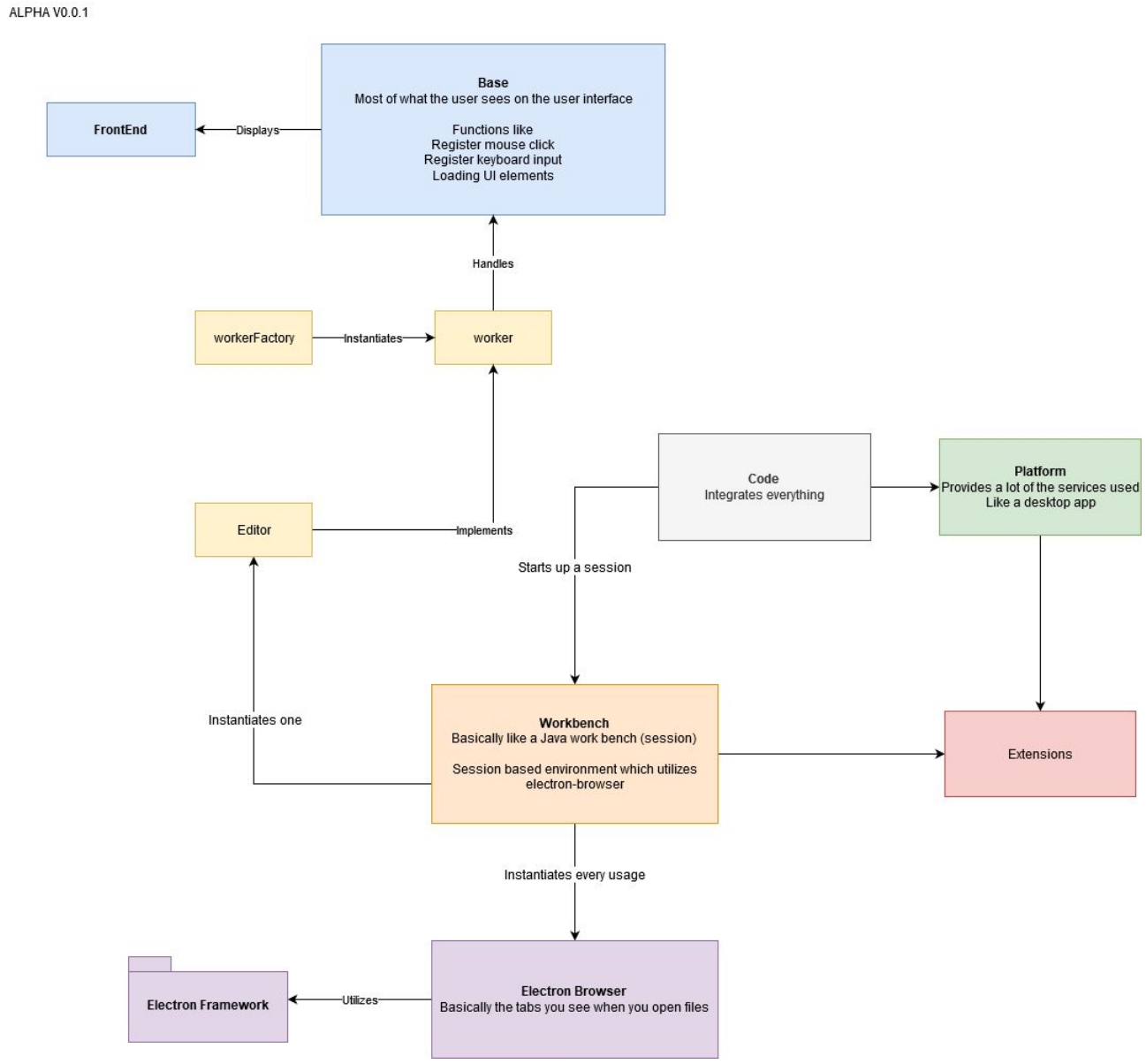
Starting from `DefaultWorkerFactory`, the base interface of a **WebWorker**, defines the base functionality of `Workers`, including creating, disposing, getting moduleIds, and `postMessage` which allows `Workers` to communicate their information.

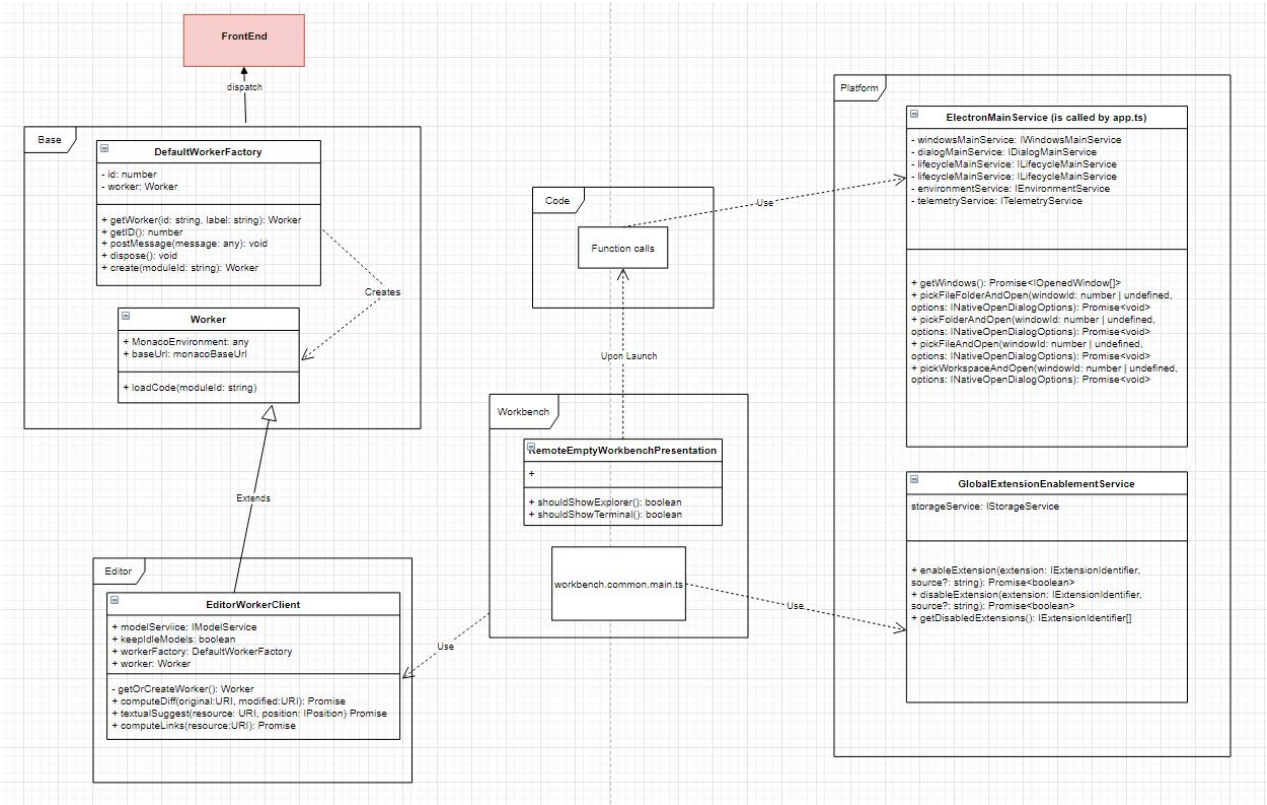
`editorSimpleWorker` extends the `Worker` classes created from the `DefaultWorkerFactory` to attach common `Model` functionalities which act as the text manipulation functions commonly seen on the editors such as: `getValue`, `getLineCount`, `getLineContent`, etc..

Finally, `editorWorkerServiceImpl` instantiates `EditorWorkerClient` instances which act as the text editors users see when editing files in VSCode and tracks changes in the text editor portion of the window

Outline and UML

https://drive.google.com/file/d/1XTCghlZxT8kOM2ysAlx2HRrhwQdKRmTm/view?usp=sharing




FrontEnd
dispatch
Base
DefaultWorkerFactory
- id: number
- worker: Worker
+ getWorker(id: string, label: string): Worker
+ getID(): number
+ postMessage(message: any): void
+ dispose(): void
+ create(moduleId: string): Worker
Creates
Worker
+ MonacoEnvironment: any
+ baseUrl: monacoBaseUrl
+ loadCode(moduleId: string)
Extends
Editor
EditorWorkerClient
+ modelService: IModelService
+ keepIdleModels: boolean
+ workerFactory: DefaultWorkerFactory
+ worker: Worker
- getOrCreateWorker(): Worker
+ computeDiff(original:URI, modified:URI): Promise
+ textualSuggest(resource: URI, position: IPosition) Promise
+ computeLinks(resource:URI): Promise
Use
Code
Function calls
Use
Upon Launch
Workbench
RemoteEmptyWorkbenchPresentation
+
+ shouldShowExplorer(): boolean
+ shouldShowTerminal(): boolean
workbench.common.main.ts
Use
Platform
ElectronMainService (is called by app.ts)
- windowsMainService: IWindowsMainService
- dialogMainService: IDialogMainService
- lifecycleMainService: ILifecycleMainService
- lifecycleMainService: ILifecycleMainService
- environmentService: IEnvironmentService
- telemetryService: ITelemetryService
+ getWindows(): Promise<IOpenedWindow[]>
+ pickFileFolderAndOpen(windowId: number | undefined, options: INativeOpenDialogOptions): Promise<void>
+ pickFolderAndOpen(windowId: number | undefined, options: INativeOpenDialogOptions): Promise<void>
+ pickFileAndOpen(windowId: number | undefined, options: INativeOpenDialogOptions): Promise<void>
+ pickWorkspaceAndOpen(windowId: number | undefined, options: INativeOpenDialogOptions): Promise<void>
GlobalExtensionEnablementService
storageService: IStorageService
+ enableExtension(extension: IExtensionIdentifier, source?: string): Promise<boolean>
+ disableExtension(extension: IExtensionIdentifier, source?: string): Promise<boolean>
+ getDisabledExtensions(): IExtensionIdentifier[]

Extension API

Many core features of VSCode are built as extensions using the Extension API. The Extension API covers how to interact with VSCode.

This includes how to create themes, how to extend the workbench (title bar, activity bar, panels), how to support a new programming language (syntax highlighting, snippet completion), enhance the debugger, and many other features.

The steps to creating an extension are as follows:

```
npm install -g yo generator-code
yo code
```

The main file to edit is `/src/extension.ts` inside the `activate` function.

This is also TypeScript, and you have access to any npm modules you would like. As a result, it is very flexible.

More details can be found here:

https://code.visualstudio.com/api/get-started/your-first-extension

Software Design Process

We considered the following software design processes, and in the end decided to settle with **Kanban**. Below you will find why we didn't choose a certain process, and what made Kanban appear more appealing to us.

Waterfall

- **Pros**: A clear layout of what to follow is laid out for the team. For example, when tasks like "requirement definition", then "system and software design" are planned out ahead of time, it is easier to go based on this when the requirements are well understood. When we will be fixing bugs, the problem, expected behaviour will be understood clearly, and so waterfall is a good choice.
- **Cons**: If the application is in testing phase, it is very difficult to go back to stage one if something is to be added. Since VS Code is a very big software, adding one small feature may require changing many files. We have to be completely sure that we are done this stage, and cannot risk going backwards, otherwise this process might take a very long time.
- **Overall decision**: We considered this to be a good solution.

Reuse-oriented

- **Pros**: As a large, well-established project, VS Code already provides many resources to work off of. This is especially good if we want to develop any new extensions since Microsoft already provides a native extension generator. This will allow us to deliver new code faster.
- **Cons**: This process model is not adequate if we are developing a feature that is currently vastly out of scope for VS Code, since we would end up spending more time developing new components than reusing and integrating existing ones. If we are unable to do an extension as the feature request, then adding a native feature to the VS Code software would prove to be difficult. There are not many existing components to work off of. We would have to create our own.
- **Overall decision**: We considered this to be an inappropriate solution.

XP

- **Pros**: If there is a difficult task that needs to be completed, XP asks for people, not processes. Basically, this means that when using XP process, we will have frequent pair programming, and this can result in more efficient development and in a good amount of time.
- **Cons**: Not good for work-life balance. XP requires all team members to be entirely dedicated to the project, and that is not feasible for us students who have other courses and obligations.
- **Overall decision**: We considered this to be an inappropriate solution.

Kanban

- **Pros**: The good thing about Kanban is that it is a gradual, incremental process with a lot of flexibility. It doesn't have to revolve around adopting pure Agile principles. Perhaps one of the most interesting features about Kanban is the daily stand-ups, which are very beneficial, and also the Kanban board. The Kanban board can be laid out in a variety of ways, for example: to do, doing, done. This will be especially helpful if we can expand this to be to do, doing, testing, verifying, committing, done, or something of that matter. This will allow us to have a clear approach on how to tackle bug fixes, or adding features. It's also very easy to add new tasks, we just "pull" them in.
- **Cons**: It is hard to enforce WIP limits. Moreover, even though a Kanban board is there, we might be tempted to just do work in a different way, rather than strictly following the board. This will result in not the optimal code base. We must follow the board.
- **Overall decision**: We considered this to be a good solution.

Incremental

- **Pros**: If we are planning to develop a large feature, we can develop and deliver new successive versions of it. With each version, we can get valuable feedback from end-users (whoever posted the thread on GitHub), faster than had we gone with a strictly plan-driven process.
- **Cons**: As we add new increments to our codebase, the quality of our code tends to degrade, unless considerable time and effort is made to refactor it. As the code grows, it becomes increasingly difficult to maintain. Given that us students only have around a month to work on (with other coursework) such a feature, we cannot maintain our codebase to be perfectly structured throughout.
- **Overall decision**: We considered this to be an inappropriate solution.

Conclusion

Overall, we considered Waterfall and Kanban to be the best processes to choose from.

Kanban offers us more flexibility in deciding the stages of development, it provides an easy way to track work progress and shareability with the rest of the team, while Waterfall lacks any of the mentioned tools. Additionally, due to Waterfall's strict development timeline, if for example we want to backtrack and fix a bug, we would have to revert to a previous phase, re-evaluate, and start the development process all over again. With Kanban, it offers us more flexibility in this matter, as it is agile based, allowing us to fix bugs without having to restart from the beginning.