Learning VS Code source

Tom Rochette <tom.rochette@coreteks.org>

November 2, 2024 — 36c8eb68

1 2018-05-26

- Read `package.json` to discover what packages VS Code depends on
- Observe the root directory structure, and more specifically the `extensions` and `src` directories which contain the bulk of the source code
  - A lot of the code in the `extensions` directory appears to be dedicated to programming language support
    - The remainder of the extensions seem to provide functionality for things that aren't "core" to vscode, such as `configuration-editing`, `emmet`, `extension-editing` and some color themes
- If you look at the `.vscode/launch.json`, you will find all the tasks that can be executed from within VS Code debugger. One task of interest is `Launch VS Code` which will take care of launching VS Code for us so that we may debug it
  - In this file you will also discover that it runs `${workspaceFolder}/scripts/code.bat`, which is the next script we'll take a look at
- In `./scripts/code.bat`, we discover that this script will run `yarn` if the `node_modules` directory is missing, download the electron binaries if necessary and call `gulp compile` if the `out` directory is missing, then finally start the electron/vs code binary in the `.build/electron` directory
- We then start to look for common entry points file such as `index.ts/js` or `main.ts/js`, for which we find a match in the `src` directory
- We take a quick look around, trying to find where electron is likely to be instantiated... There's a lot of code in `src/main.js` that would be better elsewhere to make it easier to navigate this file
- Close to the bottom of the file we discover the code we are interested in as a call to `app.once('ready', ...)`
  - Once the app is ready, we want to call `src/bootstrap-amd` and pass `vs/code/electron-main/main` as our entry point (per the signature of the exported function in `./src/bootstrap-amd`)
    - Here we can go to two places, either `src/bootstrap-amd` or `src/vs/code/electron-main/main`
      - We take a quick peek at both files and we can quickly tell that `src/bootstrap-amd` is used mainly to load `src/vs/code/electron-main/main` which is the file we're going to be interested in
- Once again, we quickly look around `src/vs/code/electron-main/main` and find that the main logic is at the bottom of the file
- First the command line arguments are parsed
- Then services are bootstrapped/instantiated
- Finally the `CodeApplication` is started up
- This leads us to look into `src/vs/code/electron-main/app.ts`
- As the file is quite large, we start by skimming through it, looking at the available methods on the `CodeApplication` class as well as its properties
- Looking at the constructor, we can see that a lot of objects are given to it. We also observe the use of the @... syntax (those are decorators)
  - In this case (and for most constructors), this is how VS Code does service (dependencies) injection
- One will also notice that most, if not all parameters have a visibility assigned to it. What this does is that it will create an associated property in the class as well as assigning the parameter value to this

property in the constructor. Thus, instead of writing

you simply write

- Upon its creation, the `CodeApplication` class will register various event listeners on the electron app object
- If we remember, in `src/vs/code/electron-main/main`, after the `CodeApplication` object is instantiated, we call `startup()` on it. So, we want to take a look at what that method does
- Without knowing too much about the VS Code source, it appears that we are instantiating an IPC server (inter-process communication) and then the shared process
- After that is done, we initialize some more services in `CodeApplication::initServices`, such as the update service (which I guess takes care of checking for VS Code updates) and the telemetry (data about VS Code feature usage)
- We finally get to the point where we're about to open a window in `CodeApplication::openFirstWindow`!
  - This leads us to go read the class `WindowsManager` in `src/vs/code/electron-main/windows.ts`. Once again, this file is pretty large, so we want to skim it to see what it contains (functions, classes, properties, methods)
- There are a few large classes in `src/vs/code/electron-main/windows.ts` that I'd want to extract to make the file smaller and simpler (less cognitive load). However, the issue is that those classes are not declared as exported, and thus are only available in the local file. It would be possible to move these classes to other files and import them, but by doing so it would also "communicate" that others can use it, which is what having the classes as not exported prevents, at the cost of making single files larger and harder to comprehend
- We know that the constructor is first called, then from `CodeApplication::openFirstWindow`, we see that `WindowsManager::ready` and `WindowsManager::open` are both called.
  - In the `constructor` we instantiate the `Dialogs` class (takes care of open/save dialog windows) and the `WorkspacesManager` class (takes care of workspace management, such as open/save)
  - In `ready` event listeners are registered
  - In `open` there is a lot of logic associated with the window finally opening

1.1 Notes

- If you start VS Code using the debug feature, you will not be able to open the Chrome DevTools (at this moment, 2018-05-26) because only 1 process is allowed to attach to the Chrome DevTools instance, and that process is the VS Code editor that started the debugged VS Code instance

2 2018-07-08

Today I want to find out how VS Code restores a windows sessions when you start it. Apparently, if you run it as `code .`, it will not restore the same set of windows than if you called it simply with `code`.

- In `src/vs/code/electron-main/launch.ts`, the `LaunchService::startOpenWindow` appears to implement logic based on how many arguments were given. In all cases, we end up doing a call to the `IWindowsMainService::open` method.
  - Note that in both cases, the path we're opening is within the `args` variable, which is passed to the `cli` property of the `IOpenConfiguration` object.
- The implementation of `IWindowsMainService` we are interested in lives in `src/vs/code/electron-main/windows.ts`.
- In the `WindowsManager::open` method, we rapidly discover that the windows that will be opened will be retrieved in `WindowsManager::getPathsToOpen`. In there, we can observe that the windows that will be opened depend on whether something was passed from the API, we forced an empty window, we're extracting paths from the cli or we should restore from the previous session.
  - If we arrive at this last case, we can see that the logic is to call `WindowsManager::doGetWindowsFromLastSession`, which is pretty self-explanatory, and will retrieve the previous set of windows from the last session. This is what happens when you start `code` using `code`

- In the case where we pass a path, this path is in `openConfig.cli._`. In this case, the windows that were previously opened, and part of `this.windowsState.openedWindows` (where `this` is a `WindowsManager` object)
  * Here we wonder how the `windowsState.openedWindows` state gets restored on VS Code start. To figure that out, we start at the `WindowsManager.constructor` method. There we find `this.windowsState = this.stateService.getItem<IWindowsState>(WindowsManager.windowsStateSto || { openedWindows: [] };`, which states to use get a `IWindowState` object from the `stateService` if one exists or to create an object with no opened windows. If we assume that this windows state is the same regardless of how we start VS Code, then it is not there that the difference in opened windows will occur.