Visual Studio Code consists of a layered and modular `core` (found as `src/vs`) that can be extended using extensions. Extensions are run in a separate process referred to as the `extension host.` Extensions are implemented by utilizing the extension API.

Layers

The `core` is partitioned into the following layers: - `base`: Provides general utilities and user interface building blocks. - `platform`: Defines service injection support and the base services for VS Code. - `editor`: The "Monaco" editor is available as a separate downloadable component. - `workbench`: Hosts the "Monaco" editor and provides the framework for "viewlets" like the Explorer, Status Bar, or Menu Bar, leveraging Electron to implement the VS Code desktop application. - `code`: The entry point to the desktop app that stitches everything together, this includes the Electron main file and the CLI for example.

Target Environments

The `core` of VS Code is fully implemented in TypeScript. Inside each layer the code is organized by the target runtime environment. This ensures that only the runtime specific APIs are used. In the code we distinguish between the following target environments: - `common`: Source code that only requires basic JavaScript APIs and run in all the other target environments - `browser`: Source code that requires the `browser` APIs like access to the DOM - may use code from: `common` - `node`: Source code that requires `nodejs` APIs - may use code from: `common` - `electron-sandbox`: Source code that requires the `browser` APIs like access to the DOM and a small subset of APIs to communicate with the Electron main process (anything exposed from `src/vs/base/parts/sandbox/electron-sandbox/globals.ts` - may use code from: `common`, `browser`, `electron-sandbox` - `electron-browser`: Source code that requires the Electron renderer-process APIs - may use code from: `common`, `browser`, `node` - `electron-main`: Source code that requires the Electron main-process APIs - may use code from: `common`, `node`

Dependency Injection

The code is organized around services of which most are defined in the `platform` layer. Services get to its clients via `constructor injection`.

A service definition is two parts: (1) the interface of a service, and (2) a service identifier - the latter is required because TypeScript doesn't use nominal but structural typing. A service identifier is a decoration (as proposed for ES7) and should have the same name as the service interface.

Declaring a service dependency happens by adding a corresponding decoration

to a constructor argument. In the snippet below `@IModelService` is the service identifier decoration and `IModelService` is the (optional) type annotation for this argument. When a dependency is optional, use the `@optional` decoration otherwise the instantiation service throws an error.

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

Use the instantiation service to create instances for service consumers, like so `instantiationService.createInstance(Client)`. Usually, this is done for you when being registered as a contribution, like a Viewlet or Language.

VS Code Editor source organization

- the `vs/editor` folder should not have any `node` or `electron-browser` dependencies.
- `vs/editor/common` and `vs/editor/browser` - the code editor core (critical code without which an editor does not make sense).
- `vs/editor/contrib` - code editor contributions that ship in both VS Code and the standalone editor. They depend on `browser` by convention and an editor can be crafted without them which results in the feature brought in being removed.
- `vs/editor/standalone` - code that ships only with the standalone editor. Nothing else should depend on `vs/editor/standalone`
- `vs/workbench/contrib/codeEditor` - code editor contributions that ship in VS Code.

Workbench Contrib

The VS Code workbench (`vs/workbench`) is composed of many things to provide a rich development experience. Examples include full text search, integrated git and debug. At its core, the workbench does not have direct dependencies to all these contributions. Instead, we use an internal (as opposed to real extension API) mechanism to contribute these contributions to the workbench.

Contributions that are contributed to the workbench all live inside the `vs/workbench/contrib` folder. There are some rules around this folder: - there cannot be any dependency from outside `vs/workbench/contrib` into `vs/workbench/contrib` - every contribution should expose its internal API from

a single file (e.g. `vs/workbench/contrib/search/common/search.ts`) - a contribution is allowed to depend on the internal API of another contribution (e.g. the git contribution may depend on `vs/workbench/contrib/search/common/search.ts`) - a contribution should never reach into the internals of another contribution (internal is anything inside a contribution that is not in the single common API file) - think twice before letting a contribution depend on another contribution: is that really needed and does it make sense? Can the dependency be avoided by using the workbench extensibility story maybe?