Extension Author Guide for Supporting Virtual Workspaces

We have recently announced the Remote Repository feature in the VS Code Insiders Build. It lets you browse and edit files and folders directly on GitHub.

Open Remote Repository... opens VSCode on a folder or workspace located on a virtual file system. We call this a **virtual workspace**. When a virtual workspace is open in a VS Code window, this is shown by a label in the remote indicator in the lower left corner, similar to remote windows.

By nature, not all extensions will be able to fully work with virtual workspaces. Some extensions use tools that rely on resources being available on disk, need synchronous file access or don't have the necessary file system abstractions. When in a virtual workspace, VS Code will indicate to the user the restricted mode and that some extensions are deactivated or work with limited functionality.

Still, we want to make sure as many extensions as possible work in that setup and that we have a good user experience with the **Remote Repository** feature as well as other features leveraging virtual resources - from connecting to ftp-servers to working with cloud resource.

This guide shows how extensions can test against virtual workspaces, helps adopting and introduces the new `virtualWorkspaces` capability property.

Is my extension affected?

When a extension has no code but is a pure theme, keybinding, snippets, grammar extension, then it can run in a virtual workspace and no adoption is necessary.

Extension with code, that means extensions that define a `main` entry point, require inspection and, possibly, adoption.

Run your extension against a virtual workspace

Run the **Open Remote Repository...** command from the Command Palette. **Notice** this command is currently only available in the VS Code Insiders version. The command shows a quick pick dialog and you can paste in any GitHub URL, or choose to search for a specific repository or pull request.

This opens a VSCode window for a virtual workspace where all resources are virtual.

Review that the code is ready for virtual resources

The API support for virtual file system has already been around for quite a while. You can check out the file system provider API, if you are interested. A file system provider is registered for a new URI scheme (e.g. `vscode-vfs`) and

resources on that file system will be represented by URIs using that schema (e.g. `vscode-vfs://github/microsoft/vscode/pacakge.json`)

Check how your extension deals with URIs it gets from the VSCode APIs:

- Never assume that the URI scheme is 'file'. `URI.fsPath` can only be used when the URI scheme is file.
- Look out for usages of the `fs` node module for file system operations. If possible, use the `vscode.workspace.fs` API, which delegates to the responsible file system provider.
- Check for third party components that depend on a `fs` access (e.g. a language server or a node module)
- If you run executables and tasks from commands, check whether these commands make sense in a virtual workspace window or whether they should be disabled.

Signal whether your extension can handle virtual workspaces

There's a new `capabilities` property in `package.json`, and `virtualWorkspaces` is used to signal whether an extension works with virtual workspace, or not.

No support for virtual workspaces

The example below declares that an extension does not support virtual workspaces and should not be enabled by VS Code in this setup.

```
{
  "capabilities": {
    "virtualWorkspaces": false
  }
}
```

With the latest insiders (1.57) it's now also possible to give a reason why the extension can not handle virtual workspaces:

```
{
  "capabilities": {
    "virtualWorkspaces": {
      "supported": false,
      "description": "Debugging is not possible in virtual workspaces."
    }
  }
}
```

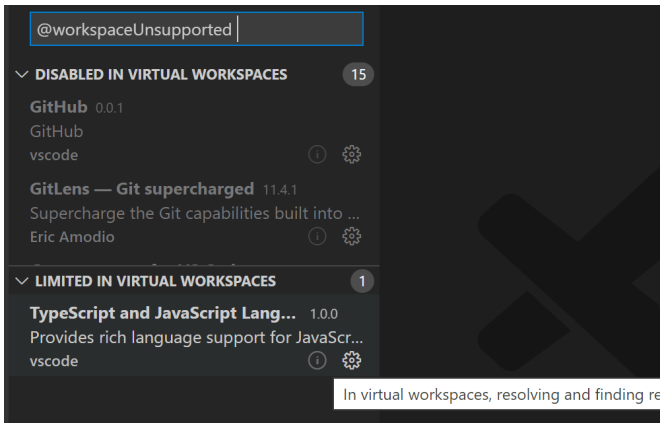
Partial and full support for virtual workspaces

When an extension works or partially works with virtual workspaces, then it should define `"virtualWorkspaces": true`.

```
{
  "capabilities": {
    "virtualWorkspaces": true
  }
}
```

With the latest insiders (1.57) it's now also possible to give a more information on what doesn't work:

```
{
  "capabilities": {
    "virtualWorkspaces": {
      "supported": "limited",
      "description": "In virtual workspaces, resolving and finding references across files i
    }
  }
}
```



The description is will be shown in the extensions view:

The extension should then disable the features that are not supported in a virtual workspace as described below.

Default

`"virtualWorkspaces": true` is the default for all extensions that have no yet filled in the `virtualWorkspaces` capability.

However, when testing, we came up list of extensions that we think should be disabled in virtual workspaces. The list can be found here. These extensions have `"virtualWorkspaces": false` as default.

Of course, extension authors are in a better position to make this decision. The `virtualWorkspaces` capability in the `package.json` will override our default and we will eventually retire our list.

Disable functionality when a virtual workspace is opened

Disable commands and view contributions

The availability of commands and views and many other contributions can be controlled through context keys in `when` clauses.

The `virtualWorkspace` context key is set when all workspace folders are located on virtual file systems. The example below shows the command `npm.publish` in the command palette only when not in a virtual workspace:

```
{
    "menus": {
      "commandPalette": [
        {
          "command": "npm.publish",
          "when": "!virtualWorkspace"
        }
      ]
    }
}
```

The `resourceScheme` context key is set to the URI scheme of the currently selected element in the explorer or the element open in the editor. In this example the `npm.runSelectedScript` command is only in the editor context menu if the underlying resource is on the local disk.

```
{
    "menus": {
      "editor/context": [
        {
          "command": "npm.runSelectedScript",
          "when": "resourceFilename == 'package.json' && resourceScheme == file"
        }
      ]
    }
}
```

Detect virtual workspaces in code

To check in code whether the current workspace consists of non-`file` schemes and is virtual you can use

```
const isVirtualWorkspace = workspace.workspaceFolders && workspace.workspaceFolders.every(f
```

Language Extensions and Virtual Workspaces

What are the expectations for language support with virtual workspaces?

It's not realistic that all extensions are able to fully work with virtual resources. Many extensions built are on tools that require synchronous file access and files on disk. It's therefore ok to only provide limited functionality, such as the 'Basic' and the 'Single-File' support as listed below.

A. Basic Language Support: * TextMate tokenization and colorization, * language specific editing support: bracket pairs, comments, on enter rules, folding markers * snippets

B. Single-File Language Support: * document symbols (outline), folding, selection ranges * document highlights, semantic highlighting, document colors * completions, hovers, signature help, find references/declarations based on symbols on the current file and on static language libraries * formatting, linked editing * syntax validation and same-file semantic validation and code actions

C. Cross-file, Workspace Aware Language Support * references across files * workspace symbols * validation of all files in the workspace/project

The rich language extensions that ship with VS Code (TypeScript, JSON, CSS, HTML, Markdown) are limited to Single-File Language Support when working on virtual resources.

We are working on a UI for VS Code to indicate that the Window runs in a restricted mode.

Disabling a language extension

If working on a single file is not option, language extensions can also decide to disable the extension when in a virtual workspaces.

If your extension provides both grammars and rich language support and have to disable the extension also the grammars will be disabled. To avoid this, we recommend to split off a basic language extension (grammars, language configuration, snippets) form the rich language support and have two extensions. - The basic language extension has `"virtualWorkspaces": true` and provides language id, configuration, grammar and snippets. - The rich language extension has `"virtualWorkspaces": false` contains the main file contributing language supports and commands and has a extension dependency (`extensionDependencies`) on the basic language extension. The rich language extension should keep the ID of the established extension, so the user will continue to the full functionality by installing a single extension.

You can see this with the built-in language extensions, such as JSON, which consists of a JSON extension and a JSON language feature extension.

This separation has also helps with untrusted workspaces.

Language selectors

When registering a provider for a language feature (e.g. completions, hovers, code actions..) make sure to specify the schemes the provider supports:

```
return vscode.languages.registerCompletionItemProvider({ language: 'typescript', scheme: 'fi
    provideCompletionItems(document, position, token) {
        // ...
    }
});
```

What about support in the language server protocol (LSP) for accessing virtual resources

Work is under way that will add FS support to LSP. Tracked in https://github.com/microsoft/language-server-protocol/issues/1264.

More information and feedback

Please comment in the Virtual Workspaces Tracking Issue if you have questions and suggestions.