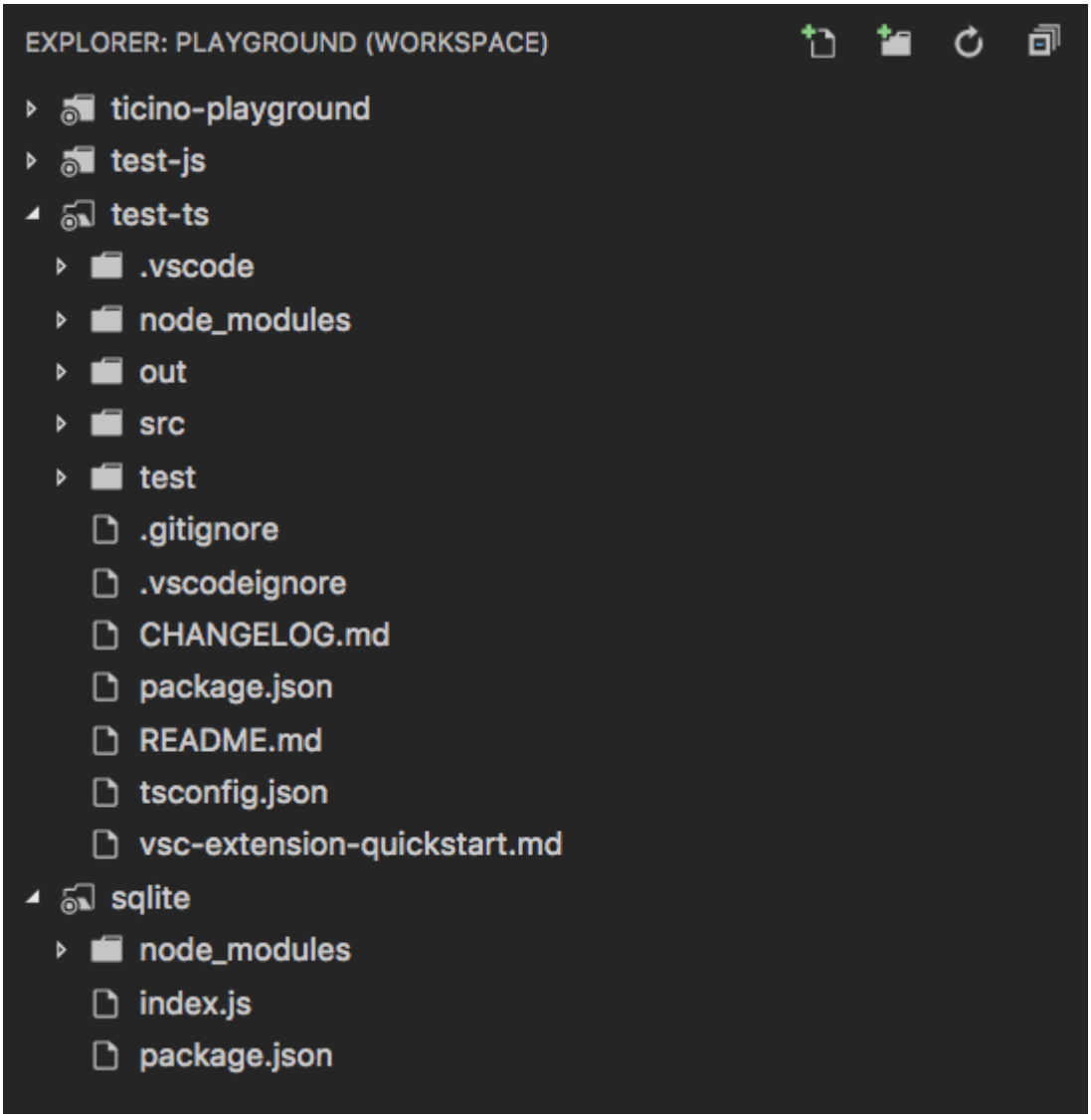
> ***Request:*** *If you have prepared your extension for Multi-root Workspaces, we would love you to add* `multi-root ready` *to your* `package.json`*, i.e.* `"keywords": ["multi-root ready"]`*. If you upgrade to* `VSCE 1.32.0` *this can be added to any existing keywords and will not be counted against the maximum of 5 allowed keywords.*

Synopsis

This wiki page explains the impact of the new multi-root-workspace feature on extensions. You will learn the concepts, new APIs and understand if and how to make your extension ready for multi-root-workspace support.

A multi-root workspace is a new way how to work with Visual Studio Code. At its core, it allows opening multiple folders in the same instance from any location that is accessible on disk. All parts of the VS Code user interface adapt to the number of folders opened, e.g. the file explorer will show all folders in a single tree and the full-text search will search across all folders. It is in the best interest of the users that extensions also adapt to supporting multiple folders.



There are numerous ways how to create a multi-root workspace, the simplest one is to just open multiple folders from the command line:

```
code-insiders <folder1> <folder2>...
```

All workspace metadata is stored in a simple JSON file that can be saved and shared (`File | Save Workspace As...`) with others:

```
{
  "folders": [
    {
      "path": "some-path"
    },
    {
      "path": "some-other-path"
    }
  ],
  "settings": {
    "...any settings..."
  }
}
```

In its most simple form, a VS Code workspace is just a collection of folders (called `WorkspaceFolder` in the API) and associated settings that should apply whenever the workspace is opened. Each `WorkspaceFolder` can have additional metadata associated (for example a `name` property).

This guide will help you as an extension author to make your extension ready for multi-root workspaces. It touches on three major pieces (basics, settings, language client/server) and is joined by samples from our samples repository.

Do I need to do anything?

Here is a simple checklist:

- If your extension is making use of the (now deprecated) `workspace.rootPath` property to work on the currently opened folder, then you are affected. See the section 'Eliminating workspace.rootPath below'.
- If your extension contributes settings, then you should review whether some of the settings can be applied on a resource (= file location) level instead of being global. Resource settings are more powerful because a user can choose to configure settings differently per workspace folder. Similarly, if you do not contribute settings but you modify settings programmatically, then you should review that you modify the settings using the proper scope. See the 'Settings' section below.
- If you are implementing a language server, then you are affected since up to now a language server only had to handle a single folder. In the new multi-folder setup, a language server should be able to handle multiple folders. See the section 'Language Client/Language Server' below.
- If you are implementing a debug adapter, then you are most likely not affected because a debug adapter cannot access the extension APIs (like `workspace.rootPath`) and debug configuration variables like `${workspaceFolder}` (or the deprecated `${workspaceRoot}`) are already resolved into full paths before they are passed into the debug adapter.

Eliminating `rootPath`

The basic APIs to work with multi-root workspaces are:

Method	Description
`workspace.workspaceFolders`	Access to all `WorkspaceFolder` (can be `undefined` when no workspace is opened!).
`workspace.onDidChangeWorkspaceFolders`	Be notified when `WorkspaceFolder` are added or removed.
`workspace.getWorkspaceFolder(uri)`	Get the `WorkspaceFolder` for a given resource (can be `undefined` when a resource is not part of any workspace folder!).

Your extension should be capable of working with any number of `WorkspaceFolder`, including 0, 1, or many folders. The `WorkspaceFoldersChangeEvent` carries information about the added or removed `WorkspaceFolder`. To find out to which `WorkspaceFolder` a given resource belongs to, use the `workspace.getWorkspaceFolder(uri)` method.

Note: Using `workspace.getWorkspaceFolder(uri)` with the URI of a `WorkspaceFolder` will return that instance of the `WorkspaceFolder`.

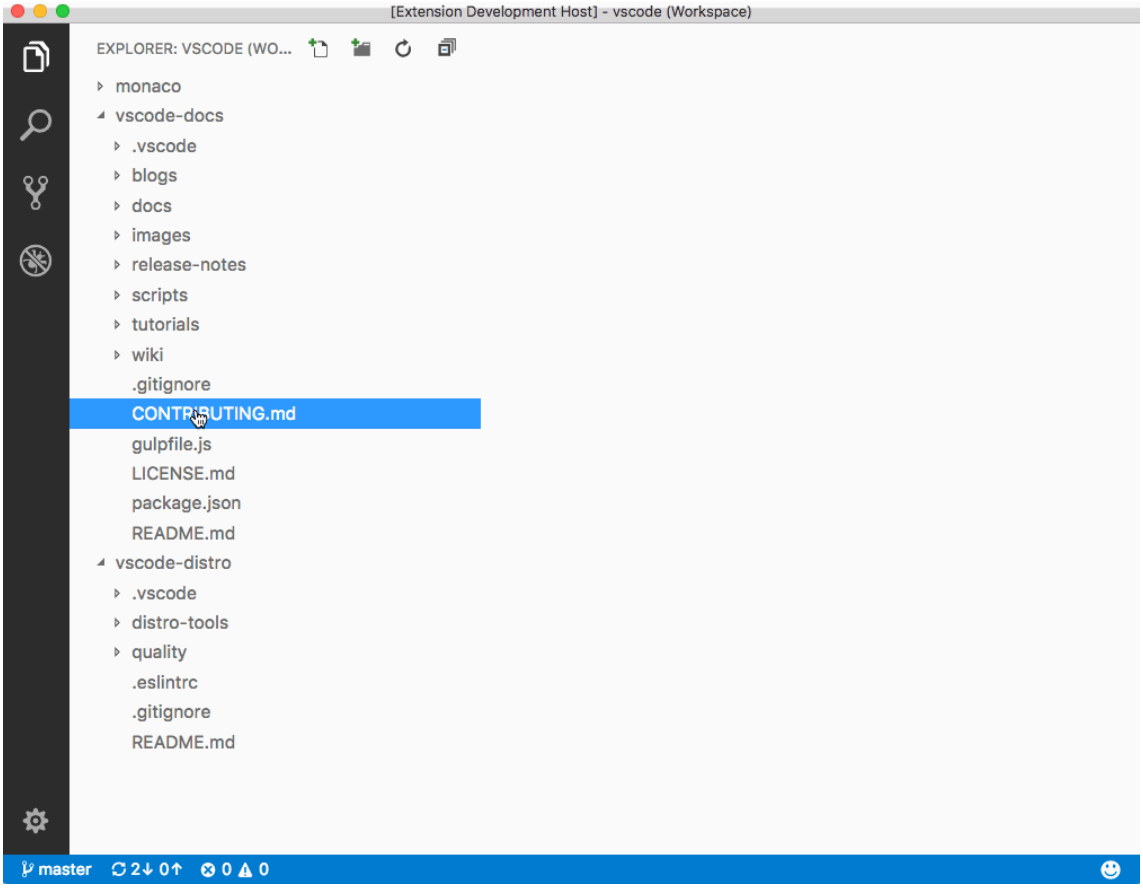
Each `WorkspaceFolder` provides access to some metadata:

Property	Description
`uri`	The associated uri for this workspace folder. The Uri-type was intentionally chosen such that future releases of the editor can support workspace folders that are not stored on the local disk, e.g. `ftp://server/workspaces/foo`.
`index`	The 0-based index of the folder as configured by the user.
`name`	The name of the folder (defaults to the folder name).

Note 1: A user is free to configure folders for a workspace that are overlapping. E.g. a workspace can consist of a parent folder as well as any of its children. It is up to the extension to be clever here and avoid duplicate work. For example, a task that scans all files of a folder should not duplicate the work by scanning again for a child folder, if any.

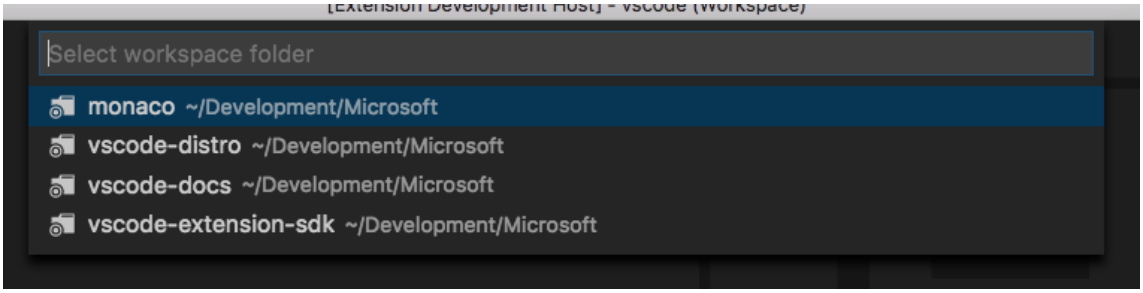
Note 2: A workspace folder might use a `uri` which does *not* resolve to a file on disk. So, it must not always be a `file`-uri, but VS Code will soon support workspace folders from remote locations.

The `basic-multi-root-sample` extension is demonstrating the use of this API by showing the workspace folder of the currently opened file on the left-hand side of the status bar.



New helpful API

In order to make it easier to work with multiple-roots from your extension, we added some new API. For example, if you need to ask the user for a specific `WorkspaceFolder`, you can use the new `showWorkspaceFolderPick` method that will open a picker and returns the result. This method will return `undefined` in case the user did not pick any folder or in case no workspace is opened.



In addition, we introduced the `RelativePattern` type and support it in the API where we ask for glob patterns to match on file paths. There may be scenarios where you want to restrict the glob matching on a specific `WorkspaceFolder` and `RelativePattern` allows this by providing a `base` for the pattern to match against. You can use `RelativePattern` in:

- `workspace.createFileSystemWatcher(pattern)`

- `workspace.findFiles(include, exclude))`
- `DocumentFilter#pattern`

The type is a class as follows:

```
/**
 * A relative pattern is a helper to construct glob patterns that are matched
 * relatively to a base path. The base path can either be an absolute file path
 * or a [workspace folder](#WorkspaceFolder).
 */
class RelativePattern {

  /**
   * A base file path to which this pattern will be matched against relatively.
   */
  base: string;

  /**
   * A file glob pattern like `*.{ts,js}` that will be matched on file paths
   * relative to the base path.
   *
   * Example: Given a base of `/home/work/folder` and a file path of
`/home/work/folder/index.js`,
   * the file glob pattern will match on `index.js`.
   */
  pattern: string;

  /**
   * Creates a new relative pattern object with a base path and pattern to match.
This pattern
   * will be matched on file paths relative to the base path.
   *
   * @param base A base file path to which this pattern will be matched against
relatively.
   * @param pattern A file glob pattern like `*.{ts,js}` that will be matched on
file paths
   * relative to the base path.
   */
  constructor(base: WorkspaceFolder | string, pattern: string)
}
```

You can create a relative pattern via the following call:

```
// Construct a relative pattern for the first root folder
const relativePattern = new
vscode.RelativePattern(vscode.workspace.workspaceFolders[0], '*.ts');
```

Settings

With the introduction of multi-root workspaces, we also revisited how settings can apply. We differentiate between where a setting is persisted and how a setting applies.

Settings can be stored in various locations:

Location	Description
User Data	Global `settings.json` file in the user data directory that applies to any VS Code instance.
Workspace File (*new*)	Settings stored within the `.code-workspace` file of a multi-root workspace, which applies whenever the workspace is opened.
Workspace Folder	Settings stored inside a `.vscode` folder of a workspace folder, which applies depending on opening the folder or a workspace with that folder (see below).

Let's have a closer look at **Workspace Folder** settings that are stored within the `.vscode` folder: If you are opening just that folder in VS Code, all the settings of this folder apply to VS Code as before. However, once you make this folder part of a multi-root workspace, the situation is different. We no longer support all settings in this setup simply because you could have multiple folders configured in the workspace, each having settings that could potentially conflict.

To solve this, we distinguish between scopes a setting can have. You as extension author have to make the decision which scope applies to your settings:

Scope	Description
`window`	The setting is applied to the entire VS Code instance.
`resource`	The setting is applied depending on an active resource.

By default, all settings have the `window` scope, however, we encourage you to support settings on the `resource` scope. Settings that apply on the window level are not supported once they are defined within a workspace folder, and as soon as the user entered a multi-root workspace. Settings that apply on a resource level however are supported, and as such, each workspace folder can have different values for these settings.

Note: When a setting is defined as a `resource` scoped setting, then you have to use the new configuration API that fetches the setting that applies to a resource (see the Settings API section below).

Settings Configuration

To declare a setting scope, simply define the scope as part of your setting from the `package.json` file. The example below is copied from the Configuration Sample extension:

```
"contributes": {
  "configuration": [
    {
      "title": "Configuration Samples",
      "properties": {
        "conf.view.showOnWindowOpen": {
          "type": "string",
          "enum": [
            "explorer",
            "search",
```

```
            "scm",
            "debug",
            "extensions"
          ],
          "default": "explorer",
          "description": "Window configuration: View to show always when a window
opens",
          "scope": "window"
        },
        "conf.resource.insertEmptyLastLine": {
          "type": "object",
          "default": {},
          "description": "Resource configuration: Configure files using glob
patterns to have an empty last line always",
          "scope": "resource"
        }
      }
    }
  ]
}
```

Settings API

The configuration example above defines a setting scoped to a resource. To fetch its value you use the `workspace.getConfiguration()` API and pass the URI of the resource as the second parameter. You can see Example 3 how the setting is used in the basic-multi-root sample.

Under the hood, resource settings are resolved with simple logic: We try to find a `WorkspaceFolder` for the resource that is passed into the `getConfiguration` API. If such a folder exists, and that folder defines the setting, it will be returned. Otherwise, the normal logic applies for finding the setting on a parent level: it could be defined within the workspace file or on the user level.

Note: You do not have to be aware if the user has opened a workspace or not when using the `getConfiguration` API with resource scope. Just make sure to always pass the resource scope URI around and we will do the resolution of the setting based on the user's setup.

Perspectives

An extension author, you should have the following two perspectives while defining a setting.

1. **End User:** Given a setting, an end-user should know where he/she can customize this setting. By defining a setting as `resource` or `window` scoped, the user can be able to customize it at the right targets. It means, User can open settings and can customize a window (`window.zoomLevel`) or resource (`editor.wordWrap`) setting in User/Workspace Settings. But when a user lands into Folder Settings, the user can only customize resource settings (`editor.wordWrap`). VS Code will use the setting's classification information to provide the right proposals in intelli-sense and will warn the user if customizing `window` settings in Folder settings.

2. **Extension author:** Extension author's main purpose is to define a setting and read its value and apply it. As mentioned before now, there is a new target `Folder Settings` where a resource scoped setting can only be customizable. So extension author should be knowing if a setting is associated with a resource or not and thereby, classify the setting. If it is a resource setting, ask for the value of the setting by passing the

resource for which the value has to be applied. Otherwise, you can just ask for the value without passing any resource. API will give you back the value user-customized for this setting.

Run time Diagnostics

While extension development, we provide some run time assistance by logging warnings when we detect the extension is accessing configuration not in an expected way. Following are the diagnostics we show and actions to be taken when you see them.

> *[ext.name]: Accessing a resource scoped configuration without providing a resource is not expected. To get the effective value for '${key}', provide the URI of a resource or 'null' for any resource.*

It is suggested to pass in a resource when you are accessing a resource scoped configuration. In case of getting the value for any resource, pass `null` for the resource.

> *[ext.name]: Accessing a window scoped configuration for a resource is not expected. To associate '${key}' to a resource, define its scope to 'resource' in configuration contributions in 'package.json'.*

- If they are not resource scoped settings, you do not need to pass in a resource while accessing the setting.
- If they are resource scoped settings, you have to define them as Resource scoped while registering in the package.json.

Refer to Configuration Sample extension for more information.

Language Client / Language Server

Since language servers usually act on a workspace, they are also affected by the introduction of multi-root workspaces. A language server extension needs to check for the following items and adopt its code accordingly:

- If the server accesses the `rootPath` or `rootURI` property of the `InitializeParams` passed in the `initialize` request, then the language server must instead use the property `workspaceFolders`. Since workspace folders can come and go dynamically the server also needs to register for `workspace/didChangeWorkspaceFolders` notifications. The documentation can be found here.
- If the server is using configuration settings the author also has to review the scope of the settings (see the Settings section above). For a language server, 'resource' scope is typically preferred since it enables a user to configure that language settings on a per-folder basis. In addition, the server has to change the way settings are accessed, (see the Language Settings section below).

Single language server or server per folder?

The first decision the author of a language server must make is whether a single server can handle all the folders in a multi-root folder setup, or whether you want to run a server per root folder. Using a single server is the recommended way. Sharing a single server is more resource-friendly than running multiple servers.

Language servers that operate on a single file, like a CSS or JSON file, or linters that validate a single file at a time, can easily handle multiple folders. We have already migrated the HTML, JSON, CSS language servers and the 'eslint' and 'tslint' Language servers. We have implemented them all as a single language server. The lsp-sample example demonstrates the use of the new protocol workspace folder and configuration protocol. We recommend that you also review the corresponding extensions when migrating your server.

Language servers that operate on multiple files with interdependencies can be different and you need to evaluate whether you want to start a server per workspace folder to isolate folders from each other. We have migrated the

TypeScript/JavaScript language server, and we were able to use a single server for multiple folders. We added support to this to the `vscode-languageclient` library, and the lsp-multi-server-sample example demonstrates the use of a server per folder.

Language settings

In the current version of the language server protocol, the client pushes settings to the server. With the introduction of a resource scope, this is not possible anymore since the actual settings values can depend on a resource. We introduce a protocol which allows servers to fetch settings from a client comparable to the `getConfiguration` API in the extension host. The protocol addition is documented here. The bundled CSS or HTML language extensions illustrate this approach.

A setting that supports file paths relative to a workspace needs special treatment. In the single folder case, the language server process is started in the root folder of the workspace. A relative path can then be resolved easily since the server's home directory corresponds to the workspace root. In the multi-root folder setup, this is no longer the case, and when the setting is defined as a resource scoped setting, then the path needs to be resolved per root folder. One approach that has worked well for us is to resolve the relative file path of a setting on the client using the new settings API. In this way, the server only sees absolute paths. An example for this can be found in vscode-tslint. It uses the `vscode-languageclient` middleware to transform the relative file paths.

Samples

Sample	Description
Basic multi root	Illustrates the `WorkspaceFolder` API and how to work with a resource scoped configuration setting.
Language Server Settings	Demonstrates how to handle configuration settings in a language server.
Multi Language Server	Starts a language server for each root folder in a workspace.

Writing Tests

Today we do not provide any extension API to add or remove workspace folders. However, you are free to run your extension tests against a `code-workspace.json` file that can contain any number of workspace folders. Modifying this file programmatically will cause the workspace to change, however, you have to account for the changes to the file bubbling through to VS Code for the update to happen (this can take up to a second).

To start your extension tests on a workspace, open the `launch.json` and add the path to the file to the `args` property:

```
{
    "name": "Launch Extension",
    "type": "extensionHost",
    "request": "launch",
    "runtimeExecutable": "${execPath}",
    "args": ["${workspaceRoot}/test/test.code-workspace", "--
extensionDevelopmentPath=${workspaceRoot}" ],
    "stopOnEntry": false,
    "sourceMaps": true,
```

```
    "outFiles": [ "${workspaceRoot}/out/src/**/*.js" ],
    "preLaunchTask": "npm"
}
```