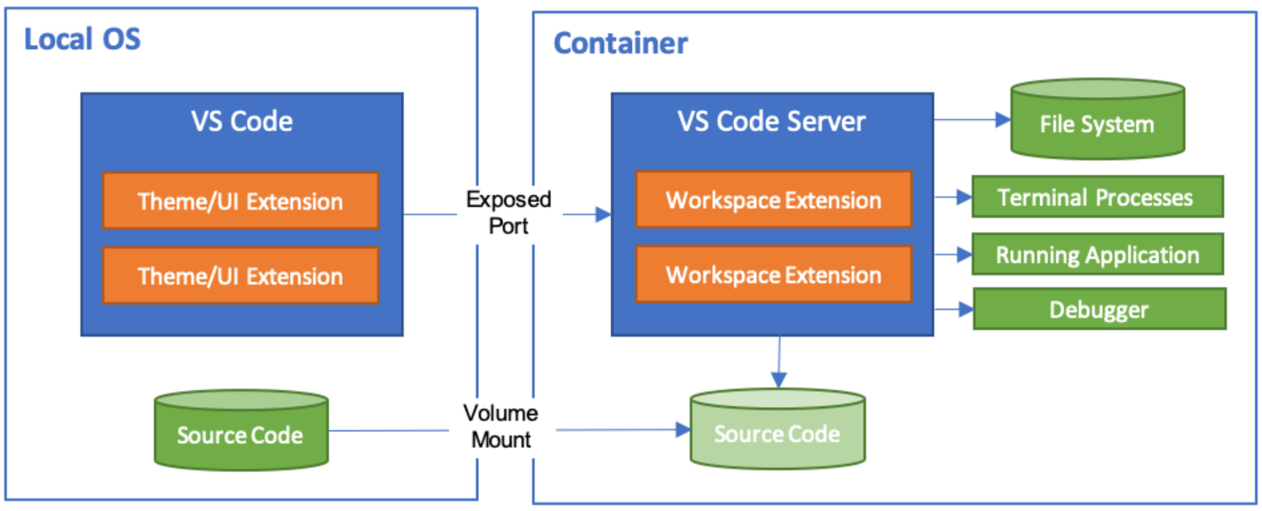
Developing inside a Container

(https://github.com/Microsoft/vscode-docs/blob/master/docs/remote/containers.md)

The **Visual Studio Code Remote - Containers** extension lets you use a Docker container (https://docker.com) as a full-featured development environment. It allows you to open any folder inside (or mounted into) a container and take advantage of Visual Studio Code's full feature set. A `devcontainer.json` file in your project tells VS Code how to access (or create) a **development container** with a well-defined tool and runtime stack. This container can be used to run an application or to sandbox tools, libraries, or runtimes needed for working with a codebase.

Workspace files are mounted from the local file system or copied or cloned into the container. Extensions are installed and run inside the container, where they have full access to the tools, platform, and file system. This means that you can seamlessly switch your entire development environment just by connecting to a different container.



This lets VS Code provide a **local-quality development experience** — including full IntelliSense (completions), code navigation, and debugging — **regardless of where your tools (or code) are located**.

Getting started

System requirements

Local: Docker Desktop 2.0+ for macOS/Windows 10 Pro/Enterprise or Docker CE/EE 18.06+ and Docker Compose 1.21+ for Linux. Docker Toolbox and Ubuntu snap packages are not supported. See the minimum requirements for VS Code (/docs/supporting/requirements) for additional details.

Containers:

- **Full support:** x86_64 Debian 8+, Ubuntu 16.04+, CentOS / RHEL 7+ based containers
- **Experimental support:** x86_64 Alpine Linux containers in Visual Studio Code Insiders (https://code.visualstudio.com/insiders/) only.

Other `glibc` based Linux containers may work if they have needed prerequisites (/docs/remote/linux).

While experimental `musl` based Alpine Linux support is available in VS Code Insiders (https://code.visualstudio.com/insiders/), some extensions installed in the container may not work due to `glibc` dependencies in native code inside the extension. See the Remote Development with Linux (/docs/remote/linux) article for details.

Installation

To get started, follow these steps:

1. Install and configure Docker (https://www.docker.com/get-started) for your operating system.

   Windows / macOS:

   1. Install Docker Desktop for Windows/Mac (https://www.docker.com/products/docker-desktop). (Docker Toolbox is not currently supported.)
   2. Right-click on the Docker taskbar item and update **Settings / Preferences** > **Shared Drives / File Sharing** with any source code locations you want to open in a container. If you run into trouble, see Docker Desktop for Windows tips (/docs/remote/troubleshooting#_docker-desktop-for-windows-tips) on avoiding common problems with sharing.
   3. **Windows**: Consider adding a `.gitattributes` file to your repos or disabling automatic line ending conversion for Git on the **Windows side** by using a command prompt to run: `git config --global core.autocrlf input` If left enabled, this setting can cause files that you have not edited to appear modified due to line ending differences. See Tips and Tricks (/docs/remote/troubleshooting#_resolving-git-line-ending-issues-in-containers-resulting-in-many-modified-files) for details.

   Linux:

   1. Follow the official install instructions for Docker CE/EE for your distribution (https://docs.docker.com/install/#supported-platforms). If you are using Docker Compose, follow the Install Docker Compose directions (https://docs.docker.com/compose/install/) as well. (Note that the Ubuntu Snap package is not supported and packages in distributions may be out of date.)
   2. Add your user to the `docker` group by using a terminal to run: `sudo usermod -aG docker $USER`
   3. Sign out and back in again so your changes take effect.
2. Install Visual Studio Code (https://code.visualstudio.com/) or Visual Studio Code Insiders (https://code.visualstudio.com/insiders/).
3. Install the Remote Development extension pack (https://aka.ms/vscode-remote/download/extension).

The Remote - Containers extension supports two primary operating models:

- You can use a container as your full-time development environment.
- You can attach to a running container to inspect it.

We will cover how to use a container as your full-time development environment first.

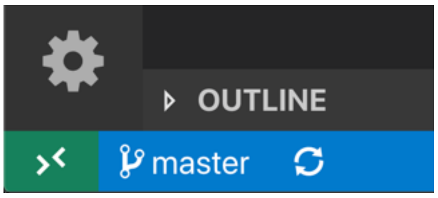
Quick start: Try a dev container

Let's start by using a sample project to try things out.

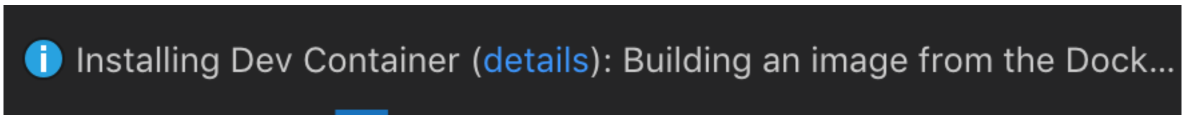
1. Clone one of the sample repositories below.

```
git clone https://github.com/Microsoft/vscode-remote-try-node
git clone https://github.com/Microsoft/vscode-remote-try-python
git clone https://github.com/Microsoft/vscode-remote-try-go
git clone https://github.com/Microsoft/vscode-remote-try-java
git clone https://github.com/Microsoft/vscode-remote-try-dotnetcore
git clone https://github.com/Microsoft/vscode-remote-try-php
git clone https://github.com/Microsoft/vscode-remote-try-rust
git clone https://github.com/Microsoft/vscode-remote-try-cpp
```

2. Start VS Code and click on the quick actions Status Bar item in the lower left corner of the window.



3. Select **Remote-Containers: Open Folder in Container...** from the command list that appears, and open the root folder of the project you just cloned.
4. The window will then reload, but since the container does not exist yet, VS Code will create one. This may take some time, and a progress notification will provide status updates. Fortunately, this step isn't necessary the next time you open the folder since the container will already exist.



Installing Dev Container (details): Building an image from the Dock...

5. After the container is built, VS Code automatically connects to it and maps the project folder from your local file system into the container. Check out the **Things to try** section of `README.md` in the repository you cloned to see what to do next.

Quick start: Open an existing folder in a container

Next we will cover how to set up a dev container for an existing project to use as your full-time development environment.

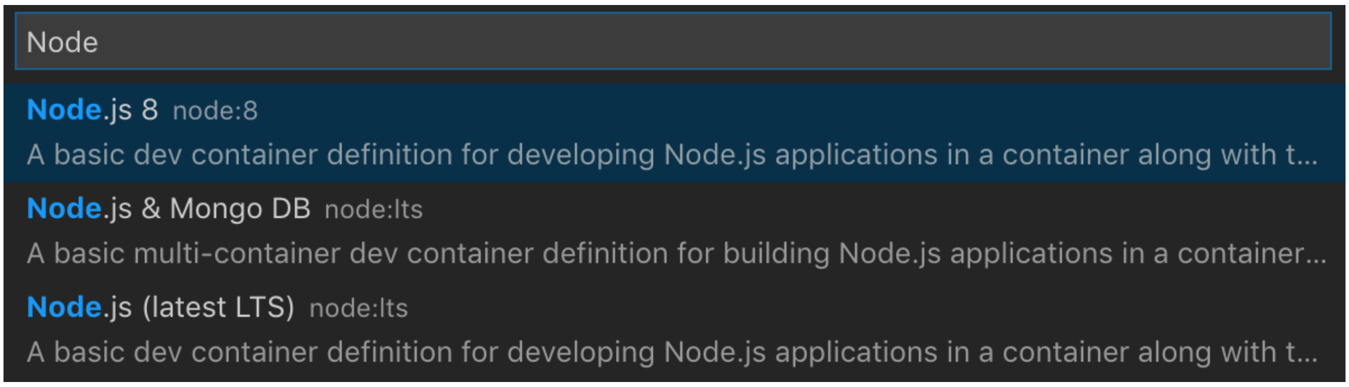
The steps are similar to those above:

1. Start VS Code, run the **Remote-Containers: Open Folder in Container...** command from the Command Palette, and select the project folder you'd like to set up the container for.

> **Note:** If you want to edit the container's contents or settings before opening the folder, you can run **Remote-Containers: Add Development Container Configuration Files...** instead.

2. Now pick a starting point for your dev container. You can either select a base **dev container definition** from a filterable list, or use an existing Dockerfile (https://docs.docker.com/engine/reference/builder/) or Docker Compose file (https://docs.docker.com/compose/compose-file/#compose-file-structure-and-examples) if one exists in the folder you selected.

> **Note:** See System requirements for information on supported containers. When using experimental support for Alpine Linux in Visual Studio Code Insiders (https://code.visualstudio.com/insiders/), note that some extensions installed in the container may not work due to `glibc` dependencies in native code inside the extension.


Node

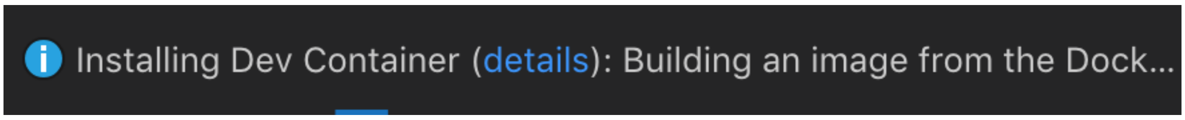
Node.js 8 node:8
A basic dev container definition for developing Node.js applications in a container along with t...

Node.js & Mongo DB node:lts
A basic multi-container dev container definition for building Node.js applications in a container...

Node.js (latest LTS) node:lts
A basic dev container definition for developing Node.js applications in a container along with t...

Note the dev container definitions displayed come from the vscode-dev-containers repository (https://aka.ms/vscode-dev-containers). You can browse the `containers` folder of that repository to see the contents of each definition.

3. After picking the starting point for your container, VS Code will add the dev container configuration files to your project (`.devcontainer/devcontainer.json`). The VS Code window will reload and start building the dev container. A progress notification provides you status updates. Note that you only have to build a dev container the first time you open it; opening the folder after the first successful build will be much quicker.


Installing Dev Container (details): Building an image from the Dock...

4. After building completes, VS Code will automatically connect to the container. You can now interact with your project in VS Code just as you could when opening the project locally. From now on, when you open the project folder, VS Code will automatically pick up and reuse your dev container configuration.

Creating a devcontainer.json file

VS Code's container configuration is stored in a `devcontainer.json` file. This file is similar to the `launch.json` file for debugging configurations, but is used for launching (or attaching to) your development container instead. You can also specify any extensions to install once the container is running or post-create commands to prepare the environment. The dev container configuration is either located under `.devcontainer/devcontainer.json` or stored as a `.devcontainer.json` file (note the dot-prefix) in the root of your project.

The **Remote-Containers: Add Development Container Configuration Files...** command adds the file to your project, where you can further customize for your needs.

For example, through a `devcontainer.json` file, you can:

- Spin up a stand-alone "sandbox" container to isolate your toolchain or speed up setup.
- Work with a container deployed application defined by an image, Dockerfile, or Docker Compose.
- Use Docker or Kubernetes (/docs/remote/containers-advanced#_using-docker-or-kubernetes-from-a-container) from inside a dev container to build and deploy your app.

The vscode-dev-containers repository (https://aka.ms/vscode-dev-containers) has examples of `devcontainer.json` for different scenarios. You can alter your configuration to:

- Install additional tools such as Git in the container.
- Automatically install extensions.
- Expose additional ports.
- Set runtime arguments.
- Reuse or extend your existing Docker Compose setup (https://aka.ms/vscode-remote/containers/docker-compose/extend).
- And more advanced container configurations (/docs/remote/containers-advanced).

Note that, if `devcontainer.json` 's supported workflows supports do not meet your needs, you can also attach to an already running container instead.

Configuration edit loop

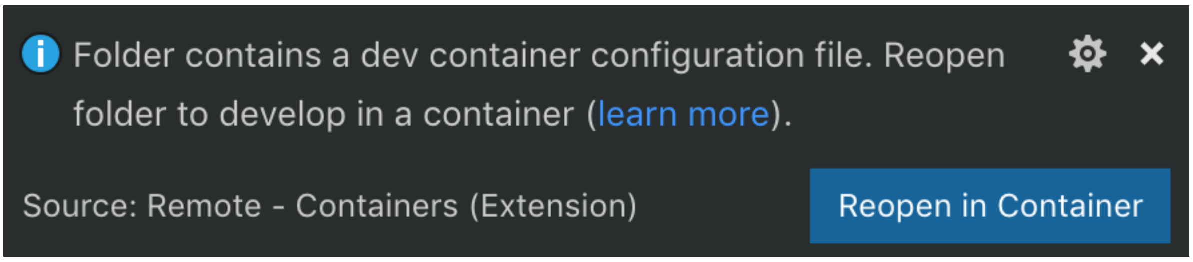
Editing your container configuration is easy. Since rebuilding a container will "reset" the container to its starting contents (with the exception of your local source code), VS Code does not automatically rebuild if you edit a container configuration file (`devcontainer.json` , `Dockerfile` , `docker-compose.yml`). Instead, there are several commands that can be used to make editing your configuration easier.

Here is the typical edit loop using these commands:

1. Start with **Remote-Containers: Add Development Container Configuration Files...** in the Command Palette (`F1`).
2. Edit the contents of the `.devcontainer` folder as required.
3. Try it with **Remote-Containers: Reopen Folder in Container**.
4. On failure:
   1. **Remote-Containers: Reopen Folder Locally**, which will open a new local window.
   2. In this local window: Edit the contents of the `.devcontainer` folder as required.
   3. Try it again: Go back to the container window, **Developer: Reload Window** from the Command Palette (`F1`)
   4. Repeat as needed.
5. If the build was successful, but you want to make more changes:
   1. Edit the contents of the `.devcontainer` folder as required when connected to the container.
   2. **Remote-Containers: Rebuild Container**.
   3. On failure: Follow the same workflow above.

Adding configuration files to public or private repositories

You can easily share a customized dev container definition for your project by adding `devcontainer.json` files to source control. By including these files in your repository, anyone that opens a local copy of your repo in VS Code will be automatically prompted to reopen the folder in a container, provided they have the Remote - Containers extension installed.


Folder contains a dev container configuration file. Reopen folder to develop in a container (learn more).

Source: Remote - Containers (Extension)

Beyond the advantages of having your team use a consistent environment and tool-chain, this also makes it easier for new contributors or team members to be productive quickly. First-time contributors will require less guidance and hit fewer issues related to environment setup.

Attaching to running containers

While using VS Code to spin up a new container can be useful in many situations, it may not match your workflow and you may prefer to "attach" VS Code to an already running container.

While `devcontainer.json` is not used in this case, you are able to use the same capabilities provided by the extension once connected. You can also use `settings.json` to specify extensions that should always be installed when you attach to a container to speed up setup.

> **Note:** See System requirements for information about supported container. When using experimental support for Alpine Linux in Visual Studio Code Insiders (https://code.visualstudio.com/insiders/), note that some extensions installed in the container may not work due to `glibc` dependencies in native code inside the extension.

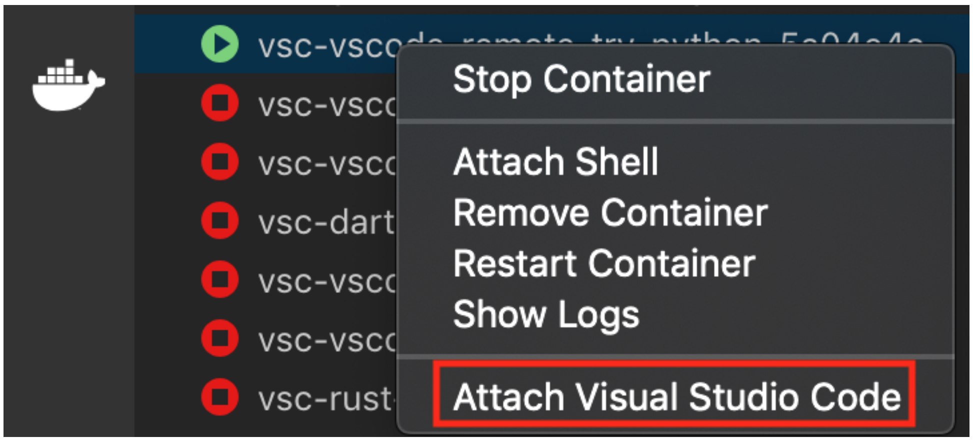
Once you have a container up and running, you can connect by either:

Option 1: Use the Attach to Running Container command

Run **Remote-Containers: Attach to Running Container...** command from the Command Palette (`F1`) and selecting a container.

Option 2: Use the Docker extension

1. Install the Docker extension (https://marketplace.visualstudio.com/items?itemName=ms-azuretools.vscode-docker) from the Extensions view (search for "docker") if it is not already installed.
2. Go to the Docker view and expand the **Containers** node in the explorer.
3. Right-click and select **Attach Visual Studio Code**.



After a brief moment, a new window will appear and you'll be connected to the running container.

Managing containers

By default, the Remote - Containers extension automatically starts the containers mentioned in the `devcontainer.json` when you open the folder. When you close VS Code, the extension automatically shuts down the containers you've connected to. You can change this behavior by adding `"shutdownAction": "none"` to `devcontainer.json` .

You can also manually manage your containers using one of the following options:

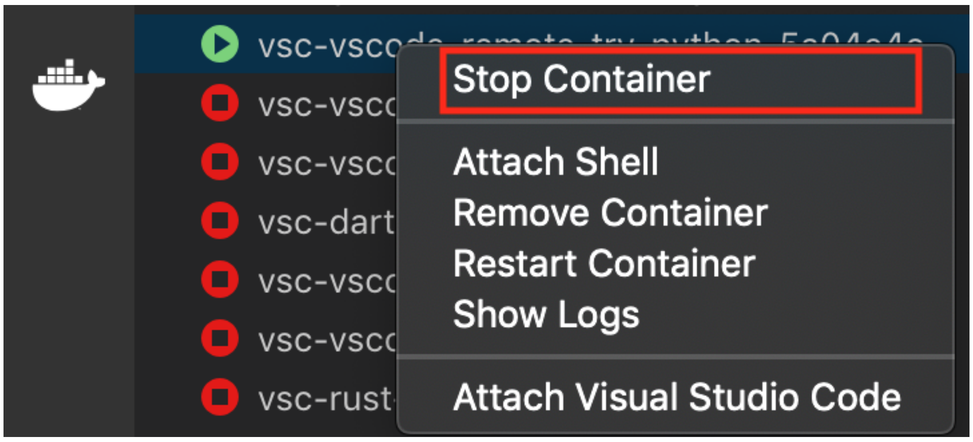
Option 1: Use the Docker extension

1. Install the Docker extension (https://marketplace.visualstudio.com/items?itemName=ms-azuretools.vscode-docker) from the Extensions view, if not already installed.

   > **Note:** Some Docker commands invoked from the Docker extension can fail when invoked from a VS Code window opened in a container. Most containers do not have the Docker command line installed. Therefore commands invoked from the Docker extension that rely on the Docker command line, for example Docker: Show Logs fail. If you need to execute these commands, open a new local VS Code window and use the Docker extension from this window or set up

Docker: Show Logs, fail. If you need to execute these commands, open a new local VS Code window and use the Docker extension from this window or set up Docker inside your container (https://aka.ms/vscode-remote/samples/docker-in-docker).

2. You can then go to the Docker view and expand the **Containers** node to see what containers are running. Right-click and select **Stop Container** to shut one down.



Option 2: Use the Docker CLI

1. Open a **local** terminal/command prompt (or use a local window in VS Code).
2. Type `docker ps` to see running containers. Use `docker ps -a` to also see any stopped containers.
3. Type `docker stop <Container ID>` from this list to stop a container.
4. If you would like to delete a container, type `docker rm <Container ID>` to remove it.

If `docker ps` does not provide enough information to identify the container you want to manage, the following command will list all development containers managed by VS Code and the folder used to generate them.

```
docker ps -a --filter="label=vsch.quality" --format "table {{.ID}}\t{{.Status}}\t{{.Image}}\tvscode-{{.Label \"vsch.quality\"}}\t{{.Label \"vsch.local.folder\"}}"
```

Option 3: Use Docker Compose

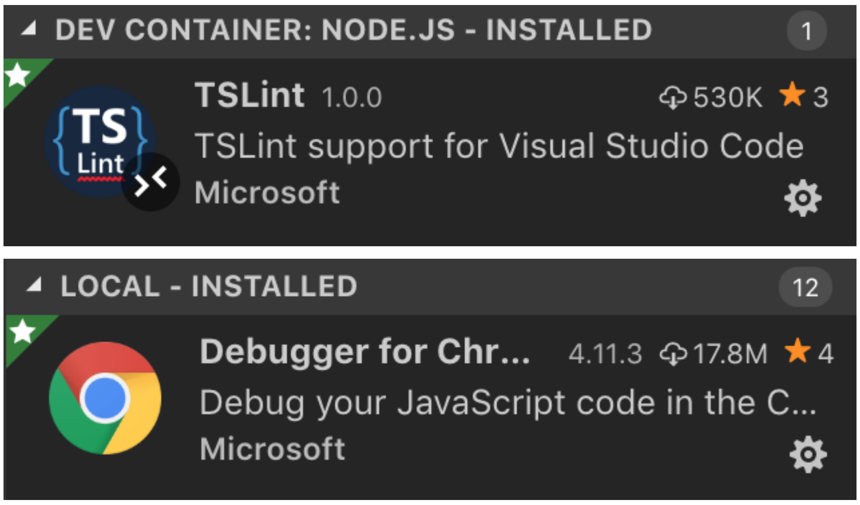
1. Open a **local** terminal/command prompt (or use a local window in VS Code).
2. Go to the directory with your `docker-compose.yml` file.
3. Type `docker-compose top` to see running processes.
4. Type `docker-compose stop` to stop the containers. If you have more than one Docker Compose file, you can specify additional Docker Compose files with the `-f` argument.
5. If you would like to delete the containers, type `docker-compose down` to both stop and delete them.

If you want to clean out images or mass-delete containers, see Cleaning out unused containers and images (/docs/remote/troubleshooting#_cleaning-out-unused-containers-and-images) for different options.

Managing extensions

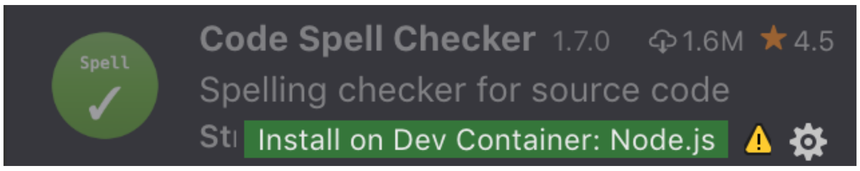
VS Code runs extensions in one of two places: locally on the UI / client side, or in the container. While extensions that affect the VS Code UI, like themes and snippets, are installed locally, most extensions will reside inside a particular container. This allows you to install only the extensions you need for a given task in a container and seamlessly switch your entire tool-chain just by connecting to a new container.

If you install an extension from the Extensions view, it will automatically be installed in the correct location. You can tell where an extension is installed based on the category grouping. There will be a **Local - Installed** category and also one for your container.


DEV CONTAINER: NODE.JS - INSTALLED
1
TSLint 1.0.0
530K
3
TSLint support for Visual Studio Code
Microsoft
LOCAL - INSTALLED
12
Debugger for Chr...
4.11.3
17.8M
4
Debug your JavaScript code in the C...
Microsoft

Note: If you are an extension author and your extension is not working properly or installs in the wrong place, see Supporting Remote Development (/api/advanced-topics/remote-extensions) for details.

Local extensions that actually need to run remotely will appear **Disabled** in the **Local - Installed** category. Select **Install** to install an extension on your remote host.


Spell
Code Spell Checker 1.7.0 1.6M 4.5
Spelling checker for source code
St
Install on Dev Container: Node.js

"Always installed" extensions

If there are extensions that you would like always installed in any container, you can update the `remote.containers.defaultExtensions` User setting (/docs/getstarted/settings). For example, if you wanted to install the GitLens (https://marketplace.visualstudio.com/items?itemName=eamodio.gitlens) and Resource Monitor (https://marketplace.visualstudio.com/items?itemName=mutantdino.resourcemonitor) extensions, you would specify their extension IDs as follows:

```
"remote.containers.defaultExtensions": [
    "eamodio.gitlens",
    "mutantdino.resourcemonitor"
]
```

Advanced: Forcing an extension to run locally / remotely

Extensions are typically designed and tested to either run locally or remotely, not both. However, if an extension supports it, you can force it to run in a particular location in your `settings.json` file.

For example, the setting below will force the Docker and Debugger for Chrome extensions to run remotely instead of their local defaults:

```
"remote.extensionKind": {
    "msjsdiag.debugger-for-chrome": "workspace",
    "ms-azuretools.vscode-docker": "workspace"
}
```

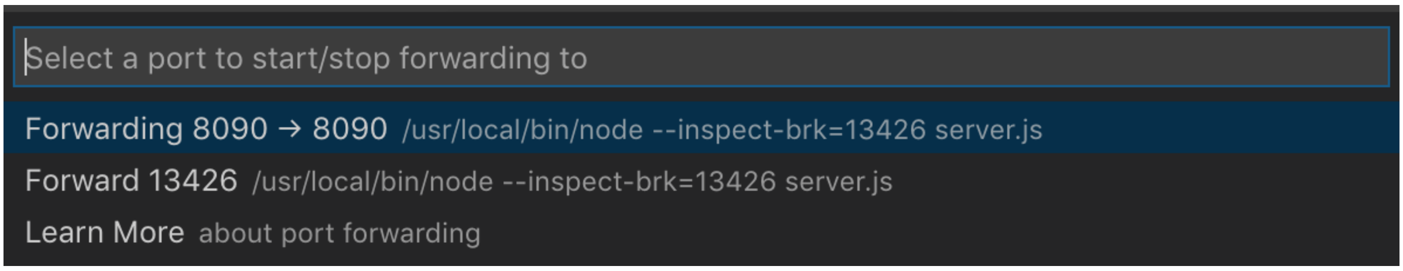
A value of `"ui"` instead of `"workspace"` will force the extension to run on the local UI/client side instead. Typically, this should only be used for testing unless otherwise noted in the extension's documentation since it **can break extensions**. See the article on Supporting Remote Development (/api/advanced-topics/remote-extensions) for details.

Forwarding a port

Containers are isolated environments, so if you want to access a server, service, or other resource inside your container, you will need to "forward" the port to your host. You can either configure your container to always expose these ports or just forward them temporarily.

Temporarily forwarding a port

Sometimes when developing you may need to access a port in your container that you didn't add to `devcontainer.json`, your Dockerfile, or Docker Compose file. If you want to **temporarily forward** a new port for the duration of the session, run the **Remote-Containers: Forward Port from Container...** command from the Command Palette (`F1`).


Select a port to start/stop forwarding to
Forwarding 8090 → 8090 /usr/local/bin/node --inspect-brk=13426 server.js
Forward 13426 /usr/local/bin/node --inspect-brk=13426 server.js
Learn More about port forwarding

After selecting a port, a notification will tell you the localhost port you should use to access the port in the container. For example, if you forwarded an HTTP server listening on port 3000, the notification may tell you that it was mapped to port 4123 on localhost. You can then connect to this remote HTTP server using `http://localhost:4123`.

Always forwarding / exposing / publishing a port

If you have ports you always want use from your host, you can set them up so they are always available. Specifically you can:

1. **Use the appPort property:** If you reference an image or Dockerfile in `devcontainer.json`, you can use the `appPort` property to publish ports to the host.

   ```
   "appPort": [ 3000, "8921:5000" ]
   ```

2. **Use the Docker Compose ports mapping:** The `ports` mapping (https://docs.docker.com/compose/compose-file#ports) can easily be added your `docker-compose.yml` file to publish additional ports.

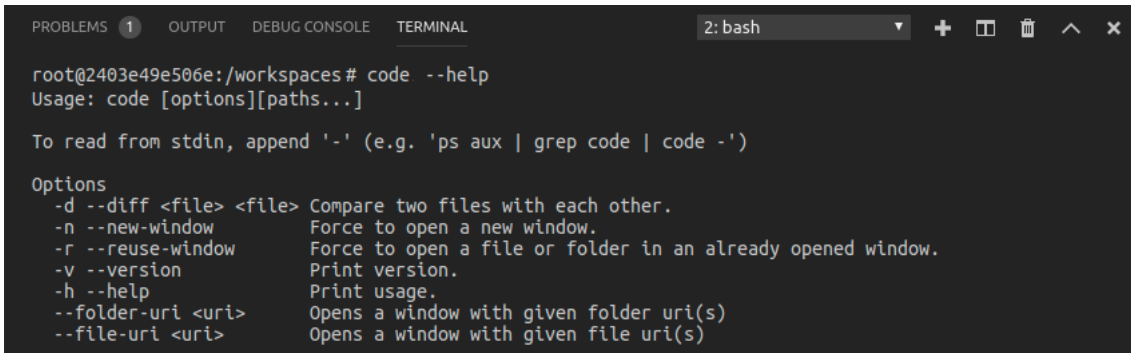
   ```
   ports:
   - "3000"
   - "8921:5000"
   ```

In each case, you'll need to rebuild your container for the setting to take effect. You can do this by running the **Remote-Containers: Rebuild Container** command in the Command Palette (`F1`) when you are connected to the container.

Opening a terminal

Opening a terminal in a container from VS Code is simple. Once you've opened a folder in a container, **any terminal window** you open in VS Code (**Terminal > New Terminal**) will automatically run in the container rather than locally.

You can also use the `code` command line from this same terminal window to perform a number of operations such as opening a new file or folder in the container. Type `code --help` to learn what options are available from the command line.



```
root@2403e49e506e:/workspaces# code --help
Usage: code [options][paths...]

To read from stdin, append '-' (e.g. 'ps aux | grep code | code -')

Options
  -d --diff <file> <file> Compare two files with each other.
  -n --new-window         Force to open a new window.
  -r --reuse-window       Force to open a file or folder in an already opened window.
  -v --version            Print version.
  -h --help               Print usage.
  --folder-uri <uri>      Opens a window with given folder uri(s)
  --file-uri <uri>        Opens a window with given file uri(s)
```

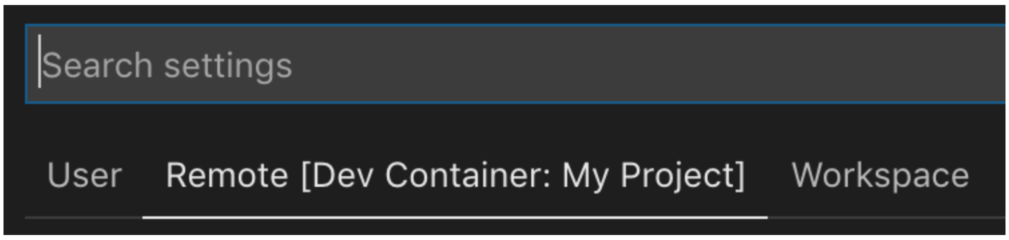
Debugging in a container

Once you've opened a folder in a container, you can use VS Code's debugger in the same way you would when running the application locally. For example, if you select a launch configuration in `launch.json` and start debugging (`F5`), the application will start on the remote host and attach the debugger to it.

See the debugging (/docs/editor/debugging) documentation for details on configuring VS Code's debugging features in `.vscode/launch.json` .

Container specific settings

VS Code's local user settings are also reused when you are connected to a dev container. While this keeps your user experience consistent, you may want to vary some of these settings between your local machine and each container. Fortunately, once you have connected to a container, you can also set container-specific settings by running the **Preferences: Open Remote Settings** command from the Command Palette (`F1`) or by selecting the **Remote** tab in the settings editor. These will override any local settings you have in place whenever you connect to the container.



Default container specific settings

You can include defaults for container specific settings in `devcontainer.json` using the `settings` property. These values will be automatically placed in the container specific settings file inside the container once it is created.

For example, adding this to `.devcontainer/devcontainer.json` will set the Java home path:

```
"settings": {
    "java.home": "/docker-java-home"
}
```

Since this just establishes the default, you are still able to change the settings as needed once the container is created.

Sharing Git credentials with your container

Using a credential helper

If you use HTTPS to clone your repositories and **have a credential helper configured (https://help.github.com/en/articles/caching-your-github-password-in-git) in your local OS, no further setup is required.** Credentials you've entered locally will be reused in the container and vice versa.

This works by copying your local `.gitconfig` file into the container on startup VS Code Server will automatically forward any git credential helper commands to your local operating system.

Note that, if you do not have your user name or email address set up locally, you may be prompted to do so. You can do this on your local machine by running the following commands:

```
git config --global user.name "Your Name"
git config --global user.email "your.email@address"
```

Using SSH keys

There are some cases when you may be cloning your repository using SSH keys instead of a credential helper. Simply mounting your local `~/.ssh` folder into the container works on macOS/Linux, but unfortunately this does not work on Windows due to the permissions set by the Windows OS. The contents of the `.ssh` folder is not automatically copied into the container since some of the keys could be used for accessing test or production servers and could pose a security risk.

However, there is a cross-platform way of copying the contents of the .ssh folder into the container when it is created, without modifying your image or Dockerfile.

- When an **image** or **Dockerfile** is referenced in `devcontainer.json` , add the following to this same file:

```
// Mount your .ssh folder to /root/.ssh-localhost so we can copy its contents
"runArgs": [ "-v", "${env:HOME}${env:USERPROFILE}/.ssh:/root/.ssh-localhost:ro" ],

// Copy the contents to the correct location and set permissions
"postCreateCommand": "mkdir -p ~/.ssh && cp -r ~/.ssh-localhost/* ~/.ssh && chmod 700 ~/.ssh && chmod 600 ~/.ssh/*"
```

- When a **Docker Compose** file is referenced, first update (or extend (/docs/remote/containers#_extending-your-docker-compose-file-for-development)) your `docker-compose.yml` with the following for the appropriate service:

```
version: '3'
services:
  your-service-name-here:
    # ...
    volumes:
      - ~/.ssh:~/.ssh-localhost:ro
```

And then add this to `devcontainer.json` to copy the keys to the correct location with the right permissions:

```
"postCreateCommand": "mkdir -p ~/.ssh && cp -r ~/.ssh-localhost/* ~/.ssh && chmod 700 ~/.ssh && chmod 600 ~/.ssh/*"
```

Note that, after you create your container for the first time, you will need to run the **Remote-Containers: Rebuild Container** command for updates to `devcontainer.json` , your Docker Compose files, or related Dockerfiles to take effect.

In-depth: Setting up a folder to run in a container

There are a few different ways VS Code Remote - Containers can be used to develop an application inside a fully containerized environment. In general, there are two primary scenarios that drive interest in this development style:

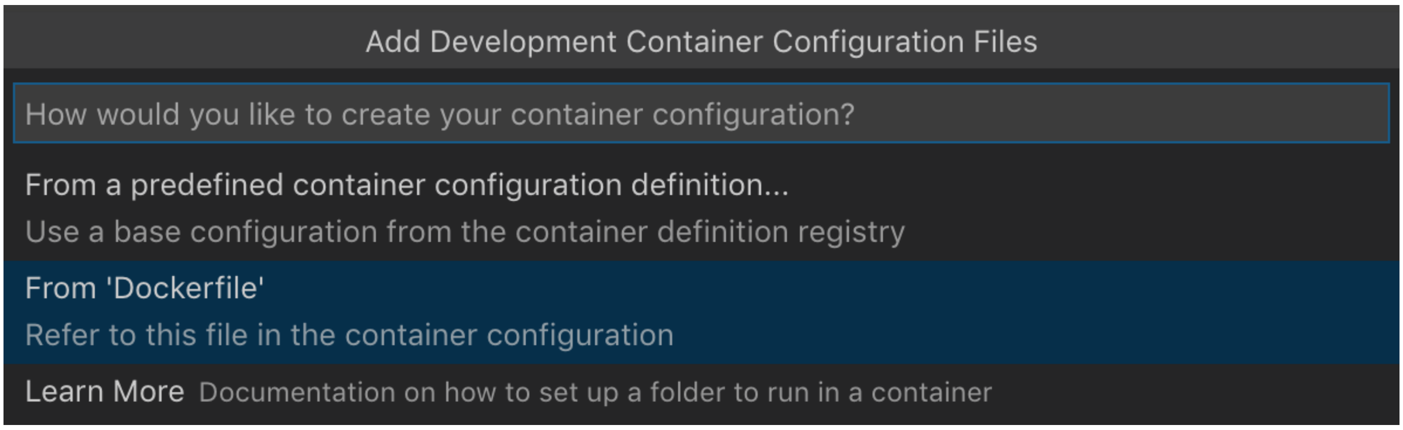
- **Stand-Alone Dev Sandboxes:** You may not be deploying your application into a containerized environment but still want to isolate your build and runtime environment from your local OS, speed up setup, or develop in an environment that is more representative of production. For example, you may be running code on your local macOS or Windows machine that will ultimately be deployed to a Linux VM or server, have different toolchain requirements for multiple projects, or want to be able to use tools/packages that could impact your local machine in an unexpected or undesired way. You can reference a container image or a Dockerfile for this purpose.
- **Container Deployed Applications**: You deploy your application into one or more containers and would like to work locally in the containerized environment. VS Code currently supports working with container-based applications defined in a number of ways:
  - Dockerfile: You are working on a single container / service that is described using a single Dockerfile.
  - Docker Compose: You are working with multiple orchestrated services that are described using a `docker-compose.yml` file.
  - In each case, you may also need to build container images and deploy to Docker or Kubernetes (/docs/remote/containers-advanced#_using-docker-or-kubernetes-from-a-container) from inside your container.
  - Attach only: While not backed by `devcontainer.json` , you can attach to an already running container if none of the workflows described in this section meet your needs.

This section will walk you through configuring your project for each of these situations. The vscode-dev-containers GitHub repository (https://aka.ms/vscode-dev-containers) also contains dev container definitions to get you up and running quickly.

Using an image or Dockerfile

You can configure VS Code to reuse an existing image from a source like DockerHub (https://hub.docker.com) or Azure Container Registry (https://azure.microsoft.com/services/container-registry/) for your dev container by adding a `.devcontainer/devcontainer.json` (or `.devcontainer.json`) config file to your project. In addition, if you are not able to find an image that meets your needs, have a single container-based project, or just want to automate the installation of several additional dependencies, you can use a Dockerfile (https://docs.docker.com/engine/reference/builder/) to generate the image instead.

To get started quickly, **open the folder** you want to work with in VS Code and run the **Remote-Containers: Add Development Container Configuration Files...** command in the Command Palette (`F1`).


Add Development Container Configuration Files
How would you like to create your container configuration?
From a predefined container configuration definition...
Use a base configuration from the container definition registry
From 'Dockerfile'
Refer to this file in the container configuration
Learn More Documentation on how to set up a folder to run in a container

You'll be asked to either select an existing Dockerfile (if one exists), or pick a pre-defined container configuration from the vscode-dev-containers repository (https://github.com/Microsoft/vscode-dev-containers) in a filterable list. VS Code will then add `devcontainer.json` and any other required files to the folder. While most of these pre-defined "dev container definitions" include a Dockerfile, you can use them as a starting point for an image instead if you prefer.

> **Note:** See System requirements for information on supported containers. When using experimental support for Alpine Linux in Visual Studio Code Insiders (https://code.visualstudio.com/insiders/), note that some extensions installed in the container may not work due to `glibc` dependencies in native code inside the extension.

You can also create your configuration manually. The difference between configuring VS Code to build a container image using a Dockerfile or just reuse an exiting image is a single property in `devcontainer.json` :

- **To use an image:** Set the `image` property. For example, this will use the .NET SDK image, publish port 8090, and install the C# VS Code extension:

```
{
    "name": "My Project",
    "image": "mcr.microsoft.com/dotnet/core/sdk:latest",
    "appPort": 8090,
    "extensions": [
        "ms-vscode.csharp"
    ]
}
```

- **To use a Dockerfile:** Set the `dockerFile` property. For example, this will cause VS Code to build the dev container image using the specified `Dockerfile` , publish port 3000, install the ES Lint extension in the container, and run `npm install` once it is done:

```
{
    "name": "My Node.js App",
    "dockerFile": "Dockerfile",
    "appPort": 3000,
    "extensions": [
        "dbaeumer.vscode-eslint"
    ],
    "postCreateCommand": "npm install"
}
```

See the devcontainer.json reference for information on other available properties such as the `appPort` , `postCreateCommand` , and the `extensions` list.

Once you have added a `.devcontainer/devcontainer.json` file to your folder, run the **Remote-Containers: Reopen Folder in Container** command (or **Remote-Containers: Open Folder in Container...** if you are not yet in VS Code) from the Command Palette (`F1`). After the container is created, the **local filesystem is automatically mapped into the container** unless you change this behavior (/docs/remote/containers-advanced#_changing-the-default-source-code-mount) and you can start working with it from VS Code.

You can also add additional local mount points to give your container access to other locations using the `runArgs` property.

For example, you can mount your home / user profile folder:

```
"runArgs": [ "-v", "${env:HOME}${env:USERPROFILE}/.ssh:/root/local-home" ]
```

This same technique can be used to **mount or copy your local SSH keys into the container** for use with Git. See Sharing Git credentials with your container for details.

The `runArgs` property supports the same list of arguments as the `docker run` command (https://docs.docker.com/engine/reference/commandline/run/) and can be useful for a wide variety of scenarios including setting environment variables (/docs/remote/containers-advanced#_adding-environment-variables).

If your application was built using C++, Go, or Rust, or another language that uses a **ptrace-based debugger**, the `runArgs` property can also be used to configure needed runtime security and capability settings.

For example:

```
{
    "name": "My Go App",
    "dockerFile": "Dockerfile",
    "extensions": [
        "ms-vscode.go"
    ],
    "runArgs": [
        "--cap-add=SYS_PTRACE",
        "--security-opt",
        "seccomp=unconfined" ]
}
```

While less efficient than a custom Dockerfile, you can also use the `postCreateCommand` property to **install additional software** that **may not be in your base image** or for cases where you would prefer **not to modify a deployment Dockerfile** you are reusing.

For example, here is a `devcontainer.json` that adds `git` and the Git Lens (https://marketplace.visualstudio.com/items?itemName=eamodio.gitlens) extension to the base Debian 9 container image:

```
{
    "image": "debian:9",
    "extensions": [
        "eamodio.gitlens"
    ],
    "postCreateCommand": "apt-get update && apt-get install -y git"
}
```

This command is run once your source code is mounted, so you can also use the property to run commands like `npm install` or to execute a shell script in your source tree.

```
"postCreateCommand": "bash scripts/install-dev-tools.sh"
```

By default, when VS Code starts a container, it will **override the container's default command** to be `/bin/sh -c "while sleep 1000; do :; done"`. This is done because the container will stop if the default command fails or simply exits. However, this may not work for certain images. If the image you are using requires the default command be run to work properly, add the following to your `devcontainer.json` file.

```
"overrideCommand": false
```

Note that, after you create your container for the first time, you will need to run the **Remote-Containers: Rebuild Container** command for updates to `devcontainer.json` or your Dockerfile to take effect.

Installing additional software in the sandbox

Once VS Code is connected to the container, you can open a VS Code terminal and execute any command against the OS inside the container. This allows you to install new command-line utilities and spin up databases or application services from inside the Linux container.

Most container images are based on Debian or Ubuntu, where the `apt-get` command is used to install new packages.

For example:

```
apt-get update # Critical step - you won't be able to install software before you do this
apt-get install <package>
```

Documentation for the software you want to install will usually provide specific instructions, but note that you may **not need to prefix commands with `sudo`** given you are likely running as root in the container. If you are not already root, read the directions for the image you've selected to learn how to install additional software. If you would **prefer not to run as root**, see the Advanced Container Configuration (/docs/remote/containers-advanced#_adding-a-nonroot-user-to-your-dev-container) article for how to set up a separate user.

However, note that if you **rebuild** the container, you will have to **reinstall** anything you've installed manually. To avoid this problem, you can use the `postCreateCommand` property or use a custom Dockerfile instead.

Using Docker Compose

In some cases, a single container environment isn't sufficient. Fortunately, Remote - Containers supports Docker Compose (https://docs.docker.com/compose/) managed multi-container configurations.

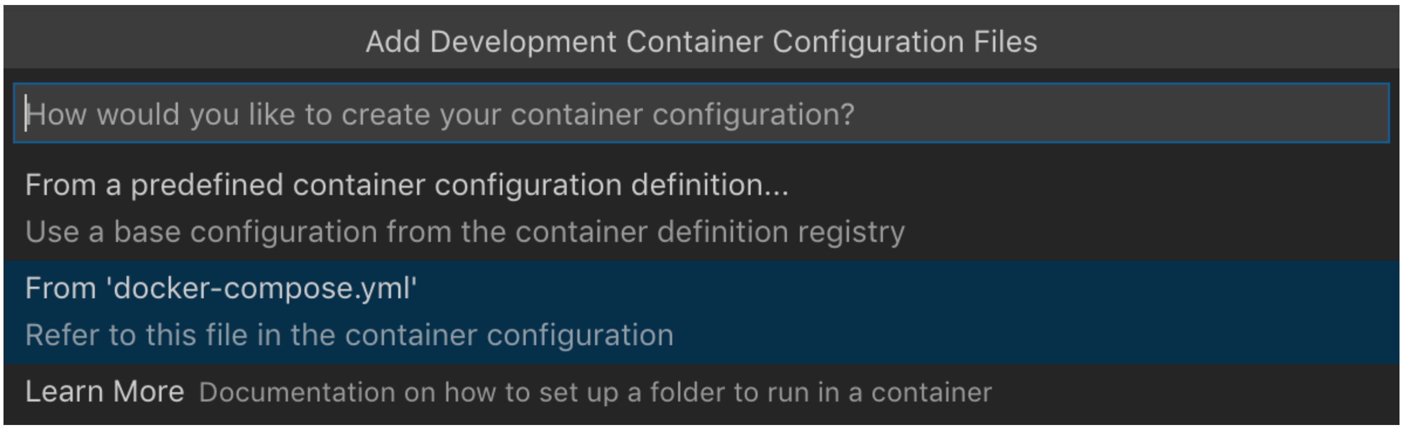
You can either:

1. Work with a service defined in an existing, unmodified `docker-compose.yml`.
2. Create a new `docker-compose.yml` (or make a copy of an existing one) that you use to develop a service.
3. Extend your existing Docker Compose configuration to develop the service.
4. Use separate VS Code windows to work with multiple Docker Compose-defined services (/docs/remote/containers-advanced#_connecting-to-multiple-containers-at-once) at once.

> **Note:** See System requirements for information on supported containers. When using experimental support for Alpine Linux in Visual Studio Code Insiders (https://code.visualstudio.com/insiders/), note that some extensions installed in the container may not work due to `glibc` dependencies in native code inside the extension.

VS Code can be configured to **automatically start any needed containers** for a particular service in a Docker Compose file. If you've already started the configured containers using the command line, VS Code will **attach to the running service** you've specified instead. This gives your multi-container workflow the same quick setup advantages described for the Docker image and Dockerfile flows above while still allowing you to use the command line if you prefer.

To get started quickly, **open the folder** you want to work with in VS Code and run the **Remote-Containers: Add Development Container Configuration Files...** command in the Command Palette (`F1`).


Add Development Container Configuration Files
How would you like to create your container configuration?
From a predefined container configuration definition...
Use a base configuration from the container definition registry
From 'docker-compose.yml'
Refer to this file in the container configuration
Learn More Documentation on how to set up a folder to run in a container

You'll be asked to either select an existing Docker Compose file (if one exists), or pick a pre-defined container configuration from the vscode-dev-containers repository (https://github.com/Microsoft/vscode-dev-containers) in a filterable list. Many of these "dev container definitions" use a Dockerfile, so select one of these definitions for a starting point for Docker Compose: Existing Docker Compose (https://aka.ms/vscode-remote/samples/existing-docker-compose), Node.js & MongoDB (https://aka.ms/vscode-remote/samples/node-mongo), Python & PostgreSQL (https://aka.ms/vscode-remote/samples/python-postgres), or Docker-in-Docker Compose (https://aka.ms/vscode-remote/samples/docker-in-docker-compose). After you make your selection, VS Code will add the appropriate

.devcontainer/devcontainer.json (or .devcontainer.json) file to the folder.

You can also create your configuration manually. To reuse a Docker Compose file unmodified, you can use the `dockerComposeFile` and `service` properties in `.devcontainer/devcontainer.json` .

For example:

```
{
    "name": "[Optional] Your project name here",
    "dockerComposeFile": "../docker-compose.yml",
    "service": "the-name-of-the-service-you-want-to-work-with-in-vscode",
    "workspaceFolder": "/default/workspace/path/in/container/to/open",
    "shutdownAction": "stopCompose"
}
```

See the devcontainer.json reference for information other available properties such as the `workspaceFolder` and `shutdownAction` .

Once you have added a `.devcontainer/devcontainer.json` file to your folder, run the **Remote-Containers: Reopen Folder in Container** command (or **Remote-Containers: Open Folder in Container...** if you are not yet in VS Code) from the Command Palette (`F1`).

If the containers are not already running, VS Code will call `docker-compose -f ../docker-compose.yml up` in this example. Note that the `service` property indicates which service in your Docker Compose file VS Code should connect to, not which service should be started. If you started them by hand, VS Code will attach to the service you specified.

You can also create a development copy of your Docker Compose file. For example, if you had `.devcontainer/docker-compose.devcontainer.yml` , you would just change the following line in `devcontainer.json` :

```
"dockerComposeFile": "docker-compose.devcontainer.yml"
```

However, a better approach is often to avoid making a copy of your Docker Compose file by **extending it with another one**. We'll cover extending a Docker Compose file in the next section.

To avoid having the container shut down if the default container command fails or exits, you can modify your Docker Compose file for the service you have specified in `devcontainer.json` as follows:

```
# Overrides default command so things don't shut down after the process ends.
command: /bin/sh -c "while sleep 1000; do :; done"
```

If you are not already, you can (bind) mount your local source code into the container using the volumes list in your Docker Compose file (https://docs.docker.com/compose/compose-file/#volumes).

For example:

```
volumes:
  # Mounts the project folder to '/workspace'. The target path inside the container
  # should match what your application expects. In this case, the compose file is
  # in a sub-folder, so we will mount '..'. You would then reference this path as the
  # 'workspaceFolder' in '.devcontainer/devcontainer.json' so VS Code starts here.
  - ..:/workspace
```

This same technique can be used to **mount or copy your local SSH keys into the container** for use with Git. See Sharing Git credentials with your container for details.

If you aren't creating a custom Dockerfile for development, you may want to install additional developer tools such as Git inside the service's container. While less efficient than adding these tools to the container image, you can also use the `postCreateCommand` property for this purpose. For example:

```
"postCreateCommand": "apt-get update && apt-get install -y git"
```

This command is run once the container is running, so you can also use the property to run commands like `npm install` or to execute a shell script in your source tree (if you have mounted it).

```
"postCreateCommand": "bash scripts/install-dev-tools.sh"
```

If your application was built using C++, Go, or Rust, or another language that uses a ptrace-based debugger, you will also need to add the following settings to your Docker Compose file:

```
# Required for ptrace-based debuggers like C++, Go, and Rust
cap_add:
- SYS_PTRACE
security_opt:
- seccomp:unconfined
```

Note that, after you create your container for the first time, you will need to run the **Remote-Containers: Rebuild Container** command for updates to `devcontainer.json` , your Docker Compose files, or related Dockerfiles to take effect.

Extending your Docker Compose file for development

Referencing an existing deployment / non-development focused `docker-compose.yml` has some potential downsides.

For example:

- Docker Compose will shut down a container if its entry point shuts down. This is problematic for situations where you are debugging and need to restart your app on a repeated basis.

- You also may not be mapping the local filesystem into the container or exposing ports to other resources like databases you want to access.
- You may want to copy the contents of your local `.ssh` folder into the container or set the ptrace options described above in Using Docker Compose.

You can solve these and other issues like them by extending your entire Docker Compose configuration with multiple `docker-compose.yml` files (https://docs.docker.com/compose/extends/#multiple-compose-files) that override or supplement your primary one.

For example, consider this additional `.devcontainer/docker-compose.extend.yml` file:

```
version: '3'
  services:
    your-service-name-here:
      volumes:
        # Mounts the project folder to '/workspace'. The target path inside the container
        # should match what your application expects. In this case, the compose file is
        # in a sub-folder, so we will mount '..'. We'll then reference this as the
        # workspaceFolder in '.devcontainer/devcontainer.json' so VS Code starts here.
        - ..:/workspace

        # [Optional] If you are using SSH keys w/Git, mount your .ssh folder to
        # /root/.ssh-localhost so we can copy its contents
        - ~/.ssh:/root/.ssh-localhost:ro

      # [Optional] Required for ptrace-based debuggers like C++, Go, and Rust
      cap_add:
        - SYS_PTRACE
      security_opt:
        - seccomp:unconfined

      # Overrides default command so things don't shut down after the process ends.
      command: /bin/sh -c "while sleep 1000; do :; done"
```

This same file can provide additional settings, such as port mappings, as needed. To use it, reference your original `docker-compose.yml` file in addition to `.devcontainer/devcontainer.extend.yml` in a specific order:

```
{
    "name": "[Optional] Your project name here",

    // The order of the files is important since later files override previous ones
    "dockerComposeFile": [
        "../docker-compose.yml",
        "docker-compose.extend.yml"
    ],

    "service": "your-service-name-here",
    "workspaceFolder": "/workspace",
    "shutdownAction": "stopCompose",

    // [Optional] If you are using SSH keys w/Git, copy them and set correct permissions
    "postCreateCommand": "mkdir -p ~/.ssh && cp -r ~/.ssh-localhost/* ~/.ssh && chmod 700 ~/.ssh && chmod 600 ~/.ssh/*"
}
```

VS Code will then **automatically use both files** when starting up any containers. You can also start them yourself from the command line as follows:

```
docker-compose -f docker-compose.yml -f .devcontainer/docker-compose.extend.yml up
```

While the `postCreateCommand` property allows you to install additional tools inside your container, in some cases you may want to have a specific Dockerfile for development. You can also use this same approach to reference a custom `Dockerfile` specifically for development without modifying your existing Docker Compose file. For example, you can update `.devcontainer/devcontainer.extend.yml` as follows:

```
version: '3'
  services:
    your-service-name-here:
       build:
         context: .
         # Location is relative to folder containing this compose file
         dockerfile: Dockerfile
       volumes:
         - ..:/workspace
         - ~/.ssh:/root/.ssh-localhost
       command: /bin/sh -c "while sleep 1000; do :; done"
```

Docker Compose dev container definitions

The following are dev container definitions that use Docker Compose:

- Existing Docker Compose (https://aka.ms/vscode-remote/samples/existing-docker-compose) - Includes a set of files that you can drop into an existing project that will reuse a `docker-compose.yml` file in the root of your project.
- Node.js & MongoDB (https://aka.ms/vscode-remote/samples/node-mongo) - A Node.js container that connects to a Mongo DB in a different container.
- Python & PostgreSQL (https://aka.ms/vscode-remote/samples/python-postgres) - A Python container that connects to PostGreSQL in a different container.
- Docker-in-Docker Compose (https://aka.ms/vscode-remote/samples/docker-in-docker-compose) - Includes the Docker CLI and illustrates how you can use it to access your local Docker install from inside a dev container by volume mounting the Docker Unix socket.

Advanced container configuration

See the Advanced Container Configuration (/docs/remote/containers-advanced) article for information on the following topics:

- Adding environment variables (/docs/remote/containers-advanced#_adding-environment-variables)
- Adding another local file mount (/docs/remote/containers-advanced#_adding-another-local-file-mount)
- Changing or removing the default source code mount (/docs/remote/containers-advanced#_changing-the-default-source-code-mount)
- Improving container disk performance (/docs/remote/containers-advanced#_improving-container-disk-performance)
- Adding a non-root user to your dev container (/docs/remote/containers-advanced#_adding-a-nonroot-user-to-your-dev-container)
- Avoiding extension reinstalls on container rebuild (/docs/remote/containers-advanced#_avoiding-extension-reinstalls-on-container-rebuild)
- Using Docker or Kubernetes from inside a container (/docs/remote/containers-advanced#_using-docker-or-kubernetes-from-a-container)
- Connecting to multiple containers at once (/docs/remote/containers-advanced#_connecting-to-multiple-containers-at-once)
- Developing inside a container on a remote Docker Machine or SSH host (/docs/remote/containers-advanced#_developing-inside-a-container-on-a-remote-docker-host)
- Reducing Dockerfile build warnings (/docs/remote/containers-advanced#_reducing-dockerfile-build-warnings)

devcontainer.json reference

Property	Type	Description
Dockerfile or image		
`image`	string	**Required** when using an image. The name of an image in a container registry (DockerHub (https://hub.docker.com), Azure Container Registry (https://azure.microsoft.com/services/container-registry/)) that VS Code should use to create the dev container.
`dockerFile`	string	**Required** when using a Dockerfile. The location of a Dockerfile (https://docs.docker.com/engine/reference/builder/) that defines the contents of the container. The path is relative to the `devcontainer.json` file. You can find a number of sample Dockerfiles for different runtimes in the vscode-dev-containers repository (https://github.com/Microsoft/vscode-dev-containers/tree/master/containers).
`context`	string	Path that the Docker build should be run from relative to `devcontainer.json`. For example, a value of `".."` would allow you to reference content in sibling directories. Defaults to `"."`.
`appPort`	integer, string, or array	A port or array of ports that should be made available locally when the container is running. Defaults to `[]`.
`workspaceMount`	string	Overrides the default local mount point for the workspace. Supports the same values as the Docker CLI `--mount` flag (https://docs.docker.com/engine/reference/commandline/run/#add-bind-mounts-or-volumes-using-the---mount-flag). Primarily useful for configuring remote containers (/docs/remote/containers-advanced#_developing-inside-a-container-on-a-remote-docker-host) or improving disk performance (/docs/remote/containers-advanced#_improving-container-disk-performance).
`workspaceFolder`	string	Sets the default path that VS Code should open when connecting to the container. Typically used in conjunction with `workspaceMount`. Defaults to the automatic source code mount location.
`runArgs`	array	An array of Docker CLI arguments (https://docs.docker.com/engine/reference/commandline/run/) that should be used when running the container. Defaults to `[]`. A run argument can refer to environment variables using the following format `${env:HOME}`
`overrideCommand`	boolean	Tells VS Code whether it should run `/bin/sh -c "while sleep 1000; do :; done"` when starting the container instead of the container's default command. Defaults to `true` since the container can shut down if the default command fails. Set to `false` if the default command must run for the container to function properly.
`shutdownAction`	enum: `none`, `stopContainer`	Indicates whether VS Code should stop the container when the VS Code window is closed / shut down. Defaults to `stopContainer`.
Docker Compose		
`dockerComposeFile`	string or array	**Required.** Path or an ordered list of paths to Docker Compose files relative to the `devcontainer.json` file.
`service`	string	**Required.** The name of the service VS Code should connect to once running.
`workspaceFolder`	string	Sets the default path that VS Code should open when connecting to the container (which is often the path to a volume mount where the source code can be found in the container). Defaults to `"/"`.
`shutdownAction`	enum: `none`, `stopCompose`	Indicates whether VS Code should stop the containers when the VS Code window is closed / shut down. Defaults to `stopCompose`.
General		
`name`	string	A display name for the container.
`extensions`	array	An array of extension IDs that specify the extensions that should be installed inside the container when it is created. Defaults to `[]`.
`settings`	object	Adds default `settings.json` values into a container/machine specific settings file.

Property	Type	Description
`postCreateCommand`	string or array	A command string or list of command arguments to run after the container is created. Use `&&` in a string to execute multiple commands. For example, `"yarn install"`, `["yarn", "install"]`, or `"apt-get update && apt-get install -y git"`. It fires after your source code has been mounted, so you can also run shell scripts from your source tree. For example: `bash scripts/install-dev-tools.sh`. Defaults to none.
`devPort`	integer	Allows you to force a specific port that the VS Code Server should use in the container. Defaults to a random, available port.

If you've already built the container and connected to it, be sure to run **Remote-Containers: Rebuild Container** from the Command Palette (`F1`) to pick up the change.

Known limitations

Remote - Containers limitations

- Windows container images are **not** yet supported.
- Experimental Alpine container support is available in VS Code Insiders (https://code.visualstudio.com/insiders/) only.
- Using a remote Docker Host is possible, but requires additional setup steps (/docs/remote/containers-advanced#_developing-inside-a-container-on-a-remote-docker-host).
- All roots/folders in a multi-root workspace will be opened in the same container, regardless of whether there are configuration files at lower levels.
- The unofficial Ubuntu Docker **snap** package for Linux is **not** supported. Follow the official Docker install instructions for your distribution (https://docs.docker.com/install/#supported-platforms).
- Docker Toolbox on Windows is not supported.
- Docker variants or alternate containerization tool kits like Podman (https://podman.io) are not supported.
- If you clone a Git repository using SSH and your SSH key has a passphrase, VS Code's pull and sync features may hang when running remotely. Either use an SSH key without a passphrase, clone using HTTPS, or run `git push` from the command line to work around the issue.
- Local proxy settings are not reused inside the container, which can prevent extensions from working unless the appropriate proxy information is configured (for example global `HTTP_PROXY` or `HTTPS_PROXY` environment variables with the appropriate proxy information).

See here for a list of active issues (https://aka.ms/vscode-remote/containers/issues) related to Containers.

Docker limitations

- First-time installs of Docker Desktop for Windows will require an additional "sharing" step to give your container access to local source code. However, this step may not work with certain AAD (email-based) identities. See Docker Desktop for Windows tips (/docs/remote/troubleshooting#_docker-desktop-for-windows-tips) and Enabling file sharing in Docker Desktop (/docs/remote/troubleshooting#_enabling-file-sharing-in-docker-desktop) for details and workarounds.

- You may see errors if you sign into Docker with your email address instead of your Docker ID. This is a known issue and can be resolved by signing in with your Docker ID instead. See Docker issue #935 (https://github.com/docker/hub-feedback/issues/935#issuecomment-300361781) for details.

- If you see high CPU spikes for `com.docker.hyperkit` on macOS, this may be due to a known issue with Docker for Mac (https://github.com/docker/for-mac/issues/1759). See the Docker issue for details.

- If you see either of these messages building a Dockerfile, you may be hitting a known Docker issue with Debian 8 (Jessie):

  ```
  W: Failed to fetch http://deb.debian.org/debian/dists/jessie-updates/InRelease
  E: Some index files failed to download. They have been ignored, or old ones used instead
  ```

  See Resolving Dockerfile build failures (/docs/remote/troubleshooting#_resolving-dockerfile-build-failures-for-images-using-debian-8) for a workaround.

See the Docker troubleshooting guide for Windows (https://docs.docker.com/docker-for-windows/troubleshoot) or Mac (https://docs.docker.com/docker-for-mac/troubleshoot), consult Docker Support Resources (https://success.docker.com/article/best-support-resources) for more information.

Docker Extension limitations

Some Docker commands invoked from the Docker extension can fail when invoked from a VS Code window opened in a container. Most containers do not have the Docker command line installed. Therefore commands invoked from the Docker extension that rely on the Docker command line, for example **Docker: Show Logs**, fail. If you need to execute these commands, open a new local VS Code window and use the Docker extension from this window or set up Docker inside your container (https://aka.ms/vscode-remote/samples/docker-in-docker).

Extension limitations

Many extensions will work inside dev containers without modification. However, in some cases, certain features may require changes. If you run into an extension issue, see here for a summary of common problems and solutions (/docs/remote/troubleshooting#_extension-tips) that you can mention to the extension author when reporting the issue.

In addition, while experimental Alpine support is available in VS Code Insiders (https://code.visualstudio.com/insiders/), some extensions installed in the container may not work due to `glibc` dependencies in native code inside the extension. See the Remote Development with Linux (/docs/remote/linux) article for details.

Common questions

I am seeing errors when trying to mount the local filesystem into a container

Right-click on the Docker task bar item. On Windows, go to the **Settings > Shared Drives** tab and check the drive(s) where your source code is located. On macOS, go the **Preferences > File Sharing** tab and make sure the folder containing your source code is under a file path specified in the list.

See Docker Desktop for Windows tips (/docs/remote/troubleshooting#_docker-desktop-for-windows-tips) for information on workarounds to common Docker for Windows issues.

I am seeing "W: Failed to fetch ..." when building a Dockerfile

If you see "W: Failed to fetch http://deb.debian.org/debian/dists/jessie-updates/InRelease (http://deb.debian.org/debian/dists/jessie-updates/InRelease)" when building a Dockerfile, you may be hitting a known Docker issue with Debian 8 (Jessie). See Resolving Dockerfile build failures (/docs/remote/troubleshooting#_resolving-dockerfile-build-failures-for-images-using-debian-8) for a workaround.

I'm seeing an error about a missing library or dependency

Some extensions rely on libraries not found in the certain Docker images. See Installing additional software for help with resolving the problem.

Can I connect to multiple containers at once?

A VS Code window can only connect to one window currently, but you can open a new window and attach to an already running container or use a common Docker Compose file with multiple `devcontainer.json` files (/docs/remote/containers-advanced#_connecting-to-multiple-containers-at-once) to automate the process a bit more.

Can I work with containers on a remote host?

Yes, you can either attach to a container running on a remote host or create a specialized `devcontainer.json` to tell VS Code how to work with your remote environment. To access the host, you can either connect to a publicly exposed Docker daemon TCP port or use SSH to tunnel into a remote VM running Docker. See Developing inside a container on a remote Docker host (/docs/remote/containers-advanced#_developing-inside-a-container-on-a-remote-docker-host) for details.

How can I build or deploy container images into my local Docker / Kubernetes install when working inside a container?

You can build images and deploy containers by forwarding the Docker socket and installing the Docker CLI (and kubectl for Kubernetes) in the container. See the Docker-in-Docker (https://aka.ms/vscode-remote/samples/docker-in-docker), Docker-in-Docker Compose (https://aka.ms/vscode-remote/samples/docker-in-docker-compose), and Kubernetes-Helm (https://aka.ms/vscode-remote/samples/kubernetes-helm) dev container definitions for details.

What are the connectivity requirements for the VS Code Server when it is running in a container?

The VS Code Server requires outbound HTTPS (port 443) connectivity to:

- `update.code.visualstudio.com`
- `marketplace.visualstudio.com`
- `vscode.blob.core.windows.net`
- `*.vo.msecnd.net` (Azure CDN)
- `*.gallerycdn.vsassets.io` (Azure CDN)

All other communication between the server and the VS Code client is accomplished through an authenticated, random, TCP port automatically exposed via the Docker CLI. You can find a list of locations VS Code itself needs access to in the network connections article (/docs/setup/network#_common-hostnames).

Finally, some extensions (like C#) download secondary dependencies from `download.microsoft.com` or `download.visualstudio.microsoft.com`. Others (like Visual Studio Live Share (https://docs.microsoft.com/visualstudio/liveshare/reference/connectivity#requirements-for-connection-modes)) may have additional connectivity requirements. Consult the extension's documentation for details if you run into trouble.

As an extension author, what do I need to do to make sure my extension works?

The VS Code extension API hides most of the implementation details of running remotely so many extensions will just work inside dev containers without any modification. However, we recommend that you test your extension in a dev container to be sure that all of its functionality works as expected. See the article on Supporting Remote Development (/api/advanced-topics/remote-extensions) for details.

What other resources are there that may be able to answer my question?

The following articles may help answer your question:

- Advanced Container Configuration (/docs/remote/containers-advanced) or Tips and Tricks (/docs/remote/troubleshooting#_containers-tips)
- Dockerfile reference (https://docs.docker.com/engine/reference/builder/)
- Docker Compose file reference (https://docs.docker.com/compose/compose-file/)
- Docker Desktop for Windows troubleshooting guide (https://docs.docker.com/docker-for-windows/troubleshoot) and FAQ (https://docs.docker.com/docker-for-windows/faqs/)
- Docker Desktop for Mac troubleshooting guide (https://docs.docker.com/docker-for-mac/troubleshoot) and FAQ (https://docs.docker.com/docker-for-mac/faqs/)
- Docker Support Resources (https://success.docker.com/article/best-support-resources)

Questions or feedback

- See Tips and Tricks (/docs/remote/troubleshooting#_containers-tips) or the FAQ (/docs/remote/faq).
- Search on Stack Overflow (https://stackoverflow.com/questions/tagged/vscode-remote).
- Add a feature request (https://aka.ms/vscode-remote/feature-requests) or report a problem (https://aka.ms/vscode-remote/issues/new).
- Create a development container definition (https://aka.ms/vscode-dev-containers) for others to use.
- Contribute to our documentation (https://github.com/Microsoft/vscode-docs) or VS Code itself (https://github.com/Microsoft/vscode).
- See our CONTRIBUTING (https://aka.ms/vscode-remote/contributing) guide for details.

Was this documentation helpful?

Yes **No**

7/18/2019

7/18/2019

IN THIS ARTICLE

Tweet this link (https://twitter.com/intent/tweet?original_referer=https://code.visualstudio.com/docs/remote/containers#_quick-start-try-a-dev-container&ref_src=twsrc%5Etfw&text=Developing%20inside%20a%20Container%20using%20Visual%20Studio%20Code%20Remote%20Development&tw_p=twe start-try-a-dev-container&via=code)

Subscribe(/feed.xml)

Ask questions(https://stackoverflow.com/questions/tagged/vscode)

Follow @code(https://go.microsoft.com/fwlink/?LinkID=533687)

Request features(https://go.microsoft.com/fwlink/?LinkID=533482)

Report issues(https://www.github.com/Microsoft/vscode/issues)

Watch videos(https://www.youtube.com/channel/UCs5Y5_7XK8HLDX0SLNwkd3w)

Hello from Seattle. Follow @code (https://go.microsoft.com/fwlink/?LinkID=533687) Star 80,029