Requirements

Our standard development setup uses docker. This means that any Linux distribution can be used as long as docker is available.

Setting up Docker

First install docker through the package manager of your Linux distribution.

Make sure that the docker daemon is running. In most distribution this can be done through `systemctl` (refer to the documentation of your OS to find the exact commands):

```
systemctl enable docker # this will make sure that the daemon is started at boot time
systemctl start docker # start the docker daemon
```

Additionally you need to give your user permissions to run docker containers. You can do this by adding your user to the group `docker` (on certain distributions, like CentOS 7, this group has a different name. For example on CentOS 7 you need to add your user to the group `dockerroot`). You can do this by running the following command:

```
sudo usermod -aG docker $USER
```

After that you should log out and then log in again to. Then you should be able to run docker as a user. If you still need to setup a `bin` directory, you can hold off with login out and back in until you have done that.

Set up a bin Directory

We use shell scripts to wrap our build commands. These should be in your `PATH` so that you can run them from any location within your home directory.

First make sure `$HOME/bin` exists:

```
mkdir -p $HOME/bin
```

Then add the following line to your `.bashrc`, `.profile`, or `.zshrc` (depending on your login shell and configuration):

```
export PATH=$HOME/bin:$PATH
```

Make sure `$HOME/bin` is the first entry in the `PATH`. After this is done you need to log out and back in.

The Wrapper Scripts

The main way we call into docker is a quite long script. Therefore we use the following wrapper script:

```
#!/usr/bin/env bash

set -e

if [ -t 1 ] ; then
    TERMINAL_ARGS=-it `# Run in interactive mode and simulate a TTY`
else
    TERMINAL_ARGS=-i `# Run in interactive mode`
fi

docker run --rm `# delete (temporary) image after return` \
    ${TERMINAL_ARGS}  \
    --privileged=true `# Run in privileged mode ` \
    --cap-add=SYS_PTRACE \
    --security-opt seccomp=unconfined \
    -u $(id -u):$(id -g) \
    -v "${HOME}:${HOME}" `# Mount home directory` \
    -v /etc/passwd:/etc/passwd:ro \
    -v /etc/group:/etc/group:ro \
    -w="$(pwd)" \
    -e CC=clang \
    -e CXX=clang++ \
    -e BOOST_ROOT=/opt/boost_1_72_0 \
    -e USE_CCACHE=ON \
    ${ccache_args} \
   foundationdb/devel:centos7-latest scl enable devtoolset-8 rh-python36 rh-
ruby27 -- "$@"
```

If you want to set additional environment variables in your docker environment you can add them to the command using the following syntax:

```
 -e NAME=VALUE
```

Copy the above script and paste it into a new file. Save this file as `$HOME/bin/fdb-dev` and make it an executable:

```
chmod +x $HOME/bin/fdb-dev
```

Additionally we recommend using `clangd` for development.`clangd` is also installed in the docker container - but in order to use it you need an executable. This is easy to do: create a new file and paste the following content into it:

```
#!/usr/bin/env bash

set -e

fdb-dev clangd
```

save it as `$HOME/bin/clangd`, and make it an executable:

```
chmod +x $HOME/bin/clangd
```

Congratulations, your FoundationDB development machine is now setup. The first time you call into `fdb-dev`, docker will fetch the image. This will take a few minutes and you need to be online for this to succeed.

Building FoundationDB

To build fdb you don't need to start a shell in the docker container (although you can by calling `fdb-dev bash`). The following script should check out the code and build fdb

```
cd $HOME
git clone git@github.com:apple/foundationdb.git
mkdir build
cd build
fdb-dev cmake -G Ninja -DCMAKE_EXPORT_COMPILE_COMMANDS=ON -DUSE_CCACHE=ON ../foundationdb
fdb-dev ninja
```

Note: If you get an error like `ld.lld: error: undefined symbol: std::__once_callable` set `FORCE_BOOST_BUILD` to `ON` in CMakeCache.txt

Note: If you get an error like `WARNING: IPv4 forwarding is disabled. Networking will not work.` restart the docker daemon.

Note: Sometimes the cmake step can potentially run into a bizarre error. Turned out it's caused because of clock skew. Fix the clock with `sudo hwclock --hctosys` if you have clock skew.

Ccache

The version of ccache currently shipping in the `foundationdb-dev` image (ccache v3.1.6) defaults to a maximum of 1GB of files stored in the cache. It is recommended to increase this size as much as possible for faster builds. Run the following to increase the cache size to 100GB:

```
$ fdb-dev ccache --max-size 100G
```

You can also view ccache stats by running `$ fdb-dev ccache -s`.

Setting up a Development Environment

We use `clangd` for code navigation and completion. `clangd` is a language server for C++ and can be used by many editors (Emacs, Vim, Atom, etc). This section describes how to set up Visual Studio Code - this is one of the easiest editors to set up.

After installing VS Code you have to install the `clangd` extension. After start VS Code click on the extension tab:

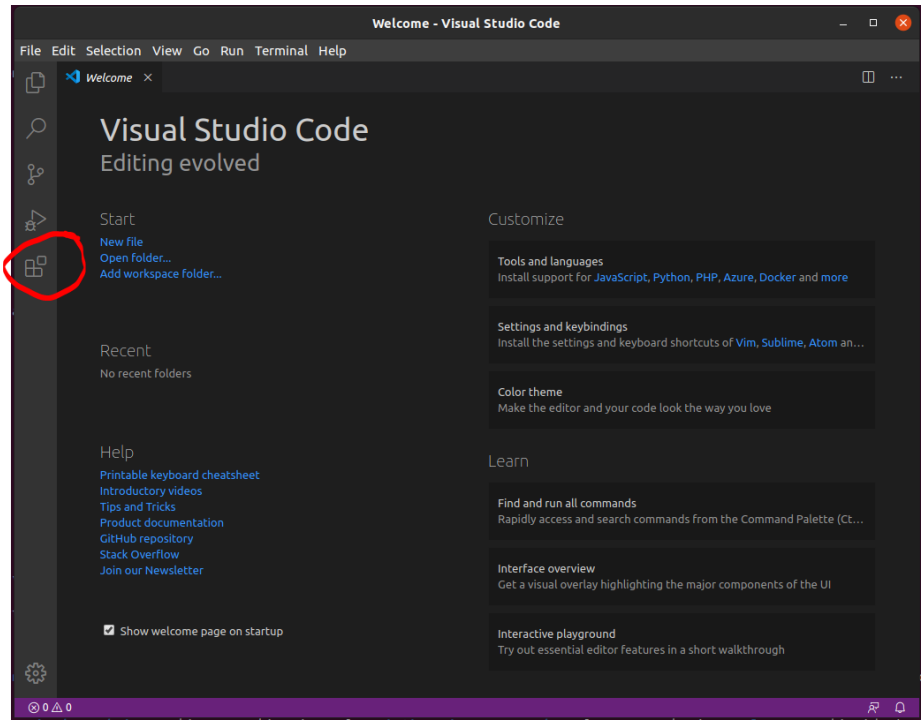

Figure 1: startup

In the search box type `clangd` to search for the extension. You should get exactly one result. After clicking on `Install` you are all done:

Now you should be open to open a folder (`File→Open Folder...` or press `Ctrl+k Ctrl+O`) and open the source folder. After opening a C++ file, you should be able to navigate code and use auto completion (it might take a while for all features to work, as `clangd` will need to index the code).

Warning: The first time you open a C++ file, VS Code will ask you whether you want to install the C++ extension. Don't install those, they will break stuff!

Troubleshooting

If just doing the above doesn't work you will need to make sure that you're including a `compile_commands.json` file in the root directory of your FDB folder.

This file is generated in your `~/build` folder when you run CMAKE with the `-DCMAKE_EXPORT_COMPILE_COMMANDS=ON` option (see above).

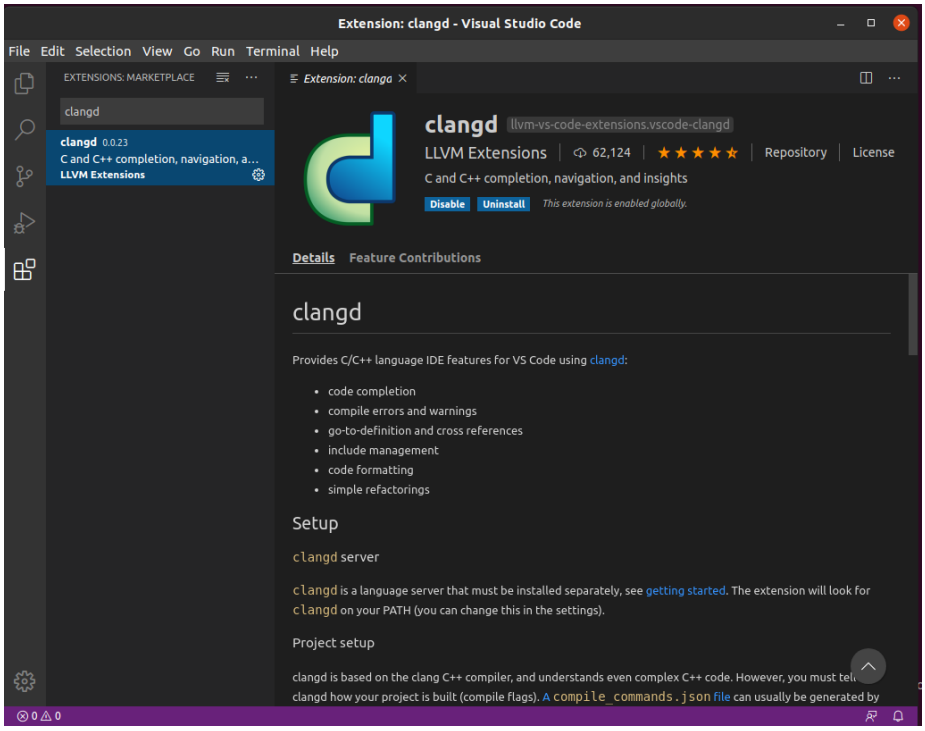

Figure 2: image

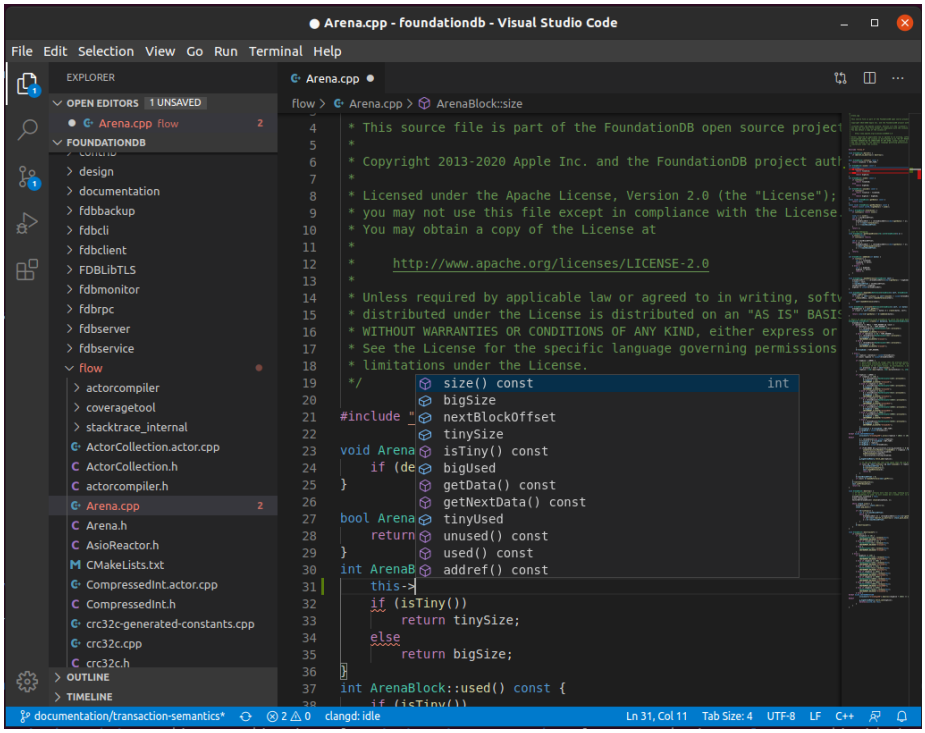

Figure 3: image

You can either run `make processed_compile_commands` (this may not work) to get this file into your local fdb project directory or create a symbolic link to it: `ln -s /home/<user>/build/compile_commands.json /home/<user>/foundationdb/`

Setting up clang-format with clangd on VSCode

Since FDB doesn't use a linter as part of it's CI (yet) its important that developers format their code locally before commiting (using clang-format).

To set up clang-format on VSCode follow these steps (assumes clangd extension is already installed):

1. Go to settings (type ? + ,). In the search bar look for format
2. Set the Editor: Default Formatter to `llvm-vs-code-extensions.vscode-clangd`
3. Check the box Editor: `Format On Save`
4. Make sure Editor: `Format On Save Mode to file`
5. Now each time you save a file it should clang format any changes

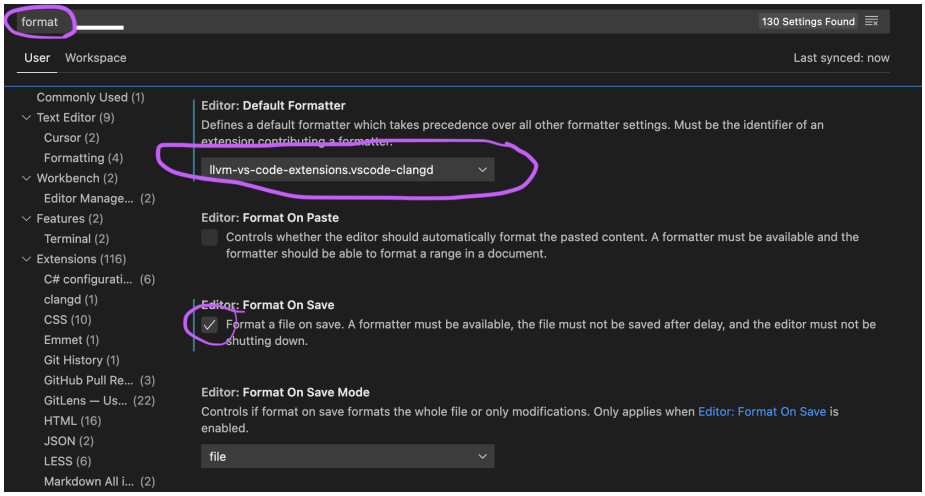

Figure 4: image