# Alpaquita Linux: Debugging apps running in Docker with JetBrains and VSCode

Java, C, and Python

Alpaquita Linux
Revision 1.0
December 2024

bell/soft

# Contents

# 1. Overview

If you have a setup to deploy your application using containers, debugging your application by connecting to debugging server that is running in your development docker container has the following advantages:

- Environment consistency: minimize environment-related bugs.
- Isolation of dependencies: less clutter in your local machine, avoid version conflicts.
- Easy cleanup and reset: restart, delete or stop your container as necessary.

The goal of this guide is to display how to remotely debug an application written in Java, Python, or C running inside a docker container. We will use CLion, IntelliJ IDEA, and VSCode to demonstrate this. We do not cover PyCharm in the guide, because at the time of writing, the remote debugging feature is limited to the paid version of their software.

Each section includes a sample application for those without an existing project who still want to follow along.

Although this document covers remote debugging with three popular languages using three IDEs, the key patterns in remote debugging are the same for any environment:

1. Find out whether your IDE can connect to a debugging server.
2. Learn which debugging tool your IDE uses. For example, PyCharm uses `pydevd-pycharm` and VSCode uses `debugpy` for Python by default.
3. Adjust or create a Docker image that exposes the debugging server over a network.
4. Configure your IDE to use the debugging server exposed by a Docker container for debugging.

After studying the examples in this guide, you should be able to adapt them to your own needs. For example, you can use `delve` debugger to debug Go applications using GoLand. And as an alternative to IntelliJ IDEA, use Eclipse for Java debugging and so forth.

The next parts demonstrate how to use popular IDEs to debug applications written in Java, Python, or C.

# 2. Java

We will use a sample Spring application, Spring PetClinic, to explain how to remotely debug Java docker applications. Spring PetClinic was dockerized using Liberica as a base image.

We will also use IntelliJ IDEA, as it is popular among Java developers.

## Prerequisites

- Docker installed and running
- Installed IntelliJ IDEA

## Dockerizing the PetClinic application

> **Note:**
>
> If you already have a containerized Java application, you can skip this step and replace the `petclinic-liberica` with your application image name.

## Application overview

Spring PetClinic is a CRUD application that uses technologies such as Spring Boot, Thymeleaf, Bootstrap, in-memory database H2.

Overall, it is a well-known and suitable application for use as an example.

## Clone the PetClinic application from GitHub

Use the following command to clone the sample application.

```
$ git clone https://github.com/spring-projects/spring-petclinic.git
```

## Dockerfile

Place the contents of the Dockerfile below into a file `Dockerfile` inside the `spring-petclinic` directory we have just cloned.

```
# Create a stage for resolving and downloading dependencies.
FROM bellsoft/liberica-openjdk-alpine:21 AS deps

# Download dependencies as a separate step to take advantage of Docker's
# caching.
WORKDIR /build

COPY --chmod=0755 mvnw mvnw
COPY .mvn/ .mvn/

RUN --mount=type=bind,source=pom.xml,target=pom.xml \
    --mount=type=cache,target=/root/.m2 \
    ./mvnw dependency:go-offline -DskipTests


# Create a stage for building the application based on the stage with
# downloaded dependencies.
FROM deps AS package

WORKDIR /build
COPY ./src src/
RUN --mount=type=bind,source=pom.xml,target=pom.xml \
    --mount=type=cache,target=/root/.m2 \
    ./mvnw package -DskipTests && \
    mv target/$(./mvnw help:evaluate -Dexpression=project.artifactId \
        -q -DforceStdout)-$(./mvnw help:evaluate -Dexpression=project.version \
        -q -DforceStdout).jar \
        target/app.jar


# Create a stage for extracting the application into separate layers.
FROM package AS extract

WORKDIR /build

RUN java -Djarmode=layertools -jar target/app.jar \
        extract --destination target/extracted

# Create a new stage for running the application that contains the minimal
# runtime dependencies. We use liberica-openjre-alpine:21
# because there is no need to use a full blown JDK just to run the app.
FROM bellsoft/liberica-openjre-alpine:21 AS final

# Create a non-privileged user that the app will run under.
ARG UID=10001
RUN adduser \
    --disabled-password \
    --gecos "" \
    --home "/nonexistent" \
    --shell "/sbin/nologin" \
    --no-create-home \
    --uid "${UID}" \
    appuser
USER appuser

# Copy the executable from the "package" stage.
COPY --from=extract build/target/extracted/dependencies/ ./
COPY --from=extract build/target/extracted/spring-boot-loader/ ./
COPY --from=extract build/target/extracted/snapshot-dependencies/ ./
COPY --from=extract build/target/extracted/application/ ./

# Expose 8080, as we will use this as the application port.
EXPOSE 8080

# Specifying the command that will executed when the container starts.
ENTRYPOINT [ "java", "org.springframework.boot.loader.launch.JarLauncher" ]
```

## Building the `petclinic-liberica` image

Use the following commands to build the `petclinic-liberica` image.

```
$ cd spring-petclinic
$ docker build --tag petclinic-liberica .
```

In this section we have created the example production image that we want to debug.

# Creating a debugging image

## Overview

To debug the PetClinic application, we use Java Debug Wire Protocol (JDWP), which is the protocol used for communication between a debugger (the IDE) and the Java virtual machine (the PetClinic app). It helps to perform debugging tasks such as setting breakpoints, stepping through code, and inspecting variables in IntelliJ IDEA.

Liberica and many other docker java images come with JDWP; therefore, you do not need to install other tools. We will instruct JVM to use JDWP using the command line arguments.

`JAVA_TOOL_OPTIONS` environment variable can be used to specify command line options for java launcher. The content of the `JAVA_TOOL_OPTIONS` environment variable is a list of arguments separated by space.

We can append a command line option required to start java debugging server to `JAVA_TOOL_OPTIONS`.

The option needed by java launcher is the following:

```
-agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=*:5005
```

**Tip**

If you want to use a port other than 5005 for debugging server, change `adress=5005` to `address=<your_port>`.

**Tip**

Change `suspend=n` to `suspend=y` if you want the app to be suspended immediately before the main class is loaded. The app will wait until you connect to the java debugging server.

## Edit Dockerfile.debug

Copy the contents of the Dockerfile above into a file `Dockerfile.debug` in the `spring-petclinic` directory.

**Important:**

This section assumes you are using java launcher for starting your app. If you use another method to run your application inside a container, the approach described here may not work. In that case, look for an alternative way to add `jdwp` capabilities to your launcher.

```
# Change this to your production image
FROM petclinic-liberica

# Add command line option for enabling JDWP debugging server
ENV JAVA_TOOL_OPTIONS="${JAVA_TOOL_OPTIONS} \
    -agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=*:5005"
```

## Build and run debugging image

Use the following commands to build the debugging image. Remember to expose the port that was specified, in this case `5005`, defined in the `JAVA_TOOL_OPTIONS` environment variable.

```
$ docker build --tag petclinic-liberica-debug -f Dockerfile.debug .
$ docker run -p8080:8080 -p5005:5005 petclinic-liberica-debug
Picked up JAVA_TOOL_OPTIONS:
-agentlib:jdwp=transport=dt_socket,server=y,suspend=y,address=*:5005
Listening for transport dt_socket at address: 5005
...
```

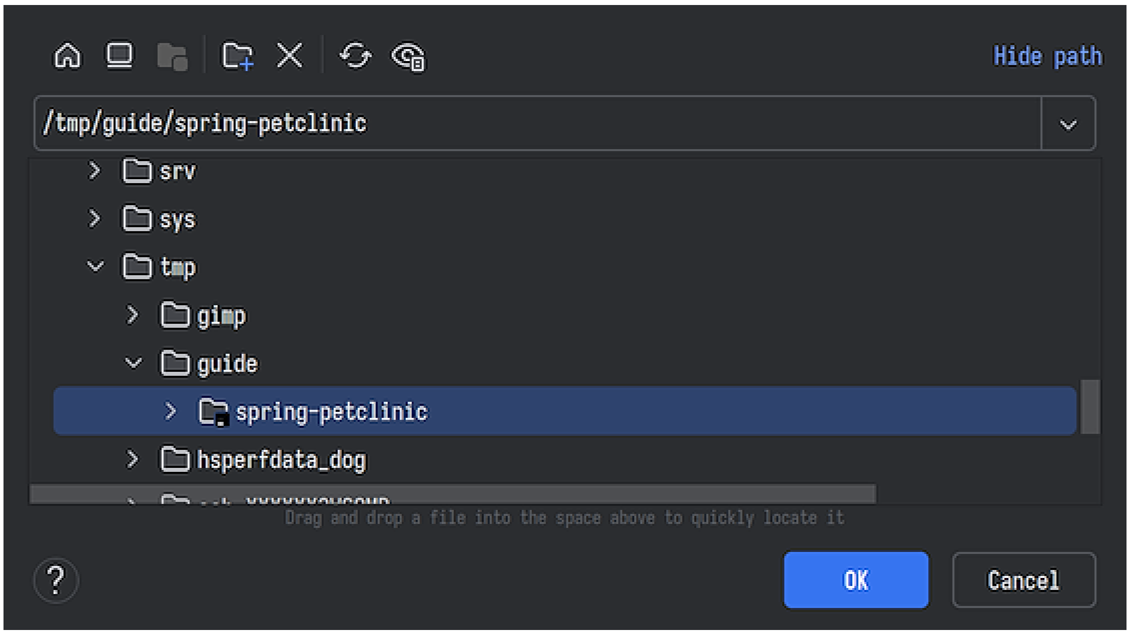
As stated in the output, the application is listening for a debugging session on port 5005.

## Remote debugging using IntelliJ IDEA

Before proceeding, ensure the debugging container is running and the Java debugging server is set up, waiting at port 5005.

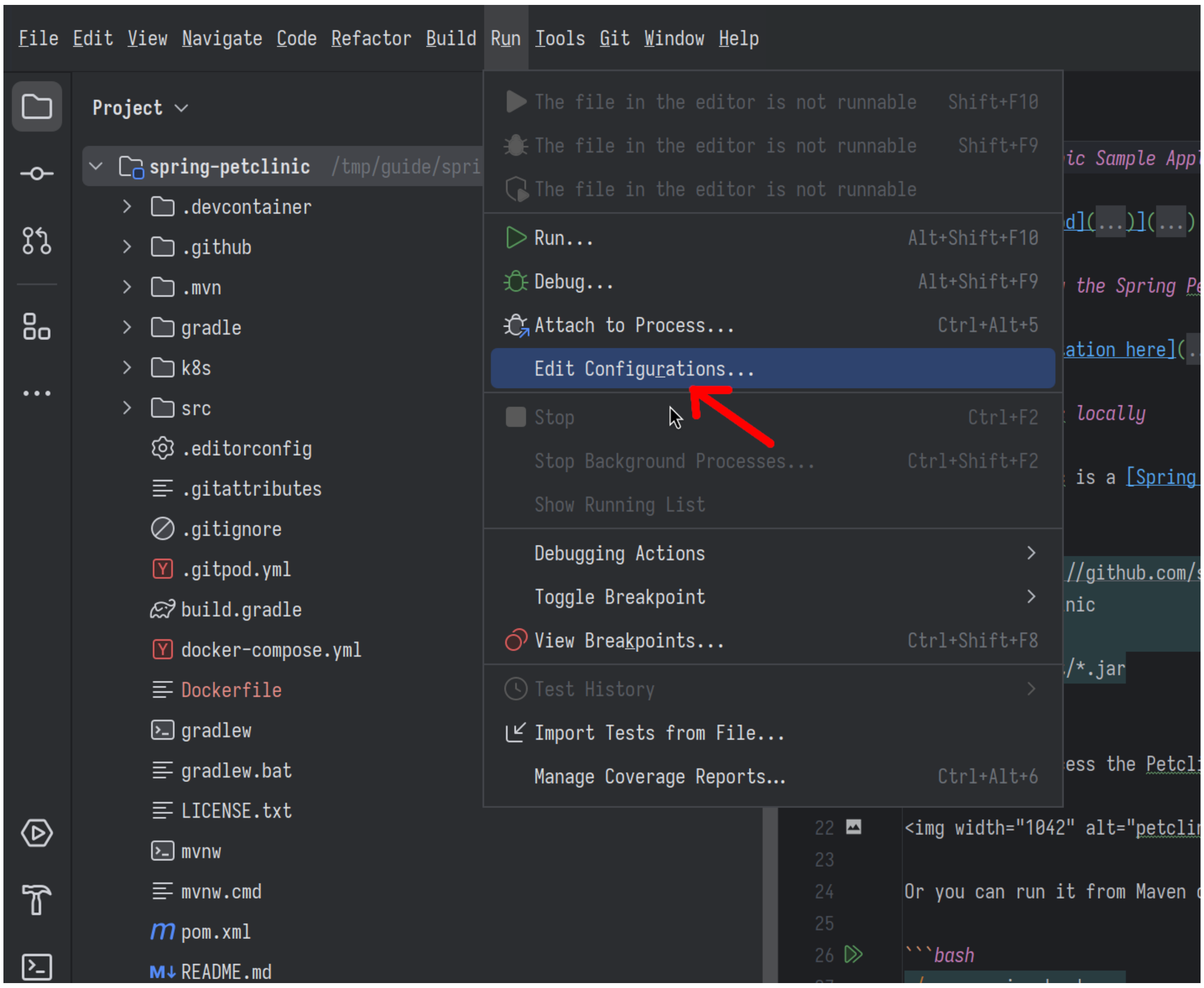
## Configure remote debugging

1. Open the previously cloned `spring-petclinic` directory using IntelliJ IDEA by clicking **File** > **Open** and then selecting the path to `spring-petclinic` directory. Click **OK**.

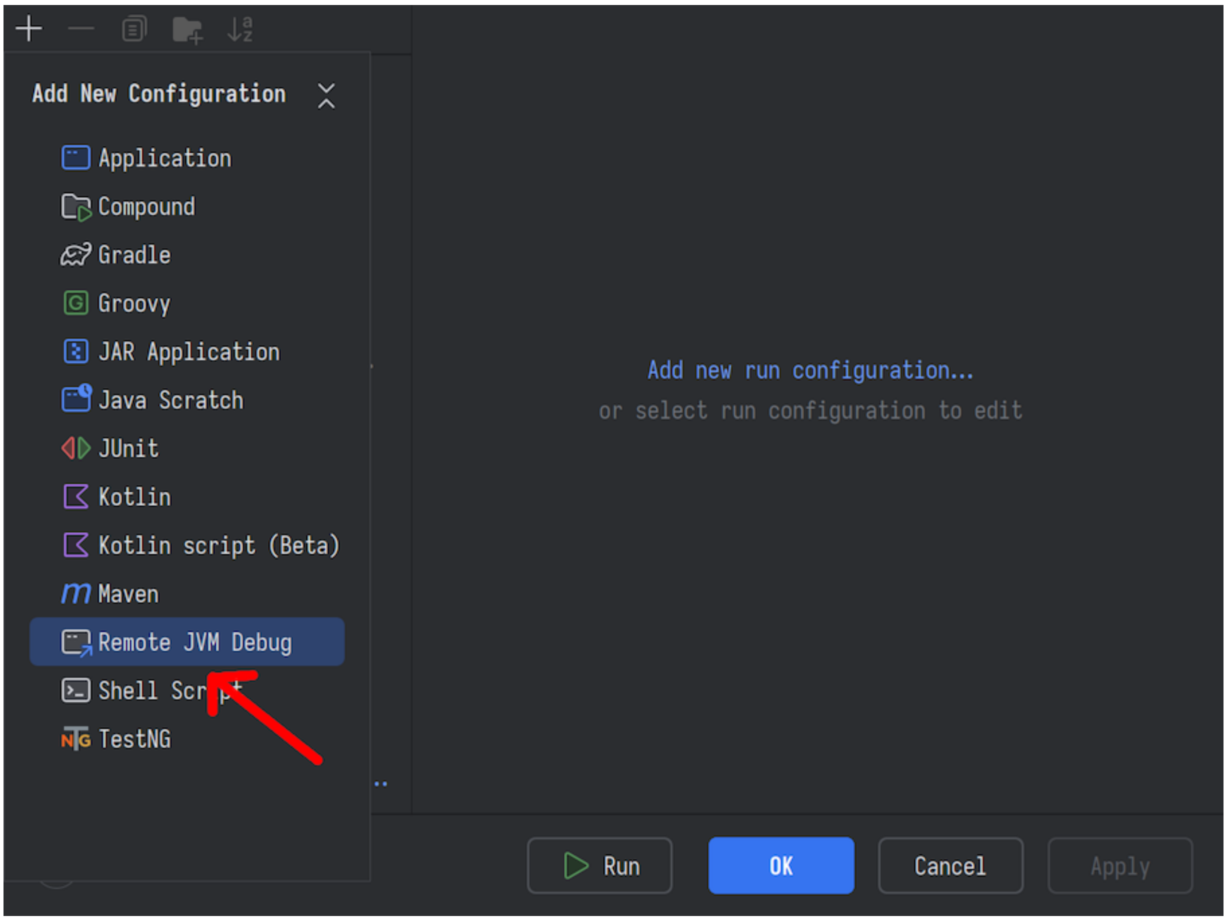

Hide path
/tmp/guide/spring-petclinic
srv
sys
tmp
gimp
guide
spring-petclinic
hsperfdata_dog
Drag and drop a file into the space above to quickly locate it
?
OK
Cancel

2. Click **Open as Maven project** and click **Trust Project**. You should be now in the `spring-petclinic` project.

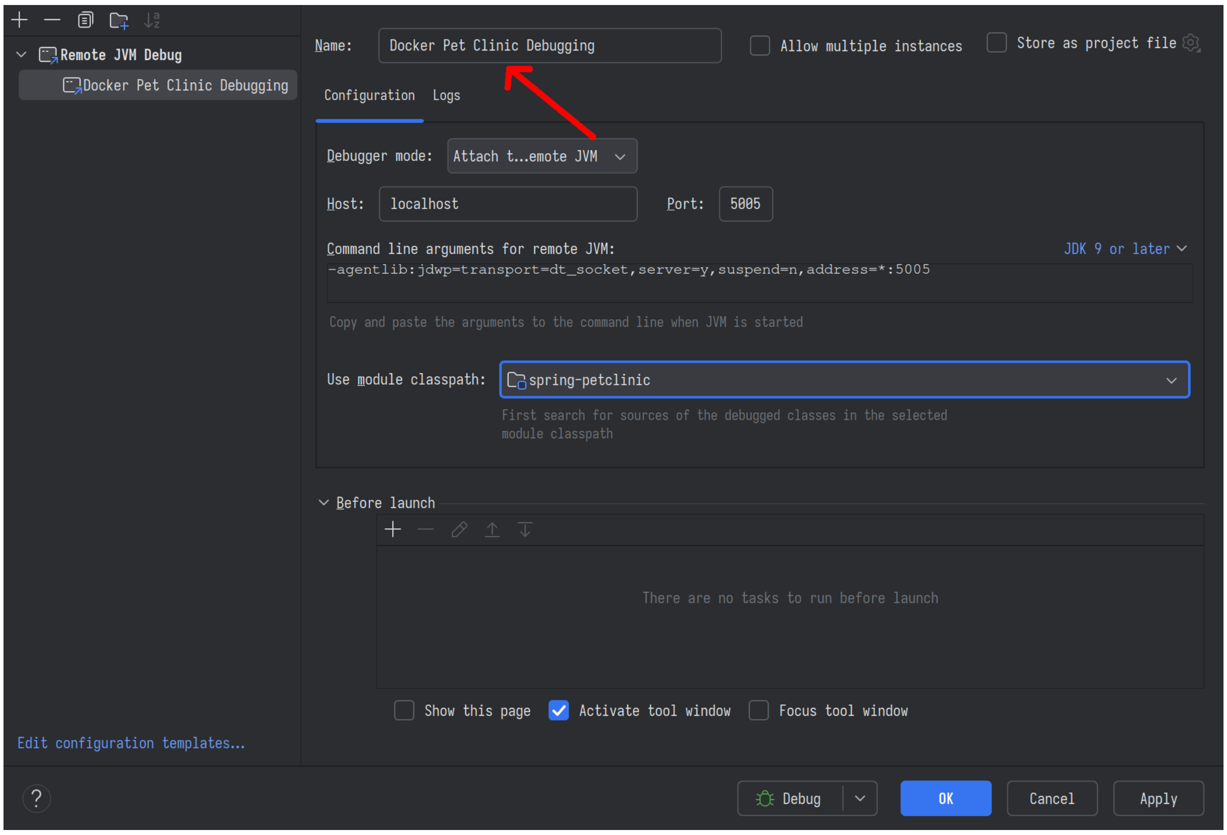
3. From the main menu, select **Run** > **Edit Configurations**.


File Edit View Navigate Code Refactor Build Run Tools Git Window Help
Project
spring-petclinic /tmp/guide/spri
.devcontainer
.github
.mvn
gradle
k8s
src
.editorconfig
.gitattributes
.gitignore
.gitpod.yml
build.gradle
docker-compose.yml
Dockerfile
gradlew
gradlew.bat
LICENSE.txt
mvnw
mvnw.cmd
pom.xml
README.md
The file in the editor is not runnable Shift+F10
The file in the editor is not runnable Shift+F9
The file in the editor is not runnable
Run... Alt+Shift+F10
Debug... Alt+Shift+F9
Attach to Process... Ctrl+Alt+5
Edit Configurations...
Stop Ctrl+F2
Stop Background Processes... Ctrl+Shift+F2
Show Running List
Debugging Actions
Toggle Breakpoint
View Breakpoints... Ctrl+Shift+F8
Test History
Import Tests from File...
Manage Coverage Reports... Ctrl+Alt+6

4. Click "+" (plus) on the top left to add a new configuration and select **Remote JVM Debug**.



You can give this configuration a meaningful name, such as "Docker Pet Clinic Debugging".


Name: Docker Pet Clinic Debugging

Debugger mode: Attach t…emote JVM

Host: localhost  Port: 5005

Command line arguments for remote JVM: JDK 9 or later

```
-agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=*:5005
```

Copy and paste the arguments to the command line when JVM is started

Use module classpath: spring-petclinic

First search for sources of the debugged classes in the selected module classpath

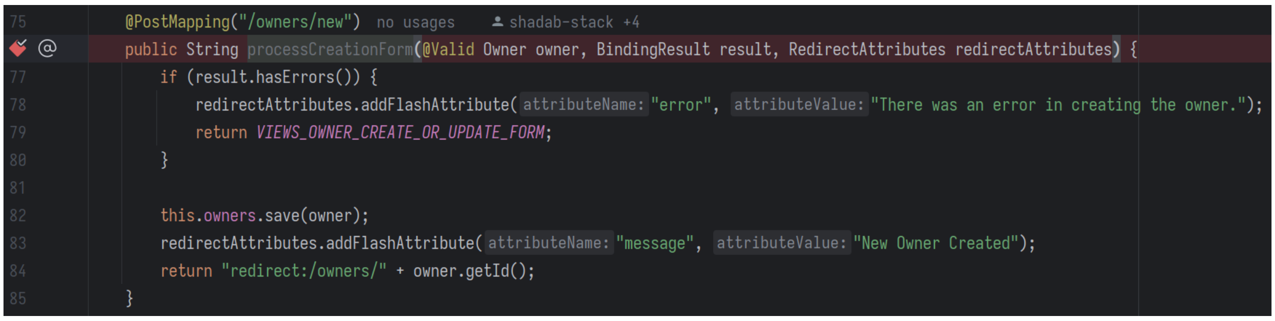
Before launch

There are no tasks to run before launch

Note that the default values for **Host** and **Port** are `localhost` and `5005` respectively. Update these to match the host address and port configured in the previous steps if you changed the values.

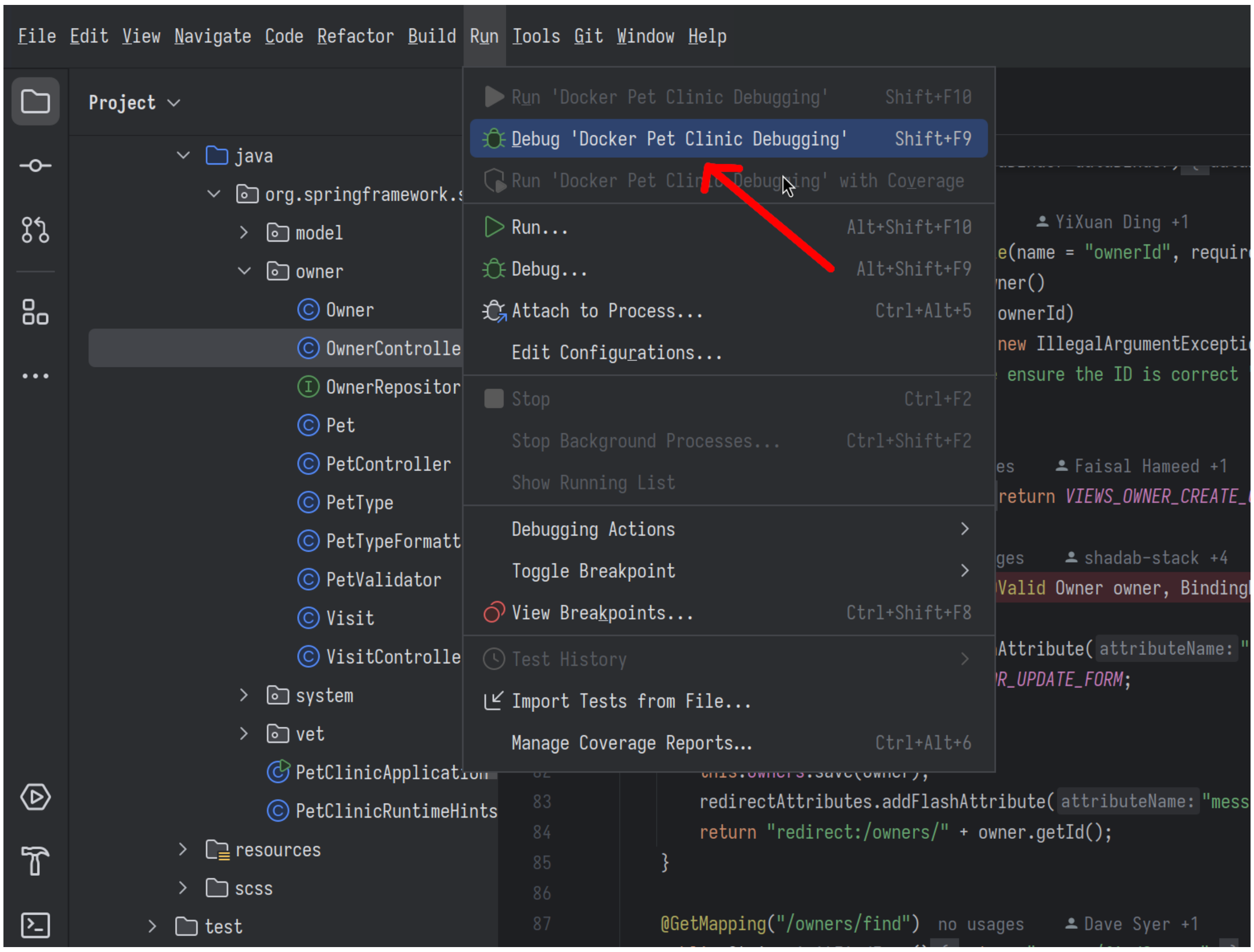
5. Click **OK** to save the configuration.

To ensure debugging is working as expected, let's insert a breakpoint at the `processCreationForm` method of the `OwnerController` class.

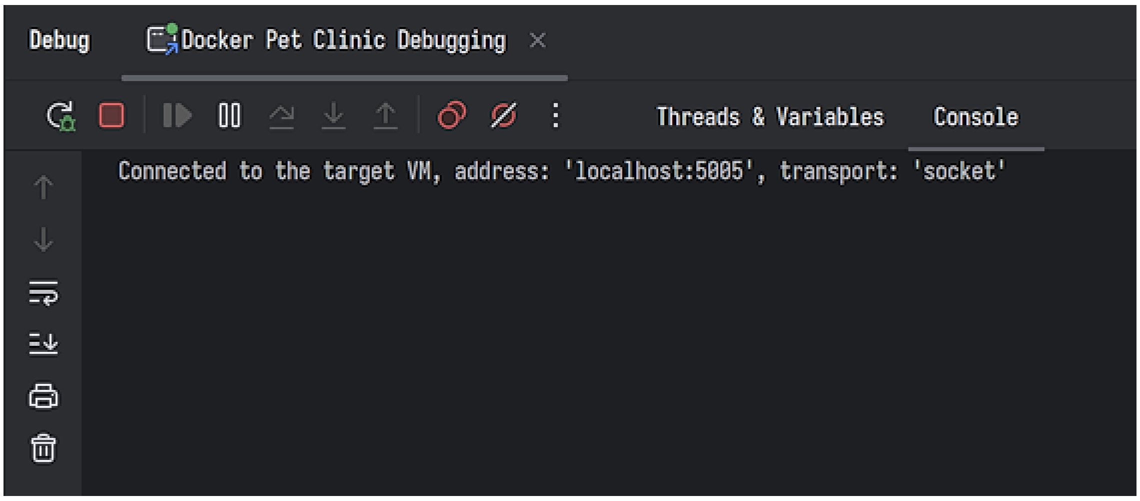

```
@PostMapping("/owners/new")  no usages   shadab-stack +4
public String processCreationForm(@Valid Owner owner, BindingResult result, RedirectAttributes redirectAttributes) {
    if (result.hasErrors()) {
        redirectAttributes.addFlashAttribute( attributeName: "error",  attributeValue: "There was an error in creating the owner.");
        return VIEWS_OWNER_CREATE_OR_UPDATE_FORM;
    }

    this.owners.save(owner);
    redirectAttributes.addFlashAttribute( attributeName: "message",  attributeValue: "New Owner Created");
    return "redirect:/owners/" + owner.getId();
}
```

Let's try to debug the application using the configuration we have created.

From the main menu, click **Run** > **Debug 'Docker Pet Clinic Debugging'**(or select the name assigned to the debug configuration earlier).


File Edit View Navigate Code Refactor Build Run Tools Git Window Help
Project
Run 'Docker Pet Clinic Debugging' Shift+F10
Debug 'Docker Pet Clinic Debugging' Shift+F9
Run 'Docker Pet Clinic Debugging' with Coverage
Run... Alt+Shift+F10
Debug... Alt+Shift+F9
Attach to Process... Ctrl+Alt+5
Edit Configurations...
Stop Ctrl+F2
Stop Background Processes... Ctrl+Shift+F2
Show Running List
Debugging Actions
Toggle Breakpoint
View Breakpoints... Ctrl+Shift+F8
Test History
Import Tests from File...
Manage Coverage Reports... Ctrl+Alt+6
java
org.springframework.s
model
owner
Owner
OwnerControlle
OwnerRepositor
Pet
PetController
PetType
PetTypeFormatt
PetValidator
Visit
VisitControlle
system
vet
PetClinicApplication
PetClinicRuntimeHints
resources
scss
test

You should see a message in the debugging console that you are connected to `localhost:5005` or your specified port over the network.

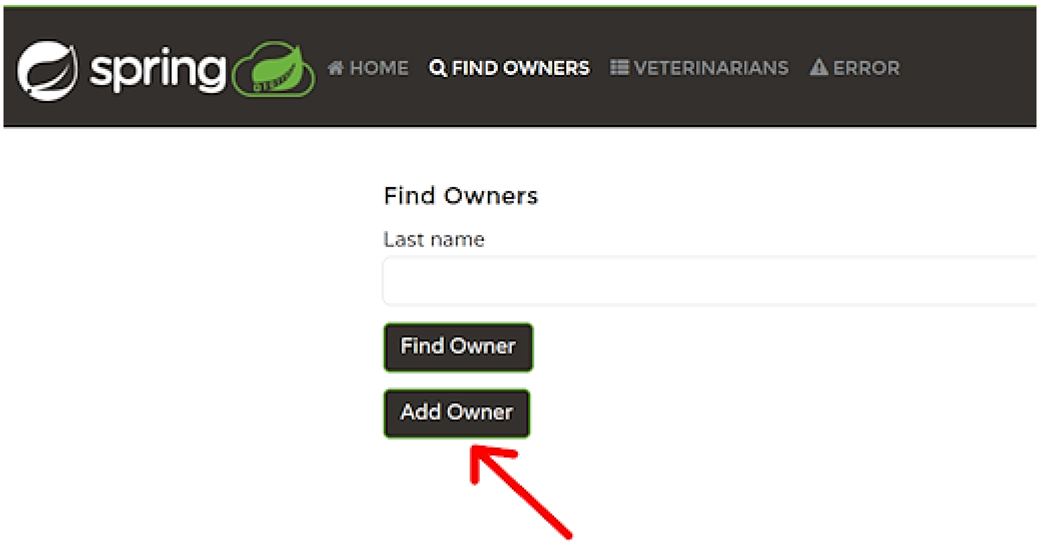


```
Connected to the target VM, address: 'localhost:5005', transport: 'socket'
```

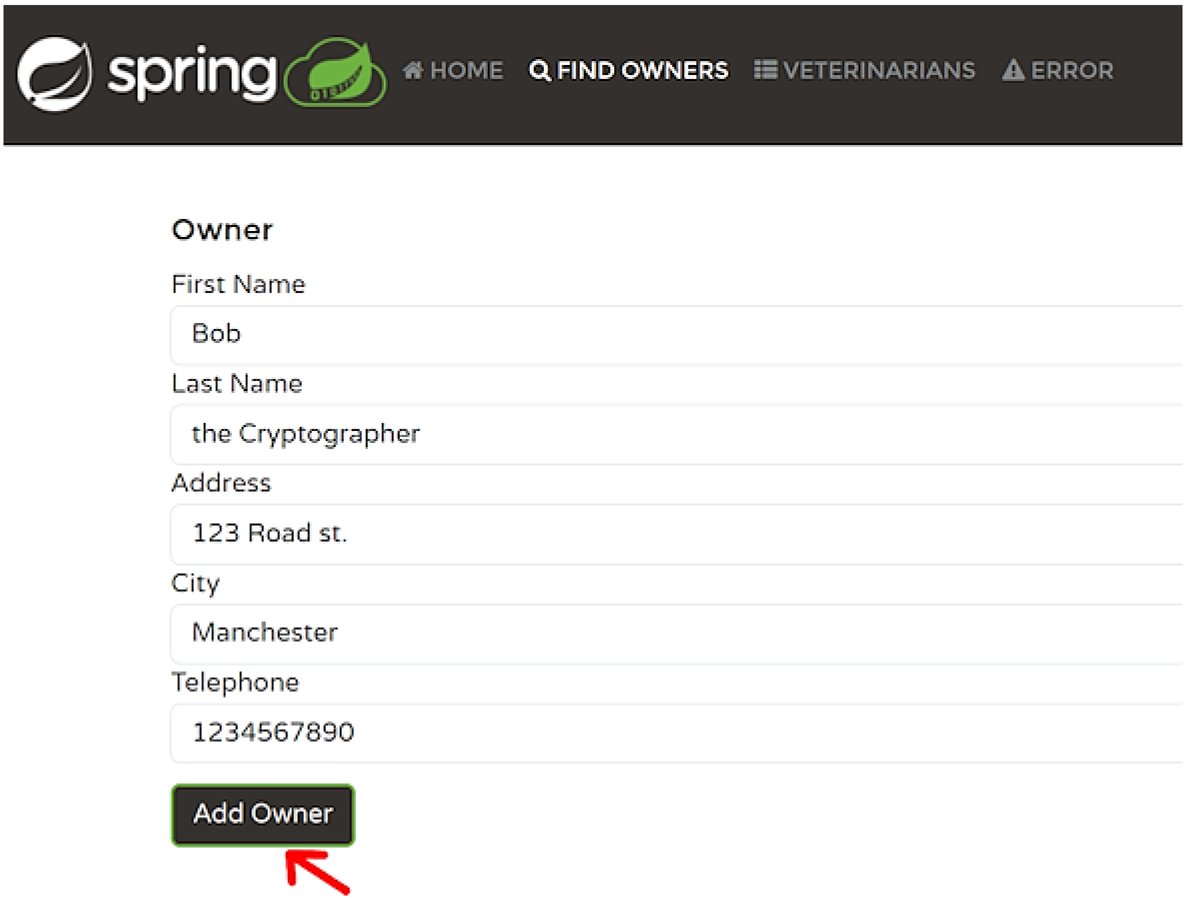
`processCreationForm` method is responsible for handling POST requests to the endpoint /owners/new. It manages the creation of a new Owner entity, validates the form input, and redirects users based on the result of the validation.

Let’s try to debug that method.

1. In your browser, go to http://localhost:8080 and click **Add Owner** on the **Find Owners** section.



2. Fill the form and click **Add Owner**.



Owner

| | |
|---|---|
| First Name | Bob |
| Last Name | the Cryptographer |
| Address | 123 Road st. |
| City | Manchester |
| Telephone | 1234567890 |

Now the web page waits, because we have instructed the debugger to stop at the `processCreationForm` method in the `OwnerController` class.

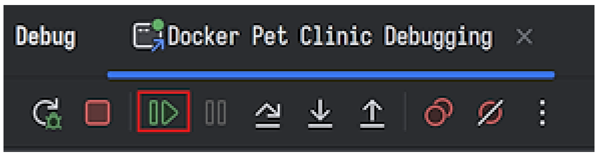
We can see that the method first instantiates an owner object with the data we provided and then saves it to the `owners` which is a Spring Repository.

We can step through the code, set other breakpoints, evaluate expressions, and more.

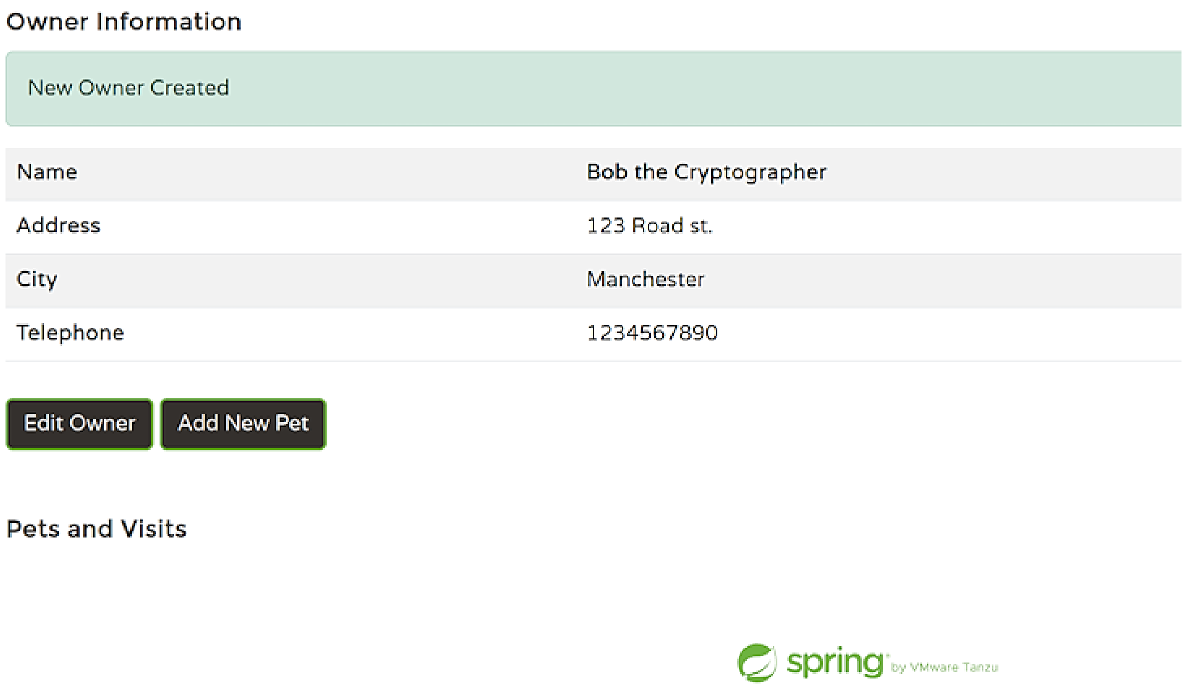
**Note:**

See the debugging section of IntelliJ IDEA Documentation to get more information about debugging in IntelliJ IDEA.

3. Click **Continue** to let the application continue normally.



Now we see that the application continues and finishes the POST request normally. Every time the application runs the code we marked with a breakpoint, it stops and waits for an input in IntelliJ IDEA, so we can debug it further.



Owner Information

New Owner Created

| | |
|---|---|
| Name | Bob the Cryptographer |
| Address | 123 Road st. |
| City | Manchester |
| Telephone | 1234567890 |

# 3. Python

## Prerequisites

- Docker installed and running
- Working VSCode with a Python extension installed

We will debug the FastAPI application running on uvicorn web server. It should be easy to adjust the setup to other web frameworks like Flask or Django.

We use `debugpy` module since it is provided by Python Debugger extension and bundled with the Python extension from VSCode marketplace.

## Dockerizing a FastAPI application

**Note:**

Skip this section if you already have a python image of your application and replace the base image in `Dockerfile.debug` with your application image name.

The final structure of the sample project should look like the following:

```
$ tree
.
├── Dockerfile
├── Dockerfile.debug
├── requirements.txt
└── src
    └── main.py

2 directories, 4 files
```

## Application: quadratic equation solver

We have a simple application that returns the roots of the following quadratic equation `ax^2 + bx + c = 0`.

```
$ curl -s \
    --request GET \
    --url 'http://localhost:8000/solve_quadratic?a=1&b=-8&c=15' \
    | jq
```

```
{
  "x1": 5.0,
  "x2": 3.0
}
```

## main.py

Create a directory `src` and copy the contents of the following python code into `main.py` in the `src` directory.

**Note:**

We have introduced a not-so-subtle bug in the logic where we calculate x1 and x2. We should be dividing by `(2 * a)` instead of `(2 * c)`.

```
from fastapi import FastAPI, HTTPException
from math import sqrt
from typing import Dict

app = FastAPI()

@app.get("/solve_quadratic")
def solve_quadratic(a: float, b: float = 0, c: float = 0) -> Dict[str, float]:
    if a == 0:
        raise HTTPException(status_code=400,
            detail="Coefficient 'a' cannot be zero.")

    discriminant = b**2 - 4*a*c

    if discriminant < 0:
        raise HTTPException(status_code=400,
            detail="No real solutions, discriminant is negative.")

    # Bug introduced here: it should be (2 * a)
    x1 = (-b + sqrt(discriminant)) / (2 * c)
    x2 = (-b - sqrt(discriminant)) / (2 * c)

    if x1 == x2:
        return {"x": x1}

    return {"x1": x1, "x2": x2}
```

## Dockerfile

Create a file `Dockerfile` with following contents.

```
FROM bellsoft/alpaquita-linux-python:3.12-musl

# Print log messages immediately instead of them being buffered
ENV PYTHONUNBUFFERED=1

WORKDIR /src

# Activate virtual environment
ENV VIRTUAL_ENV=/src/.venv
ENV PATH="$VIRTUAL_ENV/bin:$PATH"
RUN python3 -m venv $VIRTUAL_ENV

# Install dependencies:
COPY requirements.txt .
RUN pip3 install -r requirements.txt

# Copy source code
COPY src/ .

# Start the uvicorn web server. 8000 is the default port
CMD ["python", "-m", "uvicorn", "main:app", "--host", "0.0.0.0"]
```

## requirements.txt

Put the following content into a file `requirements.txt`

```
fastapi
uvicorn
```

Build your image with the following command.

```
$ docker build --tag fastapi-app .
```

## Building a debugging image

It is important that both local dev environment, where we debug the python code, and the container, where the application is running, have the `debugpy` module available.

We will install it in the debugging image and also instruct docker to run `debugpy` as a default command when the container starts.

### Dockerfile.debug

1. Copy the contents of the following Dockerfile into a file `Dockerfile.debug`.

   > **Note:**
   >
   > We used the default port `5678` for `debugpy` debugging server port. You can use any available port.

   ```
   # Base image
   FROM fastapi-app

   WORKDIR /src

   # Install debugpy for vscode remote debugging
   RUN pip3 install debugpy

   # Append debugpy to the uvicorn command so it will be executed in a
   debugging
   # mode
   CMD ["python", "-m", "debugpy", "--listen", "0.0.0.0:5678", \
       "-m", "uvicorn", "main:app", "--host", "0.0.0.0"]
   ```

2. Build the debugging image with the following command.

```
$ docker build --tag fastapi-app-debug -f Dockerfile.debug .
```

3. Finally, run the debugging image.

```
$ docker run --rm -p8000:8000 -p5678:5678 fastapi-app-debug
```

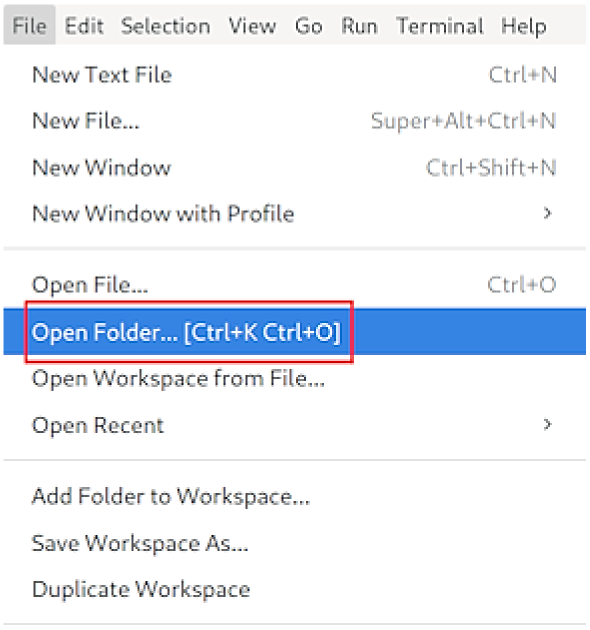
## Configuring VSCode for remote debugging

VSCode uses `debugpy` python module for debugging remotely. We will install Python Debugger for accessing the `debugpy` module.

Also, `launch.json` configuration file inside the `.vscode` directory dictates the behavior of the debugger in VSCode. We will be using this file to connect to debugging session in the container.

Before continuing, make sure your debugging docker application is running.

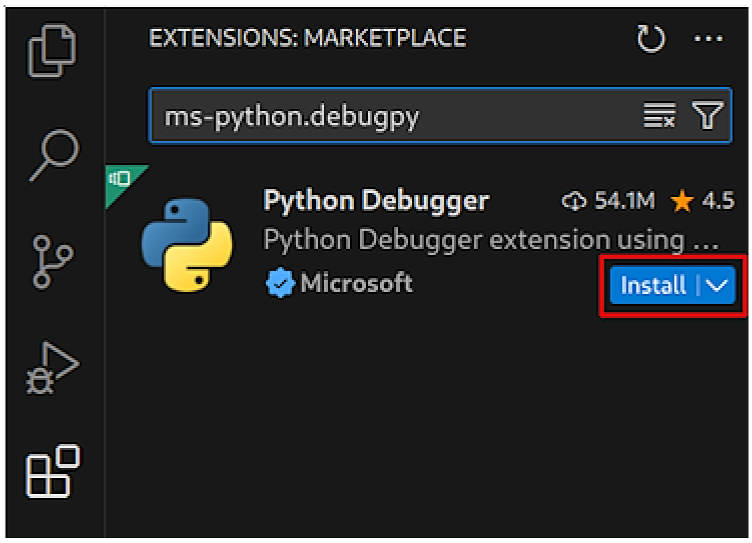
1. Let's open the source code directory of the FastAPI project.


File Edit Selection View Go Run Terminal Help
New Text File Ctrl+N
New File... Super+Alt+Ctrl+N
New Window Ctrl+Shift+N
New Window with Profile
Open File... Ctrl+O
Open Folder... [Ctrl+K Ctrl+O]
Open Workspace from File...
Open Recent
Add Folder to Workspace...
Save Workspace As...
Duplicate Workspace

2. Select the path for the parent directory of `main.py` from the previous steps. Click **Yes, I trust the authors**.

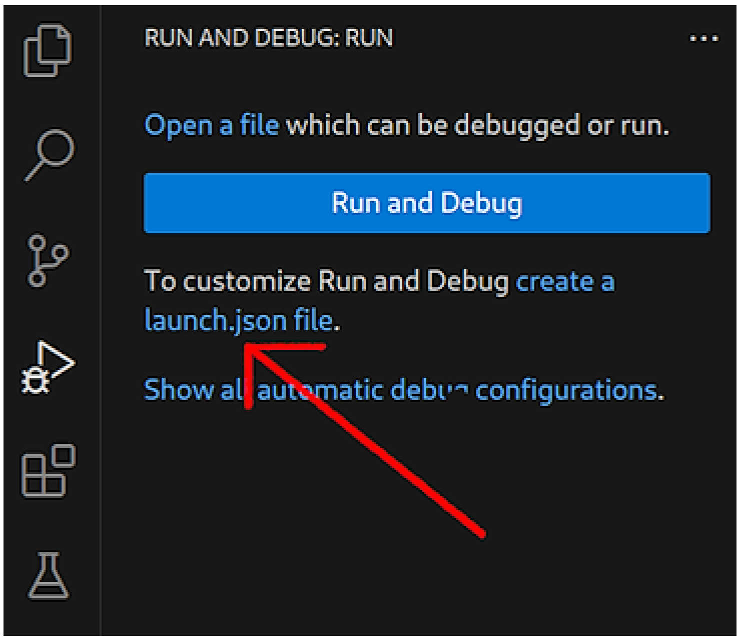
### Install extensions

From activity bar, select extensions, and search for `ms-python.debugpy`. Click **Install**.

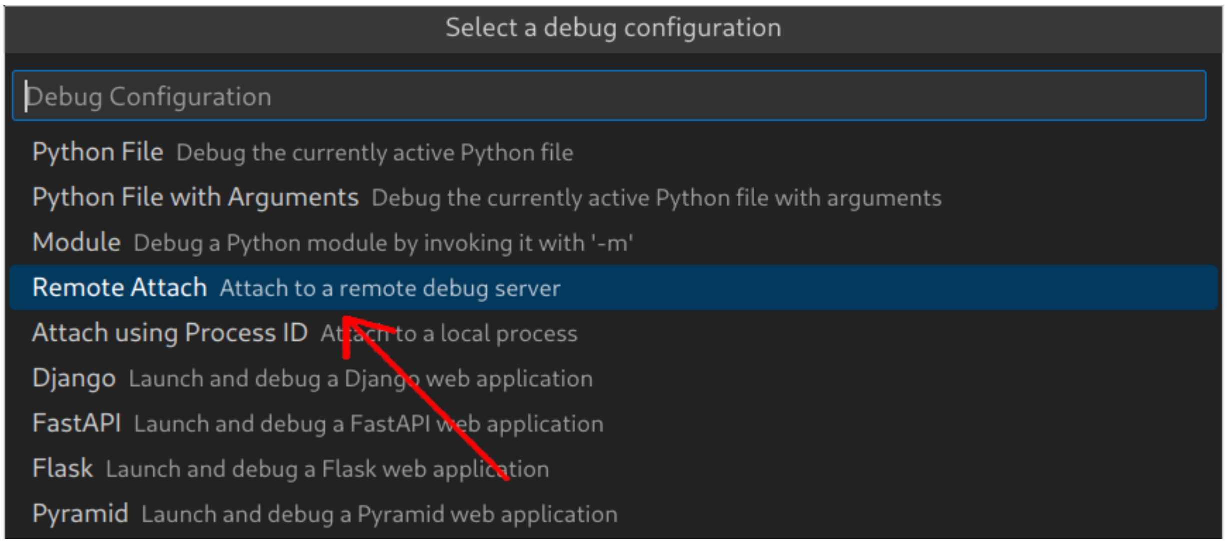


## Create launch.json

1. `launch.json` is necessary for configuring the debugger of VSCode. From activity bar, select **Run and Debug**. Click **create a launch.json file**.


RUN AND DEBUG: RUN
Open a file which can be debugged or run.
Run and Debug
To customize Run and Debug create a launch.json file.
Show all automatic debug configurations.

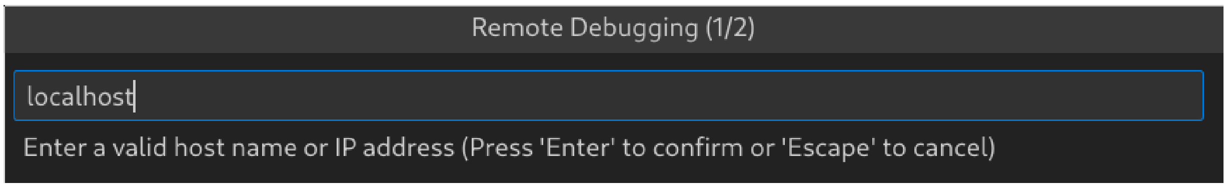
2. Select **Python Debugger** and select **Remote Attach**.


Select a debug configuration

Debug Configuration

Python File Debug the currently active Python file
Python File with Arguments Debug the currently active Python file with arguments
Module Debug a Python module by invoking it with '-m'
Remote Attach Attach to a remote debug server
Attach using Process ID Attach to a local process
Django Launch and debug a Django web application
FastAPI Launch and debug a FastAPI web application
Flask Launch and debug a Flask web application
Pyramid Launch and debug a Pyramid web application

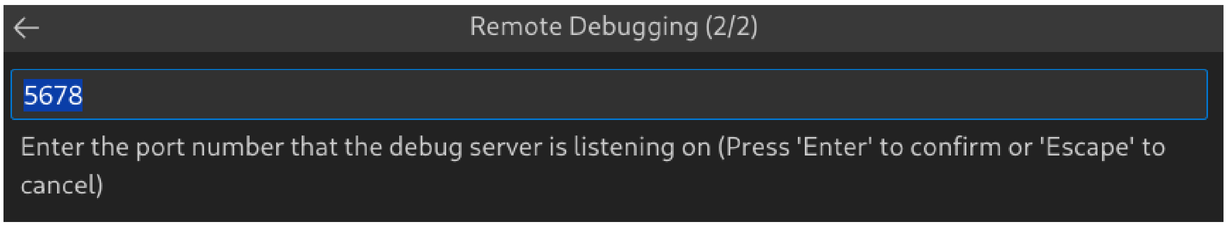
3. VSCode asks for an IP address of the debugging server. Keep the default option `localhost`.


Remote Debugging (1/2)

localhost

Enter a valid host name or IP address (Press 'Enter' to confirm or 'Escape' to cancel)

4. Either keep the default port number `5678` or change to your port in case you have used a different port.


Remote Debugging (2/2)

5678

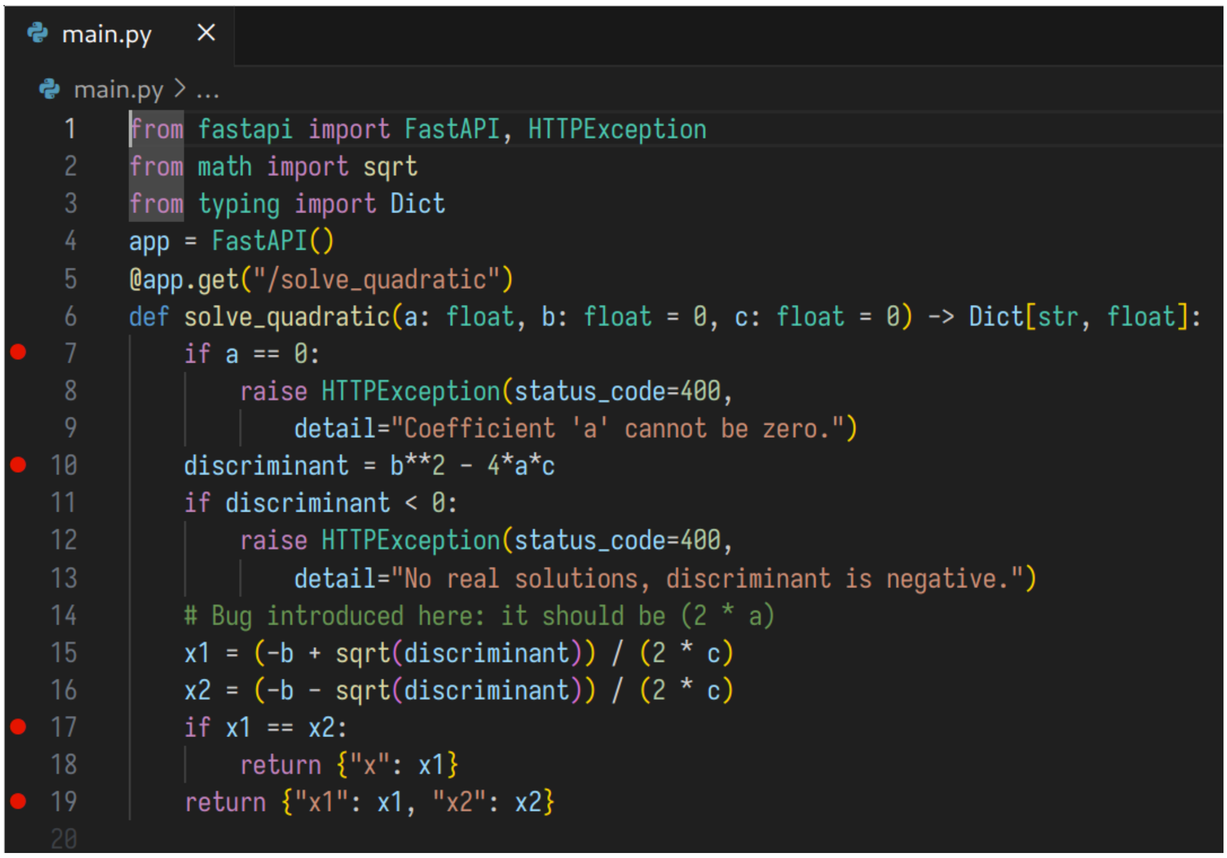
Enter the port number that the debug server is listening on (Press 'Enter' to confirm or 'Escape' to cancel)

5. Verify that VSCode created a `launch.json` file and looks similar to this:

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "Python Debugger: Remote Attach",
            "type": "debugpy",
            "request": "attach",
            "connect": {
                "host": "localhost",
                "port": 5678
            },
            "pathMappings": [
                {
                    "localRoot": "${workspaceFolder}",
                    "remoteRoot": "."
                }
            ]
        }
    ]
}
```

## Start debugging

1. Create some breakpoints in the application.



```python
from fastapi import FastAPI, HTTPException
from math import sqrt
from typing import Dict
app = FastAPI()
@app.get("/solve_quadratic")
def solve_quadratic(a: float, b: float = 0, c: float = 0) -> Dict[str, float]:
    if a == 0:
        raise HTTPException(status_code=400,
            detail="Coefficient 'a' cannot be zero.")
    discriminant = b**2 - 4*a*c
    if discriminant < 0:
        raise HTTPException(status_code=400,
            detail="No real solutions, discriminant is negative.")
    # Bug introduced here: it should be (2 * a)
    x1 = (-b + sqrt(discriminant)) / (2 * c)
    x2 = (-b - sqrt(discriminant)) / (2 * c)
    if x1 == x2:
        return {"x": x1}
    return {"x1": x1, "x2": x2}
```

2. Click **Run** > **Start Debugging**.

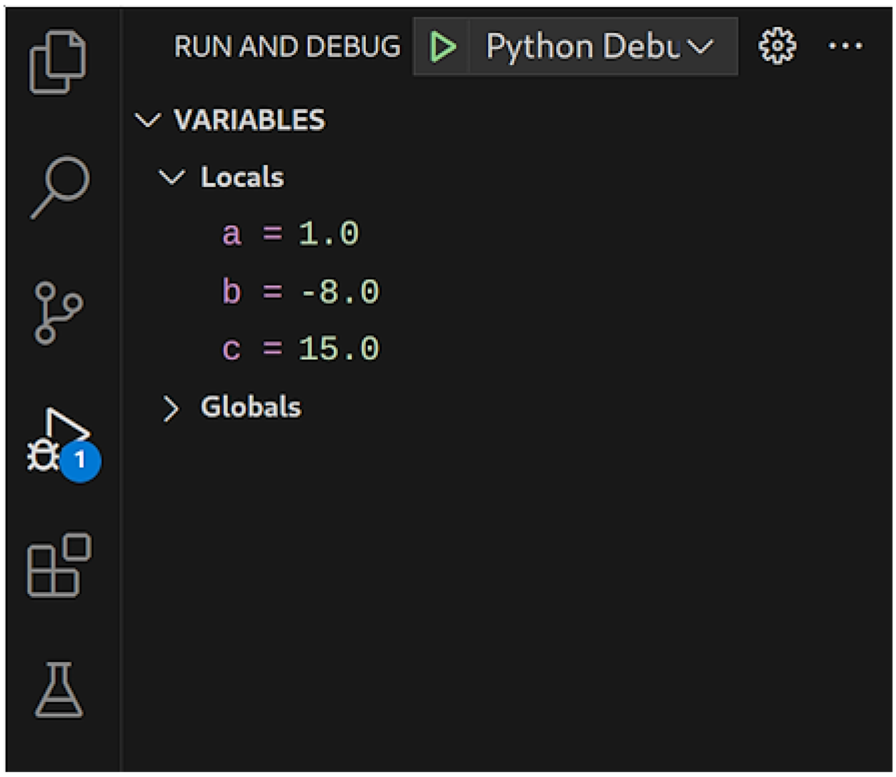
   You should now see a Debug toolbar at the top center.

3. From the terminal, send a get request to the FastAPI application as follows.

```
$ curl -s \
    --request GET \
    --url 'http://localhost:8000/solve_quadratic?a=1&b=-8&c=15' \
    | jq
```

You should see that the command "hangs". This is because we have set breakpoints in the `main.py` file and application freezes the execution for us at the breakpoints.

4. Go to **Run and Debug** from the activity bar. Observe that we have received the parameters of the request: `a`, `b` and `c` in the variables section.



```
a = 1.0
b = -8.0
c = 15.0
```

5. Click **Continue** from the Debug toolbar to jump to the next breakpoint. Click **Step over** to execute one line of code. We now see the value of `discriminant` from the variables section.

6. Step over until you reach the `x2 =` .... We see that `x1` and `x2` are calculated incorrectly.

You may debug further or stop the debugging session by pressing the red square on the debug toolbar.

This concludes the guide for remote debugging in Python.

For more information about debugging in VSCode, visit the [documentation section on the Visual Studio Code website].

# 4. C/C++

As an example in this part of the guide, we will use an application that returns a reversed version of the string received from the standard input. The application will be running inside an Alpaquita docker container. Then we will introduce a bug in the application and attempt to debug it remotely in CLion.

We will use `gdbserver`, which is a lightweight debugging server for gdb.

## Prerequisites

- Docker installed and running
- Installed CLion

## Dockerizing a C application

Upon completion of this section, you should have the following directory structure:

```
$ tree
.
├── Dockerfile
├── Makefile
└── src
    └── main.c
```

## Application: String reverser

We have a simple application that takes a string of characters from standard input, reverses it, and finally, prints the reversed string to the standard output.

Let’s provide the input "noel sees leon" to the program.

```
$ docker run -it reverser:latest
noel sees leon
Reversed string:
noel sees leon
```

## main.c

```c
#include <string.h>
#include <stdio.h>


/* reverse:  reverse string s in place */
void reverse(char s[]) {
    int n = strlen(s);

    for (int i = 0,j = n - 1; i < j; i++, j--) {
        char tmp = s[i];
        s[i] = s[j];
        s[j] = tmp;
    }
}

/* main: get string from stdin, reverse and print it */
int main() {
    char str[256];
    fgets(str, sizeof(str), stdin);

    reverse(str);
    printf("Reversed string: %s\n", str);
}
```

## Makefile

```makefile
CC = gcc
SRC_DIR = src
BUILD_DIR := build
CFLAGS = -Wall -Wextra -O2

BIN = reverser

main: $(BUILD_DIR)/$(BIN)

$(BUILD_DIR)/$(BIN): $(SRC_DIR)/main.c
	@mkdir -p $(BUILD_DIR)
	$(CC) $(CFLAGS) -o $@ $<
```

```
clean:
    rm -rf $(BUILD_DIR)

.PHONY: main clean
```

## Dockerfile

The following Dockerfile has two stages. The First stage is for compiling the code. The second stage is only for executing the binary, hence it is lightweight.

```
FROM bellsoft/alpaquita-linux-gcc:14.2-musl AS build
WORKDIR /build
# Copy src and Makefile
COPY src src
COPY Makefile .
# Compile the app
RUN make

# Use a lightweight base for final image
FROM bellsoft/alpaquita-linux-base:stream-musl
WORKDIR /app
# Follow best practices and use a non-root user
RUN adduser -D user
USER user
# Copy the binary
COPY --from=build /build/build/reverser .
# Execute the binary
ENTRYPOINT ["./reverser"]
```

## Modified main.c

The `main.c` above works fine. Let’s say we want to take the input from arguments, not from the standard input. While doing so, we will introduce a bug in the new `main.c`.

```
#include <string.h>
#include <stdio.h>

/* reverse:  reverse string s in place */
void reverse(char s[]) {
    int n = strlen(s);
    for (int i = 0,j = n - 1; i < j; i++, j--) {
        char tmp = s[i];
        s[i] = s[j];
        s[j] = tmp;
    }
}

/* reverse: print reversed string from args */
int main(const int argc, char *argv[]) {
    /* We accept only one argument.If we receive more or less than one
    argument, exit with error */
    if (argc != 1) {
        fprintf(stderr, "Usage: ./reverser <string>\n");
        return 1;
    }

    // If we receive a too long string, exit
    if (strlen(argv[1]) > 256) {
        fprintf(stderr, "%s", "String too long\n");
        return 1;
    }
    // Copy the argument into a string
    char str[256];
    strncpy(str, argv[1], sizeof(str) - 1);

    reverse(str);
    printf("Reversed string: %s\n", str);
}
```

The modified code prints out the "wrong usage" error:

```
$ docker build --tag reverser-arg .
[+] Building 0.8s (15/15) FINISHED
...
$ docker run --rm -it reverser-arg:latest "noel sees leon"
Usage: ./reverser <string>
```

Next, we will create a Dockerfile for debugging this image.

## Creating a debugging image

## Updated Makefile

For debugging the binary, compile the code with the `-g` and `-O0` flags. Flag `-g` adds debugging information to the binary. Flag `-O0` disables the optimization, which can rearrange, inline, or remove code, in turn, might make debugging difficult.

Make the following adjustments to accept variable `DEBUG` in `Makefile`, which is 0 by default. If you invoke `make` with `DEBUG=1`, it will adjust the `CFLAGS` by adding debugging flag `-g` and disabling optimization with `-O0`. Also, it changes the build directory and binary names.

```
CC = gcc
CFLAGS = -Wall -Wextra
SRC_DIR = src
BUILD_DIR = build
BIN = reverser

DEBUG ?= 0
ifeq ($(DEBUG), 1)
    CFLAGS := $(CFLAGS) -g -O0
    BUILD_DIR := $(BUILD_DIR)-debug
    BIN := $(BIN).debug
else
    CFLAGS := $(CFLAGS) -O3
endif

main: $(BUILD_DIR)/$(BIN)

$(BUILD_DIR)/$(BIN): $(SRC_DIR)/main.c
    @mkdir -p $(BUILD_DIR)
    $(CC) $(CFLAGS) -o $@ $<

clean:
    rm -rf $(BUILD_DIR)

.PHONY: main clean
```

## Dockerfile.debug

Change the `make` command to `make DEBUG=1`, install `gdb` and start `gdbserver`.

**Note:**

We used port 2159 for `gdbserver`, which is the registered TCP port number for "GDB Remote Debug Port". You can use any available port.

Create a file `Dockerfile.debug` with the following content:

```
FROM bellsoft/alpaquita-linux-gcc:14.2-musl AS build

WORKDIR /build
COPY Makefile .
COPY src src
# compile the code with debugging symbols
RUN make DEBUG=1


FROM bellsoft/alpaquita-linux-base:stream-musl
WORKDIR /app
# Install gdb package
RUN apk update && apk add --no-cache gdb
COPY --from=build /build/build-debug/reverser.debug .
COPY --from=build /build/src src
# Start gdbserver on port 2159 to debug the application
ENTRYPOINT ["gdbserver", ":2159","./reverser.debug"]
```

**Tip**

To minimize disk usage, you can remove the binaries and files provided by the `gdb` package and leave only the `gdbserver` binary. Note that `gdbserver` requires the `libstdc++` library.

## Building the image and running the container

Build the debugging image tagged as `reverser-arg-debug`.

To use gdb for tracing, the process group of the tracee must allow ptrace operations. By default, Docker removes the `SYS_PTRACE` capability, which restricts ptrace use inside the container. This capability needs to be re-enabled.

Additionally, Docker’s default seccomp profile blocks several system calls essential for gdb, including `ptrace`, `perf_event_open`, and `process_vm_writev`. Using `--security-opt` seccomp=unconfined will bypass seccomp filtering for all processes in the container.

Then, run the `gdbserver` with the options explained above passing the string, "noel sees leon" as the first argument to the program.

```
$ docker build --tag reverser-arg-debug -f Dockerfile.debug .

$ docker run --cap-add=SYS_PTRACE --security-opt seccomp=unconfined \
    -p2159:2159 reverser-arg-debug:latest "noel sees leon"

Process ./reverser.debug created; pid = 9
Listening on port 2159
```

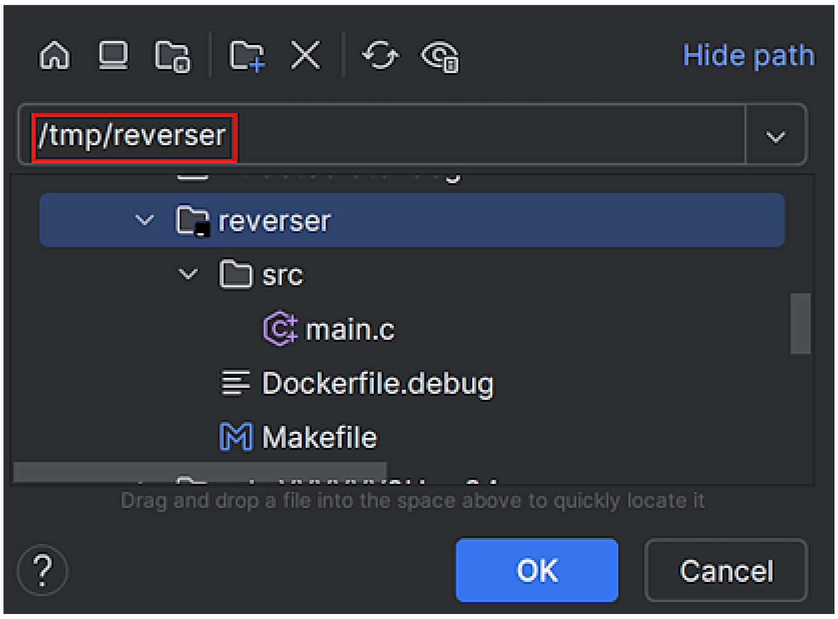
The `gdb` server is now waiting on port 2159. The next section explains how to connect to the `gdb` server using `CLion`.

## Configuring CLion for remote debugging

**Important:**

Before continuing, make sure that `gdbserver` is running inside a docker container.

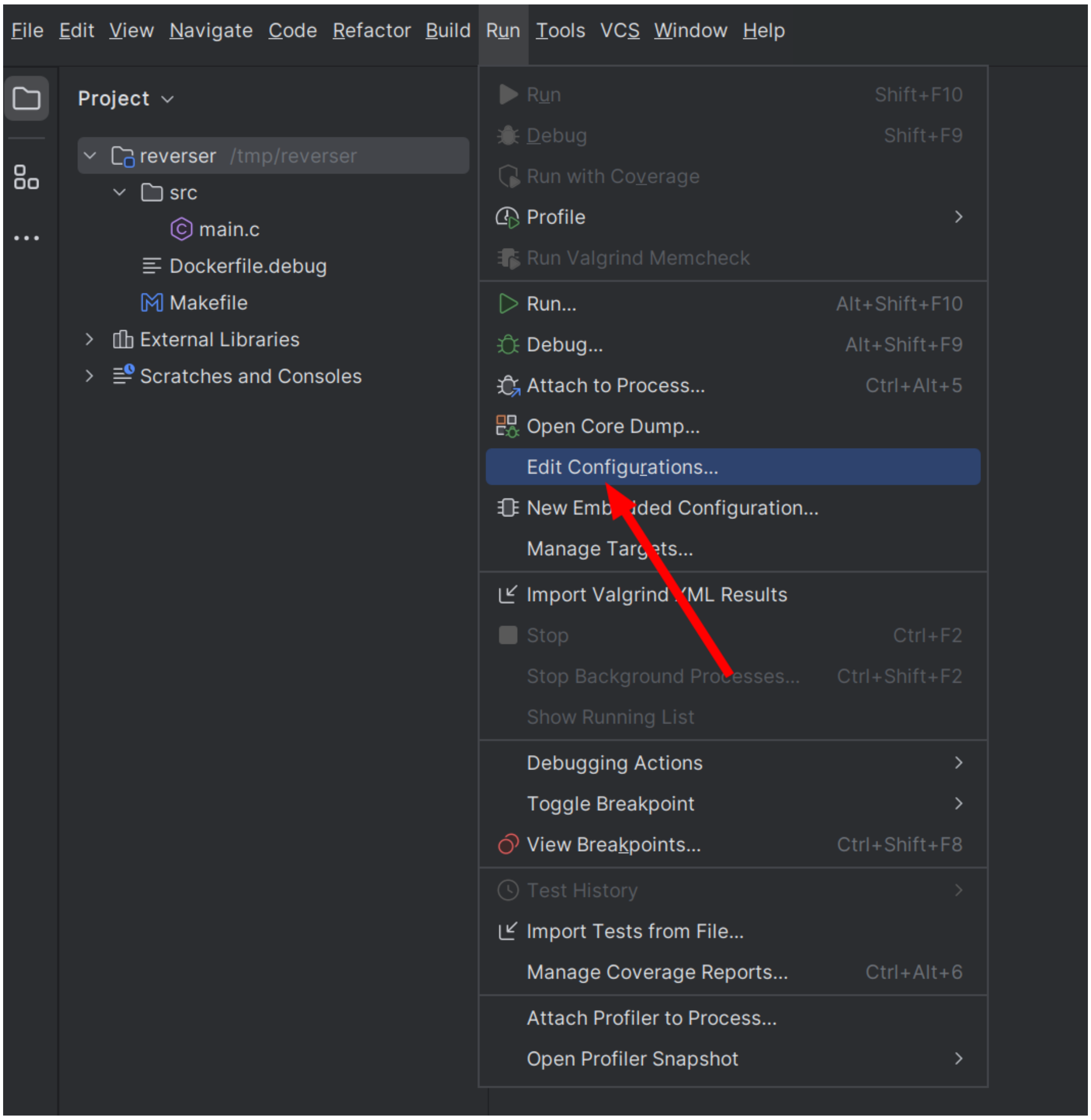
1. First, open the project directory.



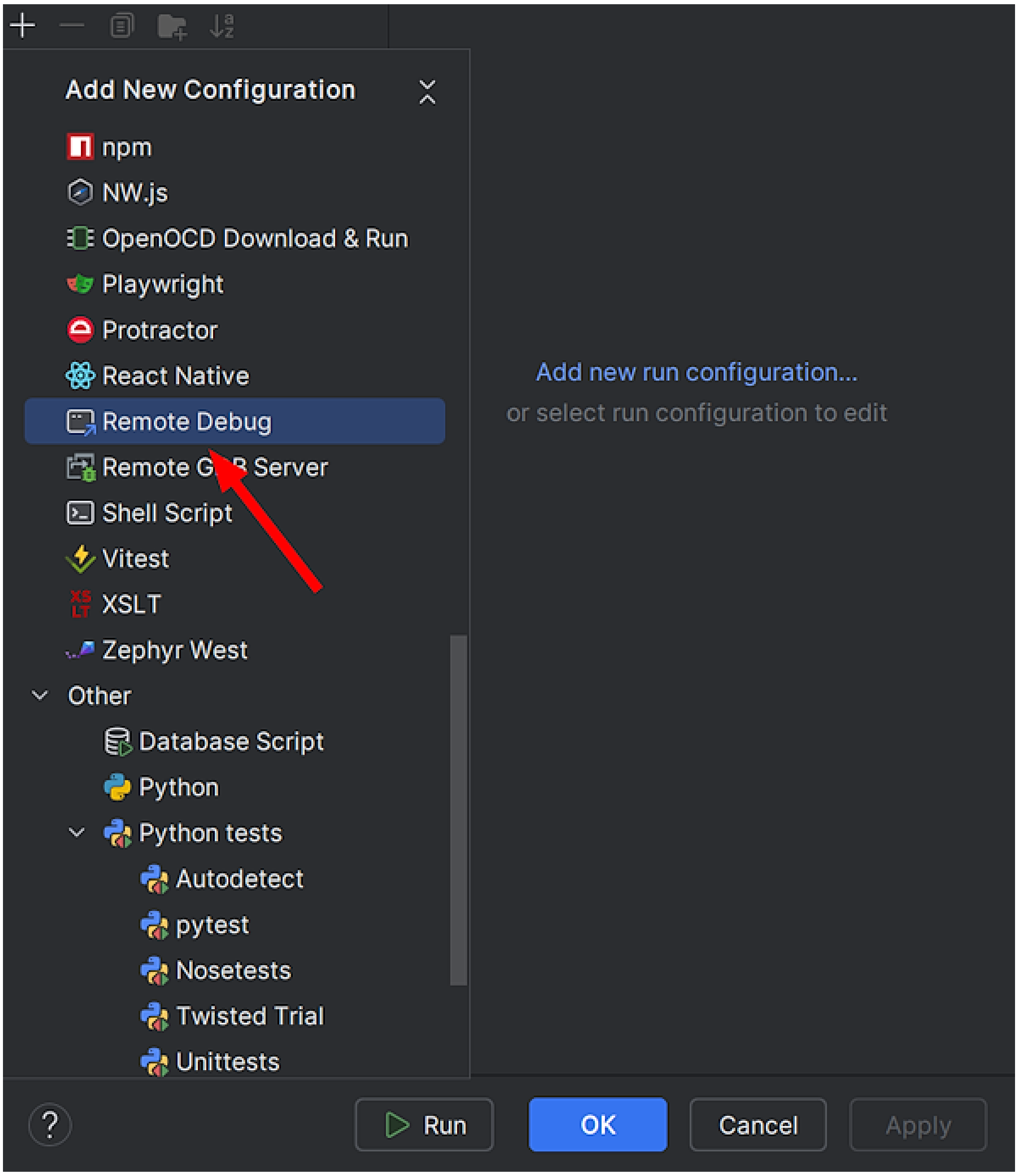
2. Click **Trust Project**.

> **Note:**
>
> Ignore the "No compilation commands found" error in the **Build** sidebar, since we will be building our project inside a docker image anyway.

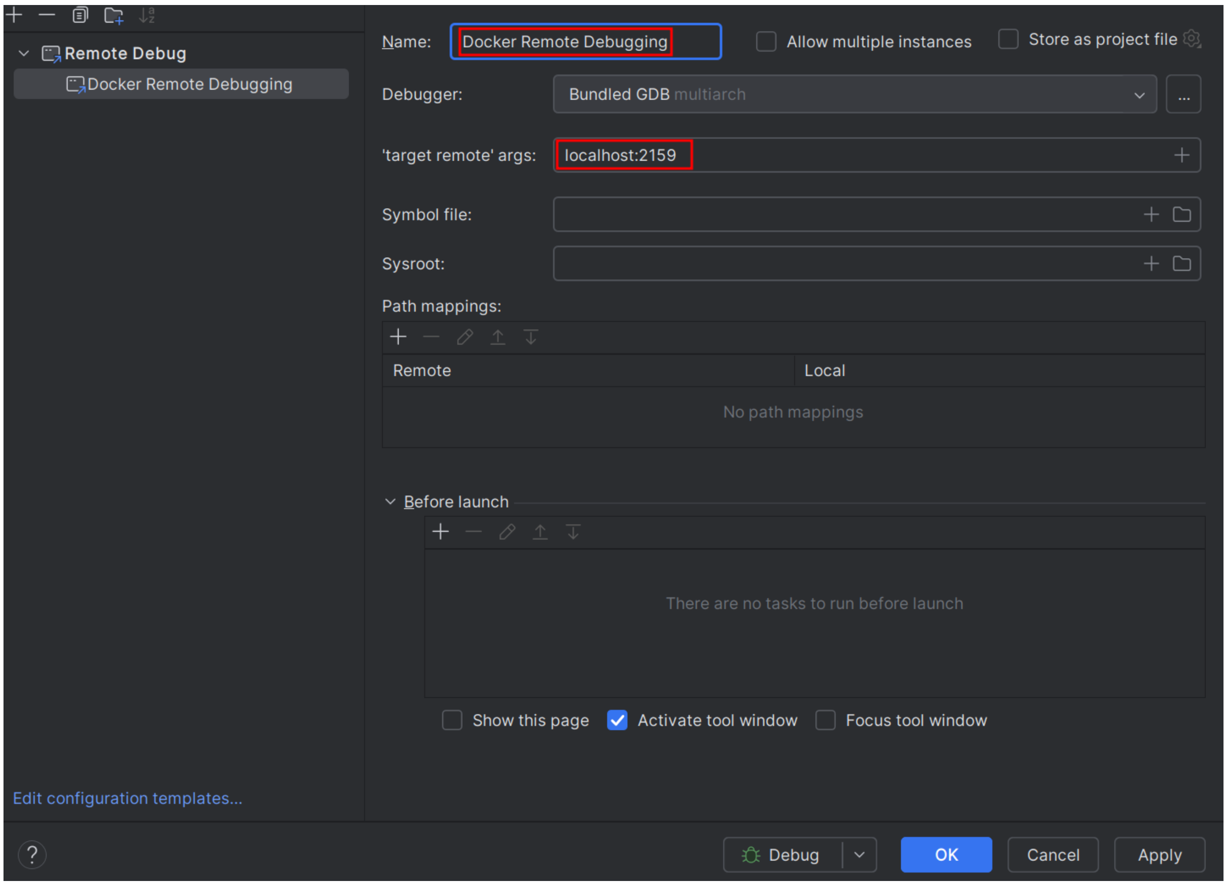
3. On the main menu, select **Run** > **Edit Configurations**.



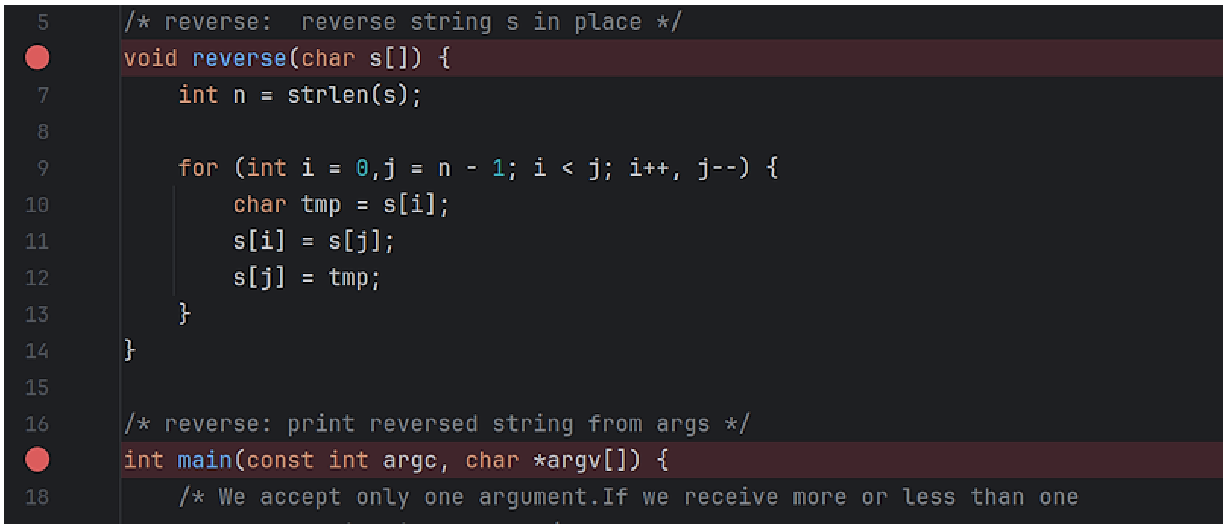
4. Click the **+** icon and then select **Remote Debug**.



5. Fill the **'target remote' args** with IP address and the port of the `gdbserver`, in this case localhost and 2159. You may want to give this configuration a meaningful name like "Docker Remote Debugging". Click **OK**.

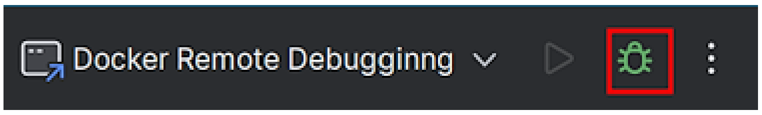


6. On the side menu, open `main.c`. Set breakpoints in the `main` and `reverse` functions as in the following image:

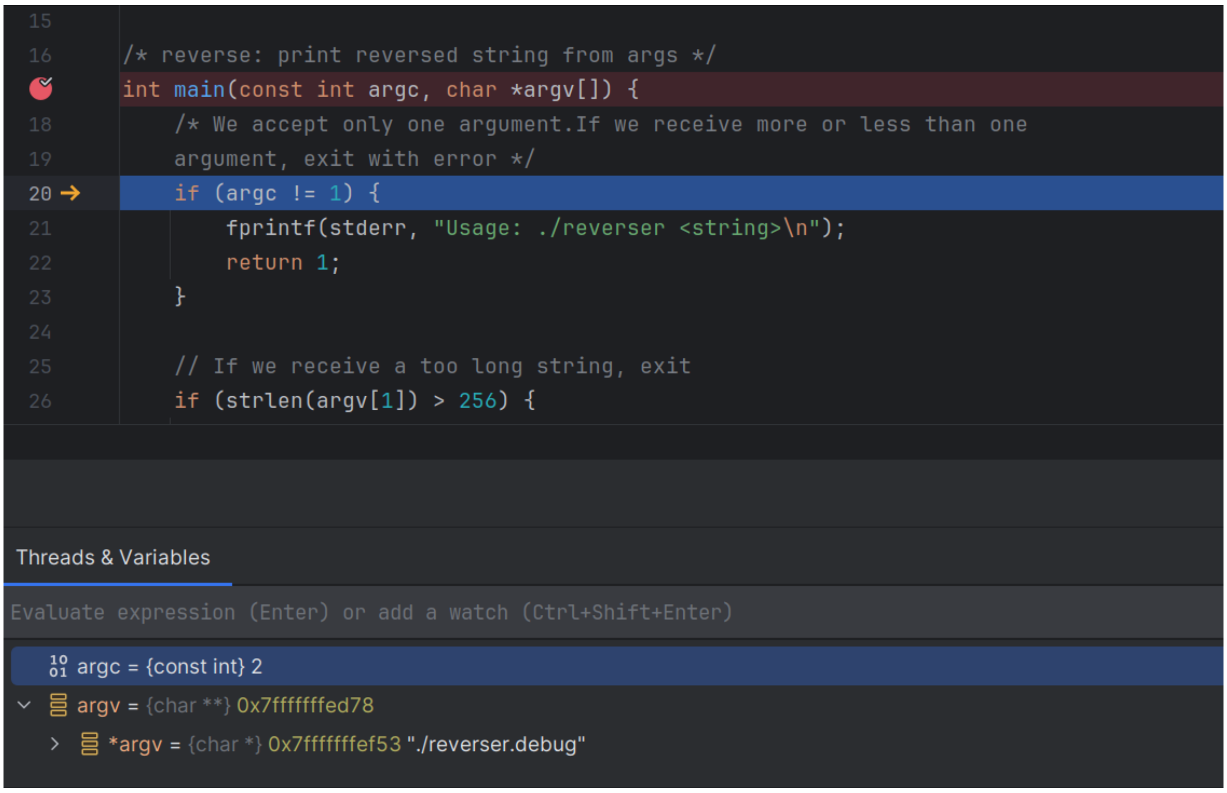


```
/* reverse:  reverse string s in place */
void reverse(char s[]) {
    int n = strlen(s);

    for (int i = 0,j = n - 1; i < j; i++, j--) {
        char tmp = s[i];
        s[i] = s[j];
        s[j] = tmp;
    }
}

/* reverse: print reversed string from args */
int main(const int argc, char *argv[]) {
    /* We accept only one argument.If we receive more or less than one
```

7. On the toolbar (top right), click the green bug icon to connect to `gdbserver` on port 2159, as we configured earlier.



The program stops at line 20 in the `main` function, and you can see an argument count (`argc`) of 2. The value of `argv[0]` is "./reverser.debug".



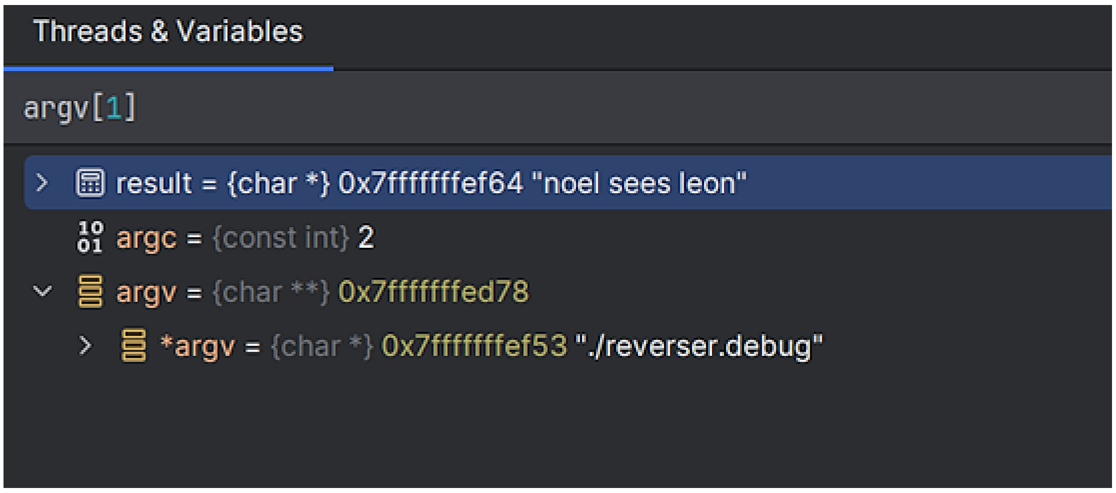
```

/* reverse: print reversed string from args */
int main(const int argc, char *argv[]) {
    /* We accept only one argument.If we receive more or less than one
    argument, exit with error */
    if (argc != 1) {
        fprintf(stderr, "Usage: ./reverser <string>\n");
        return 1;
    }

    // If we receive a too long string, exit
    if (strlen(argv[1]) > 256) {
```

Threads & Variables
Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)

```
argc = {const int} 2
argv = {char **} 0x7fffffffed78
  *argv = {char *} 0x7fffffffef53 "./reverser.debug"
```

We undoubtedly passed the string "noel sees leon" to the program as an argument in the previous steps. Let's look at the value of `argv[1]`.

8. On the debugging toolbar (bottom part of the interface), select **Threads & Variables**. At the top of the window, enter the value you want to evaluate, in this case, `argv[1]`. Press Enter.



Threads & Variables

argv[1]

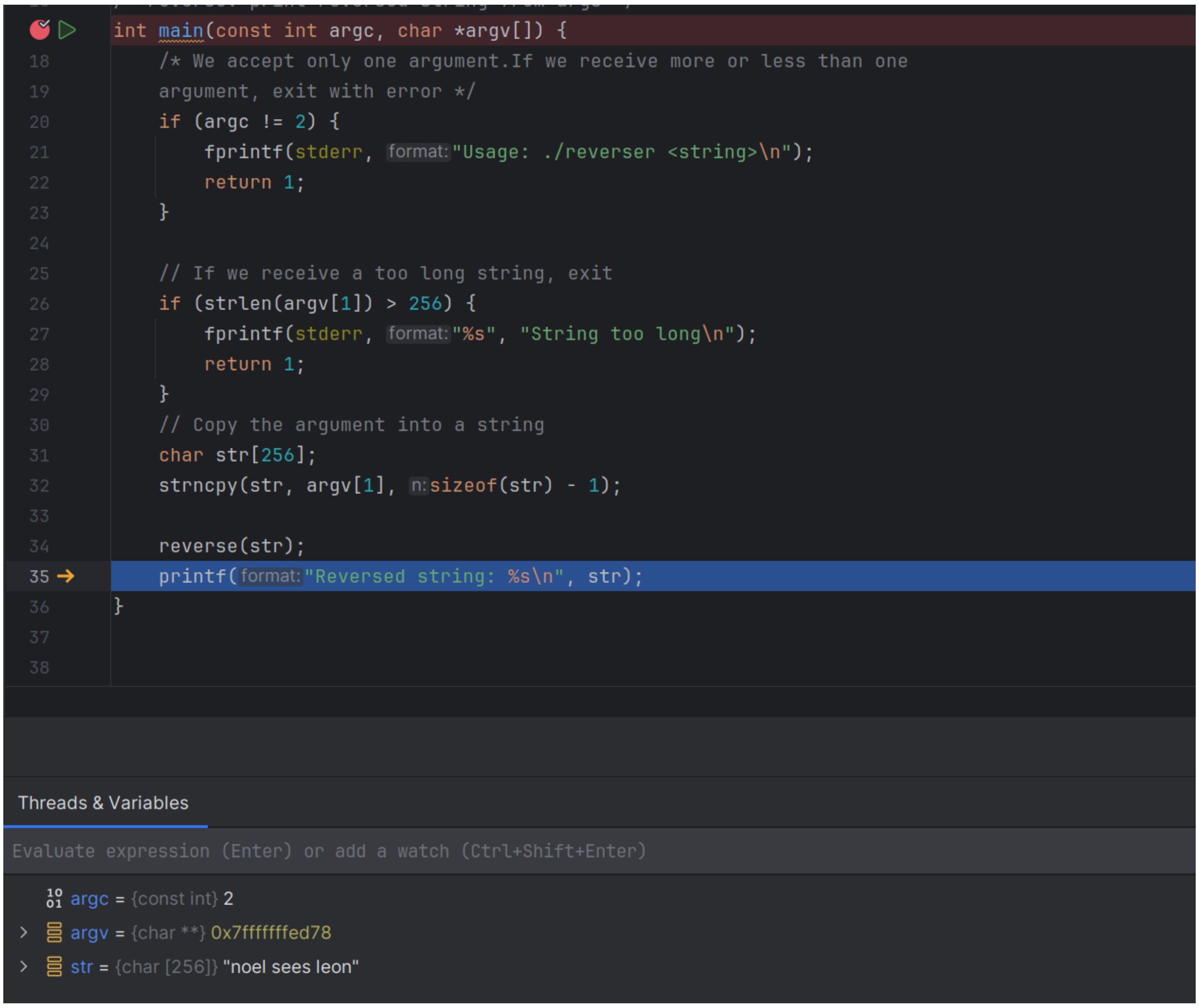
```
> result = {char *} 0x7fffffffef64 "noel sees leon"
  argc = {const int} 2
v argv = {char **} 0x7fffffffed78
  > *argv = {char *} 0x7fffffffef53 "./reverser.debug"
```

The "noel sees leon" string is displayed in the window.

The first argument of the program is always the program itself, "./reverser.debug". That's why the conditional `if (argc != 1)` fails, since we provide an argument to the program, `argc` should be 2, not 1.

9. Stop the debugging session by pressing red square on the toolbar.

Now fix the bug, re-build the image and run the container again. Then click the debug icon on the toolbar.



```
int main(const int argc, char *argv[]) {
    /* We accept only one argument.If we receive more or less than one
    argument, exit with error */
    if (argc != 2) {
        fprintf(stderr, format:"Usage: ./reverser <string>\n");
        return 1;
    }

    // If we receive a too long string, exit
    if (strlen(argv[1]) > 256) {
        fprintf(stderr, format:"%s", "String too long\n");
        return 1;
    }
    // Copy the argument into a string
    char str[256];
    strncpy(str, argv[1], n:sizeof(str) - 1);

    reverse(str);
    printf(format:"Reversed string: %s\n", str);
}
```

Threads & Variables
Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)

```
argc = {const int} 2
argv = {char **} 0x7fffffffed78
str = {char [256]} "noel sees leon"
```

The program now continues without exiting with a usage message. Click **Step over** a couple of times, inspect variables or explore the interface. Click **resume program** a few times until the program completes.

Upon finishing the program, it should display the reversed string and `gdbserver` exits.

```
$ docker run --cap-add=SYS_PTRACE --security-opt seccomp=unconfined \
    -p2159:2159reverser-arg-debug:latest "noel sees leon"

Process ./reverser.debug created; pid = 9
Listening on port 2159
Remote debugging from host ::ffff:172.17.0.1, port 39550
Reversed string: noel sees leon
```

**Note:**

See CLion's documentation for more information about debugging in CLion.

Alpaquita Linux:
Debugging apps running
in Docker with JetBrains
and VSCode

Java, C, and Python

bell/soft