Available online www.ejaet.com

European Journal of Advances in Engineering and Technology, 2024, 11(4):172-179

Research Article ISSN: 2394 - 658X

# Debugging a Go application inside a Docker Compose container

**Pallavi Priya Patharlagadda**

Pallavipriya527.p@gmail.com
United States of America

**ABSTRACT**
In any programming language, the most basic method of debugging is to use print statements or logs and write to standard out. This is undoubtedly effective. however, it gets quite challenging as our program gets bigger and the reasoning gets more intricate. It is difficult to add print statements to each code route in the application. Here's where debuggers are useful. With breakpoints and a variety of other capabilities, debuggers assist us in tracking the program's execution path. One such debugger for Go is called Delve. We will learn how to use Delve to debug Go applications in this article.

**Key words:** programming language, debugging

## INTRODUCTION

Golang has an inbuilt GCB debugger but DLV is more advanced and supports a lot of features. DLV is a command-line tool that helps with Go program debugging. It provides several tools made especially for Go developers, simplifying the process of debugging, understanding, and resolving problems with their code. You can step through your Go code, set breakpoints, examine variables, and do a lot more with DLV. With Go's DLV, developers can quickly find and address defects in their Go apps with a user-friendly and robust debugging experience.

## PROBLEM STATEMENT

In today’s software development landscape, microservices have become a popular architectural approach for building scalable and maintainable applications. Microservices is a software architectural style that structures an application as a collection of small, loosely coupled services. Each service is responsible for a specific business capability and can be independently developed, deployed, and scaled. Go is an excellent choice for developing microservices. These microservices are usually deployed as Docker containers. Docker-compose is used in multi-container Deployments. Debugging an application running inside a Docker compose needs some configuration. This paper discusses how we can connect from the host IDE (Integrated Development Environment) to an application running in the docker-compose container.

## INTRODUCTION TO DEBUGGING

Problems in an application are prevalent. The earliest method of resolving any problem is to do so during the planning stage before any code is produced. However, if we pass the planning stage and there are some bugs in the code, then it would be preferable to address the bugs in the code at the development stage itself. The sooner we fix the bugs, the lesser the fixing cost. The debugger comes in very handy here. Debugger is a ubiquitous tool in many programming languages. The foundation of a debugger is breakpoints. To monitor the application, one can set up checkpoints. As you selected, the application will terminate _before_ line.

One can then execute their test or program till it quits. The editor displays the stack trace, which includes all functions called before the breakpoint is reached. If you are exploring a new repository and would like to know whether a specific code executes and what additional instructions are involved, this trace is quite helpful. The debugger simply prints the entire trail. Inspecting the contents of variables that are accessible within the scope of a breakpoint is another option. We can either keep going or use stepping into the methods to do more debugging. These might aid in the understanding of variable values, code flow, etc.

## ADVANTAGES OF DEBUGGER

**1. Code changes are not required.**

One can view program behavior without adding debug code (fmt.Print and so on).

**2. Easy method for investigating any type of variable**

Moreover, the debugger variable's view is far more intuitive to use than any output that you may write to list variables. For instance, we can inspect any complex structure or list by selecting the "View" link. When we see a byte array that was formed from a string, we may view it as text. The debugger shows any variable elegantly.

Debugging a standalone application is very easy as IDE (Integrated Development Environment) would take care of Compiling the application, starting the application, and connecting to it. Debugging an application running inside a docker container requires some configuration.

## INTRODUCTION TO DELVE

Delve is a debugger for the Go programming language. Delve is a simple and powerful tool and is easy to invoke and use. For Go developers, Golang DLV Debugger offers a reliable and feature-rich debugging environment. With its rich feature set, which includes variable inspection, breakpoints, step-by-step execution, and more, you can efficiently debug and fix issues in your Go applications.

Delve works well with most IDEs, including Visual Studio Code, and enable remote debugging. You can improve your debugging abilities by integrating DLV into your development process. DLV supports various subcommands which can be used for different purposes. Let's dive more into this.

1. **DLV Debug:**

   Compiles your program in the current directory or the specified directory with optimizations disabled and starts debugging.

   *dlv debug [package] [flags]*

2. **DLV Attach:**

   This command will cause Delve to attach to an already running process and begin a new debug session. When exiting the debug session, one will have the option to let the process continue or kill it.

   *dlv attach pid [executable] [flags]*

3. **DLV Connect:**

   Connect to a running headless debug server with a terminal client.

   *dlv connect addr [flags]*

4. **DLV Core:**

   Examine a core dump (only supports Linux and Windows core dumps). The core command will open the specified core file and the associated executable and let you examine the state of the process when the core dump was taken.

   *dlv core <executable> <core> [flags]*

5. **DLV Dap:**

   Starts a headless TCP server communicating via Debug Adaptor Protocol (DAP). The server is always headless and needs to be connected to using a DAP client, such as VS Code, to request the launch of a binary or the attachment of a process. The client's launch configuration allows the following modes to be specified:

   Launch+exec (runs a precompiled binary; similar to 'dlv exec')
   Launch + debug (builds and releases, such as 'dlv debug')
   Launch + test (also known as "dlv test") (builds and tests)
   Launch + replay (a.k.a. 'dlv replay') replays a rr trace.
   Launch + core (plays back a core dump file, such as 'dlv core')
   Attach + local (links to an active process; similar to 'dlv attach')

   The working directory of DLV will be used to parse binary paths for the program and output. Multiple client connections (--accept-multiclient) are not accepted by this server. Alternatively, use a DAP client with attach + remote config and 'dlv [command] --headless'. The stopOnEntry launch/attach attribute can be used to control whether execution is continued at the beginning of the debug session, even while --continue is not available.

   A unique flag known as --client-addr causes the server to dial in to the host:port, where a DAP client is waiting, to start a debug session. After the debug session is over, this server process will terminate.

   *dlv dap [flags]*

6. **DLV Exec:**

   With this command, Delve will launch a precompiled binary and connect to it right away to start a fresh debugging session. Please be aware that debugging the binary may be challenging if optimizations are not deactivated during compilation. If you are using Go 1.10 or above, please think about building debugging binaries with -gcflags="all=-N -l" or -gcflags="-N -l" on previous Go versions.

*dlv exec <path/to/binary> [flags]*

7. **DLV Replay:**

   The replay command will replay an rr trace by opening a trace generated by Mozilla rr. Mozilla rr must be installed: https://github.com/mozilla/rr

   *dlv replay [trace directory] [flags]*

8. **DLV Test:**

   This command commences a new debug session and compiles a test binary without optimizations. Within the context of your unit tests, you can start a new debug session with the test command. Delve will by default debug the current directory's tests. As an alternative, you can give Delve the name of a package and it will debug the tests inside of it. Double-dashes -- are a way to give the test program arguments

   *dlv test [pkg] -- -test.run Test.v -testSomething -other-argument*

9. **DLV Trace:**

   Program execution can be tracked using the trace subcommand. Every function that matches the given regular expression will have a tracepoint set on it, and when the tracepoint is reached, information will be printed. This is helpful if you just want to know what operations your process is carrying out without starting a full debug session.

   If you simply want to see the output of the trace operations, you can redirect to stdout from the trace subcommand's output, which is reported to stderr.

   *dlv trace [package] regexp [flags]*

10. **DLV version:**

    Prints the dlv command version.

    *dlv version [flags]*

11. **DLV log:**

    Help with logging flags. The --log flag can be used to enable logging, and the --log-output flag can be used to choose which components log information.

    The component names chosen from the below list must be entered as the argument for the --log-output command, separated by commas:

    **Debugger:** Record debugger instructions
    **Gdbwire:** Record the link to the gdbserial backend
    **lldbout:** Transfer debugserver/lldb output to standard output.
    **Debuglineerr**: Log resolvable mistakes examining the.debug_line
    **Rpc**: Record every RPC message.
    **dap:** Record every DAP message.
    **fncall:** Record the function call protocol
    **minidump:** Record loading minidump
    **stack:** Record the stacktracer

    It is also possible to specify the log location by using --log-dest. The parameter will be treated as a file path if it is not a number, and as a file descriptor otherwise. In headless and dap modes, this option will also reroute the "server listening at" message.

12. **DLV Backend:**

    The backend to be utilized is specified by the --backend flag. The possible choices are:

| | |
|---|---|
| Default: | uses native everywhere else and lldb on macOS. |
| Native: | Native backend. |
| Lldb: | Uses lldb-server or debugserver. |
| Rr: | Uses mozilla rr (https://github.com/mozilla/rr). |

Environment variables can be used to configure certain backends:
DELVE_DEBUGSERVER_PATH gives the location of the lldb backend's debug server executable.
The additional flags used when invoking 'rr record' are specified by DELVE_RR_RECORD_FLAGS.
Additional flags used when invoking 'rr replay' are specified by DELVE_RR_REPLAY_FLAGS.

In addition to the above subcommands, below are some of the common options.

--accept-multiclient:

Allows a headless server to accept multiple client connections via JSON-RPC or DAP.

--allow-non-terminal-interactive:

Allows interactive sessions of Delve that don't have a terminal as stdin, stdout, and stderr

--api-version int:

Selects JSON-RPC API version when headless. New clients should use v2. Can be reset via RPCServer.SetApiVersion. See Documentation/api/json-rpc/README.md. (default 1)

--backend string:

Backend selection (see 'dlv help backend'). (default "default")

```
  --build-flags string:
            Build flags, to be passed to the compiler. For example: --build-flags="-tags=integration -
   mod=vendor -cover -v"
 --check-go-version:
           Exits if the version of Go in use is not compatible (too old or too new) with the version of Delve.
   (default true)
 --disable-aslr:
           Disables address space randomization.
 --headless:
           Run debug server only, in headless mode. The server will accept both JSON-RPC and DAP client
   connections.
 --init string:
           Init file, executed by the terminal client.
  -l, --listen string:
           Debugging server listens to address. Prefix with 'unix:' to use a Unix domain socket. (default
   "127.0.0.1:0")
  --log:
           Enable debugging server logging.
  --log-dest string:
           Writes logs to the specified file or file descriptor (see 'dlv help log').
  --log-output string:
           Comma-separated list of components that should produce debug output (see 'dlv help log')
  --only-same-user:
           Only connections from the same user that started this instance of Delve are allowed to connect.
   (default true)
  -r, --redirect stringArray:
           Specifies redirect rules for target process (see 'dlv help redirect')
   --wd string:
      Working directory for running the program.
```

## DEPLOYING A GO APPLICATION INSIDE A CONTAINER

Below is a simple Go application that starts an HTTP server on port 80.

```
Package main
import (
      "fmt"
"net/http"
)
func Example(w http.ResponseWriter, req *http.Request) {
      welcome_string := PrintTrace()
      fmt.Println(welcome_string)
      fmt.Fprintf(w, "Example\n")
}
func PrintTrace() string {
      return "Example Handler is invoked"
}
func main() {
      http.HandleFunc("/example", Example)
      fmt.Println("Server up and listening…")
      http.ListenAndServe(":80", nil)
}
```

We can run the above example file using

```
go build -o Example
./Example
```

**Output:** Server up and listening...

On another window, try to curl this http handler using cmd:

```
curl http://localhost:80/example
```

**Output:** Example

Now, let's try to run this in a Docker Container. The Dockerfile would look as below.

```
FROM golang:1.22
WORKDIR /app
EXPOSE 80

COPY go.mod go.sum src/Example.go ./
RUN go build -o example .
CMD [ "./example" ]
```

Below is the explanation of each instruction in the Dockerfile.

- *golang:1.22* is the base image.
- WORKDIR / defines the working directory, from which any relative paths referred to subsequently are sourced.
- The COPY directive copies files from the host to the image that is being created.
- RUN go build -o example. compiles our program, producing the executable file example.
- Ultimately, when the container is started, CMD gives Docker a run command for our application.

Build the Docker Image using the below file.

```
docker build. --tag example-image
```

Start the container from the example image using the below command.

```
docker run --name example-app example-image
```

To check if the container is running successfully, execute the below curl command. This should produce an output example.

```
curl http://localhost/example.
```

**Output***:* Example

Majority of the applications contain multiple containers. For example, we will have a Frontend container, Business logic Application, Backend DB container, etc. In such cases, we use Docker-compose or Kubernetes to facilitate multi-container deployments. Let's see how to launch this application using Docker Compose.

## DEPLOY THE GO APPLICATION INSIDE A DOCKER COMPOSE CONTAINER

For simplicity, I am just deploying a single container inside a Docker compose. If required, we can add multiple containers to it.

```
services:
  app:
    build:
      context: ..
      dockerfile: deploy/Dockerfile
    ports:
      - "80:80"
```

To deploy the Docker container using docker-compose, use the below command.

```
docker-compose -f docker-compose.yml up --build
```

This would bring up the container by building the image from the provided Dockerfile.

## CONNECT FROM LOCAL EDITOR TO CONTAINER APPLICATION:

To connect to a debugger running on a remote application, please follow the below steps.

1. Get the delve inside the container. Update the existing Dockerfile by adding a command to install delv.

```
RUN go install github.com/go-delve/delve/cmd/dlv@latest
```

2. Run Delve in headless mode via dlv debug --headless and then connect to it from another terminal or IDE. This will place the process in the foreground and allow it to access the terminal TTY. Below is the updated Dockerfile

```
FROM golang:1.22
WORKDIR /app

COPY go.mod go.sum src/Example.go ./

RUN go build -gcflags "all=-N -l"
 -o example .

# Install Delve for debugging
RUN go install github.com/go-delve/delve/cmd/dlv@latest
```

```
CMD [ "dlv", "--listen=:4000", "--headless=true", "--log=true", "--accept-multiclient", "--api-version=2", "exec", "/app/example" ]
```

1. In the docker-compose file, we need to expose the port 4000 on which the debugger is running. This is needed to connect from cli client to the application running inside the Docker container. Below is the updated docker-compose file.

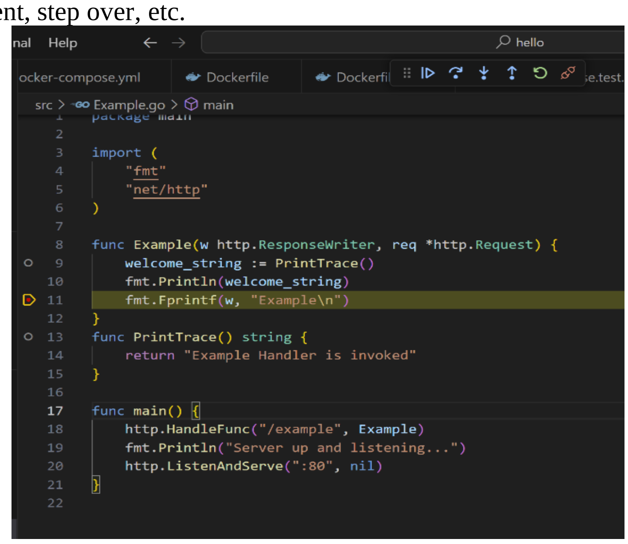
```
services:
  app:
    build:
      context: ..
      dockerfile: deploy/Dockerfile
    ports:
      - "80:80"
          - "4000:4000"
```

2. We can start the Docker containers using the command
*docker-compose -f docker-compose.yml up –build*
3. Once the server is running, we need to connect to the server either through CLI or through the IDE. In the paper, I would like to use Vscode to connect to the debug server. Below is the configuration required for launch.json.

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "Connect to server",
            "type": "go",
            "request": "attach",
            "mode": "remote",
            "port": 4000,
            "host": "127.0.0.1"
        }
    ]
}
```

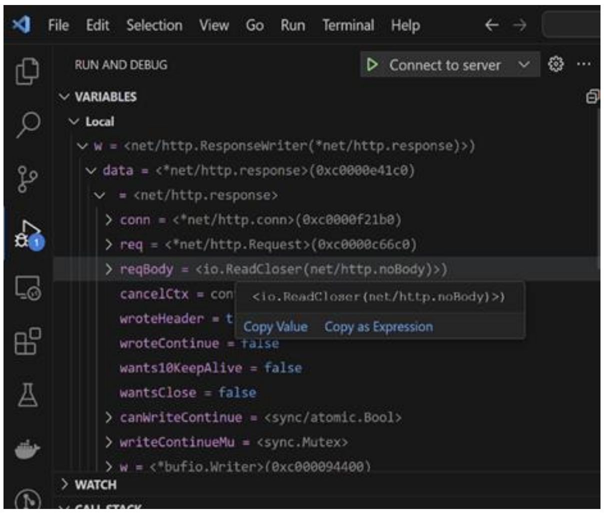
4. Once the configuration is in place, set the breakpoint at the point where you want to observe the code.
5. From the vscode, click on Run and Debug and Run connect to the server. This would connect to the debugger.
6. From another terminal run the curl command such that the example handler gets invoked.
*curl http://localhost:80/example*

We can see the breakpoint gets hit. Users can decide what to do next. Either to step into further functions, inspect the variable's content, step over, etc.



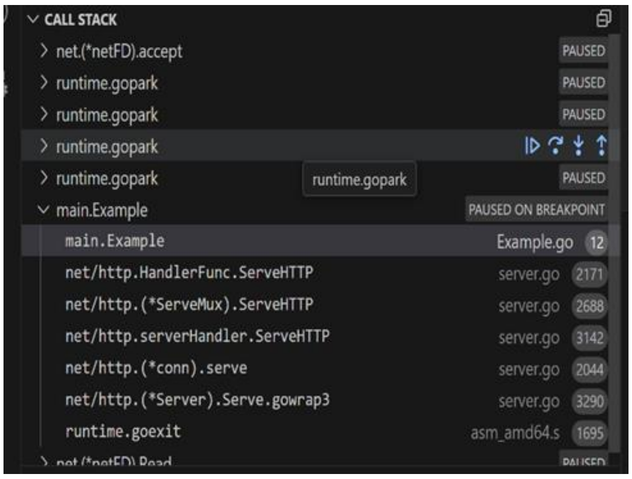
```go
package main

import (
    "fmt"
    "net/http"
)

func Example(w http.ResponseWriter, req *http.Request) {
    welcome_string := PrintTrace()
    fmt.Println(welcome_string)
    fmt.Fprintf(w, "Example\n")
}
func PrintTrace() string {
    return "Example Handler is invoked"
}

func main() {
    http.HandleFunc("/example", Example)
    fmt.Println("Server up and listening...")
    http.ListenAndServe(":80", nil)
}
```

On the left side, we can see the variables and their data as below.



```
w = <net/http.ResponseWriter(*net/http.response)>)
  data = <*net/http.response>(0xc0000e41c0)
    = <net/http.response>
      conn = <*net/http.conn>(0xc0000f21b0)
      req = <*net/http.Request>(0xc0000c66c0)
      reqBody = <io.ReadCloser(net/http.noBody)>)
      cancelCtx = con    <io.ReadCloser(net/http.noBody)>)
      wroteHeader = t
      wroteContinue = false
      wants10KeepAlive = false
      wantsClose = false
      canWriteContinue = <sync/atomic.Bool>
      writeContinueMu = <sync.Mutex>
      w = <*bufio.Writer>(0xc000094400)
```

The call trace can also be observed as below. This shows the function calls that were invoked to reach to this point.



```
net.(*netFD).accept                              PAUSED
runtime.gopark                                   PAUSED
runtime.gopark                                   PAUSED
runtime.gopark
runtime.gopark              runtime.gopark       PAUSED
main.Example                  PAUSED ON BREAKPOINT
  main.Example                           Example.go 12
  net/http.HandlerFunc.ServeHTTP         server.go 2171
  net/http.(*ServeMux).ServeHTTP         server.go 2688
  net/http.serverHandler.ServeHTTP       server.go 3142
  net/http.(*conn).serve                 server.go 2044
  net/http.(*Server).Serve.gowrap3       server.go 3290
  runtime.goexit                        asm_amd64.s 1695
```

## DISADVANTAGES OF DELVE

Debuggers alter a program's behavior.

Since GO is renowned for its concurrency patterns, your program likely contains a large number of goroutines. Code that relates to one or more of the goroutines has breakpoints set; nevertheless, the debugger is responsible for what goes on in the other goroutines. All routines are typically terminated. However, there may come a time when you wish to get debug information without having the program stop. If so, you will still require this debug information even without a debugger.

## CONCLUSION

The Debugger plays a crucial role in understanding the runtime errors. Go Delve is simple and easy to use. We can run the dlv on the CLI and it is also supported by various IDEs (Integrated Development Environment). If the application is running in a certain environment where we cannot add any logs, then dlv will be helpful to debug the errors.

## REFERENCES

[1]. https://golangbot.com/debugging-go-delve/
[2]. https://i-am-shubha.medium.com/mastering-go-debugging-unveiling-the-power-of-the-dlv-debugger-4409a3c70898
[3]. https://golangforall.com/en/post/goland-debugger.html
[4]. https://github.com/go-delve/delve/blob/master/Documentation/usage/dlv_core.md
[5]. https://github.com/go-delve/delve/blob/master/Documentation/usage/dlv_exec.md
[6]. https://github.com/go-delve/delve/blob/master/Documentation/usage/dlv_debug.md
[7]. https://github.com/go-delve/delve/blob/master/Documentation/usage/dlv_dap.md

[8]. https://github.com/go-delve/delve/blob/master/Documentation/usage/dlv_replay.md
[9]. https://github.com/go-delve/delve/blob/master/Documentation/usage/dlv_test.md
[10]. https://github.com/go-delve/delve/blob/master/Documentation/usage/dlv_log.md
[11]. https://github.com/go-delve/delve/blob/master/Documentation/usage/dlv_backend.md
[12]. https://github.com/go-delve/delve/blob/master/Documentation/usage/dlv_version.md
[13]. https://github.com/go-delve/delve/blob/master/Documentation/usage/dlv_trace.md
[14]. https://github.com/go-delve/delve/blob/master/Documentation/usage/dlv_connect.md
[15]. https://github.com/go-delve/delve/blob/master/Documentation/usage/dlv_attach.md