Profiling

- Process stacktraces
- Memory profiling
  - pprof tool
- Performance profiling
- Tracing

Forgejo is written in Go, which means Forgejo can leverage the extensive profiling tooling that is provided by Go's runtime. Profiling in Forgejo can be done in production; it is safe and with low overhead; no settings need to be enabled for that. A good knowledge of Forgejo internals helps with drawing conclusions from profiling. It is easy to draw the wrong conclusions, be overwhelmed by the data, or try to find a needle in a haystack. In the case you are reading this because you are suspecting or experiencing performance problems with Forgejo, it is recommended to make an issue or contact the Forgejo developers; this can help in narrowing down where to look or with interpreting profiling results.

The following sections go into more details about the different types of profiling that can be done in Forgejo.

Process stacktraces

This is by far the simplest way of profiling, even if it can be better described as diagnostics. Most goroutines[^0], whether short-lived or long-lived, are tracked by Forgejo as processes, which means that Forgejo knows they exist, have a description, and can cancel them at any time.

[^0]: This can be conceptually seen as lightweight thread that are managed by the Go runtime.

In the admin settings navigate to the "Stacktrace" item in the "Monitoring" menu; this is also the page where the other profiling will take place. The two buttons on above the segment determine what type of processes you want to see, "Running processes" show all non-system processes and "Stacktrace" shows all processes. It is good to note that the information of the processes are only a snapshot of what the process was doing at the time of you loading this page, it is not continuously being updated; they could no longer exist or kept running and no longer be at this particular point in the codepath.

Each process in the list has a few key details:

- Title: this is the description that was given to the process, this should identify why it was created and what the purpose is.
- Date: this indicates when the process was created.

And a stacktrace, it is possible that a process has more than one stacktrace, in which case the first stacktrace can be seen as the *main* stacktrace of that process and the others are children of that process that were not marked as new processes. The stacktrace is read from top to bottom, where each entry in the stacktrace represents a stack frame[^1]. The top entry is where the deepest stack frame and the bottom entry is the outer stack frame and where this process started its lifecycle.

[^1]: This can be conceptually seen as where the program made a function call.

Interpreting the meaning of a stackframe requires a bit of Go's runtime knowledge. In the case that a process is waiting or polling on a resource (this would often be the case because short lived process where this does not happen will likely not have any meaningful wall time and are finished quickly) would be represented by the deepeste entry being a `runtime.xxx` function, this simply tells what it currently is doing but not how it got to that point, for that you need to search for the outer entry that starts with `forgejo.org/`.

This type of diagnosis can be helpful if there are many requests that are not being processed quickly enough. By searching for the corresponding URL path on the "Running processes" page, you can find the process that is responsible for processing this particular HTTP request. However, this is only possible with a bit of luck if the monitoring page is loaded in the time period when the actual problem is blocking the process, otherwise the stacktrace would be meaningless to determine the problem. Git processes are shown to be blocked on a syscall function, which is entirely correct and expected as the syscall invokes the Git binary and logically Forgejo cannot show what Git is doing and provide a stacktrace of it, some Git processes (notably `cat-file --batch` and `cat-file --batch-check`) are *long-lived* Git processes, which are kept open (on-demand) in case another call has to be made to one of these two processes within the processing of the same HTTP request and thus the blocking is correct in the case of a syscall and it should rather be checked why the associated HTTP request takes a long time as they are closed after the handling HTTP request is finished.

Memory profiling

Before starting a memory profiling it is good to know what the **actual** memory usage of Forgejo is. Using `top` and similar tools might give a unrealistic picture, because of how unused memory by applications is represented by Linux and other operating systems. Go's runtime keep track of a lot of memory statistics which can give a precise and up to date picture[^3] of the memory usage of Forgejo.

[^3]: These memory statistics are not collected during garbage collection, but are instead calculated or fetched on demand, see documentation of runtime.ReadMemStats.

These numbers are displayed on the Admin Dashboard in the “System Status” section. Simply navigate to “Admin Configuration”; the dashboard is the default page. Note that this section is updated live and therefore the numbers will change as you view the page. The **Current memory usage** is the most important value, this is the heap memory currently used by Forgejo, although it is somewhat skewed as it only shrinks when a garbage collection cycle occurs, but this represents the memory used which is lower than the memory obtained/allocated for Forgejo as per the OS, that value is an implementation detail of Go's runtime and cannot be influenced by Forgejo.

There are other values that you should pay attention to. If a lot of memory has been allocated in a short period of time, the **Total memory allocated** value will rise steeply. Another value to watch out for is related to Go's garbage collection. Although it is fast, the application must be stopped[^4], which means that the entire execution is paused and only Go's garbage collector is running. The total time of the so-called stop-the-world pause is displayed in **Total GC pause**, this is heavily influenced by the number of garbage collections since startup, which is the **GC times** value, this is best kept low and the time between garbage collections as high as possible, a lot of garbage collections usually indicates that a lot of memory is allocated and also released immediately, then the memory allocation itself can become a CPU time bottleneck.

[^4]: For exact details of Go's garbage collector stop-the-world pause we refer to the official documentation.

You can download the heap profile from Forgejo on the stacktrace page in the admin configuration page. It is recommended to set the seconds to zero as this is not relevant for memory profiling. The output is a file named `forgejo-diagnosis-xxx.zip`, in this zip file there is a `heap.dat` file which is relevant for memory profiling. This file does not contain the actual heap, but only information that is relevant for analyzing memory usage. Forgejo enforces garbage collection before taking a snapshot of the heap, this prevents the heap from containing memory that is no longer used.

pprof tool

Go provides tooling to analyze the `heap.dat` file, make sure to have this file extracted and then execute `go tool pprof heap.dat` in a terminal in the directory that contains the `heap.dat` file.

Another complete example of how to make use of this tooling is also provided by a Go article, but to also re-iterate the most important ways to analyze:

- `topN`: Replace `N` with any number and it will display the top `N` functions that have allocated the most memory. It is important to note that this sorts by the `flat%` and not the `cum%` value, `flat%` is just the amount of memory the function itself has allocated, if the function has called another function its memory usage would be included in the `cum%` value.

- `web`: Opens up a visual goldmine to show which function uses how much memory and which functions called those functions. A good way to learn how a function was called. This does not show individual calls to function, for that the `traces <regex>` (works similar to the `list` option) can be used.
- `list <regex>`: Replace `<regex>` with a function name or a file name (case sensitive) and it will show exactly which line of source code was responsible for which amount of memory. Note that this requires the source code of Forgejo on a specific path (mainly the path where the Forgejo binary was built), so this option is usually only possible if Forgejo was built from source.

To take an example of how to effectively memory profile, in Forgejo a lot of regexps are compiled on startup and that means that they will show up in the heap, but e.g. via `topN` it will only show that `regexp.makeOnePass` and `regexp/syntax.(*compiler).inst` allocated memory and it does not show which regexps are actually taking the most memory (and how much). First we can consult the visualization of traces with `web`, we find `regexp.Compile` (this is where a bit of Go knowledges comes in handy to know what the main entry to compiling regexp is which would ultimately called by Forgejo) and in this case it's possible that only two traces are shown, one from the inserting routes in `go-chi` package and one `data.init` via `MustCompile`. For simplicity we will explore the `go-chi` trace. Executing `traces InsertRoute` shows indeed some traces to `regexp.Compile` and they actually come from the `addChild` function. You either can search this function on Github or use `list addChild` and see the following:

```
.          .    253:           if segTyp == ntRegexp {
.        1MB    254:                   rex, err := regexp.Compile(segRexpat)
.          .    255:                   if err != nil {
.          .    256:                           panic(fmt.Sprintf("chi: invalid regexp pattern
'%s' in route param", segRexpat))
.          .    257:                   }
.          .    258:                   child.prefix = segRexpat
.          .    259:                   child.rex = rex
.          .    260:           }
```

There's indeed a call to regexp compile when the route path contains regexp (that's important information to understand *why* it was allocated in the first place and what the precondition for it is). To find out which routes these are we go back to the output of `traces addChild` and follow the trace back to code in Forgejo. One trace comes from Git HTTP route and another from the API. Now to actually find the exact line in the API and Git HTTP file you can execute `granularity=lines` and then execute `traces addChild` again. This leads to finding `routers/web/githttp.go:39` and `routers/api/v1/api.go:1262`. According to the Go's memory profile the regexp created for these two routes are combined 1MB. If we want to avoid this allocation we understand now that the route must no longer be a regexp or a regexp that requires less allocated memory.[^5] As noted before, the heap profile does not contain the actual heap and thus it's not possible determine what the content of the heap was and analyze what was contained in the 1MB regexp objects.

[^5]: This approach involves also understanding the source code and behavior of another library to verify if this is possible and for this example understand why these two particular regexp routes allocated more memory than other regexp routes.

Performance profiling

When talking about profiling, it usually refers to performance profiling or, more precisely, profiling where the CPU is spending time in a time period. It should be noted that if profiling is done on a production instance, you will have no indication of which particular request or task contributed to the CPU time of a particular function; it's merely a sum of the time being spent in a function. If that information is needed, tracing should be used.

You can download the CPU profile from Forgejo on the stacktrace page in the admin configuration page. The seconds value is important here; it specifies the time period for which you want to collect a CPU profile, and this value should be chosen somewhat carefully to avoid having too much data; keep it as low as possible. After having the `forgejo-diagnosis-xxx.zip` file, extract the `cpu-profile.dat` file. You can continue with the explanation of how to use the pprof tool from the section in memory profiling; it is largely the same, but now instead of memory, the CPU time is being analyzed.

Although Go provides a CPU profiler that is offered by Forgejo, in certain scenarios using a wall-clock profiler might give better results, where I/O and CPU time are mixed into one profile. Forgejo uses felixge/fgprof to provide that. This profile is not available by default in the `forgejo-diagnosis-xxx.zip` file; it can only be retrieved by enabling it in the config `[server].ENABLE_PPROF = true` and then downloading `http://localhost:6060/debug/fgprof?seconds=5`. This must be fetched on the same machine where the Forgejo instance is running.

The results from fgprof are quite a lot and hard to take in, and using the filtering capabilities of pprof is a necessity. Forgejo has many long-lived background goroutines that are all sleeping and spending zero CPU time, which means they will show that they are spending **all** their time in `runtime.gopark`, which simply means the goroutine is sleeping and not active. This also usually means it doesn't actually take up an OS thread.[^6] Therefore you should usually start with Go's provided CPU profile and only revert to this profile if you suspect slowdowns related to I/O events. Also, by using Go's provided CPU profile, you have a way better picture of the functions that are being called, which can be used for filtering out unrelated functions in the fgprof profile.

[^6]: This has to do with the fact that goroutines are lightweight threads; if one goroutine is sleeping, it no longer takes up a thread, and another goroutine can take (now or in the future) up that thread.

When trying to optimize a function or procedure for performance, remember that optimization should be limited to what you can measure in order to avoid the root of all evil: premature optimization.[^7] This can be done by creating a

benchmark function in the Forgejo codebase, which allows CPU profiling over this specific function and avoids any other disturbance; additionally, use benchstat to compare results between benchmark runs.

[^7]: A infamous quote by Donald Knuth in Computer Programming as an Art (1974) on page 671.

Tracing

To be upfront, this type of profiling is quite powerful and gives you a lot of data in return, but is hard to understand and can be confusing to follow what the output is actually saying, it also requires an intermediate understanding of Go's runtime and Forgejo internals. The tracing implemented in Go is an execution tracer and precisely traces the execution of tasks; these tasks can be SQL queries, HTTP requests, or a queue handler, for example. It therefore can exactly tell you where a specific task was spending its time.

Similar to the CPU profile, to download the tracing profile from Forgejo, go to the stack trace page in the admin configuration page. The seconds specify for how long you want to do tracing; existing processes that were not created after the tracing started will also be included in the tracing but will contain skewed results as they are being traced somewhere in the middle of their task. After having the `forgejo-diagnosis-xxx.zip` file, extract the `tracing.dat` file.

There are two tools to analyze the tracing results, Go's native tooling that uses Catapult or the more modern alternative gotraceui[^8]. For now, we will stick with Go's native tools to explain the essentials of analyzing tracing results.

[^8]: You have to build this from source and apply this patch.

Execute `go tool trace trace.dat` after loading the file, it will open a web UI, which is the main interface for analyzing trace results. There are many options to choose from, although only one is really of interest for Forgejo traces: "User-defined tasks" in the "User-defined tasks and regions" section allows you to see all the individual traces categorized by the task of the trace; you should primarily look here to find a trace such as an HTTP request. It also allows you to quickly find slow traces in a particular categorization. For HTTP requests you should avoid taking into account `GET /user/events`; this is a long-lived request that is kept open between navigating the Forgejo UI. Each task logs the main purpose of that task; for HTTP requests, it logs the method and request URI, and SQL logs the SQL query and its arguments. To see what the task actually executed, you can click on "Goroutine overview"; this opens the trace viewer for that task.

In the trace viewer there are a few sections; only "Tasks" and "G" are really of interest. The tasks show the SQL and Git operations that were done; some Git tasks are long-lived, which means they are reused later on without needing to invoke a Git command again. Unfortunately, it is not possible to quickly see what the purpose of the task is; the logging information is not available in the trace view. It is, however, possible to deduce this partially from the stack trace that is shown

after clicking on the task. In the "G" section, "goroutines are shown, and it shows what the goroutine(s) that were associated with this task were doing; this can conceptually be seen as where CPU time was spent in the goroutine and thus gives a detailed breakdown of the execution of a particular trace. Some functions have a relatively low amount of CPU time; it's therefore recommended to make liberal use of the zoom tool (press 3) to zoom into the timescale to see in greater detail the functions that were called; to move around the timescale, press 2.

□□□ #2
□ □□□ □□□ 2026-10-31 13:34:19 UTC
□ □□□ □□□ 2026-10-31 13:34:20 UTC