Monitoring Go Applications with DynaTrace

Providing fast feedback is a key aspect of continuous integration, so kicking off a new build when changes are committed is a common practice in modern software development.

In the last weeks I had the interesting task of improving Dynatrace’s CI build triggering process. Although it fulfilled basic requirements, like stability and appropriate speed, there were some problems when coming to configuration, versioning or maintainability. For example, a part of the logic runs as Groovy code on the CI server and connecting the different tasks just relied on shell scripts with curl commands.

The redesign of the process with usage of new technologies was the perfect task for my internship at the Dynatrace Test Automation (<http://jobs.dynatrace.at/>) team. To solve all the mentioned problems, a new architecture was necessary. So the main target was to get the Groovy code out of the CI server and split logic and configuration. That was also the possibility to broaden my knowledge of modern programming languages, as I got the chance to use Go (<https://golang.org/>) for the logic part.

# Why Go and What is it!

For those who haven’t heard of Go before - what exactly is Go? Go, often also called Golang, is a native, open-source language created at Google, partly developed by the famous Ken Thompson (<https://en.wikipedia.org/wiki/Ken_Thompson>), co-creator of Unix. Its approach is to get the best out of other languages, but also to resolve their points of criticism. So, as a result, Go is a compiled, statically typed language with concurrent programming features, garbage collection and more. It can target various platforms, also including smartphones. Go is for example in use at Google, Netflix, Dropbox, Uber and SoundCloud and Docker is written completely in Go.

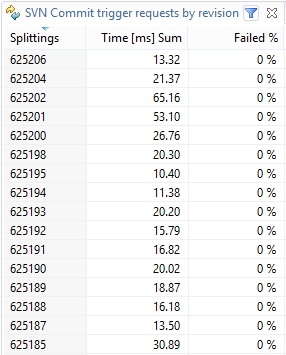
So, I basically wrote a Go application that was dealing with the logical build steps while it was reading its configuration from a JSON-formatted file. I also wanted to add automated monitoring of my Go executions when triggered by our VCS server.

# Why Monitoring Go Executions?

You may be wondering, why it could be important to monitor this application? Even though my Go code might be not that complex, it gets executed every time somebody checks in code. And there are things that could go wrong, such as parsing the incoming file from the VCS, which could possibly be malformed due to some bug earlier in the process. Maybe someone made a configuration mistake, so that wrong builds would be triggered or someone wanted to trigger builds while the CI server is in maintenance and not reachable. As you see, there are some things that could potentially go wrong.

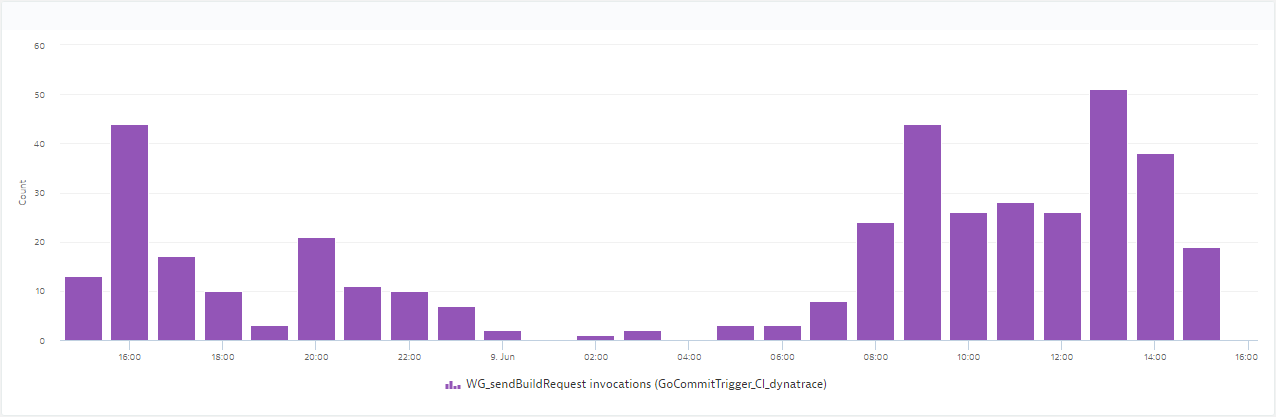
Logging everything that is (or may be) wrong appears as the standard solution, but would produce a lot of log output when building by the minute! The log output would be quite huge, maybe confusing and it would require some type of log analytics to correlate errors with build numbers.

As I have Dynatrace at my disposal I can get all the troubleshooting information much easier without going through log, Splunk or ElasticSearch. I just leverage the PurePath Technology for each incoming build request and capture parameters, return values and errors. To find errors in the process, we can just filter the PurePaths by the VCS revision number of the commit with the corrupted output using a Business Transaction. Here is a sample output showing me every Revision (Splitting Column) with its Execution Time (in ms) and whether the execution has failed or not.



*Splitting shows us the revision number and with Dynatrace’s filter functions, it is easy to find the one we are looking for.*

We can also generate a chart, which shows us the number of incoming build requests over a specific time interval.



# Instrumenting your Go App with Dynatrace

If you want to try this out feel free to download the Dynatrace Free Trial (<http://www.dynatrace.com/en/products/dynatrace-personal-license.html>) and start monitoring your own Go application! After the 30 Days Trial Period you can keep using Dynatrace to analyze your local apps – FOR LIFE!

In the following paragraphs I will show you the necessary steps to analyze any Go Application. In case you have other types of Apps such as Java, .NET, PHP, Node.js, Mobile or Web simply follow the guides as shown in the [Video Tutorials we host on YouTube](http://bit.ly/dttutorials).

## Dynatrace Native ADK

Dynatrace has agents for quite a lot of platforms, but unfortunately not for native Go applications. For these cases, you can use the so called Native ADK. The Native ADK provides C and C++ functions for instrumenting the source code (<https://community.dynatrace.com/community/display/DOCDT63/Native+ADK>). You can either work with the provided macros or directly with the functions of the ADK. I prefer the second approach, because although you have to get some values manually, you have more control about how to monitor your application.

### Using the Native ADK

You may be wondering about how to get C code into Go code. Fortunately, Go introduces the Cgo command, which allows Go packages to call C code that is written directly above an import of the pseudo-package “C”. To separate code, it is also possible to put the C code in an own C file, write a header file and import it.

The only requirement to use that feature is an installed GNU C Compiler. For Windows systems you can for example use MinGW. Of course there are some differences between C and Go, for example when it comes to data types, but Cgo provides you with the necessary converter functions between C and Go types.

With that knowledge it now isn’t hard to start working with the Native ADK. Requirement to use it is to simply have the adk and agent libraries on your system (install Dynatrace or put them into your Go workspace if you have them), set the CFLAGS and LDFLAGS with pseudo #cgo directives and also set the needed environment variables. That is all you need to start creating your PurePath.

So all together your result should for example look like this:

#### Cgo

/\*

#cgo CFLAGS: -I/go/dynatrace-6.3/adk/include

#cgo LDFLAGS: -L/go/dynatrace-6.3/adk/lib64 -l dtadk

#include "nativeAdkFunctions.h"

\*/

import "C"

If you put the adk and agent libraries into your src folder, you can simple write for example ${SRCDIR}/adk/include, which would be expanded to /go/src/projectname/adk/include.

#### Environment variables

ENV DT\_AGENTLIBRARY %DTPATH%/agent/lib64/dtagent.dll

ENV DT\_DEBUGADK true

ENV DT\_SERVER yourDynatraceServer:port

ENV LD\_LIBRARY\_PATH %DTPATH%/adk/lib64

For Windows systems you have to use the variable PATH instead of LD\_LIBRARY\_PATH.

## Monitoring Go Applications

The Native ADK provides the basic functions to create a PurePath, together with capturing parameters, return values, exceptions, log entries and more.

In the provided files “nativeAdkFunctions.h” and “nativeAdkFunctions.c” (insert future GitHub link) there are most of the functions you will need. You can edit the two constants FILE and API in the header file to your preferred labels to get going.

You basically find all the necessary information about these functions in the Native ADK documentation, but because I, as mentioned above, used the Native ADK functions directly instead of the macros, here is a bit of explanation on how to use the functions in the given file.

### Agent

The initializeAgent() function initializes the Dynatrace Agent, while the uninitializeAgent() function terminates it and shuts it down. So both functions should be called only once per application – on startup respectively shutdown.

func main() {

C.initializeAgent()

// …

C.uninitializeAgent()

}

### Sensor placement

It’s quite the same with placeSensor() and exitSensor(), but this time on a per-function-basis. Just pass the method name and the type of sensor placement (0 – simple, 1 – entry point) to placeSensor() and you’ll receive a struct with a method id and a serial no. You’ll need them for the other functions that are exetuced in the context of the function, as it is the logError() function in this example. To mark the end point of the sensor, just call exitSensor() with the previously received struct as parameter. Alternatively you can also call one of the return\*AndExitSensor() functions, which basically do the same, but also capture a return value from the given type to show it in the PurePath.

func someFunction() {

ctx := C.placeSensor(C.CString("someFunction"), C.int(1))

// …

C.logError(ctx.methodId, C.CString("Some error occurred."))

C.exitSensor(ctx) // or C.returnStringAndExitSensor(ctx, C.CString("test"))

}

### Tagging

Tagging is necessary to interact with agents on other systems and build a connected PurePath. Each new connection also needs a new tag, so it has to be fetched before each corresponding call. For example, doing this with a REST call would probably look like this:

w := &bytes.Buffer{}

req, err := http.NewRequest("POST", "http://dynatrace.com/", w)

cTag := C.getTag()

goTag := C.GoString(cTag)

C.linkClientPurepath(cTag)

req.SetBasicAuth(node.QbTarget.QBUSER, node.QbTarget.QBPASS)

req.Header.Set("Content-Type", "text/xml")

req.Header.Set("X-Dynatrace", goTag)

resp, err := client.Do(req)

Note that in addition to call C.linkClientPurepath() you also have to put the tag with the key “X-Dynatrace” into the request header.

### Troubleshooting

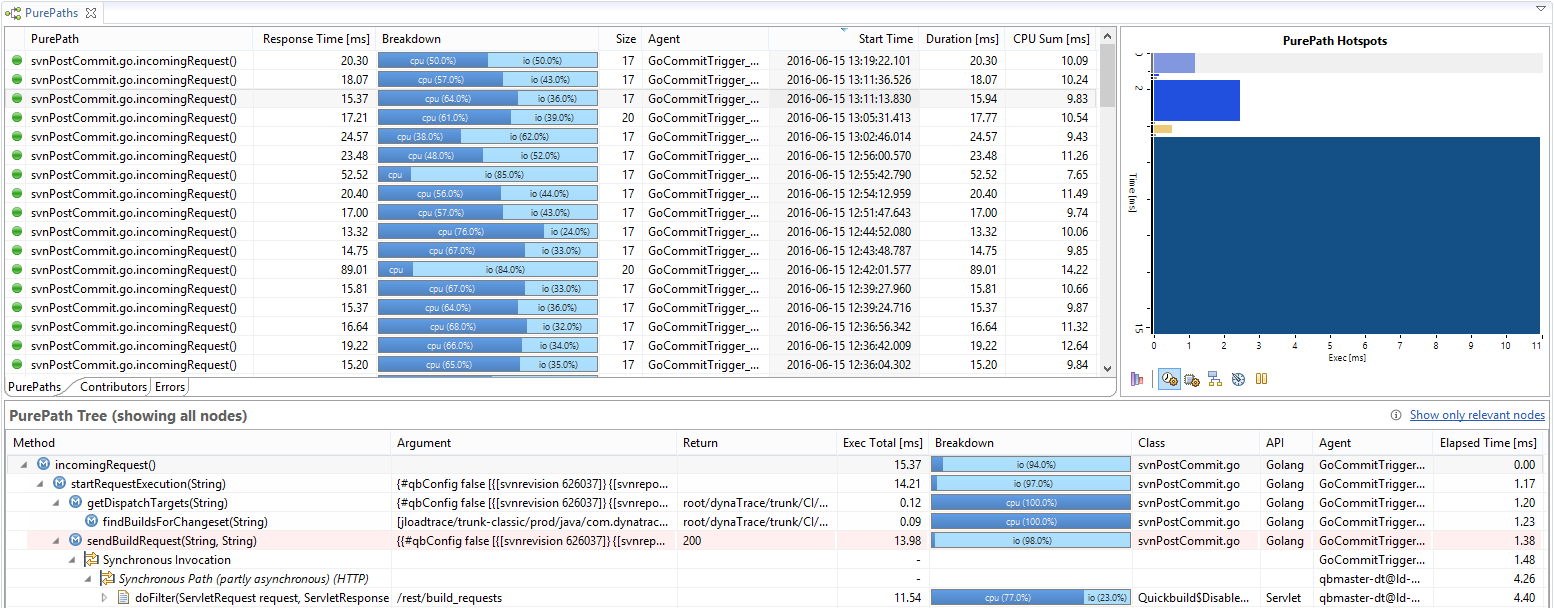
If you experience timeouts or corrupted paths, please check if you end every placeSensor() with a corresponding exitSensor() with the right context. Same of course for initializeAgent() and uninitializeAgent(). Tagging requires a running agent on the targeting system. Go likes to switch threads by itself, which leads to losing the tag from time to time. In these cases you have to lock and unlock the thread before a critical section with the functions provided by the “runtime” package (for example runtime.lockOSThread()).

### Resulting PurePath

If you configured your application the right way, you should see a nice PurePath emerging in the Dynatrace Client.

Here you can see the fluent connection between the Go application, identifiable through the API name “Golang” and monitored with the Native ADK, and the agent running on the CI server.

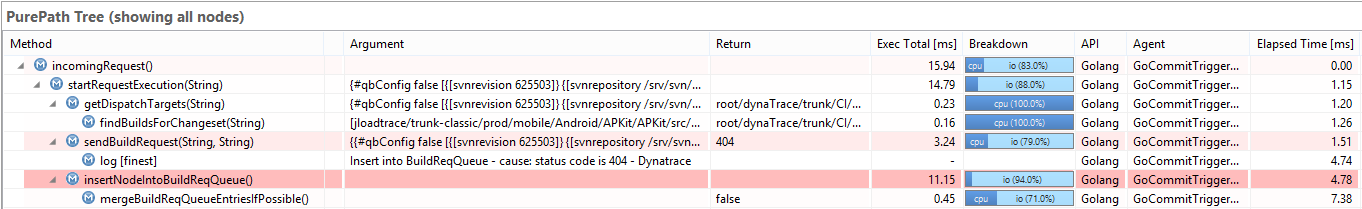
One PurePath for every Build Request



Each step is shown which its Arguments and Return Values

Trace into the build server

That would be what you see if everything goes well, but what if something goes wrong? Here is an example of the PurePath, when our build server was down for maintenance. The return value of the sendBuildRequest() function is the response status code – 404 – which is a nice indicator, that the request hasn’t reached the server. In case of errors, there will be log messages in the PurePath with a short description about what happened or where it happened.



Request failed with code 404

Log with error message

## Conclusion

The increasing usage of Go, especially at Google, doesn’t come out of nowhere. It is a nice, well-structured language that provides quite everything you may need to write applications, regardless of complexity, and may replace many C programs in the future. Multiple IDEs and text editors already support Go, which makes developing more comfortable.

Monitoring a Go application may seem a bit complex at first, but with the Native ADK and cgo there really isn’t any complexity at all. Just include the provided C code, use the installed adk and agent libraries, place sensors in your methods, capture parameters or return values if you want to and you’ll find your PurePath in Dynatrace! With Business Transactions you have every possibility to analyze the application in any way you like.

### More Resources

* GitHub Dynatrace-NativeADK-Go (or something like that?): <insert github link here>
* CGO documentation: <https://golang.org/cmd/cgo/>
* DT Native ADK: <https://community.dynatrace.com/community/display/DOCDT63/Native+ADK>
* Example project: <https://github.com/d-smith/go-dynatrace>

## About the author

Michael Bernard

Michael Bernard is an Intern at the Dynatrace Testautomation Team in Linz, Austria. He is just about to finish his bachelor’s degree in Software Engineering at the University of Applied Sciences in Hagenberg and will extend his skills with the consecutive master’s degree in the next two years. Although he hasn’t finished studying, he already had the chance to work for several companies in different areas of IT.

