Go HTTP/2 TLS Server Push

Motivation

Seit der Go Version 1.8 wird das HTTP/2 Protokoll beim HTTP Server Einsatz unterstützt und damit auch die Push Funktion. Die Plattform stack.ch unterstützt seit der Version 1.9.3 HTTP/2 und stellt die CSS und Javascript Dateien mit der Push Funktion dem Browser zur Verfügung. Damit bietet stack.ch die optimale Performance und neusten Technologien für die laufenden Domains. Dieser Blog zeigt den Bau eines minmalen Golang HTTP/2 TLS Servers mit Server Push Funktion.

HTTP/2 Vorteile

- HTTP/2 ist binär statt textuell, da binäre Protokolle effizienter zu analysieren sind, kompakter und viel weniger fehleranfällig sind.
- HTTP/2 kann mit der gleichen Verbindung mehrere Anforderungen gleichzeitig verarbeiten.
- HTTP/2 benötigt nur eine einzelne Verbindung um eine Website zu laden. Diese Verbindung bleibt bestehen, solange die Website im Browser geöffnet ist. Dies reduziert die Anzahl der Handshake-Vorgänge, die zum Herstellen einer TCP-Verbindung erforderlich sind.
- HTTP/2 verwendet die Header-Komprimierung. Da viele Header mit denselben Werten gesendet werden, verwendet HTTP/2 die HPACK-Header-Komprimierung, wodurch der Overhead reduziert wird.
- HTTP/2 ermöglicht es dem Server, Ressourcen auf den Client zu übertragen. Anstatt dass der Client nacheinander Ressourcen anfordert und der Server dann auf diese Anforderungen reagiert, kann der Server Ressourcen proaktiv in den Client-Cache übertragen.

HTTP/2 Push

Ohne HTTP Push holt sich der Browser eine HTML Datei, parsed solche und holt sich dann z.B. die CSS Styledatei:Mit HTTP/2 Push holt sich der Browser eine HTML Datei vom Server. Der Server erkennt, dass die CSS Styledatei auch geladen werden wird, und löst ein HTTP/2 Push an den Browser bezogen auf die CSS Datei aus.Der Browser lädt die CSS Datei parallel vom Server, also gleichzeitig mit der HTML Datei. Damit wird die Site schneller geladen:

TLS Cert

HTTP/2 Push setzt das HTTPS Protokoll voraus. Wir benötigen für unseren Server also ein TLS Certificate. Solches erstellen wir über das openssl Tool wie folgt:# Key considerations for algorithm "RSA" . 2048-bit
openssl genrsa -out server.key 2048

Key considerations for algorithm "ECDSA" . secp384r1
List ECDSA the supported curves (openssl ecparam -list_curves)
openssl ecparam -genkey -name secp384r1 -out server.key

Generation of self-signed(x509) public key (PEM-encodings .pem|.crt) based on the private (.key)
openssl req -new -x509 -sha256 -key server.key -out server.crt -days 3650Die erstellten Dateien server.crt und server.key referenzieren wir im nachfolgenden Go Code.

Golang HTTP/2 TLS Server

Das folgende Listing zeigt einen minimalen HTTP/2 TLS Server mit Go:package main

import (
  "log"
  "net/http"
 "os"
  "path/filepath"
 "golang.org/x/net/http2"
)

func main() {
  port := "443"
  if len(os.Args) > 1 {
  port = os.Args[1]
  }
  srv := &http.Server{
  Addr: ":" + port,
  Handler: HTTPHandler(),
 }
  http2.ConfigureServer(srv, &http2.Server{})
 log.Fatal(srv.ListenAndServeTLS("server.crt", "server.key"))
}

func HTTPHandler() http.Handler {
  return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
  path := r.URL.Path
  if path == "/" {
  pusher, ok := w.(http.Pusher)
  if ok {
 options := &http.PushOptions{
  Header: http.Header {
 "Accept-Encoding": r.Header["Accept-Encoding"],
  },
 }
  pusher.Push("/css/main.css", options)
  }
  }

workingPath, _ := os.Getwd()
  realPath := filepath.Join(workingPath, path)
  stat, err := os.Stat(realPath)
  if err != nil {
  if os.IsNotExist(err) {
 log.Println(realPath + " does not exist")
  return
  }
  }
  if stat.IsDir() {
  realPath = filepath.Join(realPath, "index.html")
  if stat, err = os.Stat(realPath); err != nil {
  if os.IsNotExist(err) {
 log.Println(realPath + " not found")
  }
  }
  }
 log.Println("serve file " + realPath)
  http.ServeFile(w, r, realPath)
 })
}Der Aufbau entspricht einem normalen HTTP Server über die http.Server Instanz. Dieser wird mit dem folgenden Befehl zu einem HTTP/2 Server:http2.ConfigureServer(srv, &http2.Server{})Die HTTP/2 Push Funktion erfolgt über die http.Pusher Abfrage. Solche ist ok true wenn der Server dies unterstützt:pusher, ok := w.(http.Pusher)
if ok {
  options := &http.PushOptions{
  Header: http.Header {
  "Accept-Encoding": r.Header["Accept-Encoding"],
  },
  }
 pusher.Push("/css/main.css", options)
}Das Accept-Encoding ist wichtig, wenn z.B. die GZip Unterstützung benötigt wird.

Feedback

War dieser Blog für Sie wertvoll. Wir danken für jede Anregung und Feedback

Kontakt

Simtech AG
Finkenweg 23
3110 Münsingen
Schweiz

Impressum

Das Copyright für sämtliche Inhalte dieser Website liegt bei Simtech AG, Schweiz.
Beachten Sie auch unsere Hinweise zum Urheberrecht, Datenschutz und Haftungsauschluss.
Jeder Hinweis auf Fehler nehmen wir gerne entgegen.

Copyright

2024 Simtech AG, All rights reserved, Powered by stack.ch written in Golang by Daniel Schmutz

https://www.simtech-ag.ch/Accept