// Sample program that implements a simple web service using the

// context to handle timeouts and pass context into the request.

package main

import (

"context"

"encoding/json"

"log"

"net/http"

"time"

)

// The key type is unexported to prevent collisions with context keys defined in

// other packages.

type key int

// userIPkey is the context key for the user IP address. Its value of zero is

// arbitrary. If this package defined other context keys, they would have

// different integer values.

const userIPKey key = 0

// User defines a user in the system.

type User struct {

Name string

Email string

}

func main() {

routes()

log.Println("listener : Started : Listening on: http://localhost:4000")

http.ListenAndServe(":4000", nil)

}

// routes sets the routes for the web service.

func routes() {

http.HandleFunc("/user", findUser)

}

// findUser makes a database call to find a user.

func findUser(rw http.ResponseWriter, r \*http.Request) {

// Create a context that timeouts in fifty milliseconds.

ctx, cancel := context.WithTimeout(context.Background(), 50\*time.Millisecond)

defer cancel()

// Save the user ip address in the context. This call returns

// a new context we now need to use. The original context is

// the parent context for this new child context.

ctx = context.WithValue(ctx, userIPKey, r.RemoteAddr)

// Create a goroutine to make the database call. Use the channel

// to get the user back.

ch := make(chan \*User, 1)

go func() {

// Get the ip address from the context for logging.

if ip, ok := ctx.Value(userIPKey).(string); ok {

log.Println("Start DB for IP", ip)

}

// Make the database call and return the value

// back on the channel.

ch <- readDatabase()

log.Println("DB goroutine terminated")

}()

// Wait for the database call to finish or the timeout.

select {

case u := <-ch:

// Respond with the user.

sendResponse(rw, u, http.StatusOK)

log.Println("Sent StatusOK")

return

case <-ctx.Done():

// If you have the ability to cancel the database

// operation the goroutine is performing do that now.

// In this example we can't.

// Respond with the error.

e := struct{ Error string }{ctx.Err().Error()}

sendResponse(rw, e, http.StatusRequestTimeout)

log.Println("Sent StatusRequestTimeout")

return

}

}

// readDatabase performs a pretend database call with

// a second of latency.

func readDatabase() \*User {

u := User{

Name: "Bill",

Email: "bill@ardanlabs.com",

}

// Create 100 milliseconds of latency.

time.Sleep(100 \* time.Millisecond)

return &u

}

// sendResponse marshals the provided value into json and returns

// that back to the caller.

func sendResponse(rw http.ResponseWriter, v interface{}, statusCode int) {

rw.Header().Set("Content-Type", "application/json")

rw.WriteHeader(statusCode)

json.NewEncoder(rw).Encode(v)

}