Implementing REST Services in Go

Mark Volkmann, Partner and Principal Software Engineer
mark@objectcomputing.com

slides at https://github.com/mvolkmann/talks

Why Go, REST, and PostgreSQL?

- **Go** is a fast server-side language
- **REST** services that access relational databases are the most common kind of server-side development
- **PostgreSQL** is a popular open source relational database

Why Go?

- Simple
  - indicated by having a small specification and fewer features than most programming languages
- Fast in compiling and running
- Statically typed
- Great for concurrency and parallelism

Learning Go

- "A Tour of Go" - https://tour.golang.org/
- "Effective Go" - https://golang.org/doc/effective_go.html
- My articles
  - **Getting Started With Go** - https://objectcomputing.com/resources/publications/sett/november-2018-way-to-go-part-1
  - **Mastering Go Syntax** - https://objectcomputing.com/resources/publications/sett/january-2019-way-to-go-part-2
  - **Go Standard Library** - https://objectcomputing.com/resources/publications/sett/april-2019-way-to-go-part-3
- and many resources

REST Services in Go

- Supported by many Go libraries
- Popular choice is Gin - https://gin-gonic.github.io/gin/
  - install with `go get github.com/gin-gonic/gin`
- Following slides show implementing REST services that manage a collection of dogs in a PostgreSQL database
  - each dog has an id, breed, and name

1) create
2) retrieve
3) update
4) delete

Go and PostgreSQL

- Several Go libraries support working with PostgreSQL databases
- Popular choice is pq - https://github.com/lib/pq
  - install with **`go get github.com/lib/pq`**
- DDL to create **`dog`** table in **`ddl.sql`**

```
create table dog (
  id serial primary key,
  breed text,
  name text
);
```

```
createdb pets
psql -d pets -f ddl.sql
```

Cross-Origin Resource Sharing (CORS)

- By default browsers can only send HTTP requests to same origin
- CORS enables sending to other origins

protocol, domain, and port

- Services enable CORS by including specific HTTP response headers
- Can allow sending requests from any domain (*) or specific ones
- Can allow only specific HTTP methods

Preflight Requests

- HTTP methods other than GET
  send a preflight OPTIONS request
  to determine allowed methods
  - so POST, PUT, and DELETE requests are preceded by an OPTIONS request
  - then actual request is sent if allowed

CORS Response Headers

- **`Access-Control-Allow-Origin`**
  - sample value "**`*`**" or "**`http://localhost:8080`**"
- **`Access-Control-Allow-Methods`**
  - sample value "**`DELETE,GET,POST,PUT`**"
- **`Access-Control-Allow-Headers`**
  - sample value "**`Content-Type`**"
- **`Access-Control-Allow-Credentials`**
  - set to **`true`** if request will contain credentials

Imports

all code on following slides is in the file **main.go**


```
package main

import (
    "database/sql" // to open database connection
    "errors"
    "fmt"
    "log"
    "net/http" // for status constants
    "strconv"  // to convert between string and int values

    "github.com/gin-gonic/gin" // HTTP web framework
    _ "github.com/lib/pq"      // Postgres driver
)
```

_ indicates we are not using anything exported by **pq**

Constants and Dog Struct

OBJECT COMPUTING

```
const allowOrigin = "http://localhost:8080"
const badRequest = http.StatusBadRequest
const forbidden = http.StatusForbidden
const ok = http.StatusOK
const serverError = http.StatusInternalServerError

type Dog struct {
    ID    int    `json:"id"`
    Breed string `json:"breed"`
    Name  string `json:"name"`
}
```

struct field names must start uppercase to be visible outside their source file; the **gin** package needs to access them

don't want uppercase properties in JSON that is produced, so alternate names are provided using struct "tags"

CORS

We are handling CORS details manually here.
Consider using https://github.com/gin-contrib/cors.

OBJECT COMPUTING

```
func shouldAllow(c *gin.Context) bool {
    origins := c.Request.Header["Origin"]
    return len(origins) > 0 && origins[0] == allowOrigin
}

// Custom middleware to enable CORS
func cors(c *gin.Context) {
    if shouldAllow(c) {
        c.Header("Access-Control-Allow-Origin", allowOrigin)
    } else {
        c.Status(forbidden)
    }
}

func options(c *gin.Context) {
    c.Header("Access-Control-Allow-Methods", "GET,POST,PUT,DELETE")
    c.Header("Access-Control-Allow-Headers", "Content-Type")
    c.Status(ok)
}
```

use * for `allowOrigin` to allow any

headers only needed for OPTIONS requests

must explicitly allow `Content-Type` header for JSON bodies

several headers are allowed by default

Error Handling


```
func handleError(c *gin.Context, statusCode int, err error) {
    c.String(statusCode, err.Error()) // writes to response body
}
```

Database Connection


```
func main() {
    // Connect to database.
    connStr := "user=postgres dbname=pets sslmode=disable"
    db, err := sql.Open("postgres", connStr)
    if err != nil {
        log.Fatal(err)
    }
```

requires SSL by default

HTTP Router Setup and Heartbeat

OBJECT COMPUTING

```
// Configure HTTP request routes.
router := gin.Default()
router.Use(cors)

// For OPTIONS request before POST.
router.OPTIONS("/dog", options)

// For OPTIONS request before PUT and DELETE.
router.OPTIONS("/dog/:id", options)

// Heartbeat
router.GET("/", func(c *gin.Context) {
    c.String(ok, "I'm alive!")
})
```

cors function defined earlier

options function defined earlier

Create Dog

OBJECT COMPUTING

```
router.POST("/dog", func(c *gin.Context) {
    var dog Dog
    if err := c.ShouldBindJSON(&dog); err != nil {
        handleError(c, badRequest, err)
        return
    }

    sql := fmt.Sprintf(
        "insert into dog (breed, name) values ('%s', '%s') returning id",
        dog.Breed,
        dog.Name)
    var id int
    err := db.QueryRow(sql).Scan(&id)
    if err != nil {
        handleError(c, serverError, err)
        return
    }

    dog.ID = id
    c.JSON(ok, dog)
})
```

get dog from request body

insert dog into database and get assigned id

return JSON representation of new dog including assigned id

Retrieve Dogs ...


```
router.GET("/dog", func(c *gin.Context) {
    if !shouldAllow(c) {
        c.Status(forbidden)
        return
    }

    rows, err := db.Query("select id, breed, name from dog")
    if err != nil {
        c.String(serverError, err.Error())
        return
    }
    defer rows.Close()
```

manually checking whether request should be allowed because GET requests are not preceded by an OPTIONS request

get all dogs from database

... Retrieve Dogs

OBJECT COMPUTING

```
        dogs := []Dog{}
        var id int
        var breed, name string

        for rows.Next() {
            if err := rows.Scan(&id, &breed, &name); err != nil {
                c.String(serverError, err.Error())
                return
            }
            dogs = append(dogs, Dog{id, breed, name})
        }

        c.JSON(ok, dogs)
    })
```

create array of **Dog** structs

create new **Dog** object and append to array

return JSON representation of dog array

Update Dog ...

OBJECT COMPUTING

```
router.PUT("/dog/:id", func(c *gin.Context) {
    id, err := strconv.Atoi(c.Param("id"))
    if err != nil {
        handleError(c, badRequest, errors.New("id must be int"))
        return
    }

    var dog Dog
    if err := c.ShouldBindJSON(&dog); err != nil {
        handleError(c, badRequest, err)
        return
    }
```

convert string URL parameter to integer

get dog from request body

... Update Dog


```
        sql := fmt.Sprintf(
            "update dog set breed='%s', name='%s' where id=%d",
            dog.Breed,
            dog.Name,
            id)
        if _, err := db.Query(sql); err != nil {
            handleError(c, serverError, err)
            return
        }

        c.Status(ok)
    })
```

update breed and name of dog in database

Delete Dog


```
router.DELETE("/dog/:id", func(c *gin.Context) {
    id, e := strconv.Atoi(c.Param("id"))
    if e != nil {
        handleError(c, badRequest, errors.New("id must be int"))
        return
    }

    sql := fmt.Sprintf("delete from dog where id=%d", id)
    if _, err := db.Query(sql); err != nil {
        handleError(c, serverError, err)
        return
    }

    c.Status(ok)
})
```

convert string URL parameter to integer

delete dog from database

Start Router

starts server

```
    router.Run(":1919")
} // end of main function started on slide 14
```

Running REST Server

- **`go run main.go`**
  or
  `go build main.go; ./main`

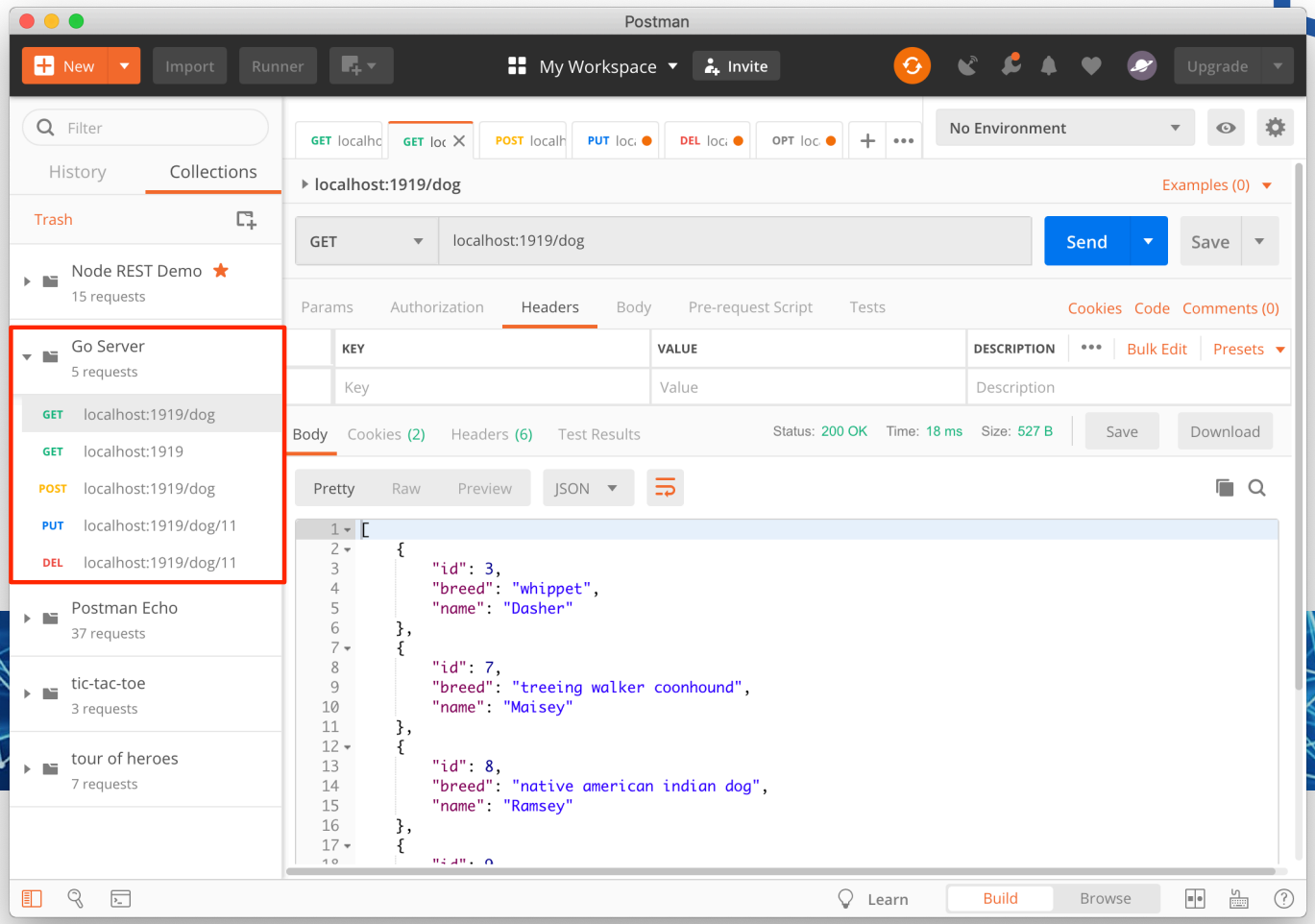
logs all configured routes

```
[GIN-debug] OPTIONS /dog                  --> main.options (4 handlers)
[GIN-debug] OPTIONS /dog/:id              --> main.options (4 handlers)
[GIN-debug] GET    /                      --> main.main.func1 (4 handlers)
[GIN-debug] POST   /dog                   --> main.main.func2 (4 handlers)
[GIN-debug] GET    /dog                   --> main.main.func3 (4 handlers)
[GIN-debug] PUT    /dog/:id               --> main.main.func4 (4 handlers)
[GIN-debug] DELETE /dog/:id               --> main.main.func5 (4 handlers)
[GIN-debug] Listening and serving HTTP on :1919
```

Testing REST Services ...

- Can use Postman to create, catalog, and execute HTTP requests
  - https://www.getpostman.com/

... Testing REST Services



Watch and Live Reload

- Go servers can automatically rebuild and restart when changes are detected
- Supported by https://github.com/codegangsta/gin
  - not related to Gin web framework, just a naming coincidence

gin Setup

- Install with **`go get github.com/codegangsta/gin`**
- Verify with **`gin -h`**
- Run with **`gin --appPort 1919 run main.go`**
  - assumes Go REST server is implemented in **`main.go`** and listens on port **`1919`**
- Web UI must send requests to gin port which defaults to 3000

consider putting this in a **`start`** script

Wrap Up

- Now you know all the basics for implementing REST services in Go!

LEARN MORE ABOUT OCI EVENTS AND TRAINING

Events:

- objectcomputing.com/events

Training:

- objectcomputing.com/training
- grailstraining.com
- micronauttraining.com

Or email info@ocitraining.com to schedule a custom training program for your team online, on site, or in our state-of-the-art, Midwest training lab.

OBJECT
COMPUTING

CONNECT WITH US

1+ (314) 579-0066

@objectcomputing

objectcomputing.com