Learning Objectives

Learners will be able to…

- **Create a simple web server using gin**
- **Create a simple gRPC server**

info

Make Sure You Know

Python or other modern programming language.

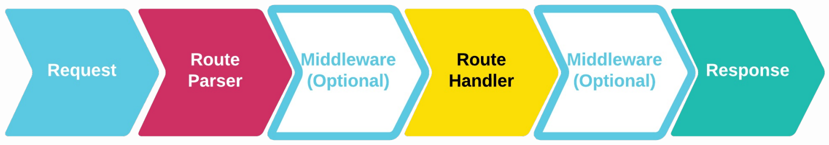
Routing with gin

We are going to create a simple web service using the `gin` framework – one of the most popular web frameworks for Go.

Key features of Gin:

- Zero Allocation Router
- Fast
- Middleware support
- Without crashing
- JSON validation
- Grouping routes
- Error management
- Rendering built-in
- Expandable

Let's see how Gin handles requests:



When a **request** comes in, Gin first checks to see if there is a **suitable route**. If a matching route is found, Gin runs that **route's handler** and intermediates in the given order.

`cmd/main.go`

Over the next few pages, you will write a simple web service that can convert a string to base64 and back, and convert unix timestamp to a human readable format.

In the `gin-example` folder:
* Create a `cmd` directory (File > New Folder...)
* Create a `main.go` file inside the `cmd` directory (File > New File...)

To initialize the module and add the `gin` dependency, open a terminal (Tools > Terminal) and run:

```
cd gin-example
go mod init gin-example
go get -u github.com/gin-gonic/gin
```

You should see the `go.mod` file appear in the filetree.

Inside of `main.go`, one of the first things is to import `gin` along with other basics:

```go
package main

import (
    "fmt"
    "net/http"

    "github.com/gin-gonic/gin"
)
```

Inside the `main` function, create a router:

```go
func main() {

    router := gin.Default()

}
```

The `router` has `GET`, `POST` and other HTTP method functions to handle requests. They take the route itself and one or more handlers as parameters.

Add a route handler for the `GET` `ping` request:

```go
router.GET("/ping", func(c *gin.Context) {
        c.String(http.StatusOK, "pong")
    })
```

The route handler has a context pointer `*gin.Context` in its parameters containing all the information about the request (e.g. headers, cookies, etc.)

`Context` also has methods for outputting the response in HTML, text, JSON, and XML formats.

In the case above, we respond to the `/ping` request with the text string `pong`.

Next we will create an http server and launch it:

```
srv := http.Server{
  Addr:    fmt.Sprintf(":%d", 3000),
  Handler: router,
}
srv.ListenAndServe()
```

To start the server, open a terminal (Tools > Terminal) and run:

```
cd gin-example
go run cmd/main.go
```

After the server has been started, you can check its functionality by opening a **second** terminal and running the command:

```
curl localhost:3000/ping
```

You will get `pong` in response.

▼ **Check the code you built**

package main

import (
“fmt”
“net/http”

```
"github.com/gin-gonic/gin"
)

func main() {
  router := gin.Default()

  router.GET("/ping", func(c *gin.Context) {
    c.String(http.StatusOK, "pong")
  })

  srv := http.Server{
    Addr:    fmt.Sprintf(":%d", 3000),
    Handler: router,
  }
  srv.ListenAndServe()
}
```

Query Processing

Let’s continue working on our application. Create a `unixtime` package:
* Create an `internal` directory inside the `gin-example` folder
* Create a `unixtime` directory inside the `internal` directory
* Create a `converter.go` file inside the `unixtime` directory
* Create a `routes.go` file inside the `unixtime` directory

`unixtime/routes.go`

In the `routes.go` file we will add the `RegisterRoutes` function for adding routes. This function takes `*gin.Engine` as a parameter. Next, let’s create a group of routes `routesUnixtime`. To create a group, use the `Group` method of `gin.Engine`.

Group creates a new router group. You should add all the routes that have common middlewares or the same path prefix. For example, all the routes that use a common middleware for authorization could be grouped.

Then we will add the `routesUnixtime.GET("/:timestamp", h.toHuman)` handler.
We pass `path` to it - `"/:timestamp"`. Note the colon. Thus we show that instead of `:timestamp` any valid string can be used. In the future, we will be able to obtain this value using the context. We also pass the request handler `h.toHuman` which we will implement next. Thus, our request will look like `localhost:3000/unixtime/3523463425`.

`unixtime/converter.go`

Now in the `converter.go` file we implement the `toHuman` method. Let’s create a `handler` structure and a `toHuman` method for it.
It takes `*gin.Context` as a parameter.
Using the `Param` method of `gin.Context` we get the timestamp. As we already remember, we added a handler for `"/:timestamp"`.
Then, in case of an error, we can terminate the request with an error by specifying the response status `http.StatusBadRequest`:

```go
if err != nil {
  c.AbortWithError(http.StatusBadRequest, err)
  return
}
```

If successful, we will return the response `c.String(http.StatusOK, tm.String())`. Where we indicate the status of the response and the response itself.

```go
// converter.go
package unixtime

import (
	"net/http"
	"strconv"
	"time"

	"github.com/gin-gonic/gin"
)

type handler struct{}

func (h handler) toHuman(c *gin.Context) {
	timestamp := c.Param("timestamp")
	parsedTimeStamp, err := strconv.ParseInt(timestamp, 10, 64)
	if err != nil {
		c.AbortWithError(http.StatusBadRequest, err)
		return
	}
	tm := time.Unix(parsedTimeStamp, 0)
	c.String(http.StatusOK, tm.String())
}
```

`base64` package

Let's write a base64 package in the same way and register our new routes in the `main.go` file `base64.RegisterRoutes(router)` and `unixtime.RegisterRoutes(router)`.

```
func main() {
    router := gin.Default()

    router.GET("/ping", func(c *gin.Context) {
        c.String(http.StatusOK, "pong")
    })

    base64.RegisterRoutes(router)
    unixtime.RegisterRoutes(router)

    srv := http.Server{
        Addr:    fmt.Sprintf(":%d", 3000),
        Handler: router,
    }
    srv.ListenAndServe()
}
```

Create a `base64` package:
* Create a `base64` directory inside the `internal` directory
* Create a `converter.go` file inside the `base64` directory
```golang
package base64
```

~~~
import (
    _base64 "encoding/base64"
    "errors"
    "net/http"

    "github.com/gin-gonic/gin"
)

var errBadRequest = errors.New("bad request")

type handler struct{}

func (h handler) decode(c *gin.Context) {
    base64, ok := c.GetQuery("base64")
    if !ok {
        c.AbortWithError(http.StatusBadRequest, errBadRequest)
        return
    }
    decoded, err := _base64.StdEncoding.DecodeString(base64)
    if err != nil {
        c.AbortWithError(http.StatusBadRequest, err)
        return
    }
    c.String(http.StatusOK, string(decoded))
}

func (h handler) encode(c *gin.Context) {
    text, ok := c.GetQuery("text")
    if !ok {
        c.AbortWithError(http.StatusBadRequest, errBadRequest)
        return
    }
    encoded := _base64.StdEncoding.EncodeToString([]byte(text))
    c.String(http.StatusOK, encoded)
}

```
~~~

- Create a `routes.go` file inside the `base64` directory
  \`\`\`golang
  package base64

  import “github.com/gin-gonic/gin”

  func RegisterRoutes(r *gin.Engine) {
  h := &handler{}

```
    routesBase64 := r.Group("/base64")
    routesBase64.GET("/decode", h.decode)
    routesBase64.GET("/encode", h.encode)
```

}
```

To start the server, open a terminal (Tools > Terminal) and run:

```
cd gin-example
go run cmd/main.go
```

After the server has been started, you can check its functionality by opening a **second** terminal and running the command:

```
curl localhost:3000/unixtime/3523463425
curl "127.0.0.1:3000/base64/encode?text=hello"
curl "127.0.0.1:3000/base64/decode?base64=aGVsbG8="
```
```

# Middleware

Middleware is a piece of code that can be executed during the processing of an HTTP request. Typically they are used to encapsulate common functionality that you need to call from various routes. We can use middleware before and/or after the processed HTTP request. Typical examples of using middleware include authorization, validation, etc.

If middleware is used before the route is processed, any changes made by it will be available in the main request handler. This is convenient to use if we want to implement checking of certain queries. If middleware is used after the handler, it will receive a response from the route handler. This can be used to modify the response from the route handler.

In `gin`, the `Use` method is used to use middleware.
Use attaches a global middleware to the router. i.e. the middleware attached through Use() will be included in the handlers chain for every single request. Even 404, 405, static files… For example, this is the right place for a logger or error management middleware.

Let’s create the `httpmiddleware` package and implement `SetRequestIdHeader`, `LogRequest`.

Create a `httpmiddleware` package:
* Create a `httpmiddleware` directory inside the `internal` directory
* Create a `headers.go` file inside the `httpmiddleware` directory
```golang
package httpmiddleware

````
import (
    "github.com/gin-gonic/gin"
    "github.com/google/uuid"
)

func SetRequestIdHeader() gin.HandlerFunc {
    return func(c *gin.Context) {
        uuid := uuid.New().String()
        c.Set("uuid", uuid)
        c.Header("X-Request-Id", uuid)
    }
}
```
````

- Create a `logger.go` file inside the `httpmiddleware` directory
  ```golang
  package httpmiddleware

  import (
  “fmt”

  ```
  "github.com/gin-gonic/gin"
  ```

)

  func LogRequest() gin.HandlerFunc {
  return func(c *gin.Context) {
  uuid, _ := c.Get(“uuid”)
  fmt.Printf(“The request with uuid %s is started ”, uuid.(string))
  c.Next()
  fmt.Printf(“The request with uuid %s is finished ”, uuid.(string))
  }
  }
  ```

Let’s look at the `SetRequestIdHeader` function. Here, using the `Set` method, we will set the value “uuid”, which will be available in the context later using the `Get` method.
Set is used to store a new key/value pair exclusively for this context. It also lazy initializes c.Keys if it was not used previously.
We will also write the header in the response: `c.Header("X-Request-Id", uuid)`.
All actions in this case will be performed before the HTTP request is processed.

In the `LogRequest` function we log the start and end times of request processing. In it we will first get the `uuid` value that we defined above. Next, we register the start time of request processing.
Using the `Next` method we will wait for the request to complete processing.
And then we register the end time of processing the request.
`Next` should be used only inside middleware. It executes the pending handlers in the chain inside the calling handler..

Now all we have to do is add `SetRequestIdHeader` and `LogRequest` to `main.go` and check what happened using `curl`.

```go
func main() {
    router := gin.Default()

    router.Use(httpmiddleware.SetRequestIdHeader())
    router.Use(httpmiddleware.LogRequest())

    router.GET("/ping", func(c *gin.Context) {
        c.String(http.StatusOK, "pong")
    })

    base64.RegisterRoutes(router)
    unixtime.RegisterRoutes(router)

    srv := http.Server{
        Addr:     fmt.Sprintf(":%d", 3000),
        Handler: router,
    }
    srv.ListenAndServe()
}
```

To start the server, open a terminal (Tools > Terminal) and run:

```
cd gin-example
go run cmd/main.go
```

After the server has been started, you can check its functionality by opening a **second** terminal and running the command:

```
curl -v localhost:3000/unixtime/3523463425
```

Here we can notice that we have a header in the response: `X-Request-Id: 64bbde41-55e1-4a56-9fa4-7a62c61d4a8a`.

gRPC

gRPC is a high-performance open source framework developed by Google for remote procedure calls (RPC). HTTP/2 is used as transport. The interface description language is Protocol Buffers.
In gRPC, a client application can directly call a method on a server application on another machine as if it were a local object, making it easier to create distributed applications and services.
Widely used to create distributed systems (microservices) and APIs.
Supports load balancing, tracking, health checking and authentication. You can read more about gRPC on the website `https://grpc.io/`.
gRPC is asynchronous by default, meaning the thread does not block when a request arrives and can serve many requests in parallel.

There are four types of APIs in gRPC:
- Unary - the client makes a request, and the server sends a response.
- Server Streaming - the client makes a request, and the server sends a data stream.
- Client Streaming - the client sends a stream of data, and the server sends one response.
- Bi-Directional Streaming - the client and server send data streams.

Protocol buffers are an extensible, language- and platform-independent mechanism for serializing structured data.
Generates anchors in native language. You define how you want your data to be structured once, and then you can use custom generated source code to easily write and read your structured data to and from different data streams using different languages.

Write a web service

Let's write a simple web service to convert a unix timestamp into a human-readable string.

First, let's describe the interface for communicating with our service. Let's create a file `./proto/unixtime/unixtime_service.proto`:

```
syntax = "proto3";

option go_package = "grpc-example/proto/unixtime";

service Unixtime {
  rpc UnixToHuman(UnixToHumanRequest) returns
        (UnixToHumanResponse);
}

message UnixToHumanRequest {
  int64 timestamp = 1;
}

message UnixToHumanResponse {
  string humantime = 1;
}
```

Here we will indicate the protocol version, the package name and describe our Unixtime service. Let's add the method `rpc UnixToHuman(UnixToHumanRequest) returns (UnixToHumanResponse);`. Now let's describe the data structures of our `UnixToHumanRequest` and `UnixToHumanResponse` messages. To describe the data structure of a message, you need to add message, the name of the structure, and inside the type, name and number of the field. Field numbers are important for backward compatibility, so do not change them when adding or removing fields.
You can read more about protobuf on the website `https://protobuf.dev/`.

For further use, it is necessary to generate client and server code for the specific language used. In our case it's Go. gRPC uses protoc a special gRPC plugin to generate code from your proto file. In our example, we will use Makefile for this. If necessary, you will need to install the required plugins:

```
cd grpc-example
go mod init grpc-example
go install google.golang.org/protobuf/cmd/protoc-gen-go@latest
go install google.golang.org/grpc/cmd/protoc-gen-go-grpc@latest
export PATH=$PATH:$(go env GOPATH)/bin
```

gRPC server

Now, having described our interface for communicating with the service and generating server and client code, let's begin writing our gRPC server.

In the file `cmd/server/main.go` we will add the server start.

```go
package main

import (
    "grpc-example/internal/grpcserver"
    "log"
)

func main() {
    err := grpcserver.Start()
    if err != nil {
        log.Fatal(err)
    }
}
```

Next you need to implement the `Start` function. To do this, create a grpc server using `grpc.NewServer()`.
`NewServer` creates a gRPC server which has no service registered and has not started to accept requests yet.
Next, we will need to register the services we need.

```go
unixtimeSrv := &unixtimeGrpc.UnixtimeGrpcServer{}
unixtimePb.RegisterUnixtimeServer(s, unixtimeSrv)
```

Where `unixtimePb` is the generated package from our proto file, and `unixtimeGrpc` implements our endpoints.

In the file `internal/endpoints/unixtime/unixtime.go` we will create the structure `UnixtimeGrpcServer` and implement the functions of our service that were described in `.proto`.

```go
package unixtime

import (
    "context"

    "grpc-example/internal/unixtimeservice"
    pb "grpc-example/proto/generated-go/proto/unixtime"
)

type UnixtimeGrpcServer struct {
    pb.UnimplementedUnixtimeServer
}

func (s *UnixtimeGrpcServer) UnixToHuman(ctx context.Context,
        req *pb.UnixToHumanRequest) (*pb.UnixToHumanResponse,
        error) {
    humanTime := unixtimeservice.ToHuman(req.Timestamp)
    return &pb.UnixToHumanResponse{Humantime: humanTime}, nil
}
```

After we have described and registered our services, we will create a new `listener` and launch our server in the file `internal/endpoints/grpcserver/grpcserver.go`

```go
package grpcserver

import (
    "net"

    "google.golang.org/grpc"

    unixtimeGrpc "grpc-example/internal/endpoints/unixtime"
    unixtimePb "grpc-example/proto/generated-go/proto/unixtime"
)

func Start() error {
    s := grpc.NewServer()

    unixtimeSrv := &unixtimeGrpc.UnixtimeGrpcServer{}
    unixtimePb.RegisterUnixtimeServer(s, unixtimeSrv)

    listener, err := net.Listen("tcp", ":3030")
    if err != nil {
        return err
    }
    if err := s.Serve(listener); err != nil {
        return err
    }
    return nil
}
```

gRPC client

To test our server, let’s write a small client `cmd/client/main.go`:

```
package main

import (
    "context"
    "fmt"
    "log"
    "os"

    "google.golang.org/grpc"
    "google.golang.org/grpc/credentials/insecure"

    unixtimePb "grpc-example/proto/generated-go/proto/unixtime"
)

func main() {
    var timestamp int64
    fmt.Print("Input timestamp: ")
    _, err := fmt.Fscan(os.Stdin, &timestamp)
    if err != nil {
        log.Fatal(err)
    }

    dialOpts := []grpc.DialOption{}
    dialOpts = append(dialOpts,
grpc.WithTransportCredentials(insecure.NewCredentials()))
    dialOpts = append(dialOpts,
grpc.WithChainUnaryInterceptor(logInterceptor()))
    conn, err := grpc.Dial("127.0.0.1:3030", dialOpts...)
    if err != nil {
        log.Fatal(err)
    }
    defer conn.Close()
    client := unixtimePb.NewUnixtimeClient(conn)
    resp, err := client.UnixToHuman(context.Background(),
&unixtimePb.UnixToHumanRequest{Timestamp: timestamp})
    if err != nil {
        log.Fatal(err)
    }
    humanTime := resp.Humantime
    fmt.Println(humanTime)
}

func logInterceptor() func(ctx context.Context, method string,
req interface{}, reply interface{}, cc *grpc.ClientConn, invoker
grpc.UnaryInvoker, opts ...grpc.CallOption) error {
    return func(ctx context.Context, method string, req
interface{}, reply interface{}, cc *grpc.ClientConn, invoker
grpc.UnaryInvoker, opts ...grpc.CallOption) error {
        log.Printf("request method: %s", method)
        return invoker(ctx, method, req, reply, cc, opts...)
    }
}
```

First, we wait for the user to enter the timestamp that needs to be converted.

```
var timestamp int64
fmt.Print("Input timestamp: ")
_, err := fmt.Fscan(os.Stdin, &timestamp)
if err != nil {
    log.Fatal(err)
}
```

Then we will create a client connection by passing the address `localhost:3030` and options for setting up the connection.
In our case, we will add `grpc.WithTransportCredentials(insecure.NewCredentials()))` setting the security of the transport.

```
conn, err := grpc.Dial("localhost:3030",
grpc.WithTransportCredentials(insecure.NewCredentials()))
if err != nil {
    log.Fatal(err)
}
defer conn.Close()
```

If necessary, you can add a TLS certificate to establish a secure connection.

```
creds := credentials.NewTLS(&tls.Config{})
dialOpts := []grpc.DialOption{}
dialOpts = append(dialOpts,
grpc.WithTransportCredentials(creds))
```

Next, we will create a client `unixtimePb.NewUnixtimeClient(conn)` and call the `UnixToHuman` method, passing the request context and the request itself to it. Where `unixtimePb` is the package generated from the `.proto` file.

```
client := unixtimePb.NewUnixtimeClient(conn)
resp, err := client.UnixToHuman(context.Background(),
&unixtimePb.UnixToHumanRequest{Timestamp: timestamp})
if err != nil {
    log.Fatal(err)
}
humanTime := resp.Humantime
fmt.Println(humanTime)
```

Interceptors

When writing an http application, we used middleware to implement common logic before and after executing handlers. We typically use middleware to write components like authorization, logging, caching, etc. The same functionality can be implemented in gRPC using a concept called Interceptors. You can use interceptors on both the client and the server.

There are two types of interceptors: `UnaryInterceptor` and `StreamInterceptor` which handle unary and stream requests respectively.

Let’s look at the logging of requests on the server. First, let’s write a simple function that will log the request method.

```
func logInterceptor(ctx context.Context, req interface{}, info
*grpc.UnaryServerInfo, handler grpc.UnaryHandler) (response
interface{}, err error) {
    log.Printf("request method: %s", info.FullMethod)
    return handler(ctx, req)
}
```

Then we will add our interceptor when creating the grpc server.

```
s := grpc.NewServer(
    grpc.ChainUnaryInterceptor(logInterceptor),
)
```

Let’s add logging on the client in a similar way.

```
func logInterceptor() func(ctx context.Context, method string,
req interface{}, reply interface{}, cc *grpc.ClientConn, invoker
grpc.UnaryInvoker, opts ...grpc.CallOption) error {
    return func(ctx context.Context, method string, req
interface{}, reply interface{}, cc *grpc.ClientConn, invoker
grpc.UnaryInvoker, opts ...grpc.CallOption) error {
        log.Printf("request method: %s", method)
        return invoker(ctx, method, req, reply, cc, opts...)
    }
}
```

and...

```
dialOpts = append(dialOpts,
grpc.WithChainUnaryInterceptor(logInterceptor()))
```

Now we are ready to test our application using `make client` after running the `make server`.