Tutorial: Developing a RESTful API with Go and Gin

Table of Contents

This tutorial introduces the basics of writing a RESTful web service API with Go and the Gin Web Framework (Gin).

You'll get the most out of this tutorial if you have a basic familiarity with Go and its tooling. If this is your first exposure to Go, please see Tutorial: Get started with Go for a quick introduction.

Gin simplifies many coding tasks associated with building web applications, including web services. In this tutorial, you'll use Gin to route requests, retrieve request details, and marshal JSON for responses.

In this tutorial, you will build a RESTful API server with two endpoints. Your example project will be a repository of data about vintage jazz records.

The tutorial includes the following sections:

1. Design API endpoints.
2. Create a folder for your code.
3. Create the data.
4. Write a handler to return all items.
5. Write a handler to add a new item.
6. Write a handler to return a specific item.

Note: For other tutorials, see Tutorials.

To try this as an interactive tutorial you complete in Google Cloud Shell, click the button below.

Prerequisites

- **An installation of Go 1.16 or later.** For installation instructions, see Installing Go.
- **A tool to edit your code.** Any text editor you have will work fine.
- **A command terminal.** Go works well using any terminal on Linux and Mac, and on PowerShell or cmd in Windows.
- **The curl tool.** On Linux and Mac, this should already be installed. On Windows, it's included on Windows 10 Insider build 17063 and later. For earlier Windows versions, you might need to install it. For more, see Tar and Curl Come to Windows.

Design API endpoints

You'll build an API that provides access to a store selling vintage recordings on vinyl. So you'll need to provide endpoints through which a client can get and add albums for users.

When developing an API, you typically begin by designing the endpoints. Your API's users will have more success if the endpoints are easy to understand.

Here are the endpoints you'll create in this tutorial.

/albums

- GET – Get a list of all albums, returned as JSON.
- POST – Add a new album from request data sent as JSON.

/albums/:id

- GET – Get an album by its ID, returning the album data as JSON.

Next, you'll create a folder for your code.

Create a folder for your code

To begin, create a project for the code you'll write.

1. Open a command prompt and change to your home directory.

   On Linux or Mac:

   ```
   $ cd
   ```

   On Windows:

```
C:\> cd %HOMEPATH%
```

2. Using the command prompt, create a directory for your code called web-service-gin.

```
$ mkdir web-service-gin
$ cd web-service-gin
```

3. Create a module in which you can manage dependencies.

   Run the `go mod init` command, giving it the path of the module your code will be in.

```
$ go mod init example/web-service-gin
go: creating new go.mod: module example/web-service-gin
```

   This command creates a go.mod file in which dependencies you add will be listed for tracking. For more about naming a module with a module path, see Managing dependencies.

Next, you'll design data structures for handling data.

Create the data

To keep things simple for the tutorial, you'll store data in memory. A more typical API would interact with a database.

Note that storing data in memory means that the set of albums will be lost each time you stop the server, then recreated when you start it.

Write the code

1. Using your text editor, create a file called main.go in the web-service directory. You'll write your Go code in this file.

2. Into main.go, at the top of the file, paste the following package declaration.

```
package main
```

   A standalone program (as opposed to a library) is always in package `main`.

3. Beneath the package declaration, paste the following declaration of an `album` struct. You'll use this to store album data in memory.

   Struct tags such as `json:"artist"` specify what a field's name should be when the struct's contents are serialized into JSON. Without them, the JSON would use the struct's capitalized field names – a style not as common in JSON.

```
// album represents data about a record album.
type album struct {
    ID     string  `json:"id"`
    Title  string  `json:"title"`
    Artist string  `json:"artist"`
    Price  float64 `json:"price"`
}
```

4. Beneath the struct declaration you just added, paste the following slice of `album` structs containing data you'll use to start.

```
// albums slice to seed record album data.
var albums = []album{
    {ID: "1", Title: "Blue Train", Artist: "John Coltrane", Price: 56.99},
    {ID: "2", Title: "Jeru", Artist: "Gerry Mulligan", Price: 17.99},
    {ID: "3", Title: "Sarah Vaughan and Clifford Brown", Artist: "Sarah Vaughar
}
```

Next, you'll write code to implement your first endpoint.

Write a handler to return all items

When the client makes a request at GET `/albums`, you want to return all the albums as JSON.

To do this, you'll write the following:

- Logic to prepare a response
- Code to map the request path to your logic

Note that this is the reverse of how they'll be executed at runtime, but you're adding dependencies first, then the code that depends on them.

Write the code

1. Beneath the struct code you added in the preceding section, paste the following code to get the album list.

   This `getAlbums` function creates JSON from the slice of `album` structs, writing the JSON into the response.

```
// getAlbums responds with the list of all albums as JSON.
func getAlbums(c *gin.Context) {
    c.IndentedJSON(http.StatusOK, albums)
}
```

   In this code, you:

- Write a `getAlbums` function that takes a `gin.Context` parameter. Note that you could have given this function any name – neither Gin nor Go require a particular function name format.

  `gin.Context` is the most important part of Gin. It carries request details, validates and serializes JSON, and more. (Despite the similar name, this is different from Go's built-in `context` package.)

- Call `Context.IndentedJSON` to serialize the struct into JSON and add it to the response.

  The function's first argument is the HTTP status code you want to send to the client. Here, you're passing the `StatusOK` constant from the `net/http` package to indicate `200 OK`.

  Note that you can replace `Context.IndentedJSON` with a call to `Context.JSON` to send more compact JSON. In practice, the indented form is much easier to work with when debugging and the size difference is usually small.

2. Near the top of main.go, just beneath the `albums` slice declaration, paste the code below to assign the handler function to an endpoint path.

   This sets up an association in which `getAlbums` handles requests to the `/albums` endpoint path.

   ```
   func main() {
       router := gin.Default()
       router.GET("/albums", getAlbums)

       router.Run("localhost:8080")
   }
   ```

   In this code, you:

   - Initialize a Gin router using `Default`.
   - Use the `GET` function to associate the GET HTTP method and `/albums` path with a handler function.

     Note that you're passing the *name* of the `getAlbums` function. This is different from passing the *result* of the function, which you would do by passing `getAlbums()` (note the parenthesis).

   - Use the `Run` function to attach the router to an `http.Server` and start the server.

3. Near the top of main.go, just beneath the package declaration, import the packages you'll need to support the code you've just written.

The first lines of code should look like this:

```
package main

import (
    "net/http"

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

4. Save main.go.

Run the code

1. Begin tracking the Gin module as a dependency.

   At the command line, use `go get` to add the github.com/gin-gonic/gin module as a dependency for your module. Use a dot argument to mean "get dependencies for code in the current directory."

   ```
   $ go get .
   go get: added github.com/gin-gonic/gin v1.7.2
   ```

   Go resolved and downloaded this dependency to satisfy the `import` declaration you added in the previous step.

2. From the command line in the directory containing main.go, run the code. Use a dot argument to mean "run code in the current directory."

   ```
   $ go run .
   ```

   Once the code is running, you have a running HTTP server to which you can send requests.

3. From a new command line window, use `curl` to make a request to your running web service.

   ```
   $ curl http://localhost:8080/albums
   ```

   The command should display the data you seeded the service with.

   ```
   [
           {
                   "id": "1",
                   "title": "Blue Train",
                   "artist": "John Coltrane",
                   "price": 56.99
   ```

```
        },
        {
                "id": "2",
                "title": "Jeru",
                "artist": "Gerry Mulligan",
                "price": 17.99
        },
        {
                "id": "3",
                "title": "Sarah Vaughan and Clifford Brown",
                "artist": "Sarah Vaughan",
                "price": 39.99
        }
]
```

You've started an API! In the next section, you'll create another endpoint with code to handle a `POST` request to add an item.

Write a handler to add a new item

When the client makes a `POST` request at `/albums`, you want to add the album described in the request body to the existing albums' data.

To do this, you'll write the following:

- Logic to add the new album to the existing list.
- A bit of code to route the `POST` request to your logic.

Write the code

1. Add code to add albums data to the list of albums.

   Somewhere after the `import` statements, paste the following code. (The end of the file is a good place for this code, but Go doesn't enforce the order in which you declare functions.)

```
// postAlbums adds an album from JSON received in the request body.
func postAlbums(c *gin.Context) {
    var newAlbum album

    // Call BindJSON to bind the received JSON to
    // newAlbum.
    if err := c.BindJSON(&newAlbum); err != nil {
        return
    }

    // Add the new album to the slice.
    albums = append(albums, newAlbum)
```

```
    c.IndentedJSON(http.StatusCreated, newAlbum)
}
```

In this code, you:

- Use `Context.BindJSON` to bind the request body to `newAlbum`.
- Append the `album` struct initialized from the JSON to the `albums` slice.
- Add a `201` status code to the response, along with JSON representing the album you added.

2. Change your `main` function so that it includes the `router.POST` function, as in the following.

```
func main() {
    router := gin.Default()
    router.GET("/albums", getAlbums)
    router.POST("/albums", postAlbums)

    router.Run("localhost:8080")
}
```

In this code, you:

- Associate the `POST` method at the `/albums` path with the `postAlbums` function.

  With Gin, you can associate a handler with an HTTP method-and-path combination. In this way, you can separately route requests sent to a single path based on the method the client is using.

Run the code

1. If the server is still running from the last section, stop it.
2. From the command line in the directory containing main.go, run the code.

```
$ go run .
```

3. From a different command line window, use `curl` to make a request to your running web service.

```
$ curl http://localhost:8080/albums \
    --include \
    --header "Content-Type: application/json" \
    --request "POST" \
    --data '{"id": "4","title": "The Modern Sound of Betty Carter","artist": "
```

The command should display headers and JSON for the added album.

```
HTTP/1.1 201 Created
Content-Type: application/json; charset=utf-8
Date: Wed, 02 Jun 2021 00:34:12 GMT
Content-Length: 116

{
    "id": "4",
    "title": "The Modern Sound of Betty Carter",
    "artist": "Betty Carter",
    "price": 49.99
}
```

4. As in the previous section, use `curl` to retrieve the full list of albums, which you can use to confirm that the new album was added.

```
$ curl http://localhost:8080/albums \
    --header "Content-Type: application/json" \
    --request "GET"
```

The command should display the album list.

```
[
        {
                "id": "1",
                "title": "Blue Train",
                "artist": "John Coltrane",
                "price": 56.99
        },
        {
                "id": "2",
                "title": "Jeru",
                "artist": "Gerry Mulligan",
                "price": 17.99
        },
        {
                "id": "3",
                "title": "Sarah Vaughan and Clifford Brown",
                "artist": "Sarah Vaughan",
                "price": 39.99
        },
        {
                "id": "4",
                "title": "The Modern Sound of Betty Carter",
                "artist": "Betty Carter",
                "price": 49.99
        }
]
```

In the next section, you'll add code to handle a GET for a specific item.

Write a handler to return a specific item

When the client makes a request to GET `/albums/[id]`, you want to return the album whose ID matches the `id` path parameter.

To do this, you will:

- Add logic to retrieve the requested album.
- Map the path to the logic.

Write the code

1. Beneath the `postAlbums` function you added in the preceding section, paste the following code to retrieve a specific album.

   This `getAlbumByID` function will extract the ID in the request path, then locate an album that matches.

```
// getAlbumByID locates the album whose ID value matches the id
// parameter sent by the client, then returns that album as a response.
func getAlbumByID(c *gin.Context) {
    id := c.Param("id")

    // Loop over the list of albums, looking for
    // an album whose ID value matches the parameter.
    for _, a := range albums {
        if a.ID == id {
            c.IndentedJSON(http.StatusOK, a)
            return
        }
    }
    c.IndentedJSON(http.StatusNotFound, gin.H{"message": "album not found"})
}
```

   In this code, you:

   - Use `Context.Param` to retrieve the `id` path parameter from the URL. When you map this handler to a path, you'll include a placeholder for the parameter in the path.
   - Loop over the `album` structs in the slice, looking for one whose `ID` field value matches the `id` parameter value. If it's found, you serialize that `album` struct to JSON and return it as a response with a `200 OK` HTTP code.

As mentioned above, a real-world service would likely use a database query to perform this lookup.

- Return an HTTP `404` error with `http.StatusNotFound` if the album isn't found.

2. Finally, change your `main` so that it includes a new call to `router.GET`, where the path is now `/albums/:id`, as shown in the following example.

```
func main() {
    router := gin.Default()
    router.GET("/albums", getAlbums)
    router.GET("/albums/:id", getAlbumByID)
    router.POST("/albums", postAlbums)

    router.Run("localhost:8080")
}
```

In this code, you:

- Associate the `/albums/:id` path with the `getAlbumByID` function. In Gin, the colon preceding an item in the path signifies that the item is a path parameter.

Run the code

1. If the server is still running from the last section, stop it.

2. From the command line in the directory containing main.go, run the code to start the server.

```
$ go run .
```

3. From a different command line window, use `curl` to make a request to your running web service.

```
$ curl http://localhost:8080/albums/2
```

The command should display JSON for the album whose ID you used. If the album wasn't found, you'll get JSON with an error message.

```
{
        "id": "2",
        "title": "Jeru",
        "artist": "Gerry Mulligan",
        "price": 17.99
}
```

Conclusion

Congratulations! You've just used Go and Gin to write a simple RESTful web service.

Suggested next topics:

- If you're new to Go, you'll find useful best practices described in Effective Go and How to write Go code.
- The Go Tour is a great step-by-step introduction to Go fundamentals.
- For more about Gin, see the Gin Web Framework package documentation or the Gin Web Framework docs.

Completed code

This section contains the code for the application you build with this tutorial.

```
package main

import (
    "net/http"

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

// album represents data about a record album.
type album struct {
    ID     string  `json:"id"`
    Title  string  `json:"title"`
    Artist string  `json:"artist"`
    Price  float64 `json:"price"`
}

// albums slice to seed record album data.
var albums = []album{
    {ID: "1", Title: "Blue Train", Artist: "John Coltrane", Price: 56.99},
    {ID: "2", Title: "Jeru", Artist: "Gerry Mulligan", Price: 17.99},
    {ID: "3", Title: "Sarah Vaughan and Clifford Brown", Artist: "Sarah Vaughan", P
}

func main() {
    router := gin.Default()
    router.GET("/albums", getAlbums)
    router.GET("/albums/:id", getAlbumByID)
    router.POST("/albums", postAlbums)

    router.Run("localhost:8080")
}

// getAlbums responds with the list of all albums as JSON.
func getAlbums(c *gin.Context) {
    c.IndentedJSON(http.StatusOK, albums)
}
```

```
// postAlbums adds an album from JSON received in the request body.
func postAlbums(c *gin.Context) {
    var newAlbum album

    // Call BindJSON to bind the received JSON to
    // newAlbum.
    if err := c.BindJSON(&newAlbum); err != nil {
        return
    }

    // Add the new album to the slice.
    albums = append(albums, newAlbum)
    c.IndentedJSON(http.StatusCreated, newAlbum)
}

// getAlbumByID locates the album whose ID value matches the id
// parameter sent by the client, then returns that album as a response.
func getAlbumByID(c *gin.Context) {
    id := c.Param("id")

    // Loop through the list of albums, looking for
    // an album whose ID value matches the parameter.
    for _, a := range albums {
        if a.ID == id {
            c.IndentedJSON(http.StatusOK, a)
            return
        }
    }
    c.IndentedJSON(http.StatusNotFound, gin.H{"message": "album not found"})
```