20-Day Go (Golang) Course: API, Web, & Deployment

Part 1: Go Fundamentals (Days 1-5)

Day 1: Introduction to Go

- **What is Go?**
  - Go, also known as Golang, is a statically typed, compiled programming language designed at Google. It is known for its simplicity, efficiency, and strong support for concurrent programming.
- **Why Learn Go?**
  - High demand in cloud-native development, DevOps, and backend services.
  - Excellent performance.
  - Simple and clean syntax.
- **Setting up the Go Development Environment**
  - **Installation:** Download and install the Go toolchain from the official website (https://golang.org/).
  - **GOPATH and Go Modules:** Understand the workspace and dependency management.
- **Basic Go Syntax**
  - Packages and imports.
  - Functions (`func`).
  - Variables (`var`, `:=`).
  - Basic data types (int, float64, string, bool).

Tasks:

1. Install Go on your machine.
2. Write and run the "Hello, World!" program.
3. Declare variables of different data types and print them.
4. Write a program that takes your name as input and prints a greeting.
5. Explore the official Go documentation.

Day 2: Control Structures and Data Structures

- **Control Structures**
  - `if-else` statements.
  - `for` loops (Go's only looping construct).
  - `switch` statements.
- **Data Structures**
  - **Arrays:** Fixed-size collections of items of the same type.
  - **Slices:** Dynamically-sized, flexible view into the elements of an array.
  - **Maps:** Key-value pairs.
  - **Structs:** Collections of fields.

Tasks:

1. Write a program to find the largest of three numbers.
2. Create a slice of integers and iterate over it to print each element.
3. Implement a simple "guess the number" game.
4. Define a `Person` struct with `Name` and `Age` fields.
5. Create a map to store the prices of different fruits.

Day 3: Functions and Pointers

- **Functions**
  - Defining and calling functions.
  - Multiple return values.
  - Variadic functions.
- **Pointers**
  - What are pointers?
  - The `&` (address of) and `*` (dereference) operators.
  - When to use pointers.

Tasks:

1. Write a function that takes two integers and returns their sum and difference.
2. Create a function that swaps the values of two variables using pointers.
3. Write a function that takes a slice of strings and returns a single concatenated string.
4. Implement a function that calculates the factorial of a number.
5. Refactor your `Person` struct to include a method that prints the person's details.

Day 4: Interfaces and Error Handling

- **Interfaces**
  - Defining and implementing interfaces.
  - The empty interface (`interface{}`).
- **Error Handling**
  - The `error` type.
  - Returning and checking for errors.

Tasks:

1. Define a `Shape` interface with an `Area()` method.
2. Implement the `Shape` interface for `Rectangle` and `Circle` structs.
3. Write a function that can calculate the area of any `Shape`.
4. Create a function that might return an error (e.g., division by zero) and handle it.
5. Read about the `fmt.Errorf` function.

Day 5: Concurrency in Go

- **Goroutines**
  - Lightweight threads managed by the Go runtime.
  - The `go` keyword.
- **Channels**
  - Typed conduits for communication between goroutines.
  - Sending and receiving data.

Tasks:

1. Write a program that launches two goroutines to print numbers from 1 to 5 and 6 to 10 concurrently.
2. Use a channel to send a message from one goroutine to another.
3. Implement a simple worker pool.
4. Explore the `sync` package, particularly `WaitGroup`.
5. Write a program that fetches data from two different URLs concurrently.

Part 2: Web Development with Go (Days 6-9)

Day 6: Building a Simple Web Server

- **The `net/http` Package**
  - `http.HandleFunc` and `http.ListenAndServe`.
  - Handling requests and responses.

Tasks:

1. Create a web server that responds with "Welcome to my website!"
2. Add a new route `/about` that provides information about the site.
3. Serve a simple HTML file.
4. Handle different HTTP methods (GET, POST).
5. Read about the `http.Request` and `http.ResponseWriter` types.

Day 7: Web Development with Gin Framework

- **Introduction to Gin**
  - A fast, feature-rich web framework for Go.
  - Installing and setting up a basic Gin server.
- **Routing in Gin**
  - Handling GET, POST, PUT, DELETE requests.
  - Route parameters and query strings.
  - Grouping routes.

Tasks:

1. Convert your simple `net/http` server to use Gin.
2. Create a route `/users/:id` that retrieves a user by their ID from a map or slice.
3. Implement an endpoint that accepts query parameters (e.g., `/search?q=golang`).
4. Group your API routes under an `/api/v1` prefix.
5. Explore Gin's documentation for more advanced routing features.

Day 8: Working with JSON and Data Binding

- **Handling JSON in Gin**
    - Gin provides simple methods like `c.JSON()` for sending JSON responses.
- **Data Binding**
    - Binding request payloads (JSON, form data) to Go structs.
    - Validation of incoming data.

Tasks:

1. Create an API endpoint that returns a list of users in JSON format using Gin.
2. Implement an endpoint that accepts a JSON payload to create a new user and binds it to a struct.
3. Add validation tags to your user struct (e.g., `binding:"required"`).
4. Handle binding errors and return appropriate error messages.
5. Explore binding for different content types, like form data.

Day 9: GORM and GORM Gen

- **Introduction to GORM**
    - A fantastic ORM library for Go.
    - Connecting to a PostgreSQL database.
- **CRUD Operations with GORM**
    - Defining models, creating tables, and performing Create, Read, Update, Delete operations.
- **Automating with GORM Gen**
    - Using GORM Gen to generate type-safe DAO code from your database schema.

Tasks:

1. Set up a connection to a PostgreSQL database using GORM in your Gin application.
2. Define a `User` model and use GORM's auto-migration to create the table.
3. Implement API endpoints for all CRUD operations on users, connecting your Gin handlers to GORM functions.
4. Install GORM Gen and generate the DAO code for your `User` model.
5. Refactor your API to use the type-safe methods provided by GORM Gen.

Part 3: Advanced Topics (Days 10-15)

Day 10: Real-time Communication with Gorilla WebSocket

- **Introduction to WebSockets**
  - Bidirectional communication between client and server.
- **Using Gorilla WebSocket with Gin**
  - Upgrading HTTP connections to WebSockets within a Gin handler.
  - Sending and receiving messages.

Tasks:

1. Create a simple chat application using WebSockets in your Gin app.
2. Broadcast messages to all connected clients.
3. Handle client connections and disconnections gracefully.
4. Implement a "user is typing" feature.
5. Explore WebSocket security considerations.

Day 11: Middleware in Gin

- **What is Middleware?**
  - Functions that execute before or after a request handler.
- **Using Gin's Middleware**
  - Default middleware (Logger, Recovery).
  - Writing custom middleware for logging, authentication, CORS, etc.

Tasks:

1. Analyze the output of Gin's default `Logger()` middleware.
2. Write a custom logging middleware that logs requests in a specific format.
3. Create a middleware for authenticating users with a simple static API key in the header.
4. Apply your auth middleware to a specific group of routes.
5. Implement a CORS middleware from scratch or use a third-party library.

Day 12: Authentication and Authorization

- **JWT (JSON Web Tokens)**
  - Creating and validating JWTs.
- **Implementing Authentication**
  - Login and registration endpoints that issue JWTs.
  - Protecting routes with JWT authentication middleware.

Tasks:

1. Implement user registration and login endpoints that return a JWT upon success.
2. Create a Gin middleware that extracts and validates the JWT from the `Authorization` header.
3. Protect your CRUD API routes so only authenticated users can access them.
4. Implement a "logout" feature (e.g., by managing a token blacklist in Redis or a database).
5. Read about different JWT libraries for Go (e.g., `golang-jwt/jwt`).

Day 13: Testing in Go

- **The `testing` Package**
  - Writing unit tests for business logic.
  - Table-driven tests for comprehensive checks.
- **Testing Gin Handlers**
  - The `net/http/httptest` package for creating mock requests.

Tasks:

1. Write unit tests for any utility or helper functions in your project.
2. Implement table-driven tests for a function with multiple inputs and expected outputs.
3. Write tests for your Gin API endpoints, checking status codes and response bodies.
4. Calculate the test coverage for your code using `go test -cover`.
5. Explore mocking your GORM database layer for more isolated handler tests.

Day 14: Project Structure and Best Practices

- **Organizing Your Go Project**
  - Common project layouts (e.g., separating handlers, models, services).
- **Go Best Practices**
  - Reading "Effective Go".
  - Using tools like `go fmt` and `go vet`.
  - Setting up a linter (`golangci-lint`).

Tasks:

1. Restructure your project into a more scalable layout (e.g., `cmd`, `internal`, `pkg`).
2. Set up `golangci-lint` for your project and fix any reported issues.
3. Read and understand the key principles of "Effective Go."
4. Refactor your code to follow best practices for error handling and clarity.
5. Add comments and documentation to your code using Go's doc comment format.

Day 15: CI/CD with Go

- **Continuous Integration and Continuous Deployment (CI/CD)**
  - Automating the build, test, and deployment process.
- **Using GitHub Actions**
  - Creating a simple workflow to build and test your Go application on every push.

Tasks:

1. Create a GitHub repository for your project if you haven't already.
2. Set up a GitHub Actions workflow that runs `go build` and `go test` automatically.
3. Add a step to your workflow to run your linter.
4. Explore how to build a Docker image within your CI pipeline.
5. Read about managing secrets (like database passwords) in GitHub Actions.

Part 4: Deployment and DevOps (Days 16-20)

Day 16: Introduction to Docker

- **What is Docker?**
  - Containers and virtualization.
- **Basic Docker Commands**
  - `docker run`, `docker build`, `docker ps`, `docker images`.

Tasks:

1. Install Docker on your machine.
2. Run a "hello-world" container to verify your installation.
3. Pull and run the official PostgreSQL Docker image.
4. Explore the Docker Hub for other useful images.
5. Learn the difference between a Docker image and a container.

Day 17: Dockerizing a Go Application

- **Creating a `Dockerfile` for Go**
  - Instructions for building a Go binary in a container.
  - Using multi-stage builds to create small, secure production images.
- **Building and Running the Go App in a Container**

Tasks:

1. Write a `Dockerfile` for your Gin API.
2. Implement a multi-stage build to compile the app and copy the binary to a minimal base image.
3. Build a Docker image of your application.
4. Run your Go API inside a Docker container.
5. Learn how to pass environment variables to your container at runtime.

Day 18: Docker Compose

- **Managing Multi-container Applications**
  - The `docker-compose.yml` file.
- **Orchestrating Your Go App and Database**
  - Defining services, networks, and volumes.

Tasks:

1. Write a `docker-compose.yml` file to define your Go API and PostgreSQL database as services.
2. Use `docker-compose up` to start your entire application stack.
3. Configure networking so your Go container can communicate with the database container.
4. Add a new service to your `docker-compose.yml` (e.g., a Redis cache).
5. Learn how to use a `.env` file with Docker Compose for configuration.

Day 19: Deploying to a Cloud Provider

- **Introduction to Cloud Deployment**
  - Overview of major cloud providers (AWS, Google Cloud, Azure).
- **Deploying a Dockerized App**
  - Using a service like AWS Elastic Beanstalk, Google Cloud Run, or DigitalOcean App Platform.

Tasks:

1. Create a free-tier account with a cloud provider.
2. Deploy your Dockerized application to the cloud using one of their container services.
3. Configure production environment variables (database credentials, JWT secret).
4. Expose your application to the internet and test the live endpoints.
5. Explore the scaling and monitoring features of the cloud platform.

Day 20: Final Project

- **Putting It All Together**
  - Build a complete web application or API from scratch, incorporating all the concepts learned in the course.
- **Project Ideas**
  - A simple e-commerce API.
  - A real-time polling application.
  - A URL shortener service.
  - A microblogging platform API.

Tasks:

1. Brainstorm and choose a final project idea.
2. Plan the API endpoints, database schema, and overall architecture.
3. Develop, test, and document your application.
4. Dockerize the application and create a `docker-compose.yml` file.
5. Deploy your final project to the cloud and share the link!