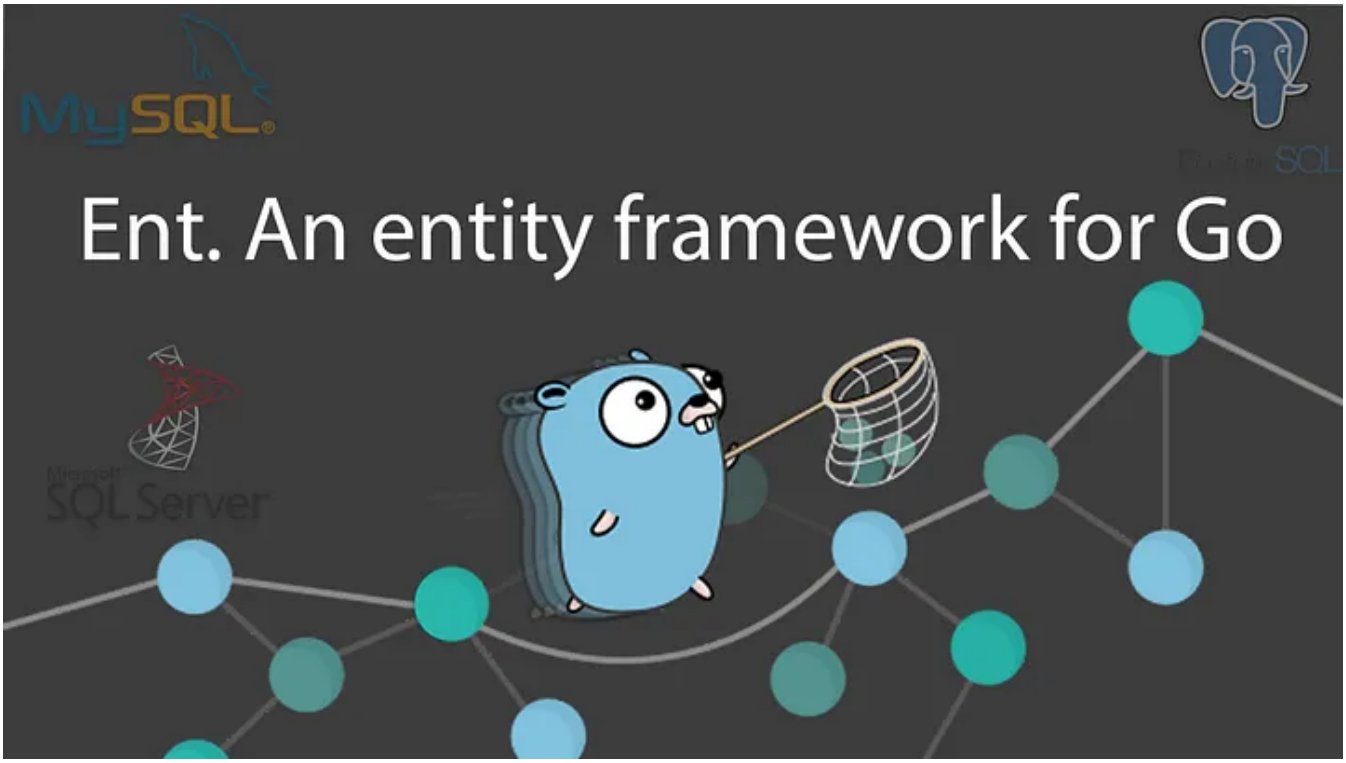
Published in Level Up Coding

Tharaka Romesh Follow
Aug 2, 2022 · 9 min read · ✨ · Listen

Let's "Go" and build an application with Ent



Golang is a language built by Google with the main focus on performance and concurrency. In recent years Go has become one of the most loved and wanted programming languages among developers. Golang is particularly suited for developing infrastructures like networked servers or even microservices. Even

though Golang had many excellent features and tools built with it, there are only a few tools available in Go that handle the data layer properly, like an ORM.

What is an ORM

> “Technique for converting data between incompatible type systems using object-oriented programming languages”

Object Relational Mapper(Mapping) is a method of writing SQL queries for a relational database using the object-oriented paradigm of your preferred programming language. An ORM will act as an interface that will wrap your tables or stored procedures in classes so you can access them using methods and properties of objects instead of writing SQL queries.

But in Go, most ORM libraries cannot handle the features below.

- Relationship support
- Prefetching
- Multicreate
- Composable Queries

The most widely used ORM in Go lang, GORM, can handle the above tasks, but there are several shortcomings in GORM, like **performance** that will hurt your application. With these issues in mind, the Meta developers have developed an ORM that can easily define any data model or graph structure in Go code called **Ent**.

What is Ent?

Ent is an ORM(Object Relational Mapping) framework built by Meta Open Source, which provides an API for modeling any database schema as Go objects. With Ent, you can run queries and aggregations and traverse graph structures. Ent supports major databases MySQL, MariaDB, PostgreSQL, SQLite, and Gremlin-based graph databases(Azure Cosmos DB). All you need to do with Ent is define the Schema for your application, and Ent will handle the rest for you. The Schema you specify will

be validated by the Ent codegen(entc), which will generate a well-typed and idiomatic API.

Why ENT is better

There are many tools in Golang like **go-pg**, **sqlx**, **sqlc**, **sql-migrate**, and **sqlboiler** which can generate type-safe code that will map the application's primitives to the database tables with struct and methods. But these tools are not a complete solution, so you will have to depend on each tool to do its part, like generating code and handling migrations when building your application. With Ent, you can have a complete framework that enables all related tasks. Ent also provides

- Statically typed and explicit API
- Queries, aggregations, and graph traversals
- Support for `context.Context`
- Enables Caching through `entcache`
- The OpenAPI Specification (OAS, formerly known as Swagger Specification) generation through `entoas`

You can see why Ent is better than other tools and ORM with these options or features. Next, let's dive into the Concepts and API in Ent.

Concepts and API of Ent

Before working with Ent, you must grasp a few concepts/keywords.

- **Schema**
  A schema defines one entity type in the graph
- **Fields**
  represent its properties
- **Edges**
  Edges represent relationships (one-to-one, one-to-many, many-to-many) in Ent.
- **Mixin**
  Mixin is an interface that allows you to create reusable pieces of a schema that

Mixin can inject into another schema. A Mixin can be a set of Feilds, Edges, or Hooks.

- **Annotations**
  Annotations allow us to add additional metadata to our schema objects like Edges and Fields.

- **Privacy**
  One of Ent's best features is the Privacy option, which defines the privacy policy for queries and mutations of entities in the database. When you define a policy for a schema, it will always be evaluated whenever queries and mutations are performed on it.

To create policies, you will have to extend the class ***ent.Policy*** holds two methods, ***EvalQuery*** and ***EvalMutation,*** responsible for read-policy and write-policy. A policy can have any number of rules defined by the user, and rules will evaluate them in the same order declared in the Schema.

Building an application with Ent

Let's move on to building an application with Ent. Here we will be building a small pokemon application with Fiber, an Express-inspired web framework written in Go.

Prerequisites

Before building the application, you must have the go 1.17 or the latest version installed and configured. We will be using MySQL as the database for this example so make sure you have a running instance of MySQL (latest of 5.6 and above).

> *You can use MySQL via Docker through this image.*

Installing Ent package

Now let's install the necessary libraries required to build our application. First, let's set up a Go module project.

```
go mod init github.com/<username>/go-ent-pokemon
```

Now let's install the necessary libraries. We will install several packages like

- **Ent**
  Since we are working with MySQL, We will also have to install the go driver for MySQL.

```
go get -d entgo.io/ent/cmd/ent
go get github.com/go-sql-driver/mysql
```

- **Viper**
  We will install Viper, a Go configuration manager that supports JSON, TOML, YAML, HCL, env, and other configuration file formats. We will need this to store the configuration for the database.

```
go get github.com/spf13/viper
```

- **Fiber**
  Fiber is a web framework built on top of Fasthttp, the **fastest** HTTP engine for Go. We will create a REST API that performs CRUD operations with Ent.

```
go get github.com/gofiber/fiber/v2
```

Creating a schema

Now that we have installed all the necessary packages Let's start creating the schemas with Ent. Let's start with creating the pokemon Schema. We will be using the Ent CLI to generate the schema files.

```
go run entgo.io/ent/cmd/ent init Pokemon
```

This will generate a schema file under the directory *ent/schema/pokemon.go,* which will look like this.

```
package schema
```

```
import (
 "time"

"entgo.io/ent"
 "entgo.io/ent/schema/edge"
 "entgo.io/ent/schema/field"
)

// Pokemon holds the schema definition for the Pokemon entity.
type Pokemon struct {
 ent.Schema
}

// Fields of the Pokemon.
func (Pokemon) Fields() []ent.Field {
 return []ent.Field{
  field.Int("id").
   StructTag(`json:"oid,omitempty"`),
  field.Text("name").
   NotEmpty(),
  field.Text("description").
   NotEmpty(),
  field.Float("weight"),
  field.Float("height"),
  field.Time("created_at").
   Default(time.Now).
   Immutable(),
  field.Time("updated_at").
   Default(time.Now),
 }
}

// Edges of the Pokemon.
func (Pokemon) Edges() []ent.Edge {
 return []ent.Edge{
  edge.To("fights", Battle.Type),
  edge.To("opponents", Battle.Type),
 }
}
```

Notice that an Edge is defined in the which refers to a Battle.
Now let's create another schema called Battle using the CLI command.

```
go run entgo.io/ent/cmd/ent init Battle
```

Now let's define the Fields and Edges for the Battle schema like below.

```
package schema
```

```
import (
 "time"

"entgo.io/ent"
 "entgo.io/ent/schema/edge"
 "entgo.io/ent/schema/field"
)

// Battle holds the schema definition for the Battle entity.
type Battle struct {
 ent.Schema
}

// Fields of the Battle.
func (Battle) Fields() []ent.Field {
 return []ent.Field{
  field.Int("id").
   StructTag(`json:"oid,omitempty"`),
  field.Text("result"),
  field.Time("created_at").
   Default(time.Now).
   Immutable(),
  field.Time("updated_at").
   Default(time.Now),
 }
}

// Edges of the Battle.
func (Battle) Edges() []ent.Edge {
 return []ent.Edge{
  edge.From("contender", Pokemon.Type).
   Ref("fights").
   Unique(),
  edge.From("oponent", Pokemon.Type).
   Ref("opponents").
   Unique(),
 }
}
```

Now that we have defined the schemas, We can run the following command from the root of our project.

```
go generate ./ent
```

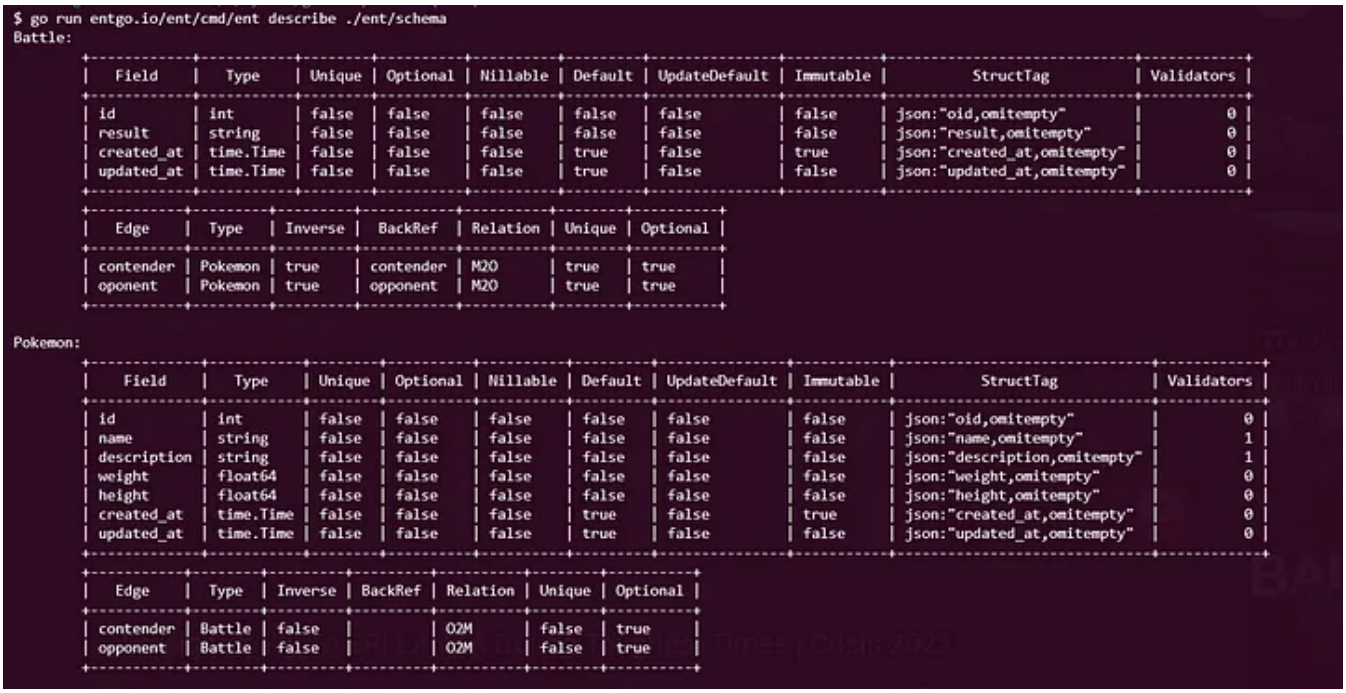
This will generate the necessary structs and other methods to help us consume or query the database.

Visualizing a schema

The entc allows you to visualize your schemas through the flag "**describe**", which will list down the schemas along with their properties. All you need to do is to run the following command.

```
go run entgo.io/ent/cmd/ent describe ./ent/schema
```

The above command will generate the following output for you.



Schema Visualization

The above table will visually represent your schemas with all their Edges(relationships between tables).

Connecting to database

Next, let's see how you can connect to a relational database. In this example, we will be using MySQL for the database. First, let's create files *main.go* and a *.env* file at the root of your projects. Let's add the following to the .env file.

```
DB_USER=root
DB_NAME=databas_name
DB_HOST=localhost
DB_PORT=3306
DB_PASSWORD=database_password
APP_PORT=4000
```

You can add the values for the *.env* file according to your development configurations. Next, let's add the content to the *main.go* file.

```
package main

func main() {
 viper.SetConfigFile(".env")
 viper.ReadInConfig()

ctx := context.Background()

url := fmt.Sprintf("%s:%s@(%s:%s)/%s?parseTime=True",
viper.Get("DB_USER"), viper.Get("DB_PASSWORD"),
viper.Get("DB_HOST"), viper.Get("DB_PORT"), viper.Get("DB_NAME"))

client, err := ent.Open(dialect.MySQL, url) // connect to MySQL

if err != nil {
  log.Fatal(err)
 }

defer client.Close()

if err := client.Schema.Create(ctx); err != nil {
  log.Fatalf("failed creating schema resources: %v", err)
 }
}
```

In the above code, you can notice that we have configured Viper to read the ***.env*** file in the project root. The database URL is created with the help of Viper and fmt.Sprintf API in golang.

Next, you can notice that the database connection is done through the **Ent.Open** API via Ent and the schemas/tables are created through the **Schema.Create(ctx)** API method.

Now that we have created the database connection let's create the web API using Fiber. The server code will look as below.

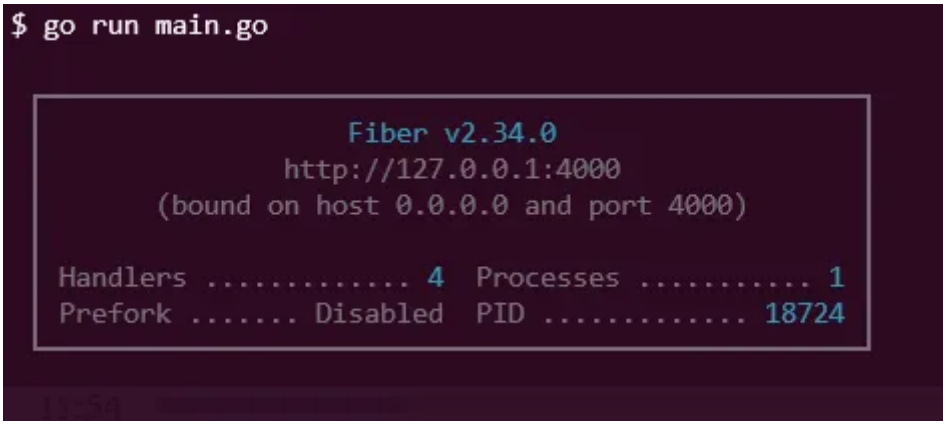
```
package main

import (
 "context"
 "fmt"
 "log"

"entgo.io/ent/dialect"
 _ "github.com/go-sql-driver/mysql"
```

```
 "github.com/gofiber/fiber/v2"
 "github.com/gofiber/fiber/v2/middleware/cors"
 "github.com/gofiber/fiber/v2/middleware/logger"
 "github.com/spf13/viper"
 "github.com/tromesh/go-ent-pokemon/ent"
)

func main() {
 viper.SetConfigFile(".env")
 viper.ReadInConfig()

ctx := context.Background()
 app := fiber.New()

url := fmt.Sprintf("%s:%s@(%s:%s)/%s?parseTime=True",
viper.Get("DB_USER"), viper.Get("DB_PASSWORD"),
viper.Get("DB_HOST"), viper.Get("DB_PORT"), viper.Get("DB_NAME"))
 client, err := ent.Open(dialect.MySQL, url)

if err != nil {
  log.Fatal(err)
 }

defer client.Close()

if err := client.Schema.Create(ctx); err != nil {
  log.Fatalf("failed creating schema resources: %v", err)
 }

app.Use(cors.New())
 app.Use(logger.New())

app.Get("/", func(c *fiber.Ctx) error {
  return c.SendString("Hello, World!")
 })

log.Fatal(app.Listen(fmt.Sprintf(":%s", viper.Get("APP_PORT"))))
}
```

You can start the web server through the following command.

```
go run main.go
```

If everything is fine you will have something similar to the image below.



Fiber server

Performing CRUD

Now that we have a web API let's create the Fiber endpoints for creating, retrieving pokemon and battle data with their relationships.

First, let's start by creating a pokemon.

```
app.Post("/create", func(c *fiber.Ctx) error {
		payload := struct {
			Name        string  `json:"name"`
			Description string  `json:"description"`
			Weight      float64 `json:"weight"`
			Height      float64 `json:"height"`
		}{}

		if err := c.BodyParser(&payload); err != nil {
			return err
		}

		pokemon, err := client.Pokemon.
			Create().
			SetName(payload.Name).
			SetDescription(payload.Description).
			SetWeight(payload.Weight).
			SetHeight(payload.Height).
			Save(ctx)
		if err != nil {
			return fmt.Errorf("failed creating pokemon: %w", err)
		}
		log.Println("pokemon created: ", pokemon)
		return c.Status(200).JSON(pokemon)
	})
```

create_pokemon.go hosted with ❤ by GitHub view raw

The above Fiber endpoint will parse the JSON body data to a struct in golang through the ***BodyParser*** API, which allows us to read the data to perform the CRUD operations. You may notice that the properties for a pokemon are set using set functions available in an entity.

Next, let's create a battle object where we will deal with a One-to-Many edge or relationship between a pokemon and a battle.

```
app.Post("/create/battle", func(c *fiber.Ctx) error {
		payload := struct {
			Result    string `json:"result"`
			Contender int    `json:"contender"`
			Oponent   int    `json:"oponent"`
		}{}

		if err := c.BodyParser(&payload); err != nil {
			return err
		}

		battle, err := client.Battle.
			Create().
			SetResult(payload.Result).
			SetContenderID(payload.Contender).
			SetOponentID(payload.Oponent).
			Save(ctx)
		if err != nil {
			return fmt.Errorf("failed creating battle: %w", err)
		}
		log.Println("battle created: ", battle)
		return c.Status(200).JSON(battle)
	})
```

create_battle.go hosted with ❤ by **GitHub** **view raw**

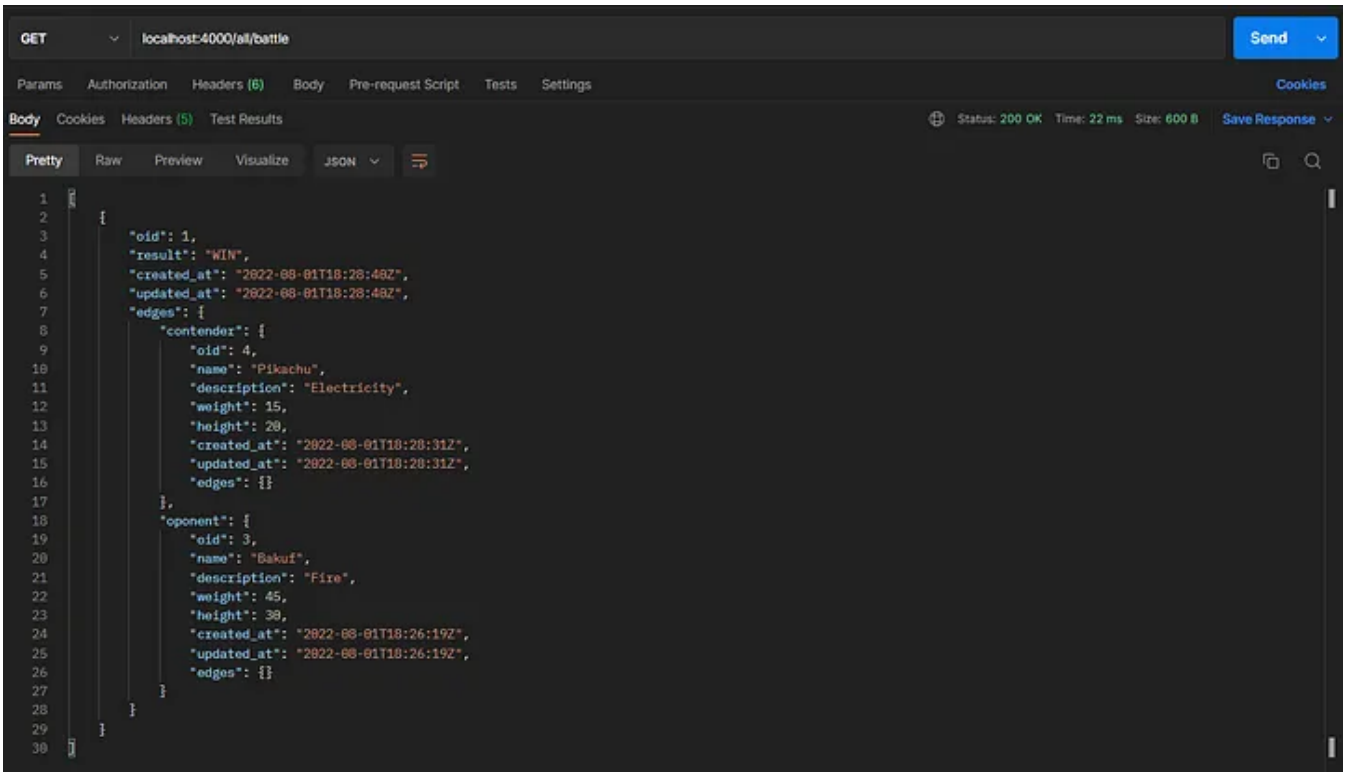
Notice that in the above Fiber endpoint, we use the set functions called ***SetContenderID()*** and ***SetOponentID(),*** which accept Id's a pokemon to create the edge/relationship between a battle and a pokemon.

Next, let's retrieve the battles and their respective pokemon using Eager Loading techniques in Ent.

```
app.Get("/all/battle", func(c *fiber.Ctx) error {
                battles, err := client.Battle.Query().WithContender().WithOponent().All(
                if err != nil {
                        return fmt.Errorf("failed querying battles: %w", err)
                }
                log.Println("returned battles:", battles)

                return c.Status(200).JSON(battles)
        })
```

get_battles.go hosted with ❤ by GitHub view raw

You can notice that we have used some special APIs called *WithContender()* and *WithOponent(),* which will retrieve the data of pokemon related to the **contender** and **opponent** edges. If you make a GET request to this particular endpoint, you will retrieve some data similar to the image below.



Making a request to /all/Battle using Postman

Notice that the **contender** and **opponent** pokemon are available inside the **edges** section of the payload.

It's not just building edges/relationships. You can even do more complex stuff like Transactions, Aggregations and Hooks with Ent. You can find the complete code to the above example through this link.

Extensions for Ent

You can easily create extensions for Ent through Ent's Extention API, allowing you to add new functionality to Ent's core. You must implement the Extension interface with Hooks, Annotations, and Templates. With the help of extensions, Ent can be easily integrated with GraphQL through the library gqlgen, a graphql server library for Golang.

Conclusion

Ent is one of the best ORM for Go with code generation, migrations, and Graphql integrations. You can agree that it provides a complete ORM Framework. Ent is now a part of the Linux Foundation, which governs other open-source software projects like Kubernetes and GraphQL. Ent also has a good and active community in slack where you can share and learn from the Ent community. For more information, you can read their excellent documentation and well-guided tutorials. At last, thank you for taking the time to read this. I would like to see your questions and comments below.

If you like my content, please do me a favour and get a cup of coffee for you and me!

Cheers!

Learn More

Let's "Go" and build an Application with gRPC

REST (REpresentational State Transfer) architecture has become the go-to method to build applications like web...

levelup.gitconnected.com

Let's “Go” and build Graphql API with gqlgen

Golang is one of the most loved programming language in the past decade, primarily because of its fast...

levelup.gitconnected.com

Jotai: Atom-based state management for React

State management has evolved a lot during the past few years. There are a lot of libraries and methods where you could...

levelup.gitconnected.com

Create Memory and Type-Safe Node.js Modules with Rust

Node is one of the most famous platforms to develop applications, which runs on the V8 engine and executes JavaScript...

levelup.gitconnected.com

Level Up Coding

Thanks for being a part of our community! Before you go:

- 🔔 Follow us: Twitter | LinkedIn | Newsletter

🚀👉 Join the Level Up talent collective and find an amazing job

Golang Programming Database Object Relational Mapping

Software Development

250 |  1

Enjoy the read? Reward the writer.[Beta]

Your tip will go to Tharaka Romesh through a third-party platform of their choice, letting them know you appreciate their story.

Give a tip

Sign up for Top Stories

By Level Up Coding

A monthly summary of the best stories shared in Level Up Coding Take a look.