

---

## protoc-gen-gorm

### Chinese Documentation

#### Purpose

A protobuf (<https://developers.google.com/protocol-buffers/>) compiler plugin designed to simplify the API calls needed to perform simple object persistence tasks. This is currently accomplished by creating a secondary .pb.gorm.go file which contains sister objects to those generated in the standard .pb.go file satisfying these criteria:

- Go field decorators/tags can be defined by option in the .proto file for GORM/SQL
- There are options for dropping fields from the PB object, or adding additional fields (not generally recommended, as it reduces clarity of .proto file)
- Converters between PB version and ORM version of the objects are included
- Hook interfaces allow for custom application logic

#### Prerequisites

**1. Protobuf Compiler** The protobuf compiler (protoc) is required.

Official instructions

Abbreviated version

**2. Golang Protobuf Code Generator** Get the golang protobuf code generator:

Before Go 1.17

```
1 go get -u github.com/golang/protobuf/protoc-gen-go
```

Starting from Go 1.17

```
1 go install github.com/golang/protobuf/protoc-gen-go@latest
```

**3. Vendored Dependencies** Retrieve and install the vendored dependencies for this project with go mod:

```
1 go mod tidy
```

---

## Installation

To use this tool, install it from code with `make install`, `go install github.com/infobloxopen/protoc-gen-gorm@latest` directly, or `go get github.com/infobloxopen/protoc-gen-gorm`.

## Usage

Once installed, the `--gorm_out=.` or `--gorm_out=${GOPATH}src` option can be specified in a `protoc` command to generate the `.pb.gorm.go` files.

Any message types with the `option (gorm.opts).ormable = true` will have the following autogenerated:

- A struct with ORM compatible types and the “ORM” suffix
- GORM tags built from the field options `[(gorm.field).tag = {..., tag: value, ...}]`.
- A `{PbType}.ToORM` and `{TypeORM}.ToPB` function
- Additional, unexposed fields added from the `option (gorm.opts) = {include: []}`, either of a built-in type e.g. `{type: "int32", name: "secret_key"}`, or an imported type, e.g. `{type: "StringArray", name: "array", package: "github.com/lib/pq"}`.
- Barebones C/U/R/D/L handlers that accept the protobuf versions (as from an API call), a context (used with the `multiaccount` option and for collection operators <https://github.com/infobloxopen/atlas-app-toolkit#collection-operators>), and a `gorm.DB` then perform the basic operation on the DB with the object
- Interface hooks for before and after each conversion that can be implemented to add custom handling.

Any services with the `option (gorm.server).autogen = true` will have basic grpc server generated:

- For service methods with names starting with `Create | Read | Update | Delete` generated implementation will call basic CRUD handlers.
- For other methods `return &MethodResponse{}`, `nil` stub is generated.

For CRUD methods to be generated correctly you need to follow specific conventions:

- Request messages for `Create` and `Update` methods should have an `Ormable` Type in a field named `payload`, for `Read` and `Delete` methods an `id` field is required. Nothing is required in the `List` request.
- Response messages for `Create`, `Read`, and `Update` require an `Ormable` Type in a field named `result` and for `List` a repeated `Ormable` Type named `results`.
- `Delete` methods require the `(gorm.method).object_type` option to indicate which `Ormable` Type it should delete, and has no response type requirements.

To customize the generated server, embed it into a new type and override any desired functions.

If conventions are not met stubs are generated for CRUD methods. As seen in the `feature_demo/demo_service` example.

---

To leverage DB specific features, specify the DB engine during generation using the `--gorm_out="engine={postgres,...}:{path}"`. Currently only Postgres has special type support, any other choice will behave as default.

The generated code can also integrate with the grpc server gorm transaction middleware provided in the atlas-app-toolkit using the service level option `option (gorm.server).txn_middleware = true`.

## Examples

Example .proto files and generated .pb.gorm.go files are included in the 'example' directory. The user file contains model examples from GORM documentation, the feature\_demo/demo\_types demonstrates the type handling and multi\_account functions, and the feature\_demo/demo\_service shows the service autogeneration.

Running `make example` will recompile all these test proto files, if you want to test the effects of changing the options and fields.

## Supported Types

Within the proto files, the following types are supported:

- standard primitive types `uint32`, `uint64`, `int32`, `int64`, `float`, `double`, `bool`, `string` map to the same type at ORM level
- google wrapper types `google.protobuf.StringValue`, `.BoolValue`, `.UInt32Value`, `.FloatValue`, etc. map to pointers of the internal type at the ORM level, e.g. `*string`, `*bool`, `*uint32`, `*float`
- [google timestamp type] (<https://github.com/golang/protobuf/blob/master/ptypes/timestamp/timestamp.proto>) `google.protobuf.Timestamp` maps to `time.Time` type at the ORM level
- custom wrapper types `gorm.types.UUID` and `gorm.types.UUIDValue`, which wrap strings and convert to a `uuid.UUID` and `*uuid.UUID` at the ORM level, from <https://github.com/satori/go.uuid>. A null or missing `gorm.types.UUID` will become a ZeroUUID (00000000-0000-0000-0000-000000000000) at the ORM level.
- custom wrapper type `gorm.types.JSONValue`, which wraps a string in protobuf containing arbitrary JSON and converts to custom `types.Jsonb` type if Postgres is the selected DB engine, otherwise it is currently dropped.
- custom wrapper type `gorm.types.InetValue`, which wraps a string and will convert to the `types.Inet` type at ORM level, which uses the golang `net.IPNet` type to hold an ip address and mask, IPv4 and IPv6 compatible, with the scan and value functions necessary to write to DBs. Like JSONValue, currently dropped if DB engine is not Postgres
- types can be imported from other .proto files within the same package (protoc invocation) or between packages. All associations can be generated properly within the same package, but cross package only the belongs-to and many-to-many will work.
- some repeated types can be automatically handled for Postgres by [github.com/lib/pq](https://github.com/lib/pq), and as long

---

as the engine is set to postgres then to/from mappings will be created (see the example called example/postgres\_arrays/postgres\_arrays.proto): - []bool: pq.BoolArray - []float64: pq.Float64Array - []int64: pq.Int64Array - []string: pq.StringArray

## Associations

The plugin supports the following GORM associations:

- Belongs-To
- Has-One
- Has-Many
- Many-To-Many

Note: polymorphic associations are currently not supported.

Association is defined by adding a field of some ormable message type(either single or repeated).

```
1 message Contact {
2     option (gorm.opts).ormable = true;
3     uint64 id = 1;
4     string name = 2;
5     repeated Email emails = 3;
6     Address home_address = 4;
7 }
```

Has-One is a default association for a single message type.

```
1 message Contact {
2     option (gorm.opts).ormable = true;
3     uint64 id = 1;
4     string first_name = 2;
5     string middle_name = 3;
6     string last_name = 4;
7     Address address = 5;
8 }
9
10 message Address {
11     option (gorm.opts).ormable = true;
12     string address = 1;
13     string city = 2;
14     string state = 3;
15     string zip = 4;
16     string country = 5;
17 }
```

Set (`gorm.field`).`belongs_to` option on the field in order to define Belongs-To.

---

---

```
1 message Contact {
2     option (gorm.opts).ormable = true;
3     uint64 id = 1;
4     string first_name = 2;
5     string middle_name = 3;
6     string last_name = 4;
7     Profile profile = 5 [(gorm.field).belongs_to = {}];
8 }
9
10 message Profile {
11     option (gorm.opts).ormable = true;
12     uint64 id = 1;
13     string name = 2;
14     string notes = 3;
15 }
```

Has-Many is a default association for a repeated message type.

```
1 message Contact {
2     option (gorm.opts).ormable = true;
3     uint64 id = 1;
4     string first_name = 2;
5     string middle_name = 3;
6     string last_name = 4;
7     repeated Email emails = 5;
8 }
9
10 message Email {
11     option (gorm.opts).ormable = true;
12     string address = 1;
13     bool is_primary = 2;
14 }
```

Set `(gorm.field).many_to_many` option on the field in order to define Many-To-Many.

```
1 message Contact {
2     option (gorm.opts).ormable = true;
3     uint64 id = 1;
4     string first_name = 2;
5     string middle_name = 3;
6     string last_name = 4;
7     repeated Group groups = 5 [(gorm.field).many_to_many = {}];
8 }
9
10 message Group {
11     option (gorm.opts).ormable = true;
12     uint64 id = 1;
13     string name = 2;
14     string notes = 3;
15 }
```

---

For each association type, except Many-To-Many, foreign keys pointing to primary keys (association keys) are automatically created if they don't exist in proto messages, their names correspond to GORM default foreign key names. GORM association tags are also automatically inserted.

## Customization

- For each association type you are able to override default foreign key and association key by setting `foreignKey` and `references` options.
- For each association type you are able to override default behavior of creating/updating the record. It's references can be created/updated depending on `disable_association_autoupdate`, `disable_association_autocreate` options, which effectively produces statements to prevent Association *auto* behavior (much like in tags were used in gormV1). Check out official association docs for more information.
- For each association type you are able to set `preload` option, which generates required code (with the same meaning gormV1 preload tag was working). Check out GORM docs.
- By default when updating child associations are wiped and replaced. This functionality can be switched to work the same way gorm handles this see [GORM]<https://gorm.io/docs/associations.html> this is done by adding one of the gorm association handler options, the options are `append` ([GORM]<https://gorm.io/docs/associations.html#Append-Associations>), `clear` ([GORM]<https://gorm.io/docs/associations.html#Clear-Associations>) and `replace` ([GORM]<https://gorm.io/docs/associations.html#Replace-Associations>).
- For Has-Many you are able to set `position_field` so additional field is created if it doesn't exist in proto message to maintain association ordering. Corresponding CRUDL handlers do all the necessary work to maintain the ordering.
- For automatically created foreign key and position field you're able to assign GORM tags by setting `foreignkey_tag` and `position_field_tag` options.
- For Many-To-Many you're able to override default join table name and column names by setting `jointable`, `joinForeignKey` and `joinReferences` options.

Check out user to see a real example of associations usage.

## Limitations

Currently only proto3 is supported.

This project is currently in development, and is expected to undergo "breaking" (and fixing) changes

---

## Contributing

Pull requests are welcome!

Any new feature should include tests for that feature.

Before pushing changes, run the tests with:

```
1 make gentool-test
```

This will run the tests in a docker container with specific known versions of dependencies.

Before the tests run, they generate code. Commit any new and modified generated code as part of your pull request.