Documentation

Hummingbird Document… / Postgres Migrations

Article

Postgres Migrations

Managing database structure changes.

Overview

Database migrations are a controlled set of incremental changes applied to a database. You can use a migration list to transition a database from one state to a new desired state. A migration can involve creating/deleting tables, adding/removing columns, changing types and constraints. The PostgresMigrations library that comes with HummingbirdPostgres provides support for setting up your own database migrations.

> **Note**
>
> If you are using Fluent then you should use the migration support that comes with Fluent.

Each migration includs an `apply` method that applies the change and a `revert` method that reverts the change.

```
struct CreateMyTableMigration: DatabaseMigration {
    func apply(connection: PostgresConnection, logger: Logger) async throws
        try await connection.query(
            """
            CREATE TABLE my_table (
                "id" text PRIMARY KEY,
                "name" text NOT NULL
            )
            """,
            logger: logger
        )
    }

    func revert(connection: PostgresConnection, logger: Logger) async throws
        try await connection.query(
            "DROP TABLE my_table",
            logger: logger
        )
    }
}
```

As an individual migration can be dependent on the results of a previous migration the order they are applied has to be the same everytime. Migrations allow for database changes to be repeatable, shared and testable without loss of data.

Adding migrations

You need to create a `DatabaseMigrations` object to store your migrations in. Only create one of these, otherwise you could confuse your database about what migrations need applied. Adding a migration is as simple as calling `add`.

```
import HummingbirdPostgres

let migrations = DatabaseMigrations()
await migrations.add(CreateMyTableMigration())
```

Applying migrations

As you need an active `PostgresClient` to apply migrations you need to run the migrate once you have called `PostgresClient.run`. It is also preferable to have run your migrations before your server is active and accepting connections. The best way to do this is use `beforeServerStarts(perform:)`.

```
var app = Application(router: router)
// add postgres client as a service to ensure it is active
app.addServices(postgresClient)
app.beforeServerStarts {
    try await migrations.apply(client: postgresClient, logger: logger, dryRu
}
```

You will notice in the code above the parameter `dryRun` is set to true. This is because applying migrations can be a destructive process and should be a supervised. If there is a change in the migration list, with `dryRun` set to true, the `apply` function will throw an error and list the migrations it would apply or revert. At that point you can make a call on whether you want to apply those changes and run the same process again except with `dryRun` set to false.

Reverting migrations

There are a number of situations where a migration maybe reverted.

- The user calls `revert(client:groups:logger:dryRun:)`. This will revert all the migrations applied to the database.
- A user removes a migration from the list. The migration still needs to be registered with the migration system as it needs to know how to revert that migration. This is done with a call to `register(_:)`. When a migration is removed it is reverted and all subsequent migrations will be reverted and then re-applied.
- A user changes the order of migrations. This is generally a user error, but if it is intentional then the first migration affected by the order change and all subsequent migrations will be reverted and then re-applied.

Migration groups

A migration group is a group of migrations that can be applied to a database independent of all other migrations outside that group. By default all migrations are added to the `.default` migration group. Each group is applied independently to your database. A group allows for a modular piece of code to add additional migrations without affecting the ordering of other migrations and causing deletion of data.

To create a group you need to extend `/PostgresMigrations/DatabaseMigrationsGroup` and add a new static variable for the migration group id.

```
extension DatabaseMigrationGroup {
    public static var myGroup: Self { .init("my_group") }
}
```

Then every migration that belongs to that group must set its group member variable

```
extension CreateMyTableMigration {
    var group: DatabaseMigrationGroup { .myGroup }
}
```

You should only use groups if you can guarantee the migrations inside it will always be independent of migrations outside the group.

The persist driver that come with HummingbirdPostgres and the job queue driver from JobsPostgres both use groups to separate their migrations from any the user might add.

See Also

Related Documentation

`protocol DatabaseMigration`

Protocol for a database migration

`actor DatabaseMigrations`

Database migration support

Database Integration

Store Data with Fluent

A tutorial that shows you how to set up Hummingbird 2 with Fluent to create and access your Galaxies.

Store Data with MongoKitten

A tutorial that shows you how to set up Hummingbird 2 with MongoKitten to create and share your kittens.