DOPS Training

09.21.2025

HyTech Racing

Checkpoint 2

Goal

When a MCAP is uploaded, a MongoDB document should be created for it in the collection. Example:

```
{
    _id: ObjectId('67f5dc9c20ee6a9547547234'),
    date_uploaded: 2025-09-14T02:33:39.100+00:00,
    location: '',
    car_model: '',
    event_type: '',
    notes: '',
    file: {
        aws_bucket: '',
        file_path: '',
        file_name: '',
    }
}
```

Connecting to MongoDB

There are multiple steps required to add/remove/update data with external data stores (MongoDB and S3)

1. Data must be prepared before inserting into the store
2. A data store client needs to connect to the store
3. Something uses the client and to update the store with new data

We want to separate these responsibilities in code to methodically ensure everything happens correctly and a easy way to easily identify where errors happen.

Model, Repository, Usecase

We use a layered design pattern when connecting data stores to our server. It allows us to separate functionality into different layers ultimately improving code maintainability and enforces a consistent manner to work with data.

The 3 layers:

- Model: Represents the structure of different types of data
- Repository: Contains all logic used to modify the data source
- Usecase: Defines specific application operations that are allowed

Each layer is under the assumption that the layer underneath it works.

Model

The Model represents the structure of different types of data. The model also handles field validation and custom serialization logic.

```
// internal/models/user_model.go

type User struct {
  ID        primitive.ObjectID `bson:"_id,omitempty" json:"id"`
  Name      string             `bson:"name" json:"name"`
}

func NewUser(name) (*User, error) {
  if name == "" {
    return nil, errors.New("name is required")
  }
  return &User{
    ID:        primitive.NewObjectID(),
    Name:      name,
  }, nil
}
```

Repository

The Repository handles accessing data and provides an interface to do so regardless of the underlying storage mechanism

```go
// internal/database/repository/user_repository.go
package repository

type UserRepository interface {
  Create(ctx context.Context, user *models.User) error
  Delete(ctx context.Context, id primitive.ObjectID) error
  List(ctx context.Context) ([]*models.User, error)
 // add more functions as needed
}

// MongoUserRepository implements UserRepository for MongoDB
type MongoUserRepository struct {
  collection *mongo.Collection
}

func NewMongoUserRepository(db *mongo.Database) *MongoUserRepository {
  return &MongoUserRepository{
    collection: db.Collection("users"),
  }
}
```

Repository (ii)

```go
func (r *MongoUserRepository) Create(ctx context.Context, user *models.User) error {
  _, err = r.collection.InsertOne(ctx, user)
  if err != nil {
    return fmt.Errorf("failed to create user: %w", err)
  }
  return nil
}

func (r *MongoUserRepository) Delete(ctx context.Context, id primitive.ObjectID) error {
  result, err := r.collection.DeleteOne(ctx, bson.M{"_id": id})
  if err != nil {
    return fmt.Errorf("failed to delete user: %w", err)
  }
  return nil
}

func (r *MongoUserRepository) List(ctx context.Context) ([]*models.User, error) {
  // logic here
  return users, nil
}
```

Usecase

The Usecase contains the “business logic” and orchestrates the flow of data between repositories and models. Each usecase method represents specific actions or operations that users can perform in the system.

```go
// internal/database/usecase/user_usecases.go

type UserUseCase struct {
  userRepo repository.UserRepository
}

func NewUserUseCase(userRepo repository.UserRepository) *UserUseCase {
  return &UserUseCase{
    userRepo: userRepo,
  }
}
```

Usecase (ii)

```go
func (uc *UserUseCase) CreateUser(ctx context.Context, name string) (*User, error) {
  user, err := models.NewUser(name)
  if err != nil {
    return nil, fmt.Errorf("validation failed: %w", err)
  }
  err = uc.userRepo.Create(ctx, user)
  if err != nil {
    return nil, fmt.Errorf("failed to create user: %w", err)
  }
  return user, nil
}

// add more usecases as needed
```

Putting it all together

We can expect the following code to create a new user! Notice how there's very little code needed to create the user. The Usecase is generally created once and can be used by the entire program.

```
func main() {
  // create mongodb client
  ctx := context.Background()
  mongoURI := "mongodb://localhost:27017"
  client, _ := mongo.Connect(ctx, options.Client().ApplyURI(mongoURI))
  db := client.Database("userapp")

  // create repository
  userRepo := NewMongoUserRepository(db)

  // create usecase
  userUseCase := NewUserUseCase(userRepo)

  // use the usecase to create a new user
  newUser, err := userUseCase.CreateUser("hytech")

  // handle error if exists
}
```

Checkpoint 2 Steps

You will have to create and add the internal `models`, `repository`, and `usecase` packages to your server.

Your folder should look like:

```
❯ tree
.
├── go.mod
├── go.sum
├── internal
│   └── db
│       ├── repository
│       │   └── car_run_repository.go
│       └── usecase
│           └── car_run_usecases.go
├── main_test.go
├── main.go
└── models
    └── car_run_model.go
```

Checkpoint 2 Steps (ii)

You should have a `CarRun` model with the following fields:

- id
- date_uploaded
- location
- car_model
- event_type
- notes
- file
  - aws_bucket
  - file_path
  - file_name

Checkpoint 2 Steps (iii)

Here is the `CarRunRepository` interface

```
type CarRunRepository interface {
  Create(ctx context.Context, carRun *models.CarRun) error
  Update(ctx context.Context, carRun *models.CarRun) error
  Delete(ctx context.Context, id primitive.ObjectID) error
  List(ctx context.Context) ([]*models.carRun, error)
}
```

You are to implement the interface with `MongoCarRunRepository`

Checkpoint 2 Steps (iv)

Create the `CarRunUseCase` with the following usecases:

- `func (uc *CarRunUseCase) CreateCarRunUseCase(ctx context.Context) (*CarRun, error)`
  - It should create a car run document with only the `id` and `date_uploaded` fields populated into the database

Finally, call this usecase in your `/upload` method.