Design Pattern Lab Project Documentation

1 Layered Architecture in the Secret Santa API

The Secret Santa API follows a structured three-layer architecture:

- **Handler Layer**: Handles HTTP requests and responses.
- **Service Layer**: Contains business logic and coordinates between handler and repository.
- **Repository Layer**: Interacts with MongoDB for data access.

And for that, I implemented the following 5 Design Patterns:

- **Constructor Injection** — dependencies are provided through constructors for loose coupling and testability.
- **Composition Root** — all dependencies are composed at the application's entry point (`main.go`).
- **Factory Method** — custom errors are created using functional options for flexible and modular error handling.
- **Singleton** — the MongoDB client is instantiated once and reused through the DI container.
- **Observer Pattern** — participants are notified of their Secret Santa match through an in-memory observer system.

2 Dependency Injection (DI) Overview

Dependency Injection (DI) follows:

- **Dependency Inversion Principle (DIP)**
- **Inversion of Control (IoC)**

Implemented via `Uber Dig`.

3 Constructor Injection

The first DI pattern used in this project is **Constructor Injection**.

What is Constructor Injection?

Definition: Constructor Injection is the act of statically defining the list of required dependencies by specifying them as parameters to the struct's constructor.

How is Constructor Injection used in this API?

Each layer depends on the one below it, but instead of creating dependencies inside the structs, they are injected through constructor functions:

Repository Layer - `NewGroupRepository`

```
func NewGroupRepository(db *mongo.Client) Repository {
    return &resource{db: db}
}
```

The repository does not create a database connection itself; instead, the `*mongo.Client` is injected when the repository is instantiated.

Service Layer - `NewGroupService`

```
func NewGroupService(repo group.Repository) Service {
    return &resource{repo: repo}
}
```

The service layer does not instantiate a repository; it receives one as a dependency.

Handler Layer - `NewGroupHandler`

```
func NewGroupHandler(svc group.Service) Handler {
    return &resource{svc: svc}
}
```

The handler does not create a service but accepts one via its constructor.

This pattern ensures that each component receives its dependencies from the outside, making it more flexible and testable.

4 Composition Root Design Pattern and Object Graph

What is an Object Graph?

An object graph represents how dependencies are related to each other in memory. In the context of Dependency Injection (DI), it visualizes how objects and their dependencies are structured and injected.

For example, in this API, the object graph looks like this:

```
Main (entry point)
          Handler (GroupHandler)
               Service (GroupService)
                    Repository (GroupRepository)
                         MongoDB Client
```

Each layer depends on another but does not create its dependencies—it receives them via Constructor Injection.

What is the Composition Root?

A Composition Root is a unique location in an application where dependencies are composed together, as close as possible to the application's entry point.

- In this API, `main.go` is the Composition Root.
- Object graph composition happens in `di.go`, ensuring separation of concerns.

How Does This API Follow the Composition Root Principle?

- Composing dependencies at the application's entry point (`main.go`).
- Constructor Injection is used throughout the application, avoiding dependency creation inside business logic.
- The DI Container (`Uber Dig`) is only referenced in the Composition Root, ensuring modularity and testability.

Dependency Injection Setup in `di.go`

```
func InitializeDI(client *mongo.Client) {
    Container = dig.New()

    Container.Provide(groupRepository.NewGroupRepository)
    Container.Provide(groupService.NewGroupService)
    Container.Provide(groupHandler.NewGroupHandler)
}
```

This ensures that all dependencies are registered and resolved only at the Composition Root.

Dependency Resolution in `main.go`

```
di.InitializeDI(mongoClient)
di.Invoke(secretSantaGroup)
```

This invokes all registered dependencies and composes the object graph at runtime.

5 Factory Method Pattern in Custom Error Handling

```
func NewCustomError(opts ...CustomErrorOption) *CustomError {
    err := &CustomError{Causes: "", Status: 0, Message: ""}
    for _, opt := range opts {
        opt(err)
    }
    return err
}
```

Example Option

```
func WithNotFound(causes, message string) CustomErrorOption {
    return func(e *CustomError) {
        e.Causes = causes
        e.Status = http.StatusNotFound
        e.Message = message
        e.Code = http.StatusText(http.StatusNotFound)
    }
}
```

6 Singleton Pattern via Dependency Injection

In Go, especially when using a DI container like `Uber Dig`, singleton behavior is achieved by default: once a constructor provides an object, the same object is reused wherever needed.

The package documentation states:

> "Multiple constructors can rely on the same type. The container creates a singleton for each retained type, instantiating it at most once when requested directly or as a dependency of another type."

In this API, the MongoDB client is created once and reused across the application:

```
mongoClient := di.InitializeMongoClient()
Container.Provide(func() *mongo.Client { return client })
```

This ensures that all components depending on `*mongo.Client` receive the same instance throughout the application's lifecycle.

7 7. Observer Pattern for Match Notification

The Observer Pattern is used to decouple the group matching logic from the notification logic. This allows each participant to react individually when matches are generated.

- `Group` implements the `Subject` interface and manages a list of observers.
- `Participant` implements the `Observer` interface by defining an `Update` method.
- When `MatchParticipants` is called, `Group.NotifyAll` triggers `Update` for each observer.
- This allows for extensibility—other types like external notifiers can be added as observers without changing existing code.

Why Runtime-Only

- `Observers` are stored in-memory and rebuilt with `RebuildObservers()` after loading a group.
- They are tagged with `json:"-"`, `bson:"-"`, and `swaggerignore:"true"` to avoid serialization and persistence.