Modular Monolith architecture

Status

Context

The modular monolith architecture is a way to structure application that could be split later once the boundary is known.

Most of the packages will be independent, but there will be a glue layer that links them together.

We will have the following packages

- boundary: this is basically what separates one functionality from another. This depends on whether you choose to split the application by domain, usecase, feature, business, by app etc.
- storage: the data that can be obtained from the storage layer, e.g tables for RDBMS, documents or collections from NoSQL, cache implementation. The types are self-contained and has no externap dependencies.
- transport: the entry layer of the app. For APIs, this can be called rest, or grpc, or graphql and even cli. There can be multiple transport available.
- config: the configuration of the app. This layer maps all environment variables to a variable.
- adapter: all external dependency, such as db, cache, message queue and even external api client or packages. Ideally the app only interacts sith the interfaces. The adapter contains configuration that will be resolved in config, but should not share types with confog.

Boundary

Boundaries should not overlap. There are no common settings between them. At the root of the boundary package is the domain types.

Each boundary will have their own domain types, usecase and repository. The repository maps the storage or external apis to the domain types. The usecase has repository nested below. The inner layer can only call the outer layer, never the opposite.

Repository

Repository is the glue layer between the application layer (aka usecase) and the domain layer. Repository is *not* the database access layer.

All external data is handled at the repository layer, and is mapped to the domain or application layer types where necessary.

Layer

```
boundary/
- auth/
  - auth.go
  - usecase/
```

```
      - usecase.go
      - repository/
        - repository.go
```

Here we see an example of deeply nested package structure.

When structuring packages, apply the following rules:

1. outer package cannot access inner package, except the main program
2. inner program can access the outer package
3. when declaring interface, the interface should not have types that belongs to other packages