

# Anime-Plan

## 后端技术栈：

- web框架Echo
- 数据库ent
- 用户鉴权jwt
- 项目分层fx

项目的主要结构如下图所示，项目的layout采用的是Go 生态系统中一组常见的老项目和新项目的布局模式

<https://github.com/golang-standards/project-layout>

```
tree -d .
.
├── bin
├── cmd
│   └── web
├── docs
└── ent
    ├── collection
    ├── enttest
    ├── hook
    ├── members
    ├── migrate
    ├── predicate
    ├── runtime
    ├── schema
    ├── subject
    └── subjectfield
    ├── internal
    │   ├── collection
    │   ├── config
    │   ├── ctrl
    │   ├── driver
    │   ├── logger
    │   ├── subject
    │   ├── subject_field
    │   └── user
    ├── public
    └── test
        ├── image
        └── server
└── web
    ├── handler
    │   ├── collection
    │   ├── subject
    │   └── user
    ├── middleware
    ├── request
    │   ├── collection
    │   ├── subject
    │   └── user
    └── response
        └── util

41 directories
```

对于项目的配置我们采用的yaml文件

```
http_host: 127.0.0.1
http_port: 2333
web_domain: 127.0.0.1:2333

mysql:
  host: 127.0.0.1
  port: 3306
  user: user
  password: password
  db: abyss

jwt: 2d7af79ab14a28f368ae68cd7ba393ed7f4d31ed2f094cebc004360dc9132916
static_directory: "./public"
```

web框架使用的是Echo，它以小巧高性能著称，只提供了一些基本的功能。

我们还使用了uber开发的依赖注入库fx，避免了全局变量的使用。我们在每一个模块中有使用fx进行分层，层层推进，这样项目的结构会更加清晰。一下是一个例子。

```
err := fx.New(
  fx.NopLogger,
  fx.Provide(config.AppConfigReader, driver.NewMysqlClient, util.NewJwtUtil),
  fx.Provide(user.NewRepo),
  fx.Provide(subject.NewRepo),
  fx.Provide(collection.NewRepo),
  fx.Provide(subjectField.NewRepo),

  ctrl.Module,
  web.Module,
  fx.Populate(&cfg, &e),
).Err()
```

它还提供了生命周期的功能。fx可以管理应用程序的启动和关闭过程，保证所有的组件按照正确的顺序进行初始化和清理。

数据库采用的是ent框架，我们确定好数据库的表的schema后，就可以使用ent来生成，可以很方便的curd。其中的很方便的是ent里有一个edge的概念，表示两个或多个表之间的联系。

```
func (Subject) Fields() []ent.Field { ↳ madehaha +1
    return []ent.Field{
        field.Uint32( name: "id").Unique().Immutable(),
        field.String( name: "image").MaxLen( i: 255).Default( s: "https://lain.bgm.tv/pic/user/l/icon.jpg"),
        field.String( name: "summary").MaxLen( i: 3000).Default( s: "No summary."),
        field.String( name: "name"),
        field.String( name: "name_cn"),
        field.Uint8( name: "episodes"), // number of episodes

        field.Uint32( name: "wish").Default( i: 0),
        field.Uint32( name: "doing").Default( i: 0),
        field.Uint32( name: "watched").Default( i: 0),
        field.Uint32( name: "on_hold").Default( i: 0),
        field.Uint32( name: "dropped").Default( i: 0),
    }
}

// Annotations of the User.

// Edges of the Subject.
func (Subject) Edges() []ent.Edge { ↳ madehaha +1
    return []ent.Edge{
        edge.To( name: "collections", Collection.Type),
        edge.To( name: "subject_field", SubjectField.Type).Unique(),
    }
}
```

可以看到在subject表中，我们为它创建了两条edge，指向收藏（collections），以及subject\_field，还可以指定edge的类型，是可以有多条还是unique的。

本项目是使用的jwt来进行用户鉴权，我们自己自定义了两个字段，用户id和用户组id。用户id很显然，是用来区分用户的。用户组id代表了用户的权限，当我们尝试创建条目的时候，我们就需要更高的用户权限。

```
type JwtUserClaims struct { 5 usages • Sober7135
    Uid uint32 `json:"uid"`
    Gid uint8 `json:"gid"`
    jwt.StandardClaims
}
```

接下来介绍一下， web部分。

```
web
├── fx.go
└── handler
    ├── collection
    │   └── collection.go
    │   └── handler.go
    ├── fx.go
    ├── subject
    │   └── handler.go
    │   └── subject.go
    └── user
        └── handler.go
        └── user.go
├── middleware
│   └── jwt.go
└── new.go
├── request
│   ├── collection
│   │   └── collection.go
│   ├── subject
│   │   └── create_subject.go
│   └── user
│       └── user.go
└── response
    ├── collection.go
    ├── subject.go
    └── user.go
├── routes.go
└── util
    ├── detail.go
    ├── functional.go
    ├── jwt.go
    └── response.go
```

12 directories, 21 files

可以看到这里也有一个fx文件。new文件是对echo进行了一些简单的设置。可以看到我们使用了validate进行request body的参数验证，可以验证是否为必要的参数，还有简单的验证如数字的大小范围等。我们是直接将这个validator直接内嵌到了echo中。我们还设置了一个静态文件夹，用于储存一些文件。

```
func New() *echo.Echo { 1 usage  ↳ Sober7135
    return echo.New()
}

type CustomValidator struct { 2 usages  ↳ Sober7135
    validator *validator.Validate
}

func (cv *CustomValidator) Validate(i interface{}) error { ↳ Sober7135
    if err := cv.validator.Struct(i); err != nil : echo.NewHTTPError(http.StatusBadRequest, err.Error()) ↗
    return nil
}

💡 func Start(app *echo.Echo, cfg *config.AppConfig) error { 1 usage  ↳ Sober7135
    addr := cfg.ListenAddr()
    logger.Info(fmt.Sprintf("Server is listening at #{addr}"))
    app.Validator = &CustomValidator{validator: validator.New()}
    app.Static( pathPrefix: "public", cfg.StaticDirectory)
    return app.Start(addr)
}
```

request和response就是一些结构体设置。

routes定义了本项目所有的路由，由于路由较多就不一一展示了。下面是部分截图

```

func AddRouters( 1 usage  ± Sober7135 +1
    app *echo.Echo, middleware JwtMiddleware, userHandler user.Handler, subjectHandler subject.Handler,
    collectionHandler collection.Handler,
) {
    // logger
    app.Use(
        echoMiddleware.LoggerWithConfig(
            echoMiddleware.LoggerConfig{...},
        ),
    )
    // cors
    app.Use(echoMiddleware.CORS())

    // User
    app.POST( path: "/register", userHandler.Register)
    app.POST( path: "/login", userHandler.Login)
    app.POST( path: "/cancel", userHandler.Cancel, middleware.UserJWTAuth)
    app.PUT( path: "/modify", userHandler.ModifyInfo, middleware.UserJWTAuth)
    app.GET( path: "/me", userHandler.GetMe, middleware.UserJWTAuth)
    app.GET( path: "/user/:id/avatar", userHandler.GetAvatar)
    app.GET( path: "/user/:member_id", userHandler.GetMember)

    // Subject
    app.GET( path: "/subject/get", subjectHandler.GetSubject)
    app.POST( path: "/subject/create", subjectHandler.CreateSubject,
        middleware.WikiJWTAuth,
    ) // verify if having permission to create subject
    app.PUT(
        path: "/subject/create", subjectHandler.CreateSubjectWithSave,
        middleware.WikiJWTAuth,
    )
}

```

可以看到用到了三个中间件，CORS是echo内置的跨域中间件，UserJWTAuth和WikiJWTAuth都是jwt验证的中间件。UserJWTAuth就是简单的对用户进行鉴权，只要存在的用户就有权限进行访问。WikiJWTAuth对有subject编辑权限进行验证，对于没有权限的用户会直接返回401。

handler用来响应http request，在进行简单处理后，就转交给ctrl（在internal/ctrl中）进行处理。以user登陆为例。  
balabala

```
func (h Handler) Login(c echo.Context) error { 2 usages • Sober7135
    var req user.UserLoginReq
    var resp response.UserLoginResp
    if err := c.Bind(&req); err != nil {
        logger.Error( msg: "Login bind failed")
        return util.Error(c, http.StatusBadRequest, err.Error())
    }
    if err := c.Validate(&req); err != nil {
        logger.Error( msg: "Login Validate failed")
        return util.Error(c, http.StatusBadRequest, err.Error())
    }
    token, err := h.ctrl.Login(req.Email, req.Password)
    if err != nil {
        logger.Error( msg: "Failed to login")
        return util.Error(c, http.StatusBadRequest, err.Error())
    }
    resp.Token = token
    return util.Success(c, http.StatusOK, resp)
}
```

```
internal/ctrl
├── collection.go
├── fx.go
└── user.go
```

1 directory, 4 files

接下来介绍一下ctrl层，前面提到handler会调用这个层，这是处于handler和数据库层之间的中间层。