**Московский государственный технический**

**университет им. Н.Э. Баумана**

Факультет «Информатика и системы управления»

Кафедра ИУ5 «Системы обработки информации и управления»

Курс «Парадигмы и конструкции языков программирования»

Отчет по лабораторным работам №1–6

«Проектирование и разработка бэкенда для социальной сети»

|  |  |  |
| --- | --- | --- |
| Выполнил: |  | Проверил: |
| студент группы ИУ5-32Б |  | преподаватель каф. ИУ5 |
| Поляков Леонид |  | Гапанюк Ю.Е. |
| Подпись и дата: |  | Подпись и дата: |

Москва, 2024 г.

# Задание

Конечной целью выполнения лабораторных работ 1-6 является разработка серверной части приложения для социальной сети – аналога Reddit. Пользователи должны иметь возможность регистрироваться в приложении, просматривать и создавать посты, писать и удалять свои комментарии, голосовать за посты.

Имеется фронтенд (работает на JS, отправляет AJAX-запросы на сервер), и контракты ручек, которые он ожидает:

1. POST /api/register – регистрация
2. POST /api/login - логин
3. GET /api/posts/ - список всех постов
4. POST /api/posts/ - добавление поста - обратите внимание - есть с урлом, а есть с текстом
5. GET /api/posts/{CATEGORY\_NAME} - список постов конкретной категории
6. GET /api/post/{POST\_ID} - детали поста с комментами
7. POST /api/post/{POST\_ID} - добавление коммента
8. DELETE /api/post/{POST\_ID}/{COMMENT\_ID} - удаление коммента
9. GET /api/post/{POST\_ID}/upvote - рейтинг поста вверх
10. GET /api/post/{POST\_ID}/downvote - рейтинг поста вниз
11. GET /api/post/{POST\_ID}/unvote - отмена голоса
12. DELETE /api/post/{POST\_ID} - удаление поста
13. GET /api/user/{USER\_LOGIN} - получение всех постов конкретного пользователя

Некоторые из них доступны всем пользователям(1, 2, 3, 5, 6, 13), остальные только авторизованным пользователям (4, 7, 8, 9, 10, 11, 12). Все API будут отдавать данные в формате JSON.

**Требования:**

1. Код должен быть написан согласно best practices (читаемость, форматирование, использование комментариев).
2. Структура проекта должна быть понятной и логичной
3. Использование линтеров для поддержания единообразия и качества кода
4. Использование баз данных для хранения и обработки данных
5. Тестирование кода, покрытие тестами методов для работы с БД и хендлеров
6. Использовать контейнеризацию и оркестрацию
7. Использовать кэширование
8. CI/CD – реализовать пайплайн, который прогоняет линтеры, тесты, собирает приложение и деплоит его на удалённую машину
9. Наличие подробной документации

# Проектирование

В качестве языка для реализации проекта был выбран язык программирования go. Причины выбора:

* Желание более тесно познакомиться с go, научиться писать на нём веб-сервисы, изучить best-practices языка
* Предметная область: поскольку go создавался для высоконагруженных веб-сервисов, он хорошо справится с нагрузками, которые могут возникнуть, если у приложения будет достаточно много пользователей
* Растущая популярность языка на российском и зарубежном рынках

**Структура проекта:**

├── bin

├── cmd

│   ├── demo

│   └── redditclone

├── coverage

├── docs

├── init

│   └── db\_tables

├── internal

│   ├── config

│   ├── models

│   │   ├── posts

│   │   ├── users

│   │   └── …

│   ├── service

│   ├── storage

│   │   ├── inmem

│   │   ├── mocks

│   │   └── tests

│   └── transport

│   ├── middleware

│   └── rest

│   └── tests

├── pkg

│   └── middleware

└── static

├── css

├── html

└── js

Для проекта была выбрана трёхслойная архитектура (транспорт ⇄ сервис ⇄ репозиторий), а структура создана по примеру репозитория <https://github.com/golang-standards/project-layout>, в котором описана стандартная структура проекта на go, используемая в большинстве подобных проектов.

**Выбор технологий:**

Одной из целей реализации данного проекта было познакомиться с как можно большим количеством технологий, используемых при разработке современных высоконагруженных и отказоустойчивых систем, для расширения своего кругозора.

* Для хранения пользователей была выбрана реляционная СУБД MySQL из-за её популярности и наличия подробной документации.
* Для хранения информации о постах была выбрана документно-ориентированная БД MongoDB. Основные причины выбора: гибкость – MongoDB не накладывает ограничений на схему данных, что позволяет при необходимости быстро её перестроить; масштабируемость – MongoDB поддерживает репликацию и шардинг из коробки; хранение данных в виде готовых JSON объектов, которые уже содержат в себе все нужные данные, что позволяет избежать сложных JOIN’ов в запросах
* Для хранения пользовательских сессий был выбран in-memory кэш Redis. Основные причины выбора: скорость – redis во много раз превосходит по скорости традиционные реляционные БД, так как не работает с диском, а хранит все данные в оперативной памяти; отсутствие необходимости надёжности хранения сессий – даже если кэш с текущими сессиями упадёт, это приведёт просто к сбросу всех текущих сессий пользователей. Наличие механизма TTL, который позволяет контролировать время жизни сессий.
* Использование Swagger в качестве документации API сервиса. Swagger основан на спецификации OpenAPI, которая является открытым стандартом для описания RESTful API. Также он позволяет автоматически генерировать спецификацию и документацию по аннотациям из кода, тем самым позволяя разрабатывать сервисы с использованием code-first подхода. Также swagger предоставляет интерактивный UI что делает документацию более понятной и удобной для восприятия
* Использование Docker для контейнеризации и docker-compose для оркестрации. Docker позволяет добиться изоляции и портативности контейнеров, что в свою очередь упрощает развёртывание приложения. Docker-compose позволяет собрать все сервисы приложения и вспомогательную инфраструктуру для его работы (например, базы данных, кеш) в одном файле и достаточно гибко настроить их под свои нужды
* Использование github-actions для CI/CD пайплайнов. Т. к. репозиторий проекта хостится на github.

# Разработка

Первой целью было создание MVP – приложения, которое реализует все основные API ручки и все данные просто хранит в памяти, без использования БД.

Было принято решение вести разработку с использованием DDD (предметно-ориентированного подхода), когда доменный слой является ядром приложения. Он содержит основные сущности, является фундаментом для всего остального приложения и не имеет зависимостей от других слоёв. Также был сделан упор на богатом доменном слое, что несёт за собой ряд преимуществ:

* Максимум логики приложения в доменном слое
* Минимум сложного кода в других слоях
* Сущность всегда валидна, нет возможности её испортить, так как мутация сущности осуществляется через её методы
* Выявляем ошибку до того, как невалидная сущность была записана в БД

Первым делом были выделены следующие сущности(домены):

* Пост
* Комментарий к посту
* Пользователь
* Сессия (создаётся при авторизации)

**post.go**

package posts  
  
import (  
 "slices"  
 "time"  
  
 "github.com/google/uuid"  
  
 "github.com/Benzogang-Tape/Reddit/internal/models/errs"  
 "github.com/Benzogang-Tape/Reddit/internal/models/jwt"  
 "github.com/Benzogang-Tape/Reddit/internal/models/users"  
)  
  
// Post model info  
//  
// @Description Post Contains all the information about a particular post in the app  
type Post struct {  
 ID users.ID `json**:**"id" bson**:**"uuid" example**:**"12345678-9abc-def1-2345-6789abcdef12" minLength**:**"36" maxLength**:**"36"`  
 Score int `json**:**"score" bson**:**"score" example**:**"-1"` // The overall balance of the post's votes  
 Views uint `json**:**"views" bson**:**"views" example**:**"1"` // How many times the post has been viewed by users  
 Type PostType `json**:**"type" bson**:**"type" example**:**"1"` // Post with text(1) or with a link(0)  
 Title string `json**:**"title" bson**:**"title" example**:**"Awesome title"`  
 URL string `json**:**"url,omitempty" bson**:**"url,omitempty" example**:**"http://localhost:8080/"`  
 Author jwt.TokenPayload `json**:**"author" bson**:**"author"` // User who created the Post  
 Category PostCategory `json**:**"category" bson**:**"category" example**:**"0"` // Number of the category to which the Post belongs  
 Text string `json**:**"text,omitempty" bson**:**"text,omitempty" example**:**"Awesome text" minLength**:**"4"` // Content of the Post  
 Votes Votes `json**:**"votes" bson**:**"votes"` // List of all the votes put by users on the post  
 Comments []\*PostComment `json**:**"comments" bson**:**"comments"` // List of all comments left by users under the post  
 Created string `json**:**"created" bson**:**"created" example**:**"2006-01-02T15:04:05.999Z" format**:**"date-time"` // Date the Post was created  
 UpvotePercentage int `json**:**"upvotePercentage" bson**:**"upvotePercentage" example**:**"75" minimum**:**"0" maximum**:**"100"` // Percentage of positive Votes to Post  
}  
  
type Posts []\*Post  
  
// PostPayload model info  
//  
// @Description PostPayload contains the necessary information to create a post  
type PostPayload struct {  
 Type PostType `json**:**"type"` // link or text  
 Title string `json**:**"title" example**:**"Awesome title"`  
 URL string `json**:**"url,omitempty" example**:**"http://localhost:8080/"`  
 Category PostCategory `json**:**"category" example**:**"0"` // Number of the category to which the Post belongs  
 Text string `json**:**"text,omitempty" example**:**"Awesome text" minLength**:**"4"` // Content of the Post  
}  
  
func NewPost(author jwt.TokenPayload, payload PostPayload) \*Post {  
 newPost := &Post{  
 ID**:** users.ID(uuid.New().String()),  
 Score**:** 1,  
 Views**:** 1,  
 Type**:** payload.Type,  
 Title**:** payload.Title,  
 Author**:** author,  
 Category**:** payload.Category,  
 Text**:** payload.Text,  
 Votes**:** Votes{author.ID**:** NewPostVote(author.ID, upVote)},  
 Comments**:** make([]\*PostComment, 0),  
 Created**:** time.Now().Format(TimeFormat),  
 UpvotePercentage**:** 100,  
 }  
 if newPost.Type == WithLink {  
 newPost.URL = payload.URL  
 }  
  
 return newPost  
}  
  
func (p \*Post) AddComment(author jwt.TokenPayload, commentBody string) \*PostComment {  
 newComment := NewPostComment(author, commentBody)  
 p.Comments = append(p.Comments, newComment)  
  
 return newComment  
}  
  
func (p \*Post) DeleteComment(commentID users.ID) error {  
 lenBeforeDelete := len(p.Comments)  
 p.Comments = slices.DeleteFunc(p.Comments, func(comment \*PostComment) bool {  
 return commentID == comment.ID  
 })  
 if lenBeforeDelete == len(p.Comments) {  
 return errs.ErrCommentNotFound  
 }  
  
 return nil  
}  
  
func (p \*Post) Upvote(userID users.ID) (\*PostVote, bool) {  
 defer p.updateUpvotePercentage()  
 vote, ok := p.getVoteByUserID(userID)  
 if !ok {  
 vote = NewPostVote(userID, upVote)  
 p.Votes[userID] = vote  
 p.Score++  
 return vote, true  
 }  
 if vote.Vote == downVote {  
 vote.Vote = upVote  
 p.Score += 2  
 }  
  
 return vote, false  
}  
  
func (p \*Post) Downvote(userID users.ID) (\*PostVote, bool) {  
 defer p.updateUpvotePercentage()  
 vote, ok := p.getVoteByUserID(userID)  
 if !ok {  
 vote = NewPostVote(userID, downVote)  
 p.Votes[userID] = vote  
 p.Score--  
 return vote, true  
 }  
 if vote.Vote == upVote {  
 vote.Vote = downVote  
 p.Score -= 2  
 }  
  
 return vote, false  
}  
  
func (p \*Post) Unvote(userID users.ID) error {  
 vote, ok := p.getVoteByUserID(userID)  
 if !ok {  
 return errs.ErrVoteNotFound  
 }  
  
 if vote.Vote == upVote {  
 p.Score--  
 } else {  
 p.Score++  
 }  
  
 delete(p.Votes, userID)  
 p.updateUpvotePercentage()  
  
 return nil  
}  
  
func (p \*Post) updateUpvotePercentage() {  
 totalVotes := len(p.Votes)  
 if totalVotes == 0 {  
 p.UpvotePercentage = 0  
 return  
 }  
 p.UpvotePercentage = ((p.Score + totalVotes) \* 100) / (totalVotes \* 2)  
}  
  
func (p \*Post) UpdateViews() \*Post {  
 p.Views++  
 return p  
}  
  
func (p \*Post) getVoteByUserID(userID users.ID) (\*PostVote, bool) {  
 postVote, ok := p.Votes[userID]  
 if !ok {  
 return nil, false  
 }  
  
 return postVote, true  
}

Для категории и типа поста были реализованы методы кастомной сериализации-десериализации(маршалинга-анмаршалига), поскольку в коде эти типы представлены как enum(тип int), а фронтенд ожидает строковый тип этих полей

**post\_attributes.go**

package posts  
  
import (  
 "encoding/json"  
 "fmt"  
 "regexp"  
 "time"  
  
 "github.com/google/uuid"  
 "go.mongodb.org/mongo-driver/bson"  
 "go.mongodb.org/mongo-driver/bson/bsontype"  
 "go.mongodb.org/mongo-driver/x/bsonx/bsoncore"  
  
 "github.com/Benzogang-Tape/Reddit/internal/models/errs"  
 "github.com/Benzogang-Tape/Reddit/internal/models/jwt"  
 "github.com/Benzogang-Tape/Reddit/internal/models/users"  
)  
  
// Vote type  
//  
// @Description Vote is an integer(1 or -1) representing the user's reaction to the Post  
type Vote int  
  
// PostCategory type  
//  
// @Description PostCategory is an integer representing the category to which post belongs  
type PostCategory int  
  
// PostType type  
//  
// @Description PostType is an integer(0 or 1) representing the type of the Post  
type PostType int  
type Votes map[users.ID]\*PostVote  
  
// Comment model info  
//  
// @Description Comment contains the text of the comment on Post  
type Comment struct {  
 Body string `json**:**"comment" example**:**"Some comment body example" minLength**:**"4"`  
}  
  
// PostComment model info  
//  
// @Description PostComment contains all information about a specific comment on a Post  
type PostComment struct {  
 Created string `json**:**"created" bson**:**"created" example**:**"2006-01-02T15:04:05.999Z" format**:**"date-time"` // Date the comment was created  
 Author jwt.TokenPayload `json**:**"author" bson**:**"author"`  
 Body string `json**:**"body" bson**:**"body" example**:**"Some comment body example" minLength**:**"4"` // Content of the comment  
 ID users.ID `json**:**"id" bson**:**"uuid" example**:**"12345678-9abc-def1-2345-6789abcdef12" minLength**:**"36" maxLength**:**"36"`  
}  
  
// PostVote model info  
//  
// @Description PostVote is a structure storing user id and his/her Vote  
type PostVote struct {  
 // ID of the user who left the comment  
 UserID users.ID `json**:**"user" bson**:**"user" example**:**"12345678-9abc-def1-2345-6789abcdef12" minLength**:**"36" maxLength**:**"36"`  
 Vote Vote `json**:**"vote" bson**:**"vote" example**:**"-1"`  
}  
  
const (  
 downVote Vote = iota - 1  
 upVote Vote = iota  
  
 withLink = "link"  
 withText = "text"  
 music = "music"  
 funny = "funny"  
 videos = "videos"  
 programming = "programming"  
 news = "news"  
 fashion = "fashion"  
  
 CategoryCount int = 6  
 UUIDLength int = 36  
  
 TimeFormat = "2006-01-02T15:04:05.999Z"  
)  
  
const (  
 Music PostCategory = iota  
 Funny  
 Videos  
 Programming  
 News  
 Fashion  
)  
  
const (  
 WithLink PostType = iota  
 WithText  
)  
  
var (  
 URLTemplate = regexp.MustCompile(`^((([A**-**Za**-**z]{3,9}:(?://)?)(?:[-;:&=+$,\w]+@)?[A**-**Za**-**z0**-**9.-]+(:[0**-**9]+)?|(?:www.|[-;:&=+$,\w]+@)[A**-**Za**-**z0**-**9.-]+)((?:/[+~%/.\w-\_]\*)?\??(?:[-+=&;%@.\w\_]\*)#?(?:\w\*))?)$`)  
)  
  
var (  
 postCategories = map[PostCategory]string{  
 0**:** music,  
 1**:** funny,  
 2**:** videos,  
 3**:** programming,  
 4**:** news,  
 5**:** fashion,  
 }  
 postTypes = map[PostType]string{  
 0**:** withLink,  
 1**:** withText,  
 }  
)  
  
func (pc PostCategory) String() string {  
 return postCategories[pc]  
}  
  
func (pc \*PostCategory) UnmarshalJSON(category []byte) error {  
 var s string  
 if err := json.Unmarshal(category, &s); err != nil {  
 return err  
 }  
  
 ctgry, err := StringToPostCategory(s)  
 if err != nil {  
 return err  
 }  
 \*pc = ctgry  
  
 return nil  
}  
  
func (pc \*PostCategory) UnmarshalBSONValue(bt bsontype.Type, category []byte) error {  
 if bt != bson.TypeString {  
 return fmt.Errorf("invalid bson postCategory type '%s'", bt.String())  
 }  
 cat, \_, ok := bsoncore.ReadString(category)  
 if !ok {  
 return fmt.Errorf("invalid bson postCategory value")  
 }  
  
 ctgry, err := StringToPostCategory(cat)  
 if err != nil {  
 return err  
 }  
 \*pc = ctgry  
  
 return nil  
}  
  
func (pc PostCategory) MarshalJSON() ([]byte, error) {  
 return json.Marshal(pc.String())  
}  
  
func (pc PostCategory) MarshalBSONValue() (bsontype.Type, []byte, error) {  
 return bson.MarshalValue(pc.String())  
}  
  
func StringToPostCategory(s string) (PostCategory, error) {  
 var category PostCategory  
 switch s {  
 case music**:** category = Music  
 case funny**:** category = Funny  
 case videos**:** category = Videos  
 case programming**:** category = Programming  
 case news**:** category = News  
 case fashion**:** category = Fashion  
 default**:** return category, errs.ErrInvalidCategory  
 }  
  
 return category, nil  
}  
  
func (pt PostType) String() string {  
 return postTypes[pt]  
}  
  
func (pt \*PostType) UnmarshalJSON(postType []byte) error {  
 var s string  
 if err := json.Unmarshal(postType, &s); err != nil {  
 return err  
 }  
  
 switch s {  
 case withLink**:** \*pt = WithLink  
 case withText**:** \*pt = WithText  
 default**:** return errs.ErrInvalidPostType  
 }  
  
 return nil  
}  
  
func (pt \*PostType) UnmarshalBSONValue(bt bsontype.Type, postType []byte) error {  
 if bt != bson.TypeString {  
 return fmt.Errorf("invalid bson postType type '%s'", bt.String())  
 }  
 tp, \_, ok := bsoncore.ReadString(postType)  
 if !ok {  
 return fmt.Errorf("invalid bson postType value")  
 }  
  
 switch tp {  
 case withLink**:** \*pt = WithLink  
 case withText**:** \*pt = WithText  
 default**:** return errs.ErrInvalidPostType  
 }  
  
 return nil  
}  
  
func (pt PostType) MarshalJSON() ([]byte, error) {  
 return json.Marshal(pt.String())  
}  
  
func (pt PostType) MarshalBSONValue() (bsontype.Type, []byte, error) {  
 return bson.MarshalValue(pt.String())  
}  
  
func (v Votes) MarshalJSON() ([]byte, error) {  
 votes := make([]\*PostVote, 0, len(v))  
 for \_, postVote := range v {  
 votes = append(votes, postVote)  
 }  
  
 return json.Marshal(votes)  
}  
  
func (v Votes) MarshalBSONValue() (bsontype.Type, []byte, error) {  
 postVotes := make([]\*PostVote, 0, len(v))  
 for \_, vote := range v {  
 postVotes = append(postVotes, vote)  
 }  
  
 return bson.MarshalValue(postVotes)  
}  
  
func (v \*Votes) UnmarshalJSONValue(votes []byte) error {  
 postVotes := make([]\*PostVote, 0, len(votes))  
 if err := json.Unmarshal(votes, &postVotes); err != nil {  
 return err  
 }  
  
 for \_, postVote := range postVotes {  
 (\*v)[postVote.UserID] = postVote  
 }  
  
 return nil  
}  
  
func (v \*Votes) UnmarshalBSONValue(bt bsontype.Type, votes []byte) error {  
 if bt != bson.TypeArray {  
 return fmt.Errorf("invalid bson votes type '%s'", bt.String())  
 }  
  
 postVotes := make([]\*PostVote, 0)  
 if err := bson.UnmarshalValue(bson.TypeArray, votes, &postVotes); err != nil {  
 return err  
 }  
  
 \*v = map[users.ID]\*PostVote{}  
 for \_, postVote := range postVotes {  
 (\*v)[postVote.UserID] = postVote  
 }  
  
 return nil  
}  
  
func NewPostComment(author jwt.TokenPayload, commentBody string) \*PostComment {  
 return &PostComment{  
 ID**:** users.ID(uuid.New().String()),  
 Created**:** time.Now().Format(TimeFormat),  
 Author**:** author,  
 Body**:** commentBody,  
 }  
}  
  
func NewPostVote(userID users.ID, vote Vote) \*PostVote {  
 return &PostVote{  
 UserID**:** userID,  
 Vote**:** vote,  
 }  
}

**user.go**

package users  
  
import (  
 "github.com/google/uuid"  
)  
  
type Username string  
type ID string  
  
type User struct {  
 ID ID `schema**:**"-" json**:**"-"`  
 Username Username `schema**:**"username,required" json**:**"username"`  
 Password string `schema**:**"password,required" json**:**"password" minLength**:**"8" format**:**"password"`  
}  
  
// AuthUserInfo model info  
//  
// @Description AuthUserInfo stores User credentials contained in the JWT Session token.  
type AuthUserInfo struct {  
 Login Username `json**:**"username" example**:**"Valery\_Albertovich"`  
 Password string `json**:**"password" example**:**"want\_pizza" minLength**:**"8" format**:**"password"`  
}  
  
func NewUser(authInfo AuthUserInfo) \*User {  
 return &User{  
 ID**:** ID(uuid.New().String()),  
 Username**:** authInfo.Login,  
 Password**:** authInfo.Password,  
 }  
}

**session.go**

package jwt  
  
import (  
 "fmt"  
 "time"  
  
 jwt "github.com/dgrijalva/jwt-go"  
  
 "github.com/Benzogang-Tape/Reddit/internal/models/errs"  
 "github.com/Benzogang-Tape/Reddit/internal/models/users"  
)  
  
// Session model info  
//  
// @Description Session stores the JWT token of the session  
type Session struct {  
 Token string `json**:**"token" example**:**"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE3MzUyMzU3ODAsImlhdCI6MTczNDYzMDk4MCwidXNlciI6eyJ1c2VybmFtZSI6InRlc3RfdXNlciIsImlkIjoiZDNkNzc1YmEtYTFlZS00MTEwLTkwOTktMTA0ZDVkYzFkYzQ2In19.I\_3\_yHlH1QUuKavtx8xVN\_IRFMYXg3dYumzSrImA\_NM"`  
}  
  
// TokenPayload model info  
//  
// @Description TokenPayload stores the User payload contained in the JWT Session token  
type TokenPayload struct {  
 // User login  
 Login users.Username `json**:**"username" bson**:**"username" example**:**"test\_user"`  
 // User id  
 ID users.ID `json**:**"id" bson**:**"uuid" example**:**"12345678-9abc-def1-2345-6789abcdef12" minLength**:**"36" maxLength**:**"36"`  
}  
  
const (  
 SessLifespan = 24 \* time.Hour \* 7  
)  
  
func NewSession(payload TokenPayload) (\*Session, error) {  
 token := jwt.NewWithClaims(jwt.SigningMethodHS256, jwt.MapClaims{  
 "user"**:** payload,  
 "iat"**:** time.Now().Unix(),  
 "exp"**:** time.Now().Add(SessLifespan).Unix(),  
 })  
  
 tokenString, err := token.SignedString(secretKeyProvider())  
 if err != nil {  
 return nil, err  
 }  
 return &Session{  
 Token**:** tokenString,  
 }, nil  
}  
  
func (s \*Session) ValidateToken() (\*TokenPayload, error) {  
 hashSecretGetter := func(token \*jwt.Token) (any, error) {  
 method, ok := token.Method.(\*jwt.SigningMethodHMAC)  
 if !ok || method.Alg() != "HS256" {  
 return nil, fmt.Errorf("bad sign method")  
 }  
 return secretKeyProvider(), nil  
 }  
 token, err := jwt.Parse(s.Token, hashSecretGetter)  
 if err != nil || !token.Valid {  
 return nil, errs.ErrBadToken  
 }  
  
 payload, ok := token.Claims.(jwt.MapClaims)  
 if !ok {  
 return nil, errs.ErrNoPayload  
 }  
 dataFromToken, ok := payload["user"].(map[string]any)  
 if !ok {  
 return nil, errs.ErrBadToken  
 }  
  
 return &TokenPayload{  
 Login**:** users.Username(dataFromToken["username"].(string)),  
 ID**:** users.ID(dataFromToken["id"].(string)),  
 }, nil  
}

**secret\_conf.go**

package jwt  
  
import (  
 "errors"  
)  
  
type favContextKey struct{}  
  
var (  
 errEmptySecret = errors.New("jwt secret is empty")  
 Payload favContextKey = struct{}{}  
 secretKeyProvider func() []byte  
)  
  
func SetJWTSecret(secret string) error {  
 if secret == "" {  
 return errEmptySecret  
 }  
  
 secretKeyProvider = func() []byte {  
 return []byte(secret)  
 }  
  
 return nil  
}

Все основные ошибки приложения были вынесены в отдельный пакет errs

**errors.go**

package errs  
  
import (  
 "encoding/json"  
 "errors"  
 "fmt"  
 "strings"  
)  
  
var (  
 ErrNoUser = errors.New("user not found")  
 ErrNoSession = errors.New("session not found")  
 ErrInternalServerError = errors.New("internal server error")  
 ErrBadPass = errors.New("invalid password")  
 ErrUserExists = errors.New("username already exist")  
 ErrBadToken = errors.New("bad token")  
 ErrNoPayload = errors.New("no payload")  
 ErrBadPayload = errors.New("bad payload")  
 ErrInvalidURL = errors.New("url is invalid")  
 ErrResponseError = errors.New("response generation error")  
 ErrPostNotFound = errors.New("post not found")  
 ErrCommentNotFound = errors.New("comment not found")  
 ErrBadID = errors.New("bad id")  
 ErrInvalidPostID = errors.New("invalid post id")  
 ErrInvalidCommentID = errors.New("invalid comment id")  
 ErrInvalidCategory = errors.New("invalid category")  
 ErrInvalidPostType = errors.New("invalid post type")  
 ErrVoteNotFound = errors.New("no votes from the requested user")  
 ErrBadCommentBody = errors.New("comment body is required")  
 ErrUnknownPayload = errors.New("unknown payload")  
 ErrUnknownError = errors.New("unknown error")  
)  
  
type RespError interface {  
 Marshal() ([]byte, error)  
 Error() string  
}  
  
// SimpleErr model info  
//  
// @Description SimpleErr stores a brief description of an error  
type SimpleErr struct {  
 Message any `json**:**"message"` // Any type  
}  
  
// ComplexErr model info  
//  
// @Description ComplexErr contains a more detailed description of the error, including the location and cause of the error  
type ComplexErr struct {  
 Location any `json**:**"location"` // Any type  
 Param any `json**:**"param"` // Any type  
 Value any `json**:**"value"` // Any type  
 Msg any `json**:**"msg"` // Any type  
}  
  
// ComplexErrArr model info  
//  
// @Description ComplexErrArr is an array of ComplexErr returned in case of a non-obvious error  
type ComplexErrArr struct {  
 Errs []ComplexErr `json**:**"errors"`  
}  
  
func (s SimpleErr) Marshal() ([]byte, error) {  
 return json.Marshal(s)  
}  
  
func (s SimpleErr) Error() string {  
 return s.Message.(string)  
}  
  
func (c ComplexErr) Marshal() ([]byte, error) {  
 complexErrs := ComplexErrArr{  
 Errs**:** []ComplexErr{c},  
 }  
  
 return json.Marshal(complexErrs)  
}  
  
func (c ComplexErr) Error() string {  
 return fmt.Sprintf("location: %s\nparam: %s\nvalue: %s\nmsg: %s\n",  
 c.Location.(string),  
 c.Param.(string),  
 c.Value.(string),  
 c.Msg.(string),  
 )  
}  
  
func (ca ComplexErrArr) Marshal() ([]byte, error) {  
 return json.Marshal(ca)  
}  
  
func (ca ComplexErrArr) Error() string {  
 b := strings.Builder{}  
 for \_, c := range ca.Errs {  
 fmt.Fprintf(&b, "%s\n", c.Error())  
 }  
  
 return b.String()  
}  
  
func NewSimpleErr(message any) SimpleErr {  
 return SimpleErr{  
 Message**:** message,  
 }  
}  
  
func NewComplexErrArr(err ...ComplexErr) ComplexErrArr {  
 return ComplexErrArr{  
 Errs**:** err,  
 }  
}

In-memory репозитории для MVP выглядели следующим образом:

**posts\_repo.go**

package inmem  
  
import (  
 "cmp"  
 "context"  
 "slices"  
 "sync"  
  
 "github.com/pkg/errors"  
  
 "github.com/Benzogang-Tape/Reddit/internal/models/errs"  
 "github.com/Benzogang-Tape/Reddit/internal/models/jwt"  
 "github.com/Benzogang-Tape/Reddit/internal/models/posts"  
 "github.com/Benzogang-Tape/Reddit/internal/models/users"  
)  
  
type PostRepo struct {  
 storage []\*posts.Post  
 mu \*sync.RWMutex  
}  
  
func NewPostRepo() \*PostRepo {  
 return &PostRepo{  
 storage**:** make([]\*posts.Post, 0),  
 mu**:** &sync.RWMutex{},  
 }  
}  
  
func (p \*PostRepo) GetAllPosts(ctx context.Context) ([]\*posts.Post, error) { //nolint:unparam  
 postList := make([]\*posts.Post, 0, len(p.storage))  
 p.mu.RLock()  
 defer p.mu.RUnlock()  
 for \_, post := range p.storage {  
 postList = append(postList, &(\*post))  
 }  
  
 return postList, nil  
}  
  
func (p \*PostRepo) GetPostsByCategory(ctx context.Context, postCategory posts.PostCategory) ([]\*posts.Post, error) { //nolint:unparam  
 postList := make([]\*posts.Post, 0, len(p.storage)/posts.CategoryCount)  
 p.mu.RLock()  
 defer p.mu.RUnlock()  
 for \_, post := range p.storage {  
 if post.Category == postCategory {  
 postList = append(postList, &(\*post))  
 }  
 }  
  
 return postList, nil  
}  
  
func (p \*PostRepo) GetPostsByUser(ctx context.Context, userLogin users.Username) ([]\*posts.Post, error) { //nolint:unparam  
 postList := make([]\*posts.Post, 0)  
 p.mu.RLock()  
 defer p.mu.RUnlock()  
 for \_, post := range p.storage {  
 if post.Author.Login == userLogin {  
 postList = append(postList, &(\*post))  
 }  
 }  
  
 return postList, nil  
}  
  
func (p \*PostRepo) GetPostByID(ctx context.Context, postID users.ID) (\*posts.Post, error) { //nolint:unparam  
 source := "GetPostByID"  
 post, err := p.getPostByID(postID)  
 if err != nil {  
 return nil, errors.Wrap(err, source)  
 }  
   
 return post, nil  
}  
  
func (p \*PostRepo) CreatePost(ctx context.Context, postPayload posts.PostPayload) (\*posts.Post, error) {  
 author, ok := ctx.Value(jwt.Payload).(\*jwt.TokenPayload)  
 if !ok {  
 return nil, errs.ErrBadPayload  
 }  
  
 defer p.sortPosts()  
  
 newPost := posts.NewPost(\*author, postPayload)  
 p.mu.Lock()  
 defer p.mu.Unlock()  
 p.storage = append(p.storage, newPost)  
  
 return &(\*newPost), nil  
}  
  
func (p \*PostRepo) DeletePost(ctx context.Context, postID users.ID) error { //nolint:unparam  
 lenBeforeDelete := len(p.storage)  
 p.mu.Lock()  
 p.storage = slices.DeleteFunc(p.storage, func(post \*posts.Post) bool {  
 return post.ID == postID  
 })  
 p.mu.Unlock()  
 if lenBeforeDelete == len(p.storage) {  
 return errs.ErrPostNotFound  
 }  
  
 return nil  
}  
  
func (p \*PostRepo) AddComment(ctx context.Context, post \*posts.Post, comment posts.Comment) (\*posts.Post, error) {  
 author, ok := ctx.Value(jwt.Payload).(\*jwt.TokenPayload)  
 if !ok {  
 return nil, errs.ErrBadPayload  
 }  
  
 post.AddComment(\*author, comment.Body)  
  
 return &(\*post), nil  
}  
  
func (p \*PostRepo) DeleteComment(ctx context.Context, post \*posts.Post, commentID users.ID) (\*posts.Post, error) { //nolint:unparam  
 source := "DeleteComment"  
 if err := post.DeleteComment(commentID); err != nil {  
 return nil, errors.Wrap(err, source)  
 }  
  
 return &(\*post), nil  
}  
  
func (p \*PostRepo) Upvote(ctx context.Context, post \*posts.Post) (\*posts.Post, error) {  
 author, ok := ctx.Value(jwt.Payload).(\*jwt.TokenPayload)  
 if !ok {  
 return nil, errs.ErrBadPayload  
 }  
  
 post.Upvote(author.ID)  
 p.sortPosts()  
  
 return &(\*post), nil  
}  
  
func (p \*PostRepo) Downvote(ctx context.Context, post \*posts.Post) (\*posts.Post, error) {  
 author, ok := ctx.Value(jwt.Payload).(\*jwt.TokenPayload)  
 if !ok {  
 return nil, errs.ErrBadPayload  
 }  
  
 post.Downvote(author.ID)  
 p.sortPosts()  
  
 return &(\*post), nil  
}  
  
func (p \*PostRepo) Unvote(ctx context.Context, post \*posts.Post) (\*posts.Post, error) {  
 source := "Unvote"  
 author, ok := ctx.Value(jwt.Payload).(\*jwt.TokenPayload)  
 if !ok {  
 return nil, errs.ErrBadPayload  
 }  
  
 if err := post.Unvote(author.ID); err != nil {  
 return nil, errors.Wrap(err, source)  
 }  
  
 p.sortPosts()  
  
 return &(\*post), nil  
}  
  
func (p \*PostRepo) getPostByID(postID users.ID) (\*posts.Post, error) {  
 p.mu.RLock()  
 defer p.mu.RUnlock()  
 postIdx := slices.IndexFunc(p.storage, func(post \*posts.Post) bool {  
 return post.ID == postID  
 })  
 if postIdx == -1 {  
 return nil, errs.ErrPostNotFound  
 }  
  
 return &(\*p.storage[postIdx]), nil  
}  
  
func (p \*PostRepo) UpdateViews(ctx context.Context, postID users.ID) error { //nolint:unparam  
 // stub  
 return nil  
}  
  
func (p \*PostRepo) sortPosts() {  
 p.mu.Lock()  
 defer p.mu.Unlock()  
 slices.SortStableFunc(p.storage, func(a, b \*posts.Post) int {  
 return -cmp.Compare(a.Score, b.Score)  
 })  
}

**users\_repo.go**

package inmem  
  
import (  
 "context"  
 "sync"  
  
 "github.com/pkg/errors"  
  
 "github.com/Benzogang-Tape/Reddit/internal/models/errs"  
 "github.com/Benzogang-Tape/Reddit/internal/models/users"  
)  
  
type UserRepo struct {  
 storage map[users.Username]\*users.User  
 mu \*sync.RWMutex  
}  
  
func NewUserRepo() \*UserRepo {  
 return &UserRepo{  
 storage**:** make(map[users.Username]\*users.User, 42),  
 mu**:** &sync.RWMutex{},  
 }  
}  
  
func (repo \*UserRepo) Authorize(ctx context.Context, authData users.AuthUserInfo) (\*users.User, error) { //nolint:unparam  
 source := "Authorize"  
 repo.mu.RLock()  
 user, ok := repo.storage[authData.Login]  
 repo.mu.RUnlock()  
  
 if !ok {  
 return nil, errors.Wrap(errs.ErrNoUser, source)  
 }  
 if user.Password != authData.Password {  
 return nil, errors.Wrap(errs.ErrBadPass, source)  
 }  
  
 return user, nil  
}  
  
func (repo \*UserRepo) RegisterUser(ctx context.Context, authData users.AuthUserInfo) (\*users.User, error) { //nolint:unparam  
 source := "RegisterUser"  
 repo.mu.RLock()  
 \_, ok := repo.storage[authData.Login]  
 repo.mu.RUnlock()  
 if ok {  
 return nil, errors.Wrap(errs.ErrUserExists, source)  
 }  
  
 newUser := repo.createUser(authData)  
  
 return newUser, nil  
}  
  
func (repo \*UserRepo) createUser(authData users.AuthUserInfo) \*users.User {  
 newUser := users.NewUser(authData)  
 repo.mu.Lock()  
 defer repo.mu.Unlock()  
 repo.storage[newUser.Username] = newUser  
  
 return newUser  
}

**session\_repo.go**

package inmem  
  
import (  
 "context"  
 "sync"  
  
 "github.com/Benzogang-Tape/Reddit/internal/models/errs"  
 "github.com/Benzogang-Tape/Reddit/internal/models/jwt"  
)  
  
type SessionRepo struct {  
 storage map[string]\*jwt.TokenPayload  
 mu \*sync.RWMutex  
}  
  
func NewSessionRepo() \*SessionRepo {  
 return &SessionRepo{  
 storage**:** make(map[string]\*jwt.TokenPayload),  
 mu**:** &sync.RWMutex{},  
 }  
}  
  
// Need TTL  
  
func (s \*SessionRepo) CreateSession(ctx context.Context, session \*jwt.Session, payload \*jwt.TokenPayload) (\*jwt.Session, error) { //nolint:unparam  
 key := session.Token  
 s.mu.Lock()  
 defer s.mu.Unlock()  
 s.storage[key] = payload  
 return session, nil  
}  
  
func (s \*SessionRepo) CheckSession(ctx context.Context, sess \*jwt.Session) (\*jwt.TokenPayload, error) { //nolint:unparam  
 key := sess.Token  
 s.mu.RLock()  
 defer s.mu.RUnlock()  
 payload, ok := s.storage[key]  
 if !ok {  
 return nil, errs.ErrNoSession  
 }  
  
 return payload, nil  
}

Репозитории реализовывали интерфейсы, описанные в сервисе. Таким образом достигалась инверсия зависимостей (DI), один из принципов SOLID. Более высокие абстракции(слои) не зависели от конкретной реализации более низких, причём оба зависели от абстракций. Абстракции в свою очередь не зависели от конкретной реализации, а реализация зависела от абстракций.

Хоть go и не ООП язык, роль абстракций в нём выполняют интерфейсы, с помощью которых достигается полиморфизм по аналогии с ООП. Go предоставляет удобный механизм утиной типизации, что позволяет просто реализовать все необходимые методы у структуры, прописанные в интерфейсе, для того чтобы структура удовлетворяла этому интерфейсу. Согласно best practice интерфейсы описывались в месте использования.

**Сервисный слой:**

**post.go**

package service  
  
import (  
 "context"  
  
 "github.com/pkg/errors"  
  
 "github.com/Benzogang-Tape/Reddit/internal/models/errs"  
 "github.com/Benzogang-Tape/Reddit/internal/models/posts"  
 "github.com/Benzogang-Tape/Reddit/internal/models/users"  
)  
  
type PostStorage interface {  
 GetAllPosts(ctx context.Context) ([]\*posts.Post, error)  
 GetPostsByCategory(ctx context.Context, postCategory posts.PostCategory) ([]\*posts.Post, error)  
 GetPostsByUser(ctx context.Context, userLogin users.Username) ([]\*posts.Post, error)  
 GetPostByID(ctx context.Context, postID users.ID) (\*posts.Post, error)  
 CreatePost(ctx context.Context, postPayload posts.PostPayload) (\*posts.Post, error)  
 DeletePost(ctx context.Context, postID users.ID) error  
}  
  
type PostActions interface {  
 AddComment(ctx context.Context, post \*posts.Post, comment posts.Comment) (\*posts.Post, error)  
 DeleteComment(ctx context.Context, post \*posts.Post, commentID users.ID) (\*posts.Post, error)  
 Upvote(ctx context.Context, post \*posts.Post) (\*posts.Post, error)  
 Downvote(ctx context.Context, post \*posts.Post) (\*posts.Post, error)  
 Unvote(ctx context.Context, post \*posts.Post) (\*posts.Post, error)  
 UpdateViews(ctx context.Context, postID users.ID) error  
}  
  
type PostHandler struct {  
 repo PostStorage  
 actionController PostActions  
}  
  
func NewPostHandler(storage PostStorage, actions PostActions) \*PostHandler {  
 return &PostHandler{  
 repo**:** storage,  
 actionController**:** actions,  
 }  
}  
  
func (p \*PostHandler) GetAllPosts(ctx context.Context) ([]\*posts.Post, error) {  
 source := "GetAllPosts"  
 postList, err := p.repo.GetAllPosts(ctx)  
 if err != nil {  
 return nil, errors.Wrap(err, source)  
 }  
  
 return postList, nil  
}  
  
func (p \*PostHandler) GetPostsByCategory(ctx context.Context, postCategory posts.PostCategory) ([]\*posts.Post, error) {  
 source := "GetPostsByCategory"  
 postList, err := p.repo.GetPostsByCategory(ctx, postCategory)  
 if err != nil {  
 return nil, errors.Wrap(err, source)  
 }  
  
 return postList, nil  
}  
  
func (p \*PostHandler) GetPostsByUser(ctx context.Context, userLogin users.Username) ([]\*posts.Post, error) {  
 source := "GetPostsByUser"  
 postList, err := p.repo.GetPostsByUser(ctx, userLogin)  
 if err != nil {  
 return nil, errors.Wrap(err, source)  
 }  
  
 return postList, nil  
}  
  
func (p \*PostHandler) GetPostByID(ctx context.Context, postID users.ID) (\*posts.Post, error) {  
 source := "GetPostByID"  
 post, err := p.repo.GetPostByID(ctx, postID)  
 if err != nil {  
 return post, errors.Wrap(err, source)  
 }  
  
 if err = p.actionController.UpdateViews(ctx, postID); err != nil {  
 return nil, errors.Wrap(err, source)  
 }  
  
 return post.UpdateViews(), nil  
}  
  
func (p \*PostHandler) CreatePost(ctx context.Context, postPayload posts.PostPayload) (\*posts.Post, error) {  
 source := "CreatePost"  
 if postPayload.Type == posts.WithLink && !posts.URLTemplate.MatchString(postPayload.URL) {  
 return nil, errors.Wrap(errs.ErrInvalidURL, source)  
 }  
  
 return p.repo.CreatePost(ctx, postPayload)  
}  
  
func (p \*PostHandler) DeletePost(ctx context.Context, postID users.ID) error {  
 source := "DeletePost"  
 if err := p.repo.DeletePost(ctx, postID); err != nil {  
 return errors.Wrap(err, source)  
 }  
  
 return nil  
}  
  
func (p \*PostHandler) Upvote(ctx context.Context, postID users.ID) (\*posts.Post, error) {  
 source := "Upvote"  
 post, err := p.repo.GetPostByID(ctx, postID)  
 if err != nil {  
 return nil, errors.Wrap(err, source)  
 }  
  
 post, err = p.actionController.Upvote(ctx, post)  
 if err != nil {  
 return post, errors.Wrap(err, source)  
 }  
  
 return post, nil  
}  
  
func (p \*PostHandler) Downvote(ctx context.Context, postID users.ID) (\*posts.Post, error) {  
 source := "Downvote"  
 post, err := p.repo.GetPostByID(ctx, postID)  
 if err != nil {  
 return nil, errors.Wrap(err, source)  
 }  
  
 post, err = p.actionController.Downvote(ctx, post)  
 if err != nil {  
 return post, errors.Wrap(err, source)  
 }  
  
 return post, nil  
}  
  
func (p \*PostHandler) Unvote(ctx context.Context, postID users.ID) (\*posts.Post, error) {  
 source := "Unvote"  
 post, err := p.repo.GetPostByID(ctx, postID)  
 if err != nil {  
 return nil, errors.Wrap(err, source)  
 }  
  
 post, err = p.actionController.Unvote(ctx, post)  
 if err != nil {  
 return post, errors.Wrap(err, source)  
 }  
  
 return post, nil  
}  
  
func (p \*PostHandler) AddComment(ctx context.Context, postID users.ID, comment posts.Comment) (\*posts.Post, error) {  
 source := "AddComment"  
 if comment.Body == "" {  
 return nil, errs.ErrBadCommentBody  
 }  
  
 post, err := p.repo.GetPostByID(ctx, postID)  
 if err != nil {  
 return nil, errors.Wrap(err, source)  
 }  
  
 post, err = p.actionController.AddComment(ctx, post, comment)  
 if err != nil {  
 return post, errors.Wrap(err, source)  
 }  
  
 return post, nil  
}  
  
func (p \*PostHandler) DeleteComment(ctx context.Context, postID, commentID users.ID) (\*posts.Post, error) {  
 source := "DeleteComment"  
 post, err := p.repo.GetPostByID(ctx, postID)  
 if err != nil {  
 return nil, errors.Wrap(err, source)  
 }  
  
 post, err = p.actionController.DeleteComment(ctx, post, commentID)  
 if err != nil {  
 return post, errors.Wrap(err, source)  
 }  
  
 return post, nil  
}

**user.go**

package service  
  
import (  
 "context"  
  
 "github.com/pkg/errors"  
  
 "github.com/Benzogang-Tape/Reddit/internal/models/jwt"  
 "github.com/Benzogang-Tape/Reddit/internal/models/users"  
)  
  
type UserStorage interface {  
 RegisterUser(ctx context.Context, authData users.AuthUserInfo) (\*users.User, error)  
 Authorize(ctx context.Context, authData users.AuthUserInfo) (\*users.User, error)  
}  
  
type UserHandler struct {  
 Repo UserStorage  
}  
  
func NewUserHandler(u UserStorage) \*UserHandler {  
 return &UserHandler{  
 Repo**:** u,  
 }  
}  
  
func (h \*UserHandler) Register(ctx context.Context, authData users.AuthUserInfo) (\*jwt.TokenPayload, error) {  
 source := "Register"  
 newUser, err := h.Repo.RegisterUser(ctx, authData)  
 if err != nil {  
 err = errors.Wrap(err, source)  
 return nil, err  
 }  
  
 return &jwt.TokenPayload{  
 Login**:** newUser.Username,  
 ID**:** newUser.ID,  
 }, nil  
}  
  
func (h \*UserHandler) Authorize(ctx context.Context, authData users.AuthUserInfo) (\*jwt.TokenPayload, error) {  
 source := "Authorize"  
 user, err := h.Repo.Authorize(ctx, authData)  
 if err != nil {  
 err = errors.Wrap(err, source)  
 return nil, err  
 }  
  
 return &jwt.TokenPayload{  
 Login**:** user.Username,  
 ID**:** user.ID,  
 }, nil  
}

**session.go**

package service  
  
import (  
 "context"  
  
 "github.com/pkg/errors"  
  
 "github.com/Benzogang-Tape/Reddit/internal/models/errs"  
 "github.com/Benzogang-Tape/Reddit/internal/models/jwt"  
)  
  
type SessionManager interface {  
 CreateSession(ctx context.Context, session \*jwt.Session, payload \*jwt.TokenPayload) (\*jwt.Session, error)  
 CheckSession(ctx context.Context, session \*jwt.Session) (\*jwt.TokenPayload, error)  
}  
  
//go:generate mockgen -source=session.go -destination=../storage/mocks/sessions\_repo\_redis\_mock.go -package=mocks SessionAPI  
type SessionAPI interface {  
 New(ctx context.Context) (\*jwt.Session, error)  
 Verify(ctx context.Context, session \*jwt.Session) (\*jwt.TokenPayload, error)  
}  
  
type SessionHandler struct {  
 manager SessionManager  
}  
  
func NewSessionHandler(mngr SessionManager) \*SessionHandler {  
 return &SessionHandler{  
 manager**:** mngr,  
 }  
}  
  
func (s \*SessionHandler) New(ctx context.Context) (\*jwt.Session, error) {  
 source := "New session"  
 payload, ok := ctx.Value(jwt.Payload).(jwt.TokenPayload)  
 if !ok {  
 return nil, errs.ErrBadPayload  
 }  
  
 sess, err := jwt.NewSession(payload)  
 if err != nil {  
 return nil, errors.Wrap(err, source)  
 }  
  
 sess, err = s.manager.CreateSession(ctx, sess, &payload)  
 if err != nil {  
 return nil, errors.Wrap(err, source)  
 }  
  
 return sess, nil  
}  
  
func (s \*SessionHandler) Verify(ctx context.Context, session \*jwt.Session) (\*jwt.TokenPayload, error) {  
 source := "Verify session"  
 payload, err := s.manager.CheckSession(ctx, session)  
 if err != nil {  
 return nil, errors.Wrap(err, source)  
 }  
  
 return payload, nil  
}

Сервис реализовывал интерфейсы, описанные в транспортном слое.

Транспортный слой принимает запросы по http и вызывает сервисный слой, который в свою очередь вызывает слой репозитория, который в свою очередь общается с доменным слоем м взаимодействует с базами данных. Таким образом полный путь запроса выглядит так: когда приходит новый запрос, он приходит на транспортный уровень, проходит через middleware, и прокидывается в слои ниже. После обработки запроса на нижних слоях он возвращается в обратном направлении снизу вверх, доходит до транспортного слоя, и транспортный слой возвращает ответ на запрос клиенту. В нашем случае для транспортного слоя было реализовано 3 middleware: логирующий, проверяющий сессии и recover middleware.

**recover.go**

package middleware  
  
import (  
 "net/http"  
  
 "go.uber.org/zap"  
  
 "github.com/Benzogang-Tape/Reddit/internal/models/errs"  
)  
  
func Panic(next http.Handler, logger \*zap.SugaredLogger) http.Handler {  
 return http.HandlerFunc(func(w http.ResponseWriter, r \*http.Request) {  
 defer func() {  
 if err := recover(); err != nil {  
 logger.Errorw("panicMiddleware",  
 "method", r.Method,  
 "remote\_addr", r.RemoteAddr,  
 "url", r.URL.Path,  
 )  
  
 http.Error(w, errs.ErrInternalServerError.Error(), http.StatusInternalServerError)  
 logger.Infow("recovered", "cause", err)  
 }  
 }()  
 next.ServeHTTP(w, r)  
 })  
}

**jwt\_auth.go**

package middleware  
  
import (  
 "context"  
 "net/http"  
 "regexp"  
 "slices"  
 "strings"  
  
 "go.uber.org/zap"  
  
 "github.com/Benzogang-Tape/Reddit/internal/models/jwt"  
 "github.com/Benzogang-Tape/Reddit/internal/service"  
)  
  
type HTTPMethods []string  
type Endpoints map[\*regexp.Regexp]HTTPMethods  
  
var (  
 authUrls = Endpoints{  
 regexp.MustCompile(`^/api/posts$`)**:** {http.MethodPost},  
 regexp.MustCompile(`^/api/post/[0**-**9a**-**fA**-**F-]+$`)**:** {http.MethodPost},  
 regexp.MustCompile(`^/api/post/[0**-**9a**-**fA**-**F-]+/[0**-**9a**-**fA**-**F-]+$`)**:** {http.MethodDelete},  
 regexp.MustCompile(`^/api/post/[0**-**9a**-**fA**-**F-]+/upvote$`)**:** {http.MethodGet},  
 regexp.MustCompile(`^/api/post/[0**-**9a**-**fA**-**F-]+/downvote$`)**:** {http.MethodGet},  
 regexp.MustCompile(`^/api/post/[0**-**9a**-**fA**-**F-]+/unvote$`)**:** {http.MethodGet},  
 regexp.MustCompile(`^/api/post/[0**-**9a**-**fA**-**F-]+$`)**:** {http.MethodDelete},  
 }  
)  
  
func Auth(next http.Handler, sessMngr service.SessionAPI, logger \*zap.SugaredLogger) http.Handler {  
 return http.HandlerFunc(func(w http.ResponseWriter, r \*http.Request) {  
 var canBeWithoutAuth = true  
 for endpoint, methods := range authUrls {  
 if endpoint.MatchString(r.URL.Path) && slices.Contains(methods, r.Method) {  
 canBeWithoutAuth = false  
 break  
 }  
 }  
 if canBeWithoutAuth {  
 next.ServeHTTP(w, r)  
 return  
 }  
  
 session := &jwt.Session{  
 Token**:** strings.Split(r.Header.Get("Authorization"), " ")[1],  
 }  
 payload, err := sessMngr.Verify(r.Context(), session)  
 if err != nil {  
 logger.Warnw("Authorization failed",  
 "reason", err.Error(),  
 "remote\_addr", r.RemoteAddr,  
 "url", r.URL.Path,  
 )  
  
 http.Redirect(w, r, "/api/posts/", http.StatusFound)  
 return  
 }  
  
 next.ServeHTTP(w, r.WithContext(context.WithValue(r.Context(), jwt.Payload, payload)))  
 })  
}

**accesslog.go**

package middleware  
  
import (  
 "net/http"  
 "time"  
  
 "go.uber.org/zap"  
)  
  
func AccessLog(logger \*zap.SugaredLogger, next http.Handler) http.Handler {  
 return http.HandlerFunc(func(w http.ResponseWriter, r \*http.Request) {  
 start := time.Now()  
 next.ServeHTTP(w, r)  
 logger.Infow("New request",  
 "method", r.Method,  
 "remote\_addr", r.RemoteAddr,  
 "url", r.URL.Path,  
 "time", time.Since(start),  
 )  
 })  
}

Обработчики транспортного слоя:

**post.go**

package rest  
  
import (  
 "context"  
 "encoding/json"  
 "errors"  
 "io"  
 "net/http"  
 "unicode/utf8"  
  
 "github.com/gorilla/mux"  
 "go.uber.org/zap"  
  
 "github.com/Benzogang-Tape/Reddit/internal/models/errs"  
 "github.com/Benzogang-Tape/Reddit/internal/models/httpresp"  
 "github.com/Benzogang-Tape/Reddit/internal/models/posts"  
 "github.com/Benzogang-Tape/Reddit/internal/models/users"  
)  
  
//go:generate mockgen -source=post.go -destination=../../storage/mocks/posts\_repo\_mongoDB\_mock.go -package=mocks PostAPI  
type PostAPI interface {  
 GetAllPosts(ctx context.Context) ([]\*posts.Post, error)  
 GetPostsByCategory(ctx context.Context, postCategory posts.PostCategory) ([]\*posts.Post, error)  
 GetPostsByUser(ctx context.Context, userLogin users.Username) ([]\*posts.Post, error)  
 GetPostByID(ctx context.Context, postID users.ID) (\*posts.Post, error)  
 CreatePost(ctx context.Context, postPayload posts.PostPayload) (\*posts.Post, error)  
 DeletePost(ctx context.Context, postID users.ID) error  
 AddComment(ctx context.Context, postID users.ID, comment posts.Comment) (\*posts.Post, error)  
 DeleteComment(ctx context.Context, postID, commentID users.ID) (\*posts.Post, error)  
 Upvote(ctx context.Context, postID users.ID) (\*posts.Post, error)  
 Downvote(ctx context.Context, postID users.ID) (\*posts.Post, error)  
 Unvote(ctx context.Context, postID users.ID) (\*posts.Post, error)  
}  
  
type PostHandler struct {  
 logger \*zap.SugaredLogger  
 service PostAPI  
}  
  
func NewPostHandler(p PostAPI, logger \*zap.SugaredLogger) \*PostHandler {  
 return &PostHandler{  
 logger**:** logger,  
 service**:** p,  
 }  
}  
  
func validateID(alias string, params map[string]string) (id users.ID, err error) {  
 extractedID := params[alias]  
 if utf8.RuneCountInString(extractedID) != posts.UUIDLength {  
 return id, errs.ErrBadID  
 }  
  
 return users.ID(extractedID), nil  
}  
  
// GetAllPosts godoc  
//  
// @Summary Get all posts  
// @Description Get a list of posts of all users and threads  
// @Tags getting-posts  
// @ID get-all-posts  
// @Produce json  
// @Success 200 {array} posts.Post "Posts successfully received"  
// @Failure 500 {object} errs.SimpleErr "Internal server error"  
// @Router /posts/ [get]  
func (p \*PostHandler) GetAllPosts(w http.ResponseWriter, r \*http.Request) {  
 postList, err := p.service.GetAllPosts(r.Context())  
 if err != nil {  
 sendErrorResponse(w, http.StatusInternalServerError, errs.NewSimpleErr(errs.ErrUnknownError.Error()))  
 return  
 }  
  
 sendResponse(postList, w)  
}  
  
// CreatePost godoc  
//  
// @Summary Create a post  
// @Description Create a post of a specific type, category, and content  
// @Security ApiKeyAuth  
// @Tags managing-posts  
// @ID create-post  
// @Accept json  
// @Produce json  
// @Param post\_payload body posts.PostPayload true "Post data" validate(required)  
// @Success 201 {object} posts.Post "Post successfully created"  
// @Failure 400 "Bad payload"  
// @Failure 422 {object} errs.ComplexErrArr "Bad content"  
// @Failure 500 {object} errs.SimpleErr "Internal server error"  
// @Router /posts [post]  
func (p \*PostHandler) CreatePost(w http.ResponseWriter, r \*http.Request) {  
 defer r.Body.Close()  
 body, err := io.ReadAll(r.Body)  
 if err != nil {  
 w.WriteHeader(http.StatusBadRequest)  
 return  
 }  
 postPayload := posts.PostPayload{}  
 if err = json.Unmarshal(body, &postPayload); err != nil {  
 w.WriteHeader(http.StatusBadRequest)  
 return  
 }  
  
 newPost, err := p.service.CreatePost(r.Context(), postPayload)  
 switch {  
 case errors.Is(err, errs.ErrInvalidURL)**:** sendErrorResponse(w, http.StatusUnprocessableEntity, errs.NewComplexErrArr(errs.ComplexErr{  
 Location**:** "body",  
 Param**:** "url",  
 Value**:** postPayload.URL,  
 Msg**:** "is invalid",  
 }))  
 return  
 case err != nil**:** sendErrorResponse(w, http.StatusInternalServerError, errs.NewSimpleErr(errs.ErrUnknownError.Error()))  
 return  
 }  
  
 sendResponse(newPost, w, httpresp.WithStatusCode(http.StatusCreated))  
}  
  
// GetPostByID godoc  
//  
// @Summary Get a certain post  
// @Description Get information on a specific post by id  
// @Tags getting-posts  
// @ID get-post-by-id  
// @Produce json  
// @Param POST\_ID path string true "Post uuid" minlength(36) maxlength(36)  
// @Success 200 {object} posts.Post "Post successfully received"  
// @Failure 400 {object} errs.SimpleErr "Bad post id"  
// @Failure 404 {object} errs.SimpleErr "No posts with the provided id were found"  
// @Failure 500 {object} errs.SimpleErr "Internal server error"  
// @Router /post/{POST\_ID} [get]  
func (p \*PostHandler) GetPostByID(w http.ResponseWriter, r \*http.Request) {  
 postID, err := validateID("POST\_ID", mux.Vars(r))  
 if err != nil {  
 sendErrorResponse(w, http.StatusBadRequest, errs.NewSimpleErr(errs.ErrInvalidPostID.Error()))  
 return  
 }  
  
 post, err := p.service.GetPostByID(r.Context(), postID)  
 switch {  
 case errors.Is(err, errs.ErrPostNotFound)**:** sendErrorResponse(w, http.StatusNotFound, errs.NewSimpleErr(errs.ErrPostNotFound.Error()))  
 return  
 case err != nil**:** sendErrorResponse(w, http.StatusInternalServerError, errs.NewSimpleErr(errs.ErrUnknownError.Error()))  
 return  
 }  
  
 sendResponse(post, w)  
}  
  
// GetPostsByCategory godoc  
//  
// @Summary Get posts by category  
// @Description Get all posts belonging to a certain category  
// @Tags getting-posts  
// @ID get-posts-by-category  
// @Produce json  
// @Param CATEGORY\_NAME path string true "Category name"  
// @Success 200 {array} posts.Post "Posts successfully received"  
// @Failure 400 {object} errs.SimpleErr "Bad category(doesn't exist)"  
// @Failure 500 {object} errs.SimpleErr "Internal server error"  
// @Router /posts/{CATEGORY\_NAME} [get]  
func (p \*PostHandler) GetPostsByCategory(w http.ResponseWriter, r \*http.Request) {  
 postCategory, err := posts.StringToPostCategory(mux.Vars(r)["CATEGORY\_NAME"])  
 if err != nil {  
 sendErrorResponse(w, http.StatusBadRequest, errs.NewSimpleErr(errs.ErrInvalidCategory.Error()))  
 return  
 }  
  
 postList, err := p.service.GetPostsByCategory(r.Context(), postCategory)  
 if err != nil {  
 sendErrorResponse(w, http.StatusInternalServerError, errs.NewSimpleErr(errs.ErrUnknownError.Error()))  
 return  
 }  
  
 sendResponse(postList, w)  
}  
  
// GetPostsByUser godoc  
//  
// @Summary Get posts by user  
// @Description Get all posts of a certain user by his/her username  
// @Tags getting-posts  
// @ID get-posts-by-user  
// @Produce json  
// @Param USER\_LOGIN path string true "Username of user"  
// @Success 200 {array} posts.Post "Posts successfully received"  
// @Failure 400 {object} errs.SimpleErr "Bad username(doesn't exist)"  
// @Failure 500 {object} errs.SimpleErr "Internal server error"  
// @Router /user/{USER\_LOGIN} [get]  
func (p \*PostHandler) GetPostsByUser(w http.ResponseWriter, r \*http.Request) {  
 userLogin := users.Username(mux.Vars(r)["USER\_LOGIN"])  
 postList, err := p.service.GetPostsByUser(r.Context(), userLogin)  
 if err != nil {  
 sendErrorResponse(w, http.StatusInternalServerError, errs.NewSimpleErr(errs.ErrUnknownError.Error()))  
 return  
 }  
  
 sendResponse(postList, w)  
}  
  
// DeletePost godoc  
//  
// @Summary Delete a post  
// @Description Delete a specific post by its id  
// @Security ApiKeyAuth  
// @Tags managing-posts  
// @ID delete-post  
// @Param POST\_ID path string true "Post uuid" minlength(36) maxlength(36)  
// @Success 200 {object} errs.SimpleErr "Post successfully deleted"  
// @Failure 400 {object} errs.SimpleErr "Bad post id"  
// @Failure 404 {object} errs.SimpleErr "No posts with the provided id were found"  
// @Failure 500 {object} errs.SimpleErr "Internal server error"  
// @Router /post/{POST\_ID} [delete]  
func (p \*PostHandler) DeletePost(w http.ResponseWriter, r \*http.Request) {  
 postID, err := validateID("POST\_ID", mux.Vars(r))  
 if err != nil {  
 sendErrorResponse(w, http.StatusBadRequest, errs.NewSimpleErr(errs.ErrInvalidPostID.Error()))  
 return  
 }  
  
 err = p.service.DeletePost(r.Context(), postID)  
 switch {  
 case errors.Is(err, errs.ErrPostNotFound)**:** sendErrorResponse(w, http.StatusNotFound, errs.NewSimpleErr(errs.ErrPostNotFound.Error()))  
 return  
 case err != nil**:** sendErrorResponse(w, http.StatusInternalServerError, errs.NewSimpleErr(errs.ErrUnknownError.Error()))  
 return  
 }  
  
 sendErrorResponse(w, http.StatusOK, errs.NewSimpleErr("success"))  
}  
  
// Upvote godoc  
//  
// @Summary Vote up on a post  
// @Description Increase post rating by 1 vote  
// @Security ApiKeyAuth  
// @Tags voting-posts  
// @ID upvote-post  
// @Produce json  
// @Param POST\_ID path string true "Post uuid" minlength(36) maxlength(36)  
// @Success 200 {object} posts.Post "Successfully upvoted"  
// @Failure 400 {object} errs.SimpleErr "Bad post id"  
// @Failure 404 {object} errs.SimpleErr "No posts with the provided id were found"  
// @Failure 500 {object} errs.SimpleErr "Internal server error"  
// @Router /post/{POST\_ID}/upvote [get]  
func (p \*PostHandler) Upvote(w http.ResponseWriter, r \*http.Request) {  
 postID, err := validateID("POST\_ID", mux.Vars(r))  
 if err != nil {  
 sendErrorResponse(w, http.StatusBadRequest, errs.NewSimpleErr(errs.ErrInvalidPostID.Error()))  
 return  
 }  
  
 post, err := p.service.Upvote(r.Context(), postID)  
 switch {  
 case errors.Is(err, errs.ErrPostNotFound)**:** sendErrorResponse(w, http.StatusNotFound, errs.NewSimpleErr(errs.ErrPostNotFound.Error()))  
 return  
 case err != nil**:** sendErrorResponse(w, http.StatusInternalServerError, errs.NewSimpleErr(errs.ErrUnknownError.Error()))  
 return  
 }  
  
 sendResponse(post, w)  
}  
  
// Downvote godoc  
//  
// @Summary Vote down on a post  
// @Description Decrease post rating by 1 vote  
// @Security ApiKeyAuth  
// @Tags voting-posts  
// @ID downvote-post  
// @Produce json  
// @Param POST\_ID path string true "Post uuid" minlength(36) maxlength(36)  
// @Success 200 {object} posts.Post "Successfully downvoted"  
// @Failure 400 {object} errs.SimpleErr "Bad post id"  
// @Failure 404 {object} errs.SimpleErr "No posts with the provided id were found"  
// @Failure 500 {object} errs.SimpleErr "Internal server error"  
// @Router /post/{POST\_ID}/downvote [get]  
func (p \*PostHandler) Downvote(w http.ResponseWriter, r \*http.Request) {  
 postID, err := validateID("POST\_ID", mux.Vars(r))  
 if err != nil {  
 sendErrorResponse(w, http.StatusBadRequest, errs.NewSimpleErr(errs.ErrInvalidPostID.Error()))  
 return  
 }  
  
 post, err := p.service.Downvote(r.Context(), postID)  
 switch {  
 case errors.Is(err, errs.ErrPostNotFound)**:** sendErrorResponse(w, http.StatusNotFound, errs.NewSimpleErr(errs.ErrPostNotFound.Error()))  
 return  
 case err != nil**:** sendErrorResponse(w, http.StatusInternalServerError, errs.NewSimpleErr(errs.ErrUnknownError.Error()))  
 return  
 }  
  
 sendResponse(post, w)  
}  
  
// Unvote godoc  
//  
// @Summary Cancel your vote  
// @Description Withdraw your vote from the post  
// @Security ApiKeyAuth  
// @Tags voting-posts  
// @ID unvote-post  
// @Produce json  
// @Param POST\_ID path string true "Post uuid" minlength(36) maxlength(36)  
// @Success 200 {object} posts.Post "Successfully unvoted"  
// @Failure 400 {object} errs.SimpleErr "Bad post id"  
// @Failure 404 {object} errs.SimpleErr "No posts with the provided id were found"  
// @Failure 500 {object} errs.SimpleErr "Internal server error"  
// @Router /post/{POST\_ID}/unvote [get]  
func (p \*PostHandler) Unvote(w http.ResponseWriter, r \*http.Request) {  
 postID, err := validateID("POST\_ID", mux.Vars(r))  
 if err != nil {  
 sendErrorResponse(w, http.StatusBadRequest, errs.NewSimpleErr(errs.ErrInvalidPostID.Error()))  
 return  
 }  
  
 post, err := p.service.Unvote(r.Context(), postID)  
 switch {  
 case errors.Is(err, errs.ErrPostNotFound)**:** sendErrorResponse(w, http.StatusNotFound, errs.NewSimpleErr(errs.ErrPostNotFound.Error()))  
 return  
 case err != nil**:** sendErrorResponse(w, http.StatusInternalServerError, errs.NewSimpleErr(errs.ErrUnknownError.Error()))  
 return  
 }  
  
 sendResponse(post, w)  
}  
  
// AddComment godoc  
//  
// @Summary Comment on the post  
// @Description Leave a comment under a certain post  
// @Security ApiKeyAuth  
// @Tags commenting-posts  
// @ID add-comment  
// @Accept json  
// @Produce json  
// @Param comment\_payload body posts.Comment true "Comment data" validate(required)  
// @Param POST\_ID path string true "Post uuid" minlength(36) maxlength(36)  
// @Success 201 {object} posts.Post "Comment successfully left"  
// @Failure 400 {object} errs.SimpleErr "Bad payload"  
// @Failure 404 {object} errs.SimpleErr "No posts with the provided id were found"  
// @Failure 422 {object} errs.ComplexErrArr "Bad content"  
// @Failure 500 {object} errs.SimpleErr "Internal server error"  
// @Router /posts/{POST\_ID} [post]  
func (p \*PostHandler) AddComment(w http.ResponseWriter, r \*http.Request) {  
 defer r.Body.Close()  
 body, err := io.ReadAll(r.Body)  
 if err != nil {  
 w.WriteHeader(http.StatusBadRequest)  
 return  
 }  
 comment := posts.Comment{}  
 if err = json.Unmarshal(body, &comment); err != nil {  
 w.WriteHeader(http.StatusBadRequest)  
 return  
 }  
  
 postID, err := validateID("POST\_ID", mux.Vars(r))  
 if err != nil {  
 sendErrorResponse(w, http.StatusBadRequest, errs.NewSimpleErr(errs.ErrInvalidPostID.Error()))  
 return  
 }  
  
 post, err := p.service.AddComment(r.Context(), postID, comment)  
 switch {  
 case errors.Is(err, errs.ErrBadCommentBody)**:** sendErrorResponse(w, http.StatusUnprocessableEntity, errs.NewComplexErrArr(errs.ComplexErr{  
 Location**:** "body",  
 Param**:** "comment",  
 Msg**:** "is required",  
 }))  
 return  
 case errors.Is(err, errs.ErrPostNotFound)**:** sendErrorResponse(w, http.StatusNotFound, errs.NewSimpleErr(errs.ErrPostNotFound.Error()))  
 return  
 case err != nil**:** sendErrorResponse(w, http.StatusInternalServerError, errs.NewSimpleErr(errs.ErrUnknownError.Error()))  
 return  
 }  
  
 sendResponse(post, w, httpresp.WithStatusCode(http.StatusCreated))  
}  
  
// DeleteComment godoc  
//  
// @Summary Delete comment  
// @Description Delete a certain comment on a certain post  
// @Security ApiKeyAuth  
// @Tags commenting-posts  
// @ID delete-comment  
// @Accept json  
// @Produce json  
// @Param POST\_ID path string true "Post uuid" minlength(36) maxlength(36)  
// @Param COMMENT\_ID path string true "Comment uuid" minlength(36) maxlength(36)  
// @Success 200 {object} posts.Post "Comment successfully deleted"  
// @Failure 400 {object} errs.SimpleErr "Bad uuid"  
// @Failure 404 {object} errs.SimpleErr "No posts or comment with the provided id were found"  
// @Failure 500 {object} errs.SimpleErr "Internal server error"  
// @Router /posts/{POST\_ID}/{COMMENT\_ID} [delete]  
func (p \*PostHandler) DeleteComment(w http.ResponseWriter, r \*http.Request) {  
 params := mux.Vars(r)  
 postID, err := validateID("POST\_ID", params)  
 if err != nil {  
 sendErrorResponse(w, http.StatusBadRequest, errs.NewSimpleErr(errs.ErrInvalidPostID.Error()))  
 return  
 }  
 commentID, err := validateID("COMMENT\_ID", params)  
 if err != nil {  
 sendErrorResponse(w, http.StatusBadRequest, errs.NewSimpleErr(errs.ErrInvalidCommentID.Error()))  
 return  
 }  
  
 post, err := p.service.DeleteComment(r.Context(), postID, commentID)  
 switch {  
 case errors.Is(err, errs.ErrPostNotFound)**:** sendErrorResponse(w, http.StatusNotFound, errs.NewSimpleErr(errs.ErrPostNotFound.Error()))  
 return  
 case errors.Is(err, errs.ErrCommentNotFound)**:** sendErrorResponse(w, http.StatusNotFound, errs.NewSimpleErr(errs.ErrCommentNotFound.Error()))  
 return  
 case err != nil**:** sendErrorResponse(w, http.StatusInternalServerError, errs.NewSimpleErr(errs.ErrUnknownError.Error()))  
 return  
 }  
  
 sendResponse(post, w)  
  
}

**user.go**

package rest  
  
import (  
 "context"  
 "encoding/json"  
 "errors"  
 "io"  
 "net/http"  
  
 "go.uber.org/zap"  
  
 "github.com/Benzogang-Tape/Reddit/internal/models/errs"  
 "github.com/Benzogang-Tape/Reddit/internal/models/httpresp"  
 "github.com/Benzogang-Tape/Reddit/internal/models/jwt"  
 "github.com/Benzogang-Tape/Reddit/internal/models/users"  
 "github.com/Benzogang-Tape/Reddit/internal/service"  
)  
  
//go:generate mockgen -source=user.go -destination=../../storage/mocks/users\_repo\_mySQL\_mock.go -package=mocks UserAPI  
type UserAPI interface {  
 Register(ctx context.Context, authData users.AuthUserInfo) (\*jwt.TokenPayload, error)  
 Authorize(ctx context.Context, authData users.AuthUserInfo) (\*jwt.TokenPayload, error)  
}  
  
type UserHandler struct {  
 logger \*zap.SugaredLogger  
 service UserAPI  
 sessMngr service.SessionAPI  
}  
  
func NewUserHandler(u UserAPI, s service.SessionAPI, logger \*zap.SugaredLogger) \*UserHandler {  
 return &UserHandler{  
 logger**:** logger,  
 service**:** u,  
 sessMngr**:** s,  
 }  
}  
  
// RegisterUser godoc  
//  
// @Summary Register a new user  
// @Description Register in reddit-clone app  
// @Tags auth  
// @ID register-user  
// @Accept json  
// @Produce json  
// @Param credentials body users.AuthUserInfo true "User credentials for registration"  
// @Success 201 {object} jwt.Session "User registered successfully"  
// @Failure 400 "Bad request"  
// @Failure 422 {object} errs.ComplexErrArr "User already exists"  
// @Failure 500 {object} errs.SimpleErr "Internal server error"  
// @Router /register [post]  
func (h \*UserHandler) RegisterUser(w http.ResponseWriter, r \*http.Request) {  
 defer r.Body.Close()  
 body, err := io.ReadAll(r.Body)  
 if err != nil {  
 w.WriteHeader(http.StatusBadRequest)  
 return  
 }  
 credentials := users.AuthUserInfo{}  
 if err = json.Unmarshal(body, &credentials); err != nil {  
 w.WriteHeader(http.StatusBadRequest)  
 return  
 }  
  
 payload, err := h.service.Register(r.Context(), credentials)  
 switch {  
 case errors.Is(err, errs.ErrUserExists)**:** sendErrorResponse(w, http.StatusUnprocessableEntity, errs.NewComplexErrArr(errs.ComplexErr{  
 Location**:** `body`,  
 Param**:** `username`,  
 Value**:** `1`,  
 Msg**:** `already exists`,  
 }))  
 return  
 case err != nil**:** sendErrorResponse(w, http.StatusInternalServerError, errs.NewSimpleErr(errs.ErrUnknownError.Error()))  
 return  
 }  
  
 h.newSession(w, r.WithContext(context.WithValue(r.Context(), jwt.Payload, \*payload)), http.StatusCreated)  
 h.logger.Infow("New user has registered",  
 "login", credentials.Login,  
 "remote\_addr", r.RemoteAddr,  
 "url", r.URL.Path,  
 )  
}  
  
// LoginUser godoc  
//  
// @Summary Login to your account  
// @Description Login via login and password in reddit-clone app  
// @Tags auth  
// @ID login-user  
// @Accept json  
// @Produce json  
// @Param credentials body users.AuthUserInfo true "User credentials for authentication"  
// @Success 200 {object} jwt.Session "User authorized successfully"  
// @Failure 400 "Bad request"  
// @Failure 401 {object} errs.SimpleErr "Bad login or password"  
// @Failure 500 {object} errs.SimpleErr "Internal server error"  
// @Router /login [post]  
func (h \*UserHandler) LoginUser(w http.ResponseWriter, r \*http.Request) {  
 defer r.Body.Close()  
 body, err := io.ReadAll(r.Body)  
 if err != nil {  
 w.WriteHeader(http.StatusBadRequest)  
 return  
 }  
 credentials := users.AuthUserInfo{}  
 if err = json.Unmarshal(body, &credentials); err != nil {  
 w.WriteHeader(http.StatusBadRequest)  
 return  
 }  
  
 payload, err := h.service.Authorize(r.Context(), credentials)  
 switch {  
 case errors.Is(err, errs.ErrNoUser), errors.Is(err, errs.ErrBadPass)**:** sendErrorResponse(w, http.StatusUnauthorized, errs.NewSimpleErr(errs.ErrBadPass.Error()))  
 return  
 case err != nil**:** sendErrorResponse(w, http.StatusInternalServerError, errs.NewSimpleErr(errs.ErrUnknownError.Error()))  
 return  
 }  
  
 h.newSession(w, r.WithContext(context.WithValue(r.Context(), jwt.Payload, \*payload)), http.StatusOK)  
 h.logger.Infow("New log in",  
 "login", credentials.Login,  
 "remote\_addr", r.RemoteAddr,  
 "url", r.URL.Path,  
 )  
}  
  
func (h \*UserHandler) newSession(w http.ResponseWriter, r \*http.Request, statusCode int) {  
 if r.Header.Get("Content-Type") != "application/json" {  
 sendErrorResponse(w, http.StatusBadRequest, errs.NewSimpleErr(errs.ErrUnknownPayload.Error()))  
 return  
 }  
  
 sess, err := h.sessMngr.New(r.Context())  
 if err != nil {  
 w.WriteHeader(http.StatusInternalServerError)  
 return  
 }  
  
 sendResponse(sess, w, httpresp.WithStatusCode(statusCode))  
}

Функции для отправки ответа по сети:

**send\_response.go**

package rest  
  
import (  
 "encoding/json"  
 "net/http"  
  
 "github.com/Benzogang-Tape/Reddit/internal/models/errs"  
 "github.com/Benzogang-Tape/Reddit/internal/models/httpresp"  
)  
  
const DefaultContentType = "application/json; charset=utf-8"  
  
func sendResponse(data any, w http.ResponseWriter, opts ...httpresp.OptionFunc) {  
 response := &httpresp.Response{  
 Data**:** data,  
 StatusCode**:** http.StatusOK,  
 }  
  
 for \_, opt := range opts {  
 opt(response)  
 }  
  
 send(response, w)  
}  
  
func sendErrorResponse(w http.ResponseWriter, statusCode int, errMsg errs.RespError) {  
 resp, err := errMsg.Marshal()  
 if err != nil {  
 w.WriteHeader(http.StatusInternalServerError)  
 return  
 }  
  
 w.Header().Set("Content-Type", DefaultContentType)  
 w.WriteHeader(statusCode)  
 if \_, err = w.Write(resp); err != nil {  
 w.WriteHeader(http.StatusInternalServerError)  
 }  
}  
  
func send(r \*httpresp.Response, w http.ResponseWriter) {  
 resp, err := json.Marshal(r.Data)  
 if err != nil {  
 sendErrorResponse(w, http.StatusInternalServerError, errs.NewSimpleErr(errs.ErrResponseError.Error()))  
 return  
 }  
  
 if r.ContentType != "" {  
 w.Header().Set("Content-Type", r.ContentType)  
 } else {  
 w.Header().Set("Content-Type", DefaultContentType)  
 }  
  
 w.WriteHeader(r.StatusCode)  
 if \_, err := w.Write(resp); err != nil {  
 sendErrorResponse(w, http.StatusInternalServerError, errs.NewSimpleErr(errs.ErrResponseError.Error()))  
 }  
}

В функции sendResponse был использован паттерн Functional options, позволяющий довольно гибко конфигурировать объект, передающийся в функцию в качестве параметра.

Типы из файла send\_response.go:

**http\_response.go**

package httpresp  
  
type Response struct {  
 Data any  
 StatusCode int  
 ContentType string  
}  
  
type OptionFunc func(\*Response)  
  
func WithStatusCode(code int) OptionFunc {  
 return func(r \*Response) {  
 r.StatusCode = code  
 }  
}  
  
func WithContentType(contentType string) OptionFunc {  
 return func(r \*Response) {  
 r.ContentType = contentType  
 }  
}

В файле router.go описана логика инициализации роутера для всего приложения, тут же на эндпоинты навешиваются обработчики из файлов выше. Здесь же описана логика отдачи статики по пути “/”.

**router.go**

package rest  
  
import (  
 "html/template"  
 "net/http"  
  
 "github.com/gorilla/mux"  
 httpSwagger "github.com/swaggo/http-swagger"  
 "go.uber.org/zap"  
  
 \_ "github.com/Benzogang-Tape/Reddit/docs"  
 "github.com/Benzogang-Tape/Reddit/internal/transport/middleware"  
 mdwr "github.com/Benzogang-Tape/Reddit/pkg/middleware"  
)  
  
type AppRouter struct {  
 userHandler \*UserHandler  
 postHandler \*PostHandler  
}  
  
func NewAppRouter(u \*UserHandler, p \*PostHandler) \*AppRouter {  
 return &AppRouter{  
 userHandler**:** u,  
 postHandler**:** p,  
 }  
}  
  
func (rtr \*AppRouter) InitRouter(logger \*zap.SugaredLogger) http.Handler {  
 templates := template.Must(template.ParseGlob("./static/\*/\*"))  
  
 r := mux.NewRouter()  
 r.HandleFunc("/", func(w http.ResponseWriter, r \*http.Request) {  
 err := templates.ExecuteTemplate(w, "index.html", nil)  
 if err != nil {  
 http.Error(w, `Template error`, http.StatusInternalServerError)  
 return  
 }  
 }).Methods(http.MethodGet)  
  
 r.PathPrefix("/static/").Handler(http.StripPrefix("/static/", http.FileServer(http.Dir("./static"))))  
 r.PathPrefix("/swagger/").Handler(httpSwagger.WrapHandler)  
  
 r.HandleFunc("/api/register", rtr.userHandler.RegisterUser).Methods(http.MethodPost)  
 r.HandleFunc("/api/login", rtr.userHandler.LoginUser).Methods(http.MethodPost)  
 r.HandleFunc("/api/posts/", rtr.postHandler.GetAllPosts).Methods(http.MethodGet)  
 r.HandleFunc("/api/posts", rtr.postHandler.CreatePost).Methods(http.MethodPost)  
 r.HandleFunc("/api/post/{POST\_ID:[0-9a-fA-F-]+$}", rtr.postHandler.GetPostByID).Methods(http.MethodGet)  
 r.HandleFunc("/api/posts/{CATEGORY\_NAME:[0-9a-zA-Z\_-]+$}", rtr.postHandler.GetPostsByCategory).Methods(http.MethodGet)  
 r.HandleFunc("/api/user/{USER\_LOGIN:[0-9a-zA-Z\_-]+$}", rtr.postHandler.GetPostsByUser).Methods(http.MethodGet)  
 r.HandleFunc("/api/post/{POST\_ID:[0-9a-fA-F-]+$}", rtr.postHandler.DeletePost).Methods(http.MethodDelete)  
 r.HandleFunc("/api/post/{POST\_ID:[0-9a-fA-F-]+}/upvote", rtr.postHandler.Upvote).Methods(http.MethodGet)  
 r.HandleFunc("/api/post/{POST\_ID:[0-9a-fA-F-]+}/downvote", rtr.postHandler.Downvote).Methods(http.MethodGet)  
 r.HandleFunc("/api/post/{POST\_ID:[0-9a-fA-F-]+}/unvote", rtr.postHandler.Unvote).Methods(http.MethodGet)  
 r.HandleFunc("/api/post/{POST\_ID:[0-9a-fA-F-]+$}", rtr.postHandler.AddComment).Methods(http.MethodPost)  
 r.HandleFunc("/api/post/{POST\_ID:[0-9a-fA-F-]+}/{COMMENT\_ID:[0-9a-fA-F-]+$}", rtr.postHandler.DeleteComment).Methods(http.MethodDelete)  
  
 router := middleware.Auth(r, rtr.userHandler.sessMngr, logger)  
 router = mdwr.AccessLog(logger, router)  
 router = middleware.Panic(router, logger)  
  
 return router  
}

В файле demo.go происходит сборка всех зависимостей и запуск приложения в главной функции main

**demo.go**

package main  
  
import (  
 "flag"  
 "fmt"  
 "log"  
 "net/http"  
 "os"  
  
 "go.uber.org/zap"  
  
 "github.com/Benzogang-Tape/Reddit/internal/models/jwt"  
 "github.com/Benzogang-Tape/Reddit/internal/service"  
 "github.com/Benzogang-Tape/Reddit/internal/storage/inmem"  
 "github.com/Benzogang-Tape/Reddit/internal/transport/rest"  
)  
  
var port = flag.Int("port", 8081, "HTTP port")  
  
func init() {  
 os.Setenv("JWT\_SECRET", "super secret key")  
}  
  
// @title Reddit-Clone API  
// @version 1.0  
// @description Basic restfull api for reddit-clone backend.  
// @termsOfService http://swagger.io/terms/  
  
// @contact.name API Support  
// @contact.url http://www.swagger.io/support  
// @contact.email support@swagger.io  
  
// @license.name Apache 2.0  
// @license.url http://www.apache.org/licenses/LICENSE-2.0.html  
  
// @host localhost:8081  
// @BasePath /api  
  
// @securityDefinitions.apikey ApiKeyAuth  
// @in header  
// @name Authorization  
  
// @externalDocs.description OpenAPI  
// @externalDocs.url https://swagger.io/resources/open-api/  
func main() {  
 flag.Parse()  
  
 if err := jwt.SetJWTSecret(os.Getenv("JWT\_SECRET")); err != nil {  
 panic(err)  
 }  
  
 zapLogger, err := zap.NewProduction()  
 if err != nil {  
 log.Fatalln("Logger init error")  
 }  
 defer zapLogger.Sync() //nolint:errcheck  
 logger := zapLogger.Sugar()  
  
 sessionRepo := inmem.NewSessionRepo()  
 sessionHandler := service.NewSessionHandler(sessionRepo)  
  
 userStorage := inmem.NewUserRepo()  
 userHandler := service.NewUserHandler(userStorage)  
 u := rest.NewUserHandler(userHandler, sessionHandler, logger)  
  
 postStorage := inmem.NewPostRepo()  
 postHandler := service.NewPostHandler(postStorage, postStorage)  
 p := rest.NewPostHandler(postHandler, logger)  
  
 router := rest.NewAppRouter(u, p).InitRouter(logger)  
  
 addr := fmt.Sprintf(":%d", \*port)  
 logger.Infow(fmt.Sprintf("Starting server on %s", addr))  
 log.Panic(http.ListenAndServe(addr, router))  
}

На этом моменте MVP было готово. Дальше предстояло реализовать работу с базами данных

**posts\_repo\_mongoDB.go**

package storage  
  
import (  
 "context"  
 "fmt"  
  
 "github.com/pkg/errors"  
 "go.mongodb.org/mongo-driver/bson"  
 "go.mongodb.org/mongo-driver/mongo"  
 "go.mongodb.org/mongo-driver/mongo/options"  
  
 "github.com/Benzogang-Tape/Reddit/internal/models/errs"  
 "github.com/Benzogang-Tape/Reddit/internal/models/jwt"  
 "github.com/Benzogang-Tape/Reddit/internal/models/posts"  
 "github.com/Benzogang-Tape/Reddit/internal/models/users"  
)  
  
type PostRepoMongoDB struct {  
 collection AbstractCollection  
}  
  
func NewPostRepoMongoDB(collection AbstractCollection) \*PostRepoMongoDB {  
 return &PostRepoMongoDB{  
 collection**:** collection,  
 }  
}  
  
func (p \*PostRepoMongoDB) GetAllPosts(ctx context.Context) ([]\*posts.Post, error) {  
 posts := make(posts.Posts, 0)  
 sort := bson.D{{Key**:** "score", Value**:** -1}}  
 cur, err := p.collection.Find(ctx, bson.M{}, options.Find().SetSort(sort))  
 if err != nil {  
 return nil, err  
 }  
  
 if err = cur.All(ctx, &posts); err != nil {  
 return nil, err  
 }  
  
 return posts, nil  
}  
  
func (p \*PostRepoMongoDB) GetPostsByCategory(ctx context.Context, postCategory posts.PostCategory) ([]\*posts.Post, error) {  
 posts := make(posts.Posts, 0)  
 filter := bson.M{"category"**:** postCategory}  
 sort := bson.D{{Key**:** "score", Value**:** -1}}  
 cur, err := p.collection.Find(ctx, filter, options.Find().SetSort(sort))  
 if err != nil {  
 return nil, err  
 }  
  
 if err = cur.All(ctx, &posts); err != nil {  
 return nil, err  
 }  
  
 return posts, nil  
}  
  
func (p \*PostRepoMongoDB) GetPostsByUser(ctx context.Context, userLogin users.Username) ([]\*posts.Post, error) {  
 postList := make([]\*posts.Post, 0)  
 filter := bson.M{"author.username"**:** userLogin}  
 sort := bson.D{{Key**:** "created", Value**:** -1}}  
 cur, err := p.collection.Find(ctx, filter, options.Find().SetSort(sort))  
 if err != nil {  
 return nil, err  
 }  
  
 if err = cur.All(ctx, &postList); err != nil {  
 return nil, err  
 }  
  
 return postList, nil  
}  
  
func (p \*PostRepoMongoDB) GetPostByID(ctx context.Context, postID users.ID) (\*posts.Post, error) {  
 filter := bson.M{"uuid"**:** postID}  
 res := p.collection.FindOne(ctx, filter)  
 if errors.Is(res.Err(), mongo.ErrNoDocuments) {  
 return nil, errs.ErrPostNotFound  
 }  
  
 post := new(posts.Post)  
 if err := res.Decode(post); err != nil {  
 return nil, err  
 }  
  
 return post, nil  
}  
  
func (p \*PostRepoMongoDB) CreatePost(ctx context.Context, postPayload posts.PostPayload) (\*posts.Post, error) {  
 author, ok := ctx.Value(jwt.Payload).(\*jwt.TokenPayload)  
 if !ok {  
 return nil, errs.ErrBadPayload  
 }  
  
 newPost := posts.NewPost(\*author, postPayload)  
  
 if \_, err := p.collection.InsertOne(ctx, newPost); err != nil {  
 fmt.Println("\n\n\n\n", err.Error())  
 return nil, err  
 }  
  
 return newPost, nil  
}  
  
func (p \*PostRepoMongoDB) DeletePost(ctx context.Context, postID users.ID) error {  
 filter := bson.M{"uuid"**:** postID}  
 deletedCount, err := p.collection.DeleteOne(ctx, filter)  
 if err != nil {  
 return err  
 }  
 if deletedCount == 0 {  
 return errs.ErrPostNotFound  
 }  
  
 return nil  
}  
  
func (p \*PostRepoMongoDB) AddComment(ctx context.Context, post \*posts.Post, comment posts.Comment) (\*posts.Post, error) {  
 author, ok := ctx.Value(jwt.Payload).(\*jwt.TokenPayload)  
 if !ok {  
 return nil, errs.ErrBadPayload  
 }  
  
 newComment := post.AddComment(\*author, comment.Body)  
 if \_, err := p.collection.UpdateOne(  
 ctx,  
 bson.M{"uuid"**:** post.ID},  
 bson.M{"$push"**:** bson.M{"comments"**:** newComment}},  
 ); err != nil {  
 return nil, err  
 }  
  
 return post, nil  
}  
  
func (p \*PostRepoMongoDB) DeleteComment(ctx context.Context, post \*posts.Post, commentID users.ID) (\*posts.Post, error) {  
 source := "DeleteComment"  
 if err := post.DeleteComment(commentID); err != nil {  
 return nil, errors.Wrap(err, source)  
 }  
  
 if \_, err := p.collection.UpdateOne(  
 ctx,  
 bson.M{"uuid"**:** post.ID},  
 bson.M{"$pull"**:** bson.M{"comments"**:** bson.M{"uuid"**:** commentID}}},  
 ); err != nil {  
 return nil, errors.Wrap(err, source)  
 }  
  
 return post, nil  
}  
  
func (p \*PostRepoMongoDB) Upvote(ctx context.Context, post \*posts.Post) (\*posts.Post, error) {  
 source := "Upvote"  
 author, ok := ctx.Value(jwt.Payload).(\*jwt.TokenPayload)  
 if !ok {  
 return nil, errs.ErrBadPayload  
 }  
  
 newVote, created := post.Upvote(author.ID)  
 if !created {  
 filter := bson.M{  
 "uuid"**:** post.ID,  
 "votes.user"**:** newVote.UserID,  
 }  
 update := bson.M{  
 "$set"**:** bson.M{  
 "votes.$.vote"**:** newVote.Vote,  
 "upvotePercentage"**:** post.UpvotePercentage,  
 "score"**:** post.Score,  
 },  
 }  
 if \_, err := p.collection.UpdateOne(ctx, filter, update); err != nil {  
 return nil, errors.Wrap(err, source)  
 }  
  
 return post, nil  
 }  
  
 filter := bson.M{"uuid"**:** post.ID}  
 update := bson.M{  
 "$push"**:** bson.M{  
 "votes"**:** newVote,  
 },  
 "$set"**:** bson.M{  
 "upvotePercentage"**:** post.UpvotePercentage,  
 "score"**:** post.Score,  
 },  
 }  
 if \_, err := p.collection.UpdateOne(ctx, filter, update); err != nil {  
 return nil, errors.Wrap(err, source)  
 }  
  
 return post, nil  
}  
  
func (p \*PostRepoMongoDB) Downvote(ctx context.Context, post \*posts.Post) (\*posts.Post, error) {  
 source := "Downvote"  
 author, ok := ctx.Value(jwt.Payload).(\*jwt.TokenPayload)  
 if !ok {  
 return nil, errs.ErrBadPayload  
 }  
  
 filter := bson.M{"uuid"**:** post.ID}  
 newVote, created := post.Downvote(author.ID)  
 if !created {  
 filter["votes.user"] = newVote.UserID  
 update := bson.M{  
 "$set"**:** bson.M{  
 "votes.$.vote"**:** newVote.Vote,  
 "upvotePercentage"**:** post.UpvotePercentage,  
 "score"**:** post.Score,  
 },  
 }  
  
 if \_, err := p.collection.UpdateOne(ctx, filter, update); err != nil {  
 return nil, errors.Wrap(err, source)  
 }  
  
 return post, nil  
 }  
  
 update := bson.M{  
 "$push"**:** bson.M{  
 "votes"**:** newVote,  
 },  
 "$set"**:** bson.M{  
 "upvotePercentage"**:** post.UpvotePercentage,  
 "score"**:** post.Score,  
 },  
 }  
 if \_, err := p.collection.UpdateOne(ctx, filter, update); err != nil {  
 return nil, errors.Wrap(err, source)  
 }  
  
 return post, nil  
}  
  
func (p \*PostRepoMongoDB) Unvote(ctx context.Context, post \*posts.Post) (\*posts.Post, error) {  
 source := "Unvote"  
 author, ok := ctx.Value(jwt.Payload).(\*jwt.TokenPayload)  
 if !ok {  
 return nil, errs.ErrBadPayload  
 }  
  
 if err := post.Unvote(author.ID); err != nil {  
 return nil, errors.Wrap(err, source)  
 }  
  
 filter := bson.M{"uuid"**:** post.ID}  
 update := bson.M{  
 "$pull"**:** bson.M{  
 "votes"**:** bson.M{  
 "user"**:** author.ID,  
 },  
 },  
 "$set"**:** bson.M{  
 "upvotePercentage"**:** post.UpvotePercentage,  
 "score"**:** post.Score,  
 },  
 }  
 if \_, err := p.collection.UpdateOne(ctx, filter, update); err != nil {  
 return nil, errors.Wrap(err, source)  
 }  
  
 return post, nil  
}  
  
func (p \*PostRepoMongoDB) UpdateViews(ctx context.Context, postID users.ID) error {  
 source := "UpdateViews"  
 filter := bson.M{"uuid"**:** postID}  
 update := bson.M{"$inc"**:** bson.M{"views"**:** 1}}  
 if \_, err := p.collection.UpdateOne(ctx, filter, update); err != nil {  
 return errors.Wrap(err, source)  
 }  
  
 return nil  
}

**mongoDB\_abstraction.go**

package storage  
  
import (  
 "context"  
  
 "go.mongodb.org/mongo-driver/mongo"  
 "go.mongodb.org/mongo-driver/mongo/options"  
)  
  
//go:generate mockgen -source=mongoDB\_abstraction.go -destination=./mocks/mongoDB\_abstraction\_mock.go -package=mocks AbstractCollection AbstractCursor AbstractSingleResult  
type AbstractCollection interface {  
 Find(ctx context.Context, filter any, opts ...\*options.FindOptions) (AbstractCursor, error)  
 FindOne(ctx context.Context, filter any, opts ...\*options.FindOneOptions) AbstractSingleResult  
 InsertOne(ctx context.Context, document any, opts ...\*options.InsertOneOptions) (any, error)  
 UpdateOne(ctx context.Context, filter any, update any, opts ...\*options.UpdateOptions) (int64, error)  
 DeleteOne(ctx context.Context, filter any, opts ...\*options.DeleteOptions) (int64, error)  
}  
  
type AbstractCursor interface {  
 All(ctx context.Context, result any) error  
}  
  
type AbstractSingleResult interface {  
 Decode(v any) error  
 Err() error  
}  
  
type mongoCollection struct {  
 collection \*mongo.Collection  
}  
  
type mongoCursor struct {  
 cursor \*mongo.Cursor  
}  
  
type mongoSingleResult struct { //nolint:unused  
 sr \*mongo.SingleResult  
}  
  
func NewMongoCollection(collection \*mongo.Collection) \*mongoCollection {  
 return &mongoCollection{  
 collection**:** collection,  
 }  
}  
  
func (c \*mongoCollection) Find(ctx context.Context, filter any, opts ...\*options.FindOptions) (AbstractCursor, error) {  
 cursor, err := c.collection.Find(ctx, filter, opts...)  
 return &mongoCursor{  
 cursor**:** cursor,  
 }, err  
}  
  
func (c \*mongoCollection) FindOne(ctx context.Context, filter any, opts ...\*options.FindOneOptions) AbstractSingleResult {  
 return c.collection.FindOne(ctx, filter, opts...)  
}  
  
func (c \*mongoCollection) InsertOne(ctx context.Context, document any, opts ...\*options.InsertOneOptions) (any, error) {  
 return c.collection.InsertOne(ctx, document, opts...)  
}  
  
func (c \*mongoCollection) UpdateOne(ctx context.Context, filter any, update any, opts ...\*options.UpdateOptions) (int64, error) {  
 result, err := c.collection.UpdateOne(ctx, filter, update, opts...)  
 if err != nil {  
 return 0, err  
 }  
  
 return result.MatchedCount, nil  
}  
  
func (c \*mongoCollection) DeleteOne(ctx context.Context, filter any, opts ...\*options.DeleteOptions) (int64, error) {  
 result, err := c.collection.DeleteOne(ctx, filter, opts...)  
 if err != nil {  
 return 0, err  
 }  
  
 return result.DeletedCount, nil  
}  
  
func (c \*mongoCursor) All(ctx context.Context, result any) error {  
 return c.cursor.All(ctx, result)  
}  
  
func (sr \*mongoSingleResult) Decode(v any) error { //nolint:unused  
 return sr.sr.Decode(v)  
}  
  
func (sr \*mongoSingleResult) Err() error { //nolint:unused  
 return sr.sr.Err()  
}

**users\_repo\_mySQL.go**

package storage  
  
import (  
 "context"  
 "database/sql"  
  
 "github.com/pkg/errors"  
  
 "github.com/Benzogang-Tape/Reddit/internal/models/errs"  
 "github.com/Benzogang-Tape/Reddit/internal/models/users"  
)  
  
type UserRepoMySQL struct {  
 db \*sql.DB  
}  
  
func NewUserRepoMySQL(db \*sql.DB) \*UserRepoMySQL {  
 return &UserRepoMySQL{  
 db**:** db,  
 }  
}  
  
func (repo \*UserRepoMySQL) Authorize(ctx context.Context, authData users.AuthUserInfo) (\*users.User, error) { //nolint:unparam  
 source := "Authorize"  
 user := &users.User{}  
 err := repo.db.  
 QueryRow(  
 "SELECT uuid, login, password FROM users WHERE login = ?",  
 authData.Login,  
 ).Scan(&user.ID, &user.Username, &user.Password)  
  
 switch {  
 case errors.Is(err, sql.ErrNoRows)**:** return nil, errors.Wrap(errs.ErrNoUser, source)  
 case err != nil**:** return nil, err  
 }  
  
 if user.Password != authData.Password {  
 return nil, errors.Wrap(errs.ErrBadPass, source)  
 }  
  
 return user, nil  
}  
  
func (repo \*UserRepoMySQL) RegisterUser(ctx context.Context, authData users.AuthUserInfo) (\*users.User, error) { //nolint:unparam  
 source := "RegisterUser"  
 var userExists bool  
 err := repo.db.QueryRow(  
 "SELECT EXISTS(SELECT 1 FROM users WHERE login = ?)",  
 authData.Login,  
 ).Scan(&userExists)  
  
 switch {  
 case err != nil**:** return nil, err  
 case userExists**:** return nil, errors.Wrap(errs.ErrUserExists, source)  
 }  
  
 newUser, err := repo.createUser(authData)  
 if err != nil {  
 return nil, err  
 }  
  
 return newUser, nil  
}  
  
func (repo \*UserRepoMySQL) createUser(credentials users.AuthUserInfo) (\*users.User, error) {  
 newUser := users.NewUser(credentials)  
 if \_, err := repo.db.Exec(  
 "INSERT INTO users (`uuid`, `login`, `password`) VALUES (?, ?, ?)",  
 newUser.ID,  
 newUser.Username,  
 newUser.Password,  
 ); err != nil {  
 return nil, err  
 }  
  
 return newUser, nil  
}

**sessions\_repo\_redis.go**

package storage  
  
import (  
 "context"  
 "encoding/json"  
 "fmt"  
  
 "github.com/go-redis/redis"  
  
 "github.com/Benzogang-Tape/Reddit/internal/models/jwt"  
)  
  
type SessionRepoRedis struct {  
 rdb \*redis.Client  
}  
  
func NewSessionRepoRedis(client \*redis.Client) \*SessionRepoRedis {  
 return &SessionRepoRedis{  
 rdb**:** client,  
 }  
}  
  
func (s \*SessionRepoRedis) CreateSession(ctx context.Context, session \*jwt.Session, payload \*jwt.TokenPayload) (\*jwt.Session, error) { //nolint:unparam  
 key := session.Token  
 value, err := json.Marshal(payload)  
 if err != nil {  
 return nil, err  
 }  
 result, err := s.rdb.Set(key, value, jwt.SessLifespan).Result()  
 if err != nil {  
 return nil, err  
 }  
 if result != "OK" {  
 return nil, fmt.Errorf("result is not OK. Actual value: %s", result)  
 }  
  
 return session, nil  
}  
  
func (s \*SessionRepoRedis) CheckSession(ctx context.Context, sess \*jwt.Session) (\*jwt.TokenPayload, error) { //nolint:unparam  
 key := sess.Token  
 val, err := s.rdb.Get(key).Result()  
 if err != nil {  
 return nil, err  
 }  
  
 payload := &jwt.TokenPayload{}  
 if err = json.Unmarshal([]byte(val), payload); err != nil {  
 return nil, err  
 }  
  
 return payload, nil  
}

Все БД запускались в Docker-контейнерах, для удобства конфигурации и запуска использовался docker-compose

**docker-compose.yml**

version: '3'  
  
# docker rm $(docker ps -a -q) && docker volume prune -f  
  
services:  
 reddit:  
 container\_name: **redditclone** build:  
 context: **.** args:  
 PORT: **${APP\_PORT:-8081}** APP\_NAME: **${APP\_NAME}** BUILD\_DATE: **$(date -u +"%Y%m%d%H%M%SZ")** ports:  
 - **8081:${APP\_PORT:-8081}** expose:  
 - **${APP\_PORT:-8081}** env\_file:  
 - **.env** depends\_on:  
 mysql:  
 condition: **service\_healthy** mongodb:  
 condition: **service\_healthy** redis:  
 condition: **service\_healthy** restart: **on-failure** volumes:  
 - "./internal/config/config.yaml:/app/config.yaml"  
 - "./static/:/app/static/"  
 - "./docs/:/app/docs/"  
  
 mysql:  
 image: **mysql:8** container\_name: **MySQL** command: **--mysql-native-password=ON** env\_file:  
 - **.env** ports:  
 - **3306:${MYSQL\_PORT:-3306}** volumes:  
 - './init/db\_tables/:/docker-entrypoint-initdb.d/'  
 healthcheck:  
 test: ["CMD", "mysqladmin", "ping", "-h", "localhost"]  
 interval: **10s** retries: **5** timeout: **10s** mongodb:  
 image: **mongo:5** container\_name: **MongoDB** env\_file:  
 - **.env** ports:  
 - "27017-27019:${MONGO\_PORT:-27017-27019}"  
 volumes:  
 - **mongo\_data:/data/db** healthcheck:  
 test: ["CMD","mongosh", "--eval", "db.adminCommand('ping')"]  
 interval: **10s** retries: **5** timeout: **10s** redis:  
 image: **redis** container\_name: **Redis\_cache** ports:  
 - **6379:${REDIS\_PORT:-6379}** env\_file:  
 - **.env** healthcheck:  
 test: ["CMD-SHELL", "redis-cli ping | grep PONG"]  
 interval: **10s** retries: **5** timeout: **10s**volumes:  
 - mongo\_data:

Тут использовались docker-volumes для сохранения данных в базах между запусками приложения и для прокидывания необходимых файлов внутрь контейнера. Также использовались healthchecks, гарантирующие, что вся вспомогательная инфраструктура работает исправно и приложение может с ней взаимодействовать. Само приложение также было запаковано в docker контейнер для автоматизации развёртывания. При сборке docker image использовалась multistage сборка, что позволило существенно уменьшить объём конечного образа.

**Dockerfile**

FROM golang:1.23.3-alpine AS **build\_stage**LABEL authors="Benzogang-Tape"  
  
ARG **PORT**ARG **APP\_NAME**ARG **BUILD\_DATE**LABEL build\_date=$**BUILD\_DATE**WORKDIR /app  
  
COPY go.mod go.sum ./  
  
RUN go mod download  
  
COPY . .  
  
RUN go build -o /bin/$**APP\_NAME** ./cmd/$**APP\_NAME**FROM alpine AS **run\_stage**RUN mkdir /app  
WORKDIR /app  
RUN mkdir /static  
RUN mkdir /docs  
  
COPY --from=**build\_stage** /bin/$**APP\_NAME** .  
  
RUN chmod +x ./$**APP\_NAME**#EXPOSE $PORT  
#  
#ENTRYPOINT ./$APP\_NAME  
  
EXPOSE $**PORT**CMD ["sh", "-c", "./$**APP\_NAME**" ]

Для конфигурации приложения использовались переменные окружения, которые были описаны в файле .env, а также .yaml файл, описывающий все необходимые конфиги. Приоритет был следующий: сначала поиск производился по переменным среды, в случае если нужная переменная не была установлена значение для неё бралось из config.yaml файла.

**config.yaml**

# Prioritized lower than environment variables  
APP:  
 PORT: "8080"  
 NAME: **redditclone**MYSQL:  
 HOST: "mysql"  
 PORT: "3306"  
 DATABASE: **reddit** PARAMS: "charset=utf8&interpolateParams=true"  
 USER: "user"  
 PASSWORD: "password"  
 ROOT:  
 PASSWORD: "root\_pass"  
  
MONGO:  
 URI: "mongodb://"  
 HOST: "mongodb"  
 INITDB:  
 DATABASE: **reddit** COLLECTION:  
 POSTS: "posts"  
  
REDIS:  
 HOST: "redis"  
 PORT: "6379"  
 PASSWORD: ""  
  
JWT:  
 SECRET: "super secret key"

Файл config.go отвечал за инициализацию и правильное чтение конфигов

**config.go**

package config  
  
import (  
 "fmt"  
 "strings"  
  
 "github.com/spf13/viper"  
)  
  
const (  
 configFileName = "config"  
 configFileExt = "yaml"  
 configFilePath = "."  
)  
  
func ReadConfig() (\*viper.Viper, error) {  
 v := viper.New()  
 v.SetConfigName(configFileName)  
 v.AddConfigPath(configFilePath)  
 v.SetConfigType(configFileExt)  
  
 v.SetEnvKeyReplacer(strings.NewReplacer(".", "\_"))  
  
 if err := v.ReadInConfig(); err != nil {  
 return nil, fmt.Errorf("error reading config file, %s", err)  
 }  
  
 v.AutomaticEnv()  
  
 return v, nil  
}

В файле redditclone.go собираются все зависимости, производятся подключения к необходимой инфраструктуре с использованием конфигов и запускается приложение.

**redditclone.go**

package main  
  
import (  
 "context"  
 "database/sql"  
 "fmt"  
 "log"  
 "net/http"  
  
 "github.com/go-redis/redis"  
 \_ "github.com/go-sql-driver/mysql"  
 "go.mongodb.org/mongo-driver/mongo"  
 "go.mongodb.org/mongo-driver/mongo/options"  
 "go.uber.org/zap"  
  
 "github.com/Benzogang-Tape/Reddit/internal/config"  
 "github.com/Benzogang-Tape/Reddit/internal/models/jwt"  
 "github.com/Benzogang-Tape/Reddit/internal/service"  
 "github.com/Benzogang-Tape/Reddit/internal/storage"  
 "github.com/Benzogang-Tape/Reddit/internal/transport/rest"  
)  
  
// @title Reddit-Clone API  
// @version 1.0  
// @description Basic restfull api for reddit-clone backend.  
// @termsOfService http://swagger.io/terms/  
  
// @contact.name API Support  
// @contact.url http://www.swagger.io/support  
// @contact.email support@swagger.io  
  
// @license.name Apache 2.0  
// @license.url http://www.apache.org/licenses/LICENSE-2.0.html  
  
// @host localhost:8081  
// @BasePath /api  
  
// @securityDefinitions.apikey ApiKeyAuth  
// @in header  
// @name Authorization  
  
// @externalDocs.description OpenAPI  
// @externalDocs.url https://swagger.io/resources/open-api/  
func main() {  
 v, err := config.ReadConfig()  
 if err != nil {  
 panic(err)  
 }  
  
 if err = jwt.SetJWTSecret(v.GetString("jwt.secret")); err != nil {  
 panic(err)  
 }  
  
 dsn := fmt.Sprintf(  
 "%s:%s@tcp(%s:%s)/%s?%s",  
 v.GetString("mysql.user"),  
 v.GetString("mysql.password"),  
 v.GetString("mysql.host"),  
 v.GetString("mysql.port"),  
 v.GetString("mysql.database"),  
 v.GetString("mysql.params"),  
 )  
  
 fmt.Println(dsn)  
 usersDB, err := sql.Open("mysql", dsn)  
 if err != nil {  
 panic(err)  
 }  
  
 err = usersDB.Ping()  
 if err != nil {  
 panic(err)  
 }  
  
 ctx := context.Background()  
 sess, err := mongo.Connect(ctx, options.Client().ApplyURI(fmt.Sprintf(  
 "%s%s",  
 v.GetString("mongo.uri"),  
 v.GetString("mongo.host"),  
 )))  
  
 if err != nil {  
 panic(err)  
 }  
  
 postsDB := sess.Database(v.GetString("mongo.initdb.database")).Collection(v.GetString("mongo.collection.posts"))  
  
 sessionDB := redis.NewClient(&redis.Options{  
 Addr**:** fmt.Sprintf("%s:%s", v.GetString("redis.host"), v.GetString("redis.port")),  
 Password**:** v.GetString("redis.password"),  
 DB**:** 0,  
 })  
  
 if \_, err = sessionDB.Ping().Result(); err != nil {  
 panic(err)  
 }  
  
 zapLogger, err := zap.NewProduction()  
 if err != nil {  
 log.Fatalln("Logger init error")  
 }  
 defer zapLogger.Sync() //nolint:errcheck  
 logger := zapLogger.Sugar()  
  
 sessionStorage := storage.NewSessionRepoRedis(sessionDB)  
 sessionHandler := service.NewSessionHandler(sessionStorage)  
  
 userStorage := storage.NewUserRepoMySQL(usersDB)  
 userHandler := service.NewUserHandler(userStorage)  
 u := rest.NewUserHandler(userHandler, sessionHandler, logger)  
  
 mongoAbstraction := storage.NewMongoCollection(postsDB)  
 postStorage := storage.NewPostRepoMongoDB(mongoAbstraction)  
 postHandler := service.NewPostHandler(postStorage, postStorage)  
 p := rest.NewPostHandler(postHandler, logger)  
  
 router := rest.NewAppRouter(u, p).InitRouter(logger)  
  
 addr := fmt.Sprintf(":%s", v.GetString("app.port"))  
 logger.Infow(fmt.Sprintf("Starting server on %s", addr))  
 log.Panic(http.ListenAndServe(addr, router))  
}

Также было реализовано 100% покрытие тестами хендлеров и репозитория. Генерация моков репозитория производилась с помощью утилиты [github.com/golang/mock/mockgen@v1.6.0](mailto:github.com/golang/mock/mockgen@v1.6.0). Файлы покрытия генерируются в директорию coverage.

Документация API производилась с использованием библиотеки <https://github.com/swaggo/swag>. Путём описания аннотаций в коде. Спецификация генерируется в директорию docs в корне репозитория. При запуске приложения документация доступна по пути /swagger/index.html

Для контроля единообразия и качества кода в проекте настроен линтер golangci-lint. Конфигурация линтера описана в файле .golangci.yaml

**.golangci.yaml**

linters:  
 disable-all: **true** enable:  
 # default  
 - **errcheck** - **gosimple** - **govet** - **ineffassign**# - staticcheck  
 - **unused** # custom  
 - **bodyclose**# - dupl  
 - **goconst**# - gocritic  
 - **gocyclo** - **nakedret** - **gocognit** - **funlen** - **prealloc** - **revive** - **unconvert** - **unparam** - **sqlclosecheck** - **gofmt** - **goimports** - **nilnil**# - testifylint  
 - **gosec**# - lll  
 fast: **false**linters-settings:  
 errcheck:  
 check-blank: **true** govet:  
 enable:  
 - **shadow** revive:  
 confidence: **0.3** unused:  
 post-statements-are-reads: **true** dupl:  
 threshold: **100** goconst:  
 numbers: **true** gocyclo:  
 min-complexity: **20** nakedret:  
 max-func-lines: **50** gocognit:  
 min-complexity: **20** funlen:  
 lines: **80** statements: **50** prealloc:  
 for-loops: **true** unconvert:  
 fast-math: **true** unparam:  
 check-exported: **true** gofmt:  
 rewrite-rules:  
 - pattern: 'interface{}'  
 replacement: 'any'  
 - pattern: 'a[b:len(a)]'  
 replacement: 'a[b:]'  
 goimports:  
 local-prefixes: **github.com/Benzogang-Tape/Reddit** nilnil:  
 detect-opposite: **true** gosec:  
 excludes:  
 - **G115** severity: **high** lll:  
 line-length: **120** tab-width: **4**issues:  
 max-issues-per-linter: **0** max-same-issues: **0** exclude:  
 - "should have comment"  
 - "always receives"  
 - "parameter .\* is always"  
 - "comment on exported .\* should be of the form"  
 exclude-rules:  
 - linters:  
 - **lll** source: "^//go:generate "  
  
output:  
 formats:  
 - format: **colored-line-number** path: **stdout** show-stats: **true**run:  
 timeout: **2m** issues-exit-code: **1** tests: **true** go: '1.23'

Все основные сценарии и скрипты описаны в файле Makefile

**Makefile**

APP\_NAME = redditclone  
APP\_DEMO = demo  
  
.PHONY:  
.SILENT:  
.DEFAULT\_GOAL := run  
  
.PHONY: build  
build:  
 go build -v -o ./bin/${APP\_NAME} ./cmd/${APP\_NAME}  
  
.PHONY: build-demo  
build-demo:  
 go build -v -o ./bin/${APP\_DEMO} ./cmd/${APP\_DEMO}  
  
.PHONY: run  
run: swag  
 docker-compose --env-file .env up -d  
 #docker-compose --env APP\_NAME=${APP\_NAME} up -d  
  
.PHONY: run-demo  
run-demo: swag-demo build-demo  
 ./bin/${APP\_DEMO}  
  
.PHONY: lint  
lint:  
 golangci-lint run \  
 -v -c .golangci.yaml --color='always' \  
 --exclude-dirs-use-default --exclude-files './internal/storage/mocks/\*','\\*.mod','\\*.sum' \  
 --exclude-dirs 'vendor'  
  
.PHONY: test  
test: gen  
 go test -v -coverprofile=./coverage/cover.out ./...  
 make test.coverage  
  
.PHONY: test.coverage  
test.coverage:  
 go tool cover -html=./coverage/cover.out -o ./coverage/cover.html  
 go tool cover -func=./coverage/cover.out | grep "total"  
  
.PHONY: gen  
gen:  
 go generate ./...  
  
.PHONY: clean  
clean:  
 go clean  
 rm -f ./bin/${APP\_NAME} ./bin/${APP\_DEMO}  
  
.PHONY: swag  
swag:  
 swag fmt  
 swag init -g ./cmd/${APP\_NAME}/${APP\_NAME}.go  
  
.PHONY: swag-demo  
swag-demo:  
 swag fmt  
 swag init -g ./cmd/${APP\_DEMO}/${APP\_DEMO}.go

Также для проекта частично настроен CI/CD c использованием github-actions.

В дальнейшем планируется доработать пайплайн добавив отчёты о покрытии кода тестами, прогоны линтеров, сборку и деплой приложения на виртуальную машину.

**redditclone.yml**

name: **CI-CD**on:  
 push:  
 branches: [**main**, **master**]  
 pull\_request:  
 branches: [**main**, **master**]  
 types: [**opened**]  
  
#env:  
  
  
jobs:  
# lint:  
  
 test:  
 name: "Unit testing"  
 runs-on: **ubuntu-latest** if: **github.event\_name == 'push'** steps:  
 - uses: **actions/checkout@v4** - name: **Setup Go** uses: **actions/setup-go@v5** with:  
 go-version: '1.23.x'  
  
 - name: **Install deps** run: **go mod tidy** - name: **Test with Go** run: **go test ./...**# build:  
# deploy:

Полная структура проекта:

**.**

**├── Dockerfile**

**├── LICENSE**

**├── Makefile**

**├── README.md**

**├── bin**

**├── cmd**

**│   ├── demo**

**│   │   └── demo.go**

**│   └── redditclone**

**│   └── redditclone.go**

**├── coverage**

**│   ├── cover.html**

**│   └── cover.out**

**├── docker-compose.yml**

**├── docs**

**│   ├── docs.go**

**│   ├── swagger.json**

**│   └── swagger.yaml**

**├── go.mod**

**├── go.sum**

**├── init**

**│   └── db\_tables**

**│   └── users.sql**

**├── internal**

**│   ├── config**

**│   │   ├── config.go**

**│   │   └── config.yaml**

**│   ├── models**

**│   │   ├── errs**

**│   │   │   └── errors.go**

**│   │   ├── httpresp**

**│   │   │   └── http\_response.go**

**│   │   ├── jwt**

**│   │   │   ├── secret\_conf.go**

**│   │   │   └── session.go**

**│   │   ├── posts**

**│   │   │   ├── post.go**

**│   │   │   └── post\_attributes.go**

**│   │   └── users**

**│   │   └── user.go**

**│   ├── service**

**│   │   ├── post.go**

**│   │   ├── session.go**

**│   │   └── user.go**

**│   ├── storage**

**│   │   ├── inmem**

**│   │   │   ├── posts\_repo.go**

**│   │   │   ├── session\_repo.go**

**│   │   │   └── users\_repo.go**

**│   │   ├── mocks**

**│   │   │   ├── mongoDB\_abstraction\_mock.go**

**│   │   │   ├── posts\_repo\_mongoDB\_mock.go**

**│   │   │   ├── sessions\_repo\_redis\_mock.go**

**│   │   │   └── users\_repo\_mySQL\_mock.go**

**│   │   ├── mongoDB\_abstraction.go**

**│   │   ├── posts\_repo\_mongoDB.go**

**│   │   ├── sessions\_repo\_redis.go**

**│   │   ├── tests**

**│   │   │   ├── posts\_repo\_mongoDB\_test.go**

**│   │   │   └── users\_repo\_mySQL\_test.go**

**│   │   └── users\_repo\_mySQL.go**

**│   └── transport**

**│   ├── middleware**

**│   │   ├── jwt\_auth.go**

**│   │   └── recover.go**

**│   └── rest**

**│   ├── post.go**

**│   ├── router.go**

**│   ├── send\_response.go**

**│   ├── tests**

**│   │   ├── post\_test.go**

**│   │   └── user\_test.go**

**│   └── user.go**

**├── pkg**

**│   └── middleware**

**│   └── accesslog.go**

**├── sreenshots**

**│   ├── commenting.png**

**│   ├── log\_in.png**

**│   ├── news\_feed.png**

**│   └── post\_creation.png**

**└── static**

**├── css**

**│   └── main.74225161.chunk.css**

**├── html**

**│   └── index.html**

**└── js**

**├── 2.d59deea0.chunk.js**

**└── main.32ebaf54.chunk.js**

**Все файлы, не представленные в отчёте, можно найти в репозитории проекта:** [**https://github.com/Benzogang-Tape/Reddit**](https://github.com/Benzogang-Tape/Reddit)