САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Бэк-энд разработка

Отчет

Лабораторная работа №1
Boilerplat

Выполнил:
Даньшин Семён
К3340

Проверил:
Добряков Д. И.

Санкт-Петербург

2025 г.

Задача

Написать свой boilerplate на express + TypeORM + typescript. Должно быть явное разделение на модели, контроллеры, роуты.

Ход работы

1. Анализ текущей архитектуры

В проекте реализована современная архитектура с использованием Go вместо TypeScript, но принципы разделения слоев соблюдены:

2. Архитектура проекта

Структура backend:

```
backend/
├── cmd/                    # Точки входа
│   ├── migrate/            # Миграции БД
│   └── workouts/           # Основное приложение
├── internal/
│   ├── app/                # Слой приложения (контроллеры)
│   ├── domain/             # Доменные модели
│   ├── repository/         # Слой данных
│   ├── service/            # Бизнес логика
│   └── clients/            # Внешние клиенты
└── pkg/                    # Сгенерированные protobuf
```

3. Слой моделей (Domain)

Основные доменные модели (internal/domain/domain.go):

```
type Model struct {
    ID        ID
    CreatedAt time.Time
    UpdatedAt time.Time
}

type User struct {
    Model
    Email         string
    Password      string
    FirstName     string
    LastName      string
    DateOfBirth   time.Time
    Height        float32
    Weight        float32
    ProfilePicURL string
}

type Exercise struct {
    Model
```

```
    Name               string
    Description        string
    VideoURL           string
    TargetMuscleGroups []MuscleGroup
}

type Workout struct {
    Model
    UserID        ID
    RoutineID     utils.Nullable[ID]
    Notes         string
    Rating        int
    FinishedAt    time.Time
    IsAIGenerated bool
    Reasoning     string
}

type Routine struct {
    Model
    Name        string
    Description string
    UserID      ID
}
```

4. Слой репозитория (Repository Pattern)

Интерфейсы репозиториев (internal/service/service.go):

```
type userRepository interface {
    GetUserByEmail(ctx context.Context, email string) (domain.User, error)
    GetUserByID(ctx context.Context, id domain.ID) (domain.User, error)
    CreateUser(ctx context.Context, user domain.User) (domain.User, error)
    UpdateUser(ctx context.Context, user domain.User) (domain.User, error)
}

type exerciseRepository interface {
    GetExercises(ctx context.Context, muscleGroups, excludedExercises
[]domain.ID) ([]domain.Exercise, error)
    GetExerciseByID(ctx context.Context, id domain.ID) (domain.Exercise,
error)
    CreateExercise(ctx context.Context, exercise domain.Exercise,
muscleGroupsIDs []domain.ID) (domain.Exercise, error)
}

type workoutRepository interface {
    GetWorkouts(ctx context.Context, userID domain.ID, limit, offset int)
([]domain.Workout, error)
    CreateWorkout(ctx context.Context, workout domain.Workout)
(domain.Workout, error)
    GetWorkoutByID(ctx context.Context, id domain.ID) (domain.Workout,
error)
    UpdateWorkout(ctx context.Context, id domain.ID, workout
domain.Workout) (domain.Workout, error)
    DeleteWorkout(ctx context.Context, id domain.ID) error
}
```

Реализация репозитория (internal/repository/):

```
type PGXRepository struct {
    contextManager *db.ContextManager
```

```
}

func (r *PGXRepository) CreateUser(ctx context.Context, user domain.User)
(domain.User, error) {
    span, ctx := opentracing.StartSpanFromContext(ctx,
"repository.CreateUser")
    defer span.Finish()

    query := `
        INSERT INTO users (id, email, password, first_name, last_name,
date_of_birth, height, weight, created_at, updated_at, picture_profile_url)
        VALUES ($1, $2, $3, $4, $5, $6, $7, $8, $9, $10, $11)
        RETURNING *
    `

    engine := r.contextManager.GetEngineFromContext(ctx)
    userEntity := userFromDomain(user)

    err := pgxscan.Get(ctx, engine, &userEntity, query, /* параметры */)
    if err != nil {
        return domain.User{}, err
    }

    return userEntity.toDomain(), nil
}
```

5. Слой сервисов (Business Logic)

Сервисный слой (internal/service/):

```
type Service struct {
    jwtProvider                 jwtProvider
    userRepository              userRepository
    exerciseRepository          exerciseRepository
    workoutRepository           workoutRepository
    // ... другие зависимости
}

func (s *Service) CreateUser(ctx context.Context, dto dto.CreateUserDTO)
(domain.User, error) {
    span, ctx := opentracing.StartSpanFromContext(ctx,
"service.CreateUser")
    defer span.Finish()

    // Валидация
    if dto.Email == "" {
        return domain.User{}, errors.New("email is required")
    }

    // Хеширование пароля
    hashedPassword, err := utils.HashPassword(dto.Password)
    if err != nil {
        return domain.User{}, err
    }

    // Создание доменной модели
    user := domain.NewUser(
        dto.Email,
        hashedPassword,
        dto.FirstName,
```

```
        dto.LastName,
        dto.DateOfBirth,
        dto.Height,
        dto.Weight,
    )

    // Сохранение через репозиторий
    createdUser, err := s.userRepository.CreateUser(ctx, user)
    if err != nil {
        return domain.User{}, err
    }

    // Отправка welcome email
    go func() {
        if err := s.emailService.SendWelcomeEmail(context.Background(),
user.Email, user.FirstName); err != nil {
            logger.Errorf("failed to send welcome email: %v", err)
        }
    }()

    return createdUser, nil
}
```

6. Слой контроллеров (API Layer)

gRPC контроллеры (internal/app/fitness-trainer/api/):

```
type Implementation struct {
    service Service
    desc.UnimplementedUserServiceServer
}

func (i *Implementation) CreateUser(ctx context.Context, req
*desc.CreateUserRequest) (*desc.UserResponse, error) {
    span, ctx := opentracing.StartSpanFromContext(ctx, "api.CreateUser")
    defer span.Finish()

    // Преобразование запроса в DTO
    createDTO := dto.CreateUserDTO{
        Email:       req.Email,
        Password:    req.Password,
        FirstName:   req.FirstName,
        LastName:    req.LastName,
        DateOfBirth: req.DateOfBirth.AsTime(),
        Height:      req.Height,
        Weight:      req.Weight,
    }

    // Вызов сервиса
    user, err := i.service.CreateUser(ctx, createDTO)
    if err != nil {
        return nil, status.Error(codes.Internal, err.Error())
    }

    // Преобразование ответа
    return &desc.UserResponse{
        User: &desc.User{
            Id:          user.ID.String(),
            Email:       user.Email,
            FirstName:   user.FirstName,
```

```
            LastName:    user.LastName,
            DateOfBirth: timestamppb.New(user.DateOfBirth),
            Height:      user.Height,
            Weight:      user.Weight,
            CreatedAt:   timestamppb.New(user.CreatedAt),
            UpdatedAt:   timestamppb.New(user.UpdatedAt),
        },
    }, nil
}
```

7. Слой маршрутизации (Routes)

Protobuf определения (api/workouts/workouts.proto):

```
service UserService {
  rpc CreateUser(CreateUserRequest) returns (UserResponse);
  rpc GetUser(GetUserRequest) returns (UserResponse);
  rpc GetMe(google.protobuf.Empty) returns (UserResponse);
  rpc UpdateUser(UpdateUserRequest) returns (UserResponse);
  rpc
UpdateWorkoutGenerationSettings(UpdateWorkoutGenerationSettingsRequest)
returns (google.protobuf.Empty);
  rpc GetWorkoutGenerationSettings(google.protobuf.Empty) returns
(WorkoutGenerationSettingsResponse);
}

service ExerciseService {
  rpc GetExercises(GetExercisesRequest) returns (GetExercisesResponse);
  rpc CreateExercise(CreateExerciseRequest) returns (ExerciseResponse);
  rpc GetExerciseDetail(GetExerciseDetailRequest) returns
(ExerciseResponse);
  rpc GetExerciseAlternatives(GetExerciseAlternativesRequest) returns
(GetExerciseAlternativesResponse);
  rpc GetExerciseHistory(GetExerciseHistoryRequest) returns
(ExerciseHistoryResponse);
  rpc GetMuscleGroups(google.protobuf.Empty) returns
(GetMuscleGroupsResponse);
}
```

HTTP маршруты (автогенерация через gRPC-Gateway):

```
POST   /v1/users                    # CreateUser
GET    /v1/users/me                 # GetMe
PUT    /v1/users                    # UpdateUser
GET    /v1/users/{userId}           # GetUser

GET    /v1/exercises                # GetExercises
POST   /v1/exercises                # CreateExercise
GET    /v1/exercises/{exerciseId}   # GetExerciseDetail

GET    /v1/routines                 # GetRoutines
POST   /v1/routines                 # CreateRoutine
GET    /v1/routines/{routineId}     # GetRoutineDetail

GET    /v1/workouts                 # GetWorkouts
POST   /v1/workouts                 # StartWorkout
GET    /v1/workouts/{workoutId}     # GetWorkout
```

8. Конфигурация и зависимости

Dependency Injection (cmd/workouts/main.go):

```
func main() {
    // Инициализация компонентов
    contextManager := db.NewContextManager(pool)
    repository := repository.NewPGXRepository(contextManager)
    jwtProvider := jwt.NewProvider(config.JWT)
    emailService := email.NewService(kafkaProducer, "email-topic")

    // Создание сервиса с внедрением зависимостей
    service := service.New(
        unitOfWork,
        jwtProvider,
        s3Client,
        workoutGenerator,
        generateWorkoutLimiter,
        repository, // sessionRepository
        repository, // userRepository
        repository, // exerciseRepository
        repository, // routineRepository
        repository, // exerciseInstanceRepository
        repository, // muscleGroupRepository
        repository, // workoutRepository
        repository, // exerciseLogRepository
        repository, // setLogRepository
        repository, // setRepository
        repository, // expectedSetRepository
        repository, // generationSettingsRepository
        emailService,
    )

    // Регистрация gRPC сервисов
    userAPI := user.New(service)
    exerciseAPI := exercise.New(service)
    routineAPI := routine.New(service)
    workoutAPI := workout.New(service)

    // Регистрация HTTP маршрутов
    desc.RegisterUserServiceServer(grpcServer, userAPI)
    desc.RegisterExerciseServiceServer(grpcServer, exerciseAPI)
    desc.RegisterRoutineServiceServer(grpcServer, routineAPI)
    desc.RegisterWorkoutServiceServer(grpcServer, workoutAPI)
}
```

9. Middleware и обработка ошибок

Аутентификация middleware:

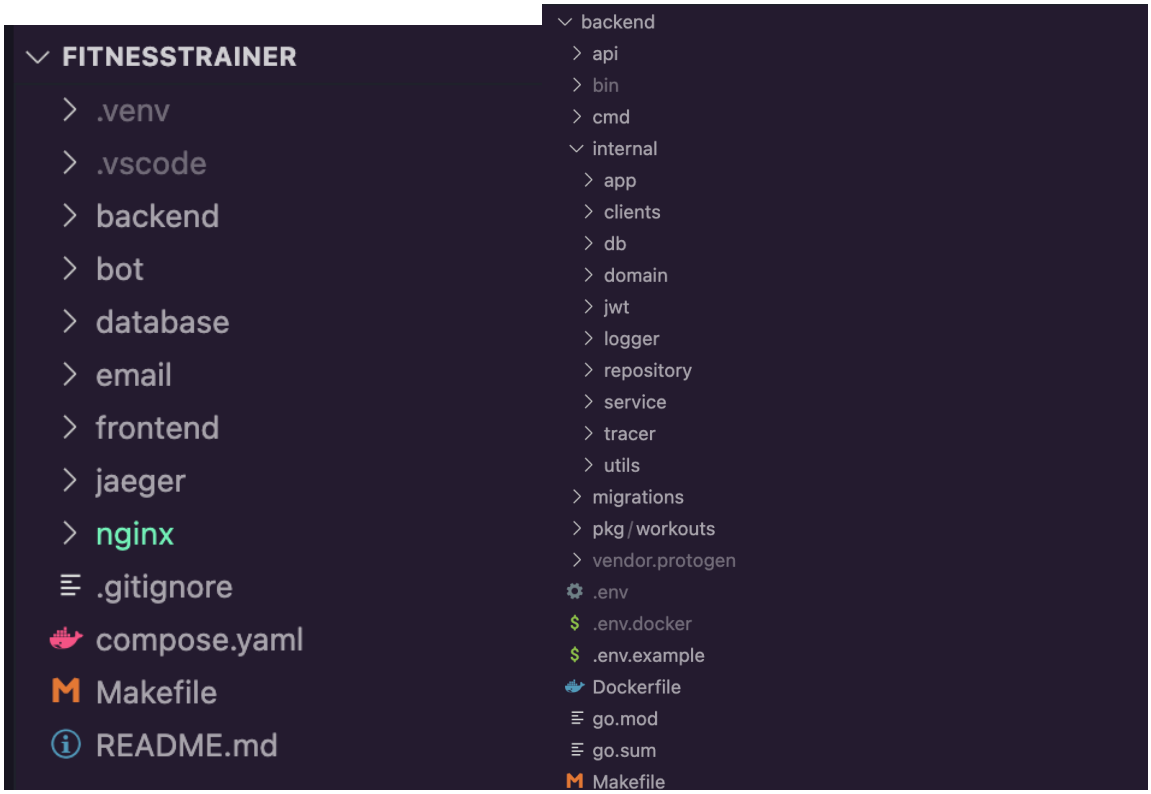
```
func (i *AuthInterceptor) UnaryServerInterceptor()
grpc.UnaryServerInterceptor {
    return func(ctx context.Context, req interface{}, info
*grpc.UnaryServerInfo, handler grpc.UnaryHandler) (interface{}, error) {
        // Проверка токена аутентификации
        userID, err := i.extractUserID(ctx)
        if err != nil {
            return nil, status.Error(codes.Unauthenticated, "invalid
token")
```

```
        }

        // Добавление userID в контекст
        ctx = context.WithValue(ctx, "userID", userID)
        return handler(ctx, req)
    }
}
```

10. Валидация и DTO

Data Transfer Objects (internal/domain/dto/):

```
type CreateUserDTO struct {
    Email       string    `validate:"required,email"`
    Password    string    `validate:"required,min=8"`
    FirstName   string    `validate:"required"`
    LastName    string    `validate:"required"`
    DateOfBirth time.Time
    Height      float32   `validate:"min=0"`
    Weight      float32   `validate:"min=0"`
}

type UpdateUserDTO struct {
    FirstName   *string
    LastName    *string
    DateOfBirth *time.Time
    Height      *float32
    Weight      *float32
}
```



Вывод

Реализован современный boilerplate с четким разделением на слои:

1. **Модели (Domain)** - чистые доменные сущности без зависимостей
2. **Репозитории (Repository)** - абстракция доступа к данным
3. **Сервисы (Service)** - бизнес-логика приложения
4. **Контроллеры (API)** - обработка HTTP/gRPC запросов
5. **Маршруты (Routes)** - определены через protobuf и автогенерация

Преимущества архитектуры:

- Четкое разделение ответственности
- Легкость тестирования каждого слоя
- Возможность замены реализации без изменения интерфейсов
- Масштабируемость и поддерживаемость кода
- Поддержка как HTTP, так и gRPC протоколов

Дополнительные возможности:

- Трассировка запросов с OpenTelemetry
- Валидация входных данных
- Обработка ошибок и возврат корректных HTTP статусов
- Dependency Injection для гибкой настройки зависимостей