САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Бэк-энд разработка

Отчет

Лабораторная работа №2

Выполнила:

Платонова Александра

Группа К3339

Проверил:
Добряков Д. И.

Санкт-Петербург

2025 г.

Задача

По выбранному варианту необходимо реализовать RESTful API средствами express + typescript (используя ранее написанный boilerplate).

Описание предметной области:

Прикладное программное обеспечение деятельности отдела заселения муниципальных общежитий администрации города. В ведении администрации города находится несколько десятков общежитий. Раньше они принадлежали предприятиям города, а теперь, после банкротства предприятий, все эти общежития переданы муниципальным властям. В последние годы бесплатные квартиры гражданам города практически не предоставляются, а количество малоимущих жителей, нуждающихся в жилье, растет. Хоть как-то улучшить жилищные условия этой категории граждан позволяет наличие муниципальных общежитий. Получить четкую картину их заселения позволит данное программное обеспечение. База данных отдела содержит информацию об общежитиях, комнатах общежитий и проживающих.

Ход работы

На первом этапе была проанализирована структура сущностей (Address, Hostel, Room, CheckInOut, Payment, Resident) и их взаимосвязи. Определены основные операции, которые должны поддерживаться для каждой сущности:

- CRUD-операции для каждой сущности
- Фильтрация (поиск по различным полям, диапазонам дат, статусам)
- Работа со связанными данными и составными запросами

Для каждой сущности создан отдельный репозиторий, содержащий:

- Методы CRUD

- Поиск по текстовым полям (с частичным совпадением)
- Фильтрация по числовым диапазонам (например, площадь комнаты)
- Работа с датами (например, поиск CheckInOut за определенный период)
- Подгрузка связанных данных

Особое внимание уделено ограничениям (например, статус Payment может быть только p, np или pp, то есть оплачено, частично оплачено, не оплачено).

Для каждой сущности создан контроллер, который:

1. Принимает HTTP-запросы (GET, POST, PUT, DELETE)
2. Валидирует входные данные (проверка ID, обязательных полей)
3. Вызывает соответствующий метод репозитория
4. Обрабатывает результаты и ошибки, возвращая клиенту:

    - Успешные ответы (201 Created, 200 OK, 204 No Content)
    - Ошибки (400 Bad Request, 404 Not Found, 500 Internal Server Error)

Все контроллеры следуют единой структуре, что в дальнейшем способствует упрощению модификации и расширения функциональных возможностей приложения.

Вывод

В ходе выполнения лабораторной работы было реализовано RESTful API приложение средствами express + typescript, в качестве основы был взят шаблон проекта, созданный в лабораторной работе №1. Логика работы с базой данных вынесена в репозитории, а обработка HTTP-запросов в контроллеры. Поставленные задачи выполнены в полном объеме.

ПРИЛОЖЕНИЕ А

Программный код

Описание репозиториев приведено в листинге А.1.

```
@EntityRepository(Address)
export class AddressRepository extends Repository<Address> {
  async createAddress(addressData: Partial<Address>): Promise<Address> {
    const address = this.create(addressData);
    return this.save(address);
  }
  async findWithFilters(filters: {
    city_district?: string;
    street?: string;
    zip_code?: string;
  }): Promise<Address[]> {
    return this.find({
      where: {
        city_district: filters.city_district ? Like(`%${filters.city_district}%`) : undefined,
        street: filters.street ? Like(`%${filters.street}%`) : undefined,
        zip_code: filters.zip_code ? Like(`%${filters.zip_code}%`) : undefined,
      },
      relations: ['hostel'],
    });
  }
```

```
  async findById(id: number): Promise<Address | undefined> {
    return this.findOne({ where: { id }, relations: ['hostel'] });
  }
  async updateAddress(id: number, updateData: Partial<Address>): Promise<Address | undefined> {
    await this.update(id, updateData);
    return this.findOne({ where: { id } });
  }
  async deleteAddress(id: number): Promise<void> {
    await this.delete(id);
  }
}
```

@EntityRepository(Hostel)

```
export class HostelRepository extends Repository<Hostel> {
  async createHostel(hostelData: Partial<Hostel>): Promise<Hostel> {
    const hostel = this.create(hostelData);
    return this.save(hostel);
  }
  async findWithFilters(filters: {
    name?: string;
    house_num?: number;
    building?: number;
    organizationId?: number;
  }): Promise<Hostel[]> {
    return this.find({
      where: {
```

```
      name: filters.name ? Like(`%${filters.name}%`) : undefined,
      house_num: filters.house_num,
      building: filters.building,
      organization: { id: filters.organizationId },
    },
    relations: ['address', 'organization', 'rooms'],
  });
 }
 async findById(id: number): Promise<Hostel | undefined> {
  return this.findOne({
   where: { id },
   relations: ['address', 'organization', 'rooms']
  });
 }
 async updateHostel(id: number, updateData: Partial<Hostel>): Promise<Hostel | undefined>
{
  await this.update(id, updateData);
  return this.findOne({ where: { id } });
 }
 async deleteHostel(id: number): Promise<void> {
  await this.delete(id);
 }
}
@EntityRepository(Room)
export class RoomRepository extends Repository<Room> {
 async createRoom(roomData: Partial<Room>): Promise<Room> {
```

```
  const room = this.create(roomData);
  return this.save(room);
}
async findWithFilters(filters: {
  floor?: number;
  beds?: number;
  minArea?: number;
  maxArea?: number;
  busy_beds?: number;
  hostelId?: number;
}): Promise<Room[]> {
  return this.find({
    where: {
      floor: filters.floor,
      beds: filters.beds,
      area: filters.minArea && filters.maxArea
        ? Between(filters.minArea, filters.maxArea)
        : undefined,
      busy_beds: filters.busy_beds,
      hostel: { id: filters.hostelId },
    },
    relations: ['hostel', 'checkIns'],
  });
}
async findById(id: number): Promise<Room | undefined> {
```

```
    return this.findOne({
      where: { id },
      relations: ['hostel', 'checkIns']
    });
  }
  async updateRoom(id: number, updateData: Partial<Room>): Promise<Room | undefined> {
    await this.update(id, updateData);
    return this.findOne({ where: { id } });
  }
  async deleteRoom(id: number): Promise<void> {
    await this.delete(id);
  }
}
@EntityRepository(CheckInOut)
export class CheckInOutRepository extends Repository<CheckInOut> {
  async createCheckInOut(checkInOutData: Partial<CheckInOut>): Promise<CheckInOut> {
    const checkInOut = this.create(checkInOutData);
    return this.save(checkInOut);
  }
  async findWithFilters(filters: {
    doc_num?: number;
    date_from?: Date;
    date_to?: Date;
    residentId?: number;
    roomId?: number;
```

```
  }): Promise<CheckInOut[]> {
    return this.find({
      where: {
        doc_num: filters.doc_num,
        date_of_issue: filters.date_from && filters.date_to
          ? Between(filters.date_from, filters.date_to)
          : undefined,
        resident: { id: filters.residentId },
        room: { id: filters.roomId },
      },
      relations: ['resident', 'room', 'payment'],
    });
  }
  async findById(id: number): Promise<CheckInOut | undefined> {
    return this.findOne({
      where: { id },
      relations: ['resident', 'room', 'payment']
    });
  }
  async updateCheckInOut(id: number, updateData: Partial<CheckInOut>): Promise<CheckInOut | undefined> {
    await this.update(id, updateData);
    return this.findOne({ where: { id } });
  }
  async deleteCheckInOut(id: number): Promise<void> {
    await this.delete(id);
```

```
  }
}
@EntityRepository(Payment)
export class PaymentRepository extends Repository<Payment> {
  async createPayment(paymentData: Partial<Payment>): Promise<Payment> {
    const payment = this.create(paymentData);
    return this.save(payment);
  }
  async findWithFilters(filters: {
    status?: 'p' | 'np' | 'pp';
    minAmount?: number;
    maxAmount?: number;
    date_from?: Date;
    date_to?: Date;
    checkInOutId?: number;
  }): Promise<Payment[]> {
    return this.find({
      where: {
        status: filters.status,
        amount: filters.minAmount && filters.maxAmount
          ? Between(filters.minAmount, filters.maxAmount)
          : undefined,
        date_pay: filters.date_from && filters.date_to
          ? Between(filters.date_from, filters.date_to)
          : undefined,
```

```
      checkInOut: { id: filters.checkInOutId },
    },
    relations: ['checkInOut'],
  });
 }
 async findById(id: number): Promise<Payment | undefined> {
  return this.findOne({
    where: { id },
    relations: ['checkInOut']
  });
 }
 async updatePayment(id: number, updateData: Partial<Payment>): Promise<Payment | undefined> {
  await this.update(id, updateData);
  return this.findOne({ where: { id } });
 }
 async deletePayment(id: number): Promise<void> {
  await this.delete(id);
 }
}
```

@EntityRepository(Resident)

```
export class ResidentRepository extends Repository<Resident> {
 async createResident(residentData: Partial<Resident>): Promise<Resident> {
  const resident = this.create(residentData);
  return this.save(resident);
 }
```

```
async findWithFilters(filters: {
  full_name?: string;
  phone?: string;
  email?: string;
  date_from?: Date;
  date_to?: Date;
}): Promise<Resident[]> {
  return this.find({
    where: {
      full_name: filters.full_name ? Like(`%${filters.full_name}%`) : undefined,
      phone: filters.phone ? Like(`%${filters.phone}%`) : undefined,
      email: filters.email ? Like(`%${filters.email}%`) : undefined,
      created_at: filters.date_from && filters.date_to
        ? Between(filters.date_from, filters.date_to)
        : undefined,
    },
    relations: ['checkIns'],
  });
}
async findById(id: number): Promise<Resident | undefined> {
  return this.findOne({
    where: { id },
    relations: ['checkIns']
  });
}
```

```
  async updateResident(id: number, updateData: Partial<Resident>): Promise<Resident | undefined> {
    await this.update(id, updateData);
    return this.findOne({ where: { id } });
  }
  async deleteResident(id: number): Promise<void> {
    await this.delete(id);
  }
}
```

Дополненное описание контроллеров приведено в листинге А.2.

Листинг А.2. – Контроллеры

```
export class AddressController {
  private repository = new AddressRepository();

  async create(req: Request, res: Response) {
    try {
      const address = await this.repository.createAddress(req.body);
      res.status(201).json(address);
    } catch (error) {
      res.status(500).json({ error: 'Failed to create address' });
    }
  }
  async getAll(req: Request, res: Response) {
    try {
      const addresses = await this.repository.findWithFilters(req.query);
```

```
    res.json(addresses);
  } catch (error) {
    res.status(500).json({ error: 'Failed to fetch addresses' });
  }
}
async getById(req: Request, res: Response) {
  try {
    const address = await this.repository.findById(Number(req.params.id));
    address ? res.json(address) : res.status(404).json({ error: 'Not found' });
  } catch (error) {
    res.status(500).json({ error: 'Failed to fetch address' });
  }
}
async update(req: Request, res: Response) {
  try {
    const address = await this.repository.updateAddress(Number(req.params.id), req.body);
    address ? res.json(address) : res.status(404).json({ error: 'Not found' });
  } catch (error) {
    res.status(500).json({ error: 'Failed to update address' });
  }
}
async delete(req: Request, res: Response) {
  try {
    await this.repository.deleteAddress(Number(req.params.id));
    res.status(204).send();
```

```
    } catch (error) {
      res.status(500).json({ error: 'Failed to delete address' });
    }
  }
}
export class HostelController {
  private repository = new HostelRepository();

  async create(req: Request, res: Response) {
    try {
      const hostel = await this.repository.createHostel(req.body);
      res.status(201).json(hostel);
    } catch (error) {
      res.status(500).json({ error: 'Failed to create hostel' });
    }
  }

  async getAll(req: Request, res: Response) {
    try {
      const hostels = await this.repository.findWithFilters(req.query);
      res.json(hostels);
    } catch (error) {
      res.status(500).json({ error: 'Failed to fetch hostels' });
    }
  }
```

```
async getById(req: Request, res: Response) {
  try {
    const hostel = await this.repository.findById(Number(req.params.id));
    hostel ? res.json(hostel) : res.status(404).json({ error: 'Not found' });
  } catch (error) {
    res.status(500).json({ error: 'Failed to fetch hostel' });
  }
}

async update(req: Request, res: Response) {
  try {
    const hostel = await this.repository.updateHostel(Number(req.params.id), req.body);
    hostel ? res.json(hostel) : res.status(404).json({ error: 'Not found' });
  } catch (error) {
    res.status(500).json({ error: 'Failed to update hostel' });
  }
}

async delete(req: Request, res: Response) {
  try {
    await this.repository.deleteHostel(Number(req.params.id));
    res.status(204).send();
  } catch (error) {
    res.status(500).json({ error: 'Failed to delete hostel' });
```

```
    }
  }
}
export class RoomController {
  private repository = new RoomRepository();
  async create(req: Request, res: Response) {
    try {
      const room = await this.repository.createRoom(req.body);
      res.status(201).json(room);
    } catch (error) {
      res.status(500).json({ error: 'Failed to create room' });
    }
  }
  async getAll(req: Request, res: Response) {
    try {
      const rooms = await this.repository.findWithFilters(req.query);
      res.json(rooms);
    } catch (error) {
      res.status(500).json({ error: 'Failed to fetch rooms' });
    }
  }
  async getById(req: Request, res: Response) {
    try {
      const room = await this.repository.findById(Number(req.params.id));
      room ? res.json(room) : res.status(404).json({ error: 'Not found' });
```

```
    } catch (error) {
      res.status(500).json({ error: 'Failed to fetch room' });
    }
  }
  async update(req: Request, res: Response) {
    try {
      const room = await this.repository.updateRoom(Number(req.params.id), req.body);
      room ? res.json(room) : res.status(404).json({ error: 'Not found' });
    } catch (error) {
      res.status(500).json({ error: 'Failed to update room' });
    }
  }
  async delete(req: Request, res: Response) {
    try {
      await this.repository.deleteRoom(Number(req.params.id));
      res.status(204).send();
    } catch (error) {
      res.status(500).json({ error: 'Failed to delete room' });
    }
  }
}
export class CheckInOutController {
  private repository = new CheckInOutRepository();
  async create(req: Request, res: Response) {
    try {
```

```
      const checkInOut = await this.repository.createCheckInOut(req.body);
      res.status(201).json(checkInOut);
    } catch (error) {
      res.status(500).json({ error: 'Failed to create check-in record' });
    }
  }
  async getAll(req: Request, res: Response) {
    try {
      const checkIns = await this.repository.findWithFilters(req.query);
      res.json(checkIns);
    } catch (error) {
      res.status(500).json({ error: 'Failed to fetch check-ins' });
    }
  }
  async getById(req: Request, res: Response) {
    try {
      const checkIn = await this.repository.findById(Number(req.params.id));
      checkIn ? res.json(checkIn) : res.status(404).json({ error: 'Not found' });
    } catch (error) {
      res.status(500).json({ error: 'Failed to fetch check-in record' });
    }
  }
  async update(req: Request, res: Response) {
    try {
      const checkIn = await this.repository.updateCheckInOut(Number(req.params.id),
req.body);
```

```
      checkIn ? res.json(checkIn) : res.status(404).json({ error: 'Not found' });
    } catch (error) {
      res.status(500).json({ error: 'Failed to update check-in record' });
    }
  }
  async delete(req: Request, res: Response) {
    try {
      await this.repository.deleteCheckInOut(Number(req.params.id));
      res.status(204).send();
    } catch (error) {
      res.status(500).json({ error: 'Failed to delete check-in record' });
    }
  }
}
export class PaymentController {
  private repository = new PaymentRepository();
  async create(req: Request, res: Response) {
    try {
      const payment = await this.repository.createPayment(req.body);
      res.status(201).json(payment);
    } catch (error) {
      res.status(500).json({ error: 'Failed to create payment' });
    }
  }
  async getAll(req: Request, res: Response) {
```

```
  try {
    const payments = await this.repository.findWithFilters(req.query);
    res.json(payments);
  } catch (error) {
    res.status(500).json({ error: 'Failed to fetch payments' });
  }
}
async getById(req: Request, res: Response) {
  try {
    const payment = await this.repository.findById(Number(req.params.id));
    payment ? res.json(payment) : res.status(404).json({ error: 'Not found' });
  } catch (error) {
    res.status(500).json({ error: 'Failed to fetch payment' });
  }
}
async update(req: Request, res: Response) {
  try {
    const payment = await this.repository.updatePayment(Number(req.params.id), req.body);
    payment ? res.json(payment) : res.status(404).json({ error: 'Not found' });
  } catch (error) {
    res.status(500).json({ error: 'Failed to update payment' });
  }
}
async delete(req: Request, res: Response) {
  try {
```

```
      await this.repository.deletePayment(Number(req.params.id));
      res.status(204).send();
    } catch (error) {
      res.status(500).json({ error: 'Failed to delete payment' });
    }
  }
}
```