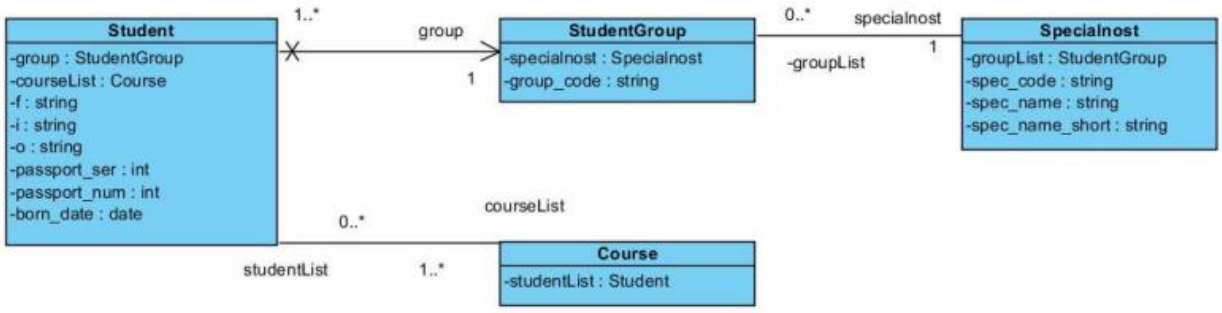
Проект: Java EE приложение "Деканат" с модулем JPA

Описание задачи:

Разработайте полнофункциональное веб-приложение "Деканат", используя Java EE и интегрирующее модуль JPA для работы с базой данных. Программа должна реализовать следующую схему сущностей:



Определите первичные и альтернативные ключи для всех сущностей.

Доработайте перечень методов сущностей, обеспечивая их полноту и корректность.

Требования к функциональности:

а) Страница специальностей:

Создайте веб-страницу для отображения списка всех доступных специальностей.

б) Страница групп по специальности:

Разработайте интерфейс для отображения списка групп, связанных с выбранной специальностью.

в) Страница студентов по группе (ленивая загрузка):

Реализуйте веб-интерфейс для отображения списка студентов, принадлежащих к выбранной группе.

Используйте ленивую загрузку для оптимизации производительности при работе с большими объемами данных.

г) Страница курсов по студенту:

Создайте страницу для отображения списка курсов, связанных с конкретным студентом.

Ключевые аспекты разработки:

Архитектура приложения на основе Java EE.

Интеграция JPA для управления данными в базе данных.

Создание пользовательского интерфейса с удобством использования.

Оптимизация производительности, особенно при использовании ленивой загрузки.

Правильная организация кода и соблюдение лучших практик Java EE.

Выполнение:

Для работы Java EE использовались следующие программы и компоненты: GlassFish, JDK, NetBeans, EJB, JPA.

Создана база данных деканата, содержащая таблицы студентов, групп, специальностей и курсов. Таблицы заполнены данными. Пример создания таблицы студентов представлен на рис.2.

Был создан веб-проект Java EE (содержание которого отображено на рис.1). Поскольку используется JPA, был создан файл конфигурации persistence.xml с указанием свойств подключения к базе данных.

Созданы классы сущностей из базы данных (Entity bean), сеансовые компоненты (Session Beans), контроллеры для обработки запросов, подготовки и передачи управления в созданные соответствующие jsp-файлы, которые отрисовывают результат.

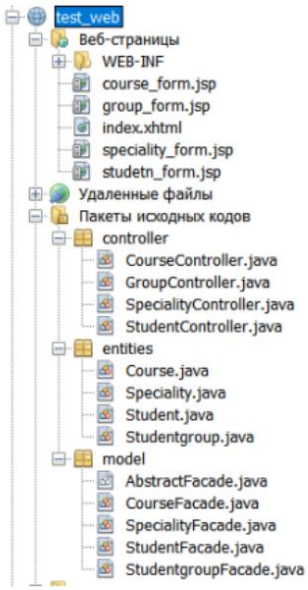


Рисунок 1

```
create table student (
id  int(3) PRIMARY KEY NOT NULL AUTO_INCREMENT,
groups_id int(3),
course_id int(3),
surname varchar(120) NOT NULL,
firstname varchar(120) NOT NULL,
lastname varchar(120),
passport varchar(120),
birthdate varchar(120),
FOREIGN KEY (groups_id)  REFERENCES studentGroup (id),
FOREIGN KEY (course_id)  REFERENCES course (id)
);
```

Рисунок 2

Часть класса Student (Entity Bean), который является отображением таблицы студентов, с соответствующими конструкторами, геттерами и сеттерами, представлена на рис.3. Классы остальных таблиц аналогичны.

```
@Entity
@Table(name = "student")
@XmlRootElement
@NamedQueries({ ////SQL запросы на получение данных из таблицы students с условием
    @NamedQuery(name = "Student.findAll", query = "SELECT s FROM Student s")
    , @NamedQuery(name = "Student.findById", query = "SELECT s FROM Student s WHERE s.id = :id")
    , @NamedQuery(name = "Student.findBySurname", query = "SELECT s FROM Student s WHERE s.surname = :surname")
    , @NamedQuery(name = "Student.findByGroupId", query = "SELECT s FROM Student s WHERE s.groupsId.id = :groupsId")
    , @NamedQuery(name = "Student.findCoursebyStudentId", query = "SELECT s FROM Student s WHERE s.courseId.id = :courseId")
    , @NamedQuery(name = "Student.findByFirstname", query = "SELECT s FROM Student s WHERE s.firstname = :firstname")
    , @NamedQuery(name = "Student.findByLastname", query = "SELECT s FROM Student s WHERE s.lastname = :lastname")
    , @NamedQuery(name = "Student.findByPassport", query = "SELECT s FROM Student s WHERE s.passport = :passport")
    , @NamedQuery(name = "Student.findByBirthdate", query = "SELECT s FROM Student s WHERE s.birthdate = :birthdate")})

public class Student implements Serializable {
    //обозначение полей таблицы
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Basic(optional = false)
    @Column(name = "id")
    private Integer id;
    @Basic(optional = false)
    @NotNull
    @Size(min = 1, max = 120)
    @Column(name = "surname")
    private String surname;
    @Basic(optional = false)
    @NotNull
```

Рисунок 3

Пример сеансового компонента StudentFacade для работы со студентами, который представляет собой простой интерфейс и скрывает сложность модели от клиента, находится на рис.4. Остальные созданы аналогично.

```java
@Stateless
public class StudentFacade extends AbstractFacade<Student> {

    @PersistenceContext(unitName = "test_webPU")
    private EntityManager em;

    @Override
    protected EntityManager getEntityManager() {
        return em;
    }
    public StudentFacade() {
        super(Student.class);
    }
    //поиск студентов по выбранной группе
    public List<Student> findByGroupId(int groupsId) {
        TypedQuery<Student> typedQuery = em.createNamedQuery("Student.findByGroupId", Student.class);
        typedQuery.setParameter("groupsId", groupsId);
        return typedQuery.getResultList();
    }
    //поиск студентов по выбранному курсу
    public List<Student> findCoursebyStudentId(int courseId) {
        TypedQuery<Student> typedQuery = em.createNamedQuery("Student.findCoursebyStudentId", Student.class);
        typedQuery.setParameter("courseId", courseId);
        return typedQuery.getResultList();
    }
}
```

Рисунок 4

На рис. 5 и 6 представлен контроллер StudentController для обработки запросов. Для работы со студентами реализована фильтрация студентов по параметру группы и курса, а так же вывод списка студентов. Остальные контроллеры реализованы аналогично.

```java
@Named(value = "studentController")
@Dependent
@WebServlet(name = "StudentController", urlPatterns = {"/StudentController"})
public class StudentController extends HttpServlet {

    @EJB
    private StudentFacade studentFacade;

    public StudentController() {
    }
    //получение списка всех студентов
    public List<Student> findAll() {
        return this.studentFacade.findAll();
    }
    //получение одного студента
    public Student find(int id) {
        return this.studentFacade.find(id);
    }
    //поиск студентов по выбранной группе
    public List<Student> findByGroupId(int groupsId) {
        return this.studentFacade.findByGroupId(groupsId);
    }
    //поиск студентов по выбранному курсу
    public List<Student> findCoursebyStudentId(int courseId) {
        return this.studentFacade.findCoursebyStudentId(courseId);
    }
```

Рисунок 5

```
//вызов метода, соответсвущего полученному значению параметра action
@Override
protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
    String action = request.getParameter("action");
    try {
        switch (action) {
            case "filter": //фильтр по группе
                filterStudent(request, response);
                break;
            case "course": //фильтр по курсу
                courseStudent(request, response);
                break;
            default: //вывод списка студентов
                listStudent(request, response);
                break;
        }
    } catch (SQLException ex) {throw new ServletException(ex);}
}
//передача списка всех студентов
private void listStudent(HttpServletRequest request, HttpServletResponse response)
        throws SQLException, IOException, ServletException {
    List<Student> student = findAll();
    request.setAttribute("student", student);
    RequestDispatcher dispatcher = request.getRequestDispatcher("studetn_form.jsp");
    dispatcher.forward(request, response);
}
//фильтр студентов по выбранной группе
private void filterStudent(HttpServletRequest request, HttpServletResponse response)
        throws SQLException, IOException, ServletException {
    int groupsId = Integer.parseInt(request.getParameter("groupsId"));
    List<Student> student = findByGroupId(groupsId);
    request.setAttribute("student", student);
    RequestDispatcher dispatcher = request.getRequestDispatcher("studetn_form.jsp");
    dispatcher.forward(request, response);
}
//фильтр студентов по выбранному курсу
private void courseStudent(HttpServletRequest request, HttpServletResponse response)
        throws SQLException, IOException, ServletException {
    int courseId = Integer.parseInt(request.getParameter("courseId"));
    List<Student> student = findCoursebyStudentId(courseId);
    request.setAttribute("student", student);
    RequestDispatcher dispatcher = request.getRequestDispatcher("course_form.jsp");
    dispatcher.forward(request, response);
```

Рисунок 6

Для отображения таблицы студентов создана страница studetn_form.jsp, которая является начальной. Здесь реализован выпадающий список с выбором группы, а также кнопка, по нажатию которой в контроллер передается параметр номера группы, после чего талица отфильтровывается и перерисовывается. На странице присутствует верхняя панель для быстрого переключения между страницами, страницы которых выполнены аналогично.

Начальная страница с отображением списка студентов на рис. 7

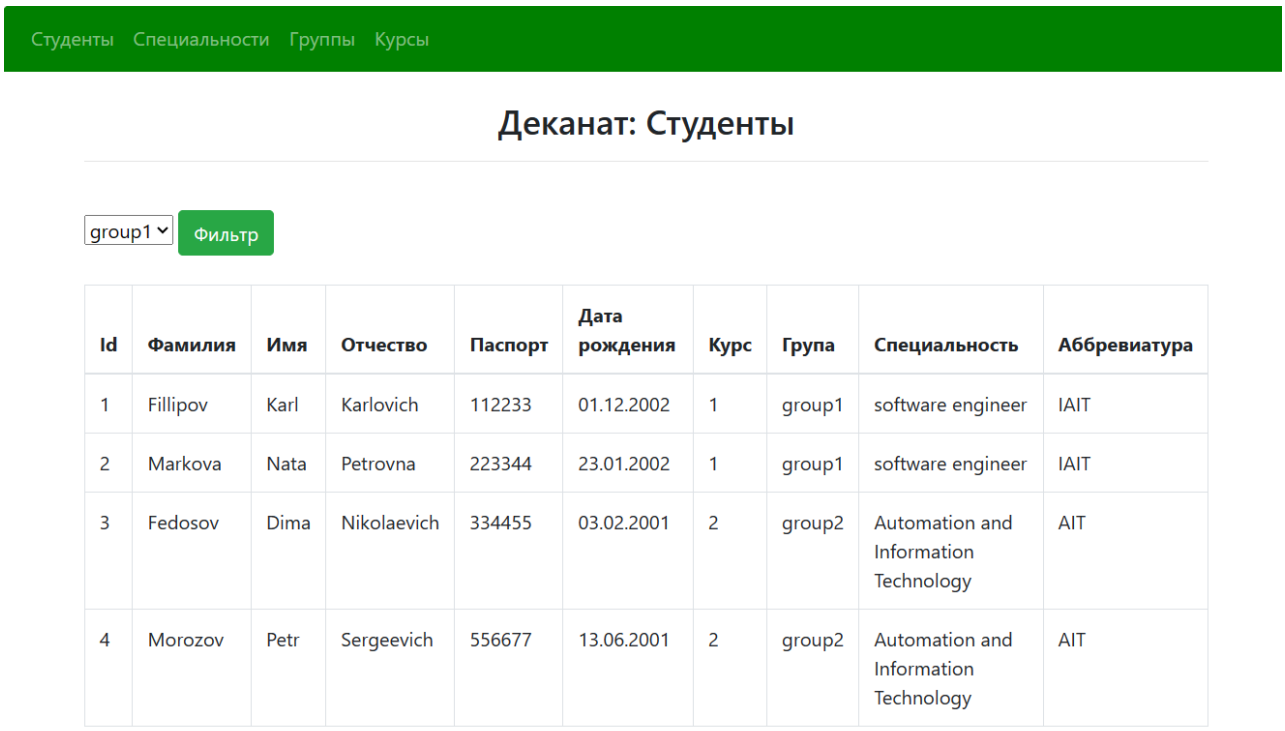


Рисунок 7

Страница «Студенты» при выборе из выпадающего списка группы. Отображение перечня студентов по выбранной группе после нажатия кнопки «Фильтр» на рис.8

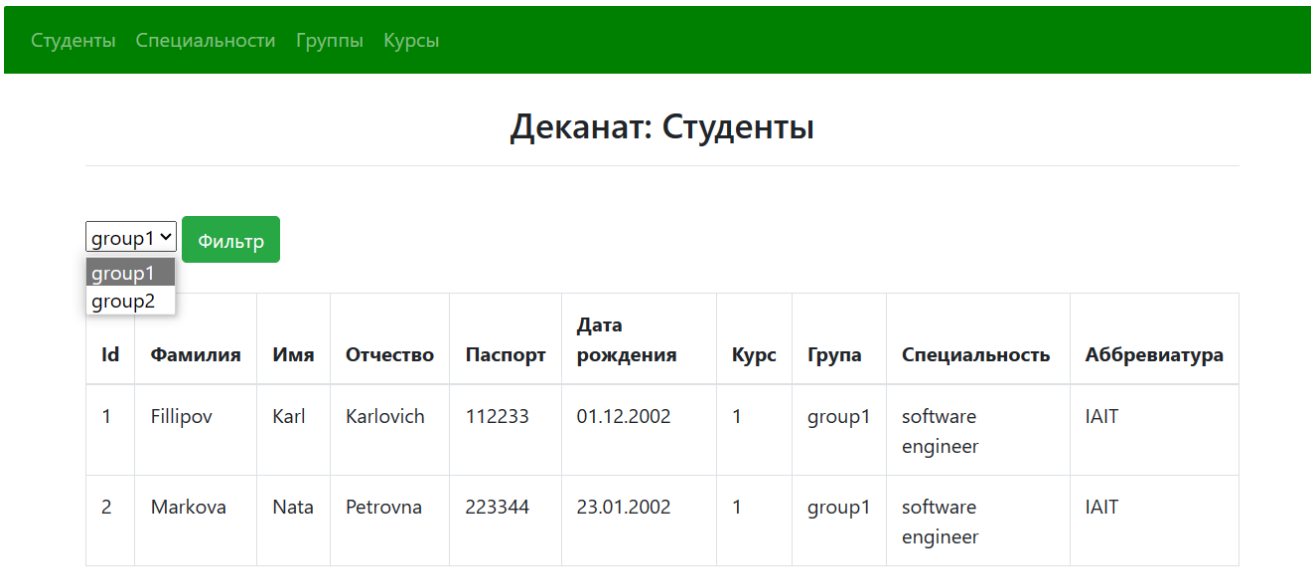


Рисунок 8

Страница «Специальности» с отображением списка специальностей на рис.9

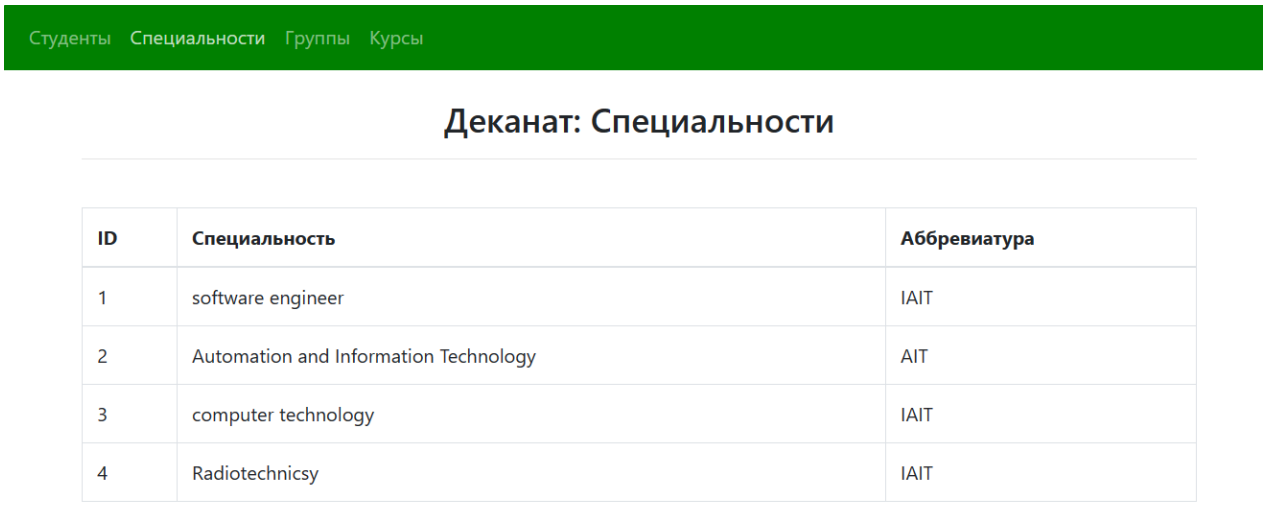


Рисунок 9

Страница «Группы» с отображением списка групп на рис.10

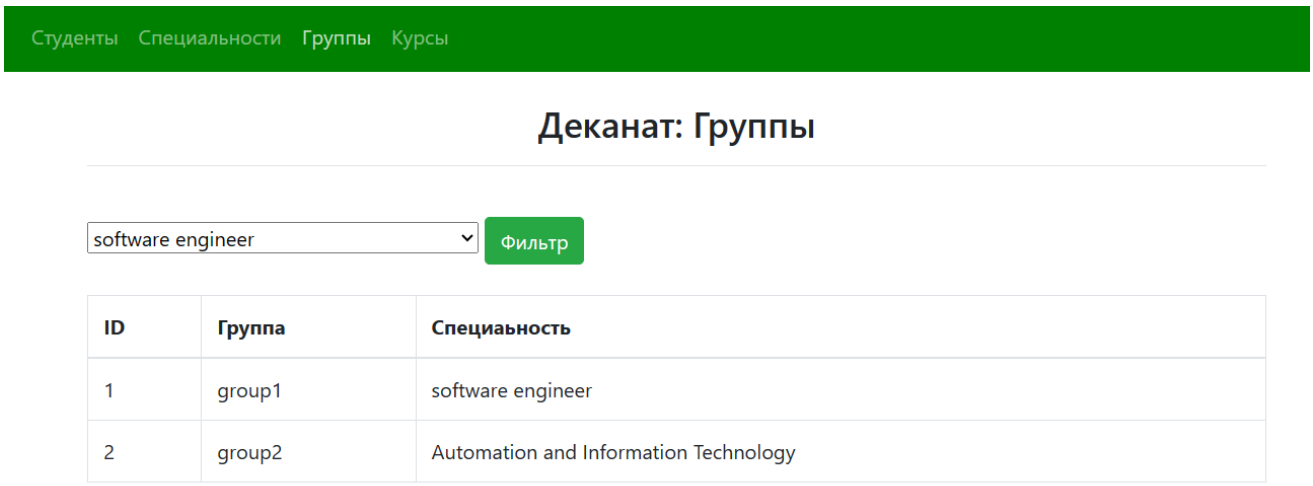


Рисунок 10

Страница «Группы» при выборе из выпадающего списка специальности. Отображение перечня групп по выбранной специальности после нажатия кнопки «Фильтр» на рис.11

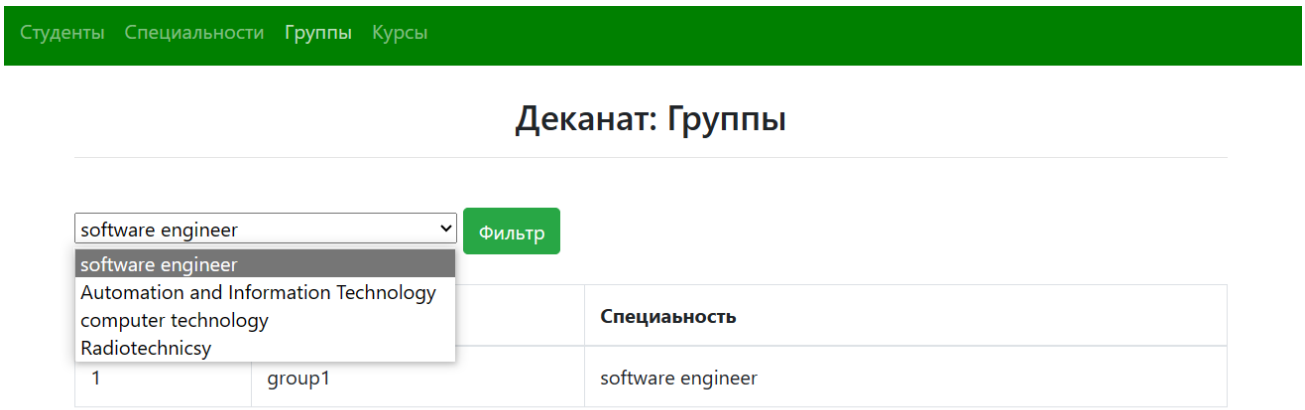


Рисунок 11

Страница «Курсы» с отображением списка курсов на рис.12

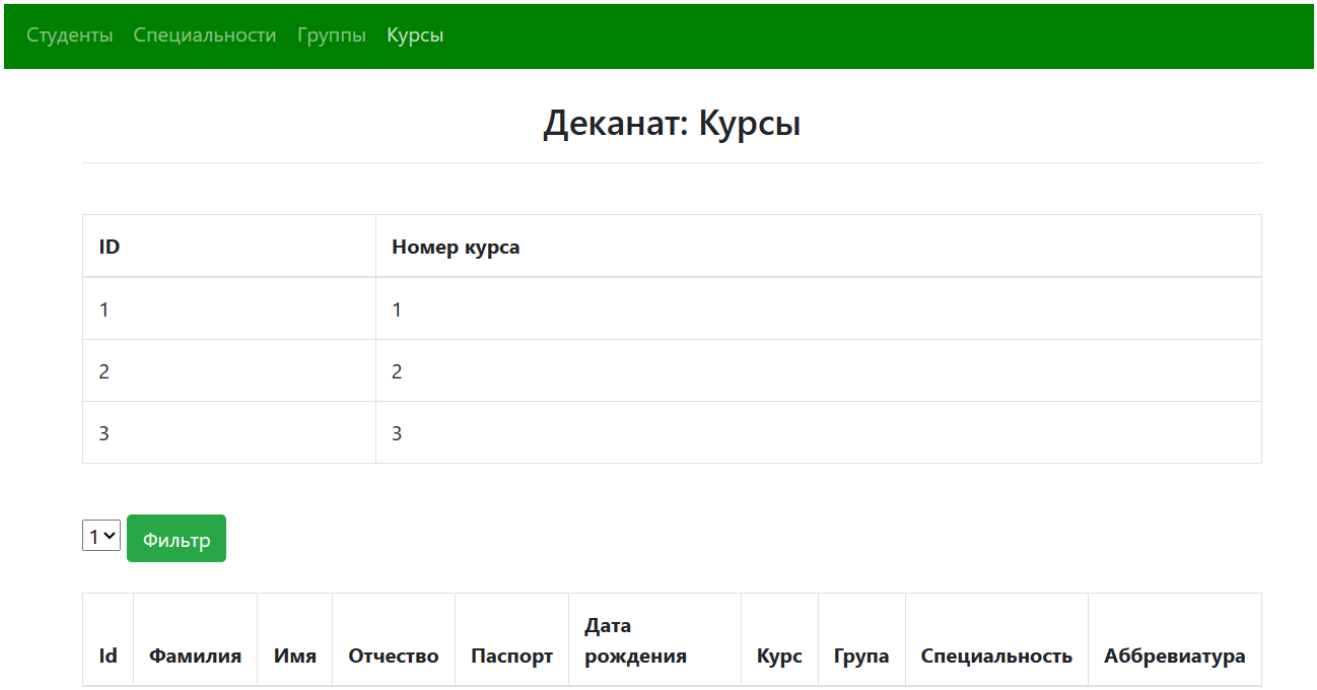


Рисунок 12

Страница «Курсы» при выборе из выпадающего списка курса. Отображение перечня студентов по выбранному курсу после нажатия кнопки «Фильтр» на рис.13

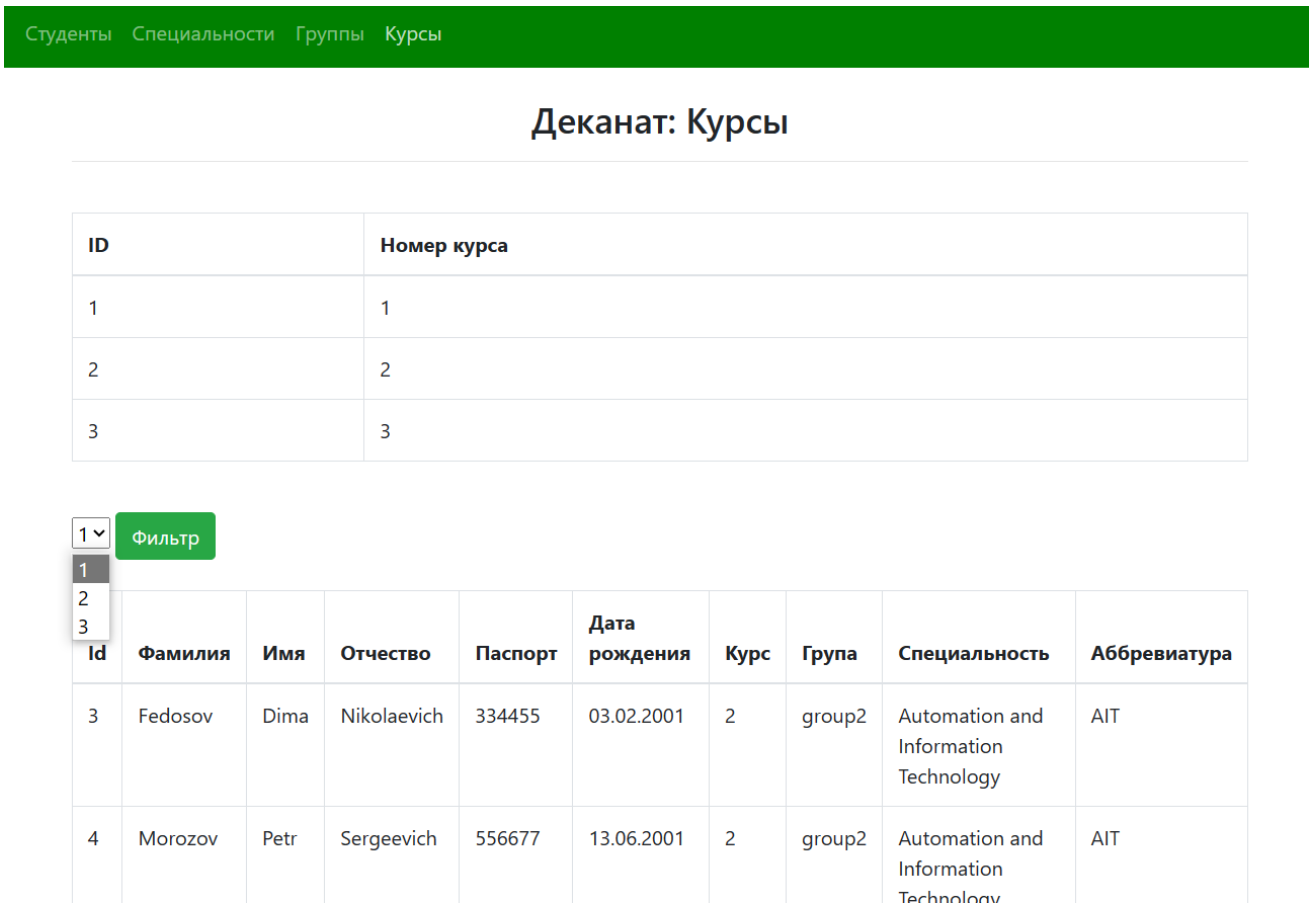


Рисунок 13

Ответы на вопросы:

1. Понятие и виды персистентности данных.

Персистентные структуры данных — это структуры данных, которые при внесении в них изменений сохраняют доступ ко всем своим предыдущим состояниям.

Есть несколько «уровней» персистентности:

- Частичная — к каждой версии можно делать запросы, но изменять можно только последнюю.
- Полная — можно делать запросы к любой версии и менять любую версию.
- Конфлюэнтная — помимо этого можно объединять две структуры данных в одну (например, сливать вместе кучи или деревья поиска).
- Функциональная — структуру можно реализовать на чистом функциональном языке: для любой переменной значение может быть присвоено только один раз и изменять значения переменных нельзя.

2. Проблема потери соответствия. Системы ORM.

Проблема потери соответствия состоит в том, что между объектно-ориентированным программированием и применением реляционных базы данных

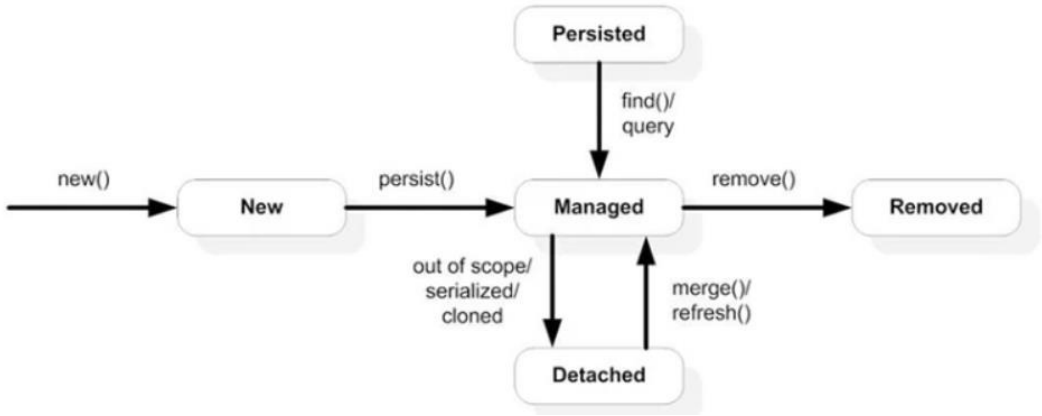
в качестве постоянного места хранения существует конфликт в модели данных. В результате данная проблема заставляет программиста писать программное обеспечение, которое должно уметь, как обрабатывать данные в объектно-ориентированном виде, так и уметь сохранять эти данные в реляционной форме. Новые библиотеки и технологии программирования для объектно-реляционного отображения во многом упростили проблему потери соответствия, одним из которых является объектно-реляционное отображение (ORM, Object-Rational Mapping).

ORM (объектно-реляционное отображение) – это технология программирования, которая связывает базы данных с концепциями объектно-ориентированных языков программирования, при этом создавая виртуальную объектную базу данных. ORM используется для упрощения процесса сохранения объектов в реляционную базу данных и их извлечения, при этом ORM сама заботится о преобразовании данных между двумя несовместимыми состояниями

3. Основные понятия JPA. Взаимодействие компонентов в JPA.

JPA (Java Persistence API) - набор интерфейсов и классов для управления отношениями объектно-ориентированных предметов с реляционной базой данных.

- **EntityManagerFactory.** Совокупность приемов и методов целенаправленного воздействия на сущность начинается с создания EntityManagerFactory, которая отвечает за поддержку соединений, отображение объектов в базу данных, память с большой скоростью доступа.
- **EntityManager** является интерфейсом, описывающим API для базовых операций над сущностью Enitity и получение данных.
- **Entity** – сущности, являющиеся постоянными объектами, которые хранятся в виде записей в базе данных. Для каждой сохраняемой сущности JPA создается новая таблица в соответствующей базе данных. Для определения Entity необходимо определить его с использованием аннотации @Entity, которая указывается на уровне класса.



4. JPA сущности: понятие, объявление, аннотации.

Для определения Entity необходимо определить его с использованием аннотации @Entity, которая указывается на уровне класса. Имя сущности должно соответствовать имени класса, но его можно изменить, используя элемент name. Каждой сущности JPA необходимо иметь наличие первичного ключа, который будет однозначно идентифицировать ее.

В большинстве случаев имя таблицы в базе данных и имя сущности не будут совпадать. В этих случаях указывают имя таблицы с помощью аннотации @Table. При отсутствии аннотации @Table имя сущности по умолчанию становится именем таблицы. Для упоминания сведений о столбце в таблице можно использовать аннотацию @Column. Аннотация @Column содержит множество элементов, таких как имя, длина, возможность обнуления и уникальность. В некоторых случаях нам может потребоваться сохранить временные значения в необходимую таблицу, для этого используется аннотация @Temporal. Аннотация @Enumered используется для сохранения типов перечислений по имени или по порядку.

5. Вложенные сущности.

Вложенные сущности — это способ хранения объектов одного типа внутри других объектов в базе данных с использованием JPA, что позволяет создавать более сложные структуры данных, где одна сущность может содержать в себе другие сущности. Это обеспечивает удобство и гибкость при работе с организацией данных в приложении.

6. Первичные ключи JPA сущностей.

Каждая сущность, сопоставленная с реляционной базой данных, должна иметь сопоставление с первичным ключом в таблице.

Спецификация EJB 3.0 поддерживает три различных способа указания идентификатора объекта:

- Аннотация идентификатора: Аннотация @Id предлагает самый простой механизм определения сопоставления с первичным ключом.
- Аннотация IDclass: Аннотация @IDClass используется для моделирования составного ключа.
- Аннотация встроенного идентификатора: Используйте аннотацию @EmbeddedId с аннотацией @Embeddable, чтобы переместить определение составного ключа внутри сущности.
-

7. Использование коллекций и валидации в JPA сущностях.

JPA валидация обладает ограниченной функциональностью, но является хорошим выбором для простейших ограничений в классах сущности, если такие ограничения могут быть отображены на DDL. Коллекции могут быть использованы в JPA сущностях для хранения связанных объектов или списков значений. Встроенная валидация в JPA позволяет определять ограничения и правила проверки данных, такие как проверка на уникальность, допустимые значения и другие правила целостности данных.

8. Связи между JPA сущностями.

- @ManyToOne – отношение между объектами «многие к одному», когда на один объект или столбец ссылается другой объект или столбец с уникальными значениями. В реляционных базах данных эти отношения реализованы при помощи внешнего ключа или первичного ключа между сущностями.
- @OneToMany – отношение между объектами «один ко многим», когда каждая строка одного объекта ссылается на множество дочерних записей в другом объекте. Дочерние записи могут иметь только одну родительскую таблицу.
- @OneToOne – отношения между объектами «один к одному», когда один элемент принадлежит только одному другому элементу – строка одного объекта относится только к одной строке другого объекта.
- @ManyToMany – отношение между объектами «многие ко многим», когда одна или несколько строк одного объекта могут принадлежать к нескольким строкам другого объекта.

9. Наследование в JPA. Отображение наследования в БД.

InheritanceType.SINGLE_TABLE — вся иерархия наследования отображается в одну таблицу. Объект хранится ровно в одной строке в этой таблице, а значение дискриминатора, хранящееся в столбце дискриминатора, указывает тип объекта. Любые поля, не используемые в суперклассе или другой

ветви иерархии, устанавливаются в NULL . Это стратегия отображения наследования по умолчанию, используемая JPA.

InheritanceType.TABLE_PER_CLASS — Каждый конкретный класс сущностей в иерархии сопоставляется с отдельной таблицей. Объект хранится ровно в одной строке в конкретной таблице для его типа. Эта конкретная таблица содержит столбец для всех полей конкретного класса, включая любые унаследованные поля. Это означает, что у братьев и сестер в иерархии наследования будет каждая своя копия полей, которые они наследуют от своего суперкласса. UNION из отдельных таблиц выполняется при запросе на суперкласса.

InheritanceType.JOINED — Каждый класс в иерархии представлен в виде отдельной таблицы, поэтому дублирование полей не происходит. Объект хранится в нескольких таблицах; по одной строке в каждой из таблиц, составляющих иерархию наследования классов. Отношение is-a между подклассом и его суперклассом представляется как отношение внешнего ключа от «subtable» к «supertable», и сопоставленные таблицы объединяются для загрузки всех полей объекта.

10. Управление сущностями через менеджеры сущностей.

В JPA интерфейс EntityManager используется, чтобы позволить приложениям управлять и искать объекты в реляционной базе данных.

EntityManager — это API, который управляет жизненным циклом экземпляров сущностей. Объект EntityManager управляет набором сущностей, которые определяются единицей персистентности. Каждый экземпляр EntityManager связан с контекстом персистентности . Контекст персистентности определяет область, в которой конкретные экземпляры сущностей создаются, сохраняются и удаляются с помощью API, доступных с помощью EntityManager. В некотором смысле контекст персистентности концептуально аналогичен контексту транзакции.

Менеджер сущностей отслеживает все объекты сущностей в контексте персистентности на предмет внесенных изменений и обновлений и сбрасывает эти изменения в базу данных. После закрытия контекста персистентности все экземпляры объекта управляемой сущности отделяются от контекста персистентности и связанного с ним менеджера сущностей и больше не управляются. Как только объект отсоединен от контекста персистентности, он больше не может управляться менеджером объектов, и любые изменения состояния этого экземпляра объекта не будут синхронизированы с базой данных.

Экземпляр объекта сущности либо управляется (прикрепляется) менеджером сущности, либо неуправляемым (отсоединяется). Когда сущность

подключена к менеджеру сущности, менеджер отслеживает любые изменения сущности и синхронизирует их с базой данных всякий раз, когда менеджер сущности решает сбросить ее состояние. Когда сущность отсоединена и, следовательно, больше не связана с контекстом персистентности, она становится неуправляемой, и изменения ее состояния не отслеживаются менеджером сущности.

Экземпляры сущностей становятся неуправляемыми и отсоединенными, когда заканчивается область транзакции или расширенный контекст персистентности. Важным следствием этого факта является то, что отдельные объекты можно сериализовать и отправить по сети удаленному клиенту. Клиент может удаленно вносить изменения в эти экземпляры сериализованных объектов и отправлять их обратно на сервер для обратного объединения и синхронизации с базой данных.

11. Язык JPQL.

JPQL (Java Persistence query language) это язык запросов, практически такой же как SQL, однако вместо имен и колонок таблиц базы данных, он использует имена классов Entity и их атрибуты. В качестве параметров запросов так же используются типы данных атрибутов Entity, а не полей баз данных. В отличии от SQL в JPQL есть автоматический полиморфизм (см. следующий вопрос). Также в JPQL используется функции которых нет в SQL: такие как KEY (ключ Map'ы), VALUE (значение Map'ы), TREAT (для приведение суперкласса к его объекту-наследнику, downcasting), ENTRY и т.п**.**

12. Criteria API.

Criteria API это тоже язык запросов, аналогичным JPQL (Java Persistence query language), однако запросы основаны на методах и объектах, то есть запросы выглядят так:

```
CriteriaBuilder cb = ...
CriteriaQuery<Customer> q = cb.createQuery(Customer.class);
Root<Customer> customer = q.from(Customer.class);
q.select(customer);
```

13. Уровни блокировок при использовании JPA

У JPA есть шесть видов блокировок, перечислим их в порядке увеличения надежности (от самого ненадежного и быстрого, до самого надежного и медленного):

1) NONE — без блокировки

2) OPTIMISTIC (или синоним READ, оставшийся от JPA 1) — оптимистическая блокировка,

3) OPTIMISTIC_FORCE_INCREMENT (или синоним WRITE, оставшийся от JPA 1) — оптимистическая блокировка с принудительным увеличением поля версионности,

4) PESSIMISTIC_READ — пессимистичная блокировка на чтение,

5) PESSIMISTIC_WRITE — пессимистичная блокировка на запись (и чтение),

6) PESSIMISTIC_FORCE_INCREMENT — пессимистичная блокировка на запись (и чтение) с принудительным увеличением поля версионности,