4.2 Взаимодействие с базой данных

В качестве системы управления базами данных (СУБД) была выбрана система MySQL версии 4.0.

Данная СУБД предоставляет достаточно широкий перечень возможностей для взаимодействия с базой данных: формирование запросов, осуществление поиска необходимых данных, синхронизация информации, а также выполнение аналитической обработки данных и получение разнообразных отчетов. MySql — это реляционная система управления базами данных. То есть данные в ее базах хранятся в виде логически связанных между собой таблиц, доступ к которым осуществляется с помощью языка запросов SQL. MySql —это достаточно быстрая, надежная и, главное, простая в использовании СУБД.

Работать с MySql можно не только в текстовом режиме, но и в графическом. Существует очень популярный визуальный интерфейс для работы с этой СУБД — MySql Workbench. Окно данной программы представлено на рисунке 3.3. Этот интерфейс позволяет значительно упростить работу с базами данных в MySql.

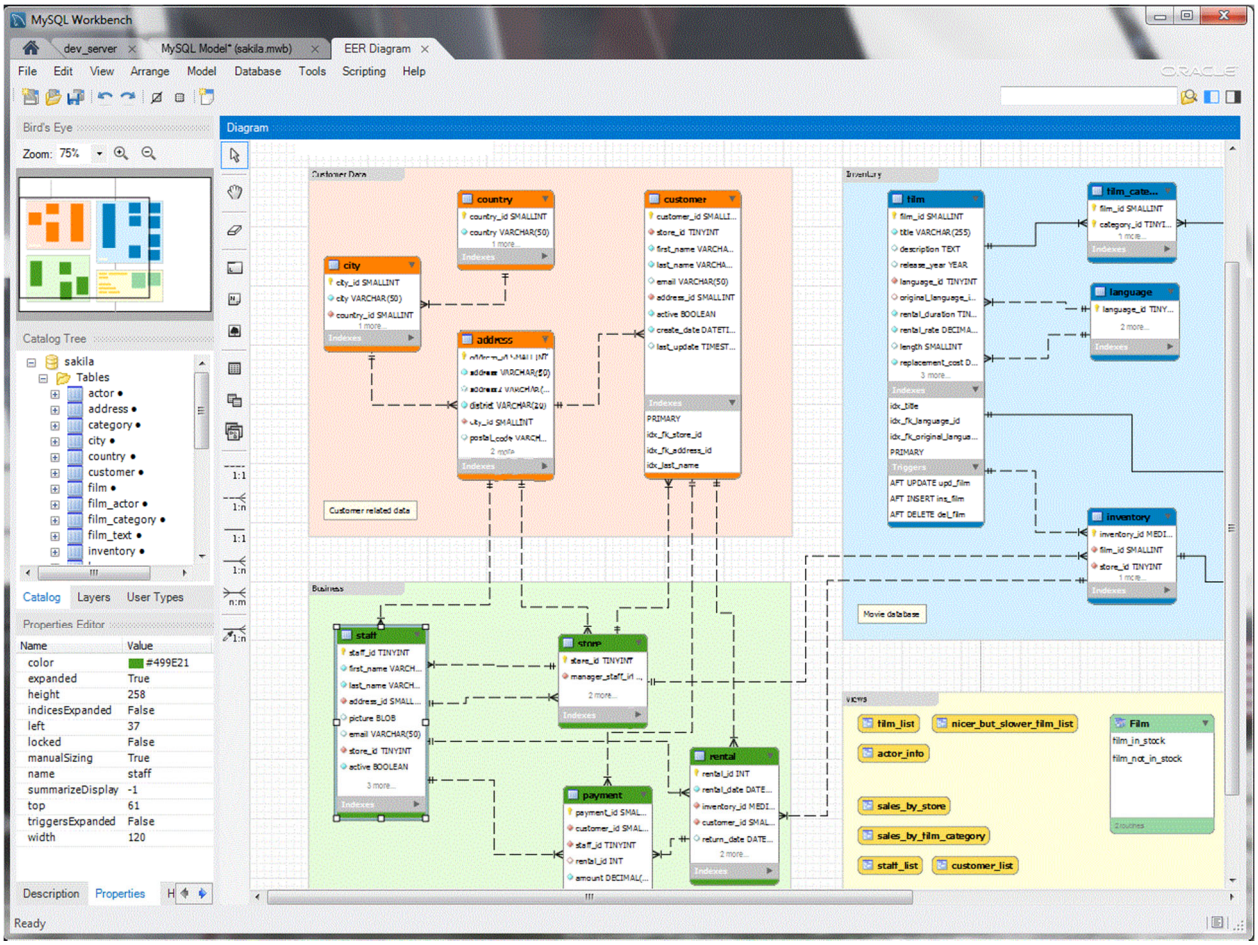

Рисунок 4.1 – Окно приложения MySql Workbench

```
@Controller
@RequestMapping("user")
public class UserController2 {
    @Autowired
    private UserService userService;
    @RequestMapping(value = "/me", method = RequestMethod.GET)
    public String getMyUserPage(Model model, @AuthenticationPrincipal UserDetails currentUser) {
        User = userService.getByNickname(currentUser.getUsername());
        model.addAttribute("userInfo", user);
        return "userPage";
    }
    @RequestMapping(value = "/{id}", method = RequestMethod.GET)
    public String getUserPage(Model model,
                              @PathVariable("id") Long id) {
        User = userService.getById(id);
        model.addAttribute("userInfo", user);
        return "userPage";
    }
    @RequestMapping(value = "/add", method = RequestMethod.POST)
    public String addUser(@ModelAttribute("userForm") @Valid User userForm,
                          BindingResult, Model model) {
        if (bindingResult.hasErrors()) {
            return "redirect:/registration";
        }
        if (!userForm.getPassword().equals(userForm.getPasswordConfirm())){
            model.addAttribute("errorMessage", "Пароли не совпадают");
            return "registrationPage";
        }
        if (!userService.add(userForm)){
            model.addAttribute("errorMessage", "Пользователь с таким именем уже существует");
            return "registrationPage";
        }
        return "redirect:/login";
    }
    @RequestMapping(value = "/edit", method = RequestMethod.PUT)
    public String editUser(@RequestBody User userForm) {
        if (!userForm.getPassword().equals(userForm.getPasswordConfirm())){
            return "redirect:/user/me/edit?error=1";
        }
        if (!userService.edit(userForm)){
            return "redirect:/user/me/edit?error=2";
        }
        return "redirect:/user/me";
    }
    @DeleteMapping(value = "/{id}")
    public String deleteUser(Model model, @AuthenticationPrincipal UserDetails currentUser,
                             @PathVariable("id") Long id) {
        User = userService.getById(id);
        userService.delete(user);
        model.addAttribute("userInfo", user);
        return "userPage";
    }
}
```

Рисунок 4.2 – Пример контроллера

Хранить логику обработки запросов в контроллере было бы громоздко и неудобно. С целью избегания этого приложение имеет 3 слоя:

– веб;

– бизнес логика;

– доступ к данным.

Контроллеры хранятся в веб слое и обращаются к сервисам, которые хранятся в слое бизнес логики. Сервисы содержат в себе всю логику изменения данных, пришедших в запросе. Для выполнения своей задачи сервисы могут обращаться к другим сервисам. На рисунке 4.3 приведён пример сервиса, отвечающего за работу с комментариями.

```
@Service
@Transactional
public class CommentServiceImpl implements CommentService {

    @Autowired
    private CommentRepository commentRepository;

    @Override
    public Iterable<Comment> getAllByAudiobook(Audiobook audiobook) {
        return commentRepository.findAllByAudiobook(audiobook);
    }

    @Override
    public void add(Comment comment) {
        commentRepository.save(comment);
    }

    @Override
    public void edit(Comment comment) { commentRepository.save(comment); }

    @Override
    public void delete(Comment comment) { commentRepository.delete(comment); }

}
```

Рисунок 4.3 – Пример сервиса

Для повышения независимости слоёв друг от друга используется принцип инверсии зависимостей. Согласно нему компоненты не знают друг о друге ничего кроме интерфейсов. Это позволяет в любой момент подменять компоненты другими с таким же интерфейсом. Он является частью большего принципа написания кода, распространяющегося не только на веб-приложения – SOLID. Этот принцип является очень важным при написании качественного и поддерживаемого кода. Каждая буква в названии отвечает за отдельный принцип, несущий свои правила написания кода:

– принцип одной ответственности (single responsibility);
– принцип открытости/закрытости (open-closed);
– принцип подстановки Барбары Лисков (Liskov substitution);
– принцип разделения интерфейса (interface segregation);
– принцип инверсии зависимостей (dependency inversion).

Spring Framework позволяет настраивать зависимости автоматически, с помощью аннотации @Autowired. На рисунке 4.4 приведён пример настройки зависимостей для контроллера аудиокниг.

```
private final AudiobookService audiobookService;
private final CommentService commentService;
private final AudiobookFileService audiobookFileService;
private final UserService userService;
private final CreatorService creatorService;

@Autowired
public AudiobookController(AudiobookService audiobookService, CommentService commentService,
AudiobookFileService audiobookFileService, UserService userService, CreatorService creatorService) {
    this.audiobookService = audiobookService;
    this.commentService = commentService;
    this.audiobookFileService = audiobookFileService;
    this.userService = userService;
    this.creatorService = creatorService;
}
```

Рисунок 4.4 – Пример настойки зависимостей

После применения всех преобразований к данным сервисы обращаются к репозиториям для внесения изменений в базу данных. В приложении реализован репозиторий для выполнения базовых операций с любой моделью. При необходимости выполнения нестандартных действий создаётся дополнительный репозиторий, наследуемый от базового, и в нём реализуются все необходимые операции. На рисунке 4.5 приведён пример базового репозитория.

```
@Repository
public interface CrudRepository<T, ID> extends Repository<T, ID> {
    <S extends T> S save(S var1);

    <S extends T> Iterable<S> saveAll(Iterable<S> var1);

    Optional<T> findById(ID var1);

    boolean existsById(ID var1);

    Iterable<T> findAll();

    Iterable<T> findAllById(Iterable<ID> var1);

    long count();

    void deleteById(ID var1);

    void delete(T var1);

    void deleteAll(Iterable<? extends T> var1);

    void deleteAll();
}
```

Рисунок 4.5 – Пример базового репозитория

Репозитории обращаются к базе данных за данными, которые в приложении хранятся в моделях – второй части архитектуры MVC. Модели содержат только данные. На рисунке 4.6 приведён пример модели.

```
@Entity
@Table(name = "user")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id", nullable = false)
    private Long id;

    @NotNull
    @Column(name = "nickname", nullable = false)
    private String nickname;

    @Column(name = "email",  nullable = true)
    private String email;

    @NotNull
    @Column(name = "password", length = 70, nullable = false)
    private String password;

    @ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn(name = "role_id", nullable = false)
    private Role role;

    public User() { }

    public Long getId() {  return id; }

    public String getNickname() { return nickname; }

    public void setNickname(String nickname) {  this.nickname = nickname; }

    public String getEmail() { return email;}

    public void setEmail(String email) { this.email = email; }

    public String getPassword() { return password; }

    public void setPassword(String password) { this.password = password; }

    public Role getRole() {return role;}

    public void setRole(Role role) { this.role = role;}|

}
```

Рисунок 4.6 – Пример модели

Для хранения данных в приложении используется база данных. Для удобной работы с ней используется Spring Data JPA. Он позволяет работать с базой данных как будто сущности в ней – это объекты, к которым можно обратиться напрямую из кода приложения.

Последняя часть архитектуры MVC – представление – реализована клиентской частью приложения в виде шаблонов html страниц, с использованием Thymeleaf и JavaScript.

Thymeleaf – это современный серверный механизм Java-шаблонов для веб- и автономных сред, способный обрабатывать HTML, XML, JavaScript, CSS и даже простой текст. Основной целью Thymeleaf является создание элегантного и удобного способа шаблонизации. Thymeleaf также был разработан с самого начала с учетом стандартов Web, особенно HTML5, что позволяет создавать полностью соответствующие стандарту шаблоны страниц. На рисунке 4.7 приведён пример шаблона HTML страницы.

```
<!DOCTYPE html>
<html lang="ru"
    xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org"
    xmlns:sec="http://www.thymeleaf.org/thymeleaf-extras-springsecurity4">
<head>
  <title>Spring Security Example </title>
  <div th:replace="fragments/header :: header-css"/>
</head>
<body>
  <div th:replace="fragments/header :: header"/>
  <div id="content" class="container mt-4">
    <div class="row justify-content-center" >
    <div class="card col-lg-5  col-md-7 col-sm-10"  th:object="${userInfo}">
      <div class="card-body" >
        <div class="row justify-content-center" >
          <div class="col-9 mb-1" th:if="${userInfo.nickname.equals(#authorization.authentication.name)}">
            <h2 class="card-title" style="text-align: center">Мой профиль</h2>
          </div>
          <div class="col-9 mb-1"
th:unless="${userInfo.nickname.equals(#authorization.authentication.name)}">
            <h2 class="card-title" style="text-align: center">Профиль пользователя <p
th:text="${userInfo.nickname}"></p></h2>
          </div>
        </div>
        <input type="hidden" required name="id" th:field="*{id}"><br>
        <div class="row justify-content-center">
          <div class="mb-2 col-9">
            <span class="" name="nickname" id="nickname">
              <p th:text="'Никнейм: '+${userInfo.nickname}"></p></span>
          </div>
        </div>
        <div class="row justify-content-center">
          <div class="mb-2 col-9" id="email" name="email">
            <div class="col" th:unless="${#strings.isEmpty(userInfo.email)}" >
              <p th:text="'Email: '+ ${userInfo.email}"></p>
            </div>
            <p class="col" th:if="${#strings.isEmpty(userInfo.email)}">Email: не указан</p>
          </div>
        </div>
        <div class="row justify-content-center">
          <div  class="mb-2 col-9" th:switch="${userInfo.role.id}">
            <p th:case="2">Статус: администратор сайта</p>
            <p th:case="3">Статус: Администратор сайта</p>
            <p th:case="1">Статус: пользователь сайта</p>
          </div>
        </div>
        <div class="row justify-content-center mt-2"
th:if="${userInfo.nickname.equals(#authorization.authentication.name)}" >
          <a id="edit-profile-btn" class="col-lg-7 col-md-8 col-sm-9 btn  btn-primary btn-block"
             th:href="@{/user/me/edit}">
            Редактировать профиль
          </a>
        </div>
      </div>
    </div>
  </div>
  </div>
  <div th:replace="fragments/footer :: footer"/>
</body>
</html>
```

Рисунок 4.7 – Пример шаблона Html страницы