Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра мультимедійних систем факультету інформатики

Web-застосунок для розвитку музичних начичок

Текстова частина до курсової роботи
за спеціальністю „Інженерія програмного забезпечення ” 121

Керівник курсової роботи
старший викладач
Борозенний С. О.

_____
(Підпис)
“___” _____ 2022 року

Виконав студент ІПЗ-БП4
Романенко М. О.
“___” _____ 2022 року

Київ 2022

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мережних технологій факультету інформатики

ЗАТВЕРДЖУЮ

Зав. кафедри мультимедійних систем,

доцент, к. ф-м. н.

_____ О. П. Жежерун

*(підпис)*_ _____

_“____” _____ 2022 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на курсову роботу

Студенту Романенку Михайлу Олександровичу факультету інформатики 4 курсу

ТЕМА: Web-застонук для розвитку музичних навичок

Зміст ТЧ до курсової роботи:

Календарний план

Перелік умовних позначень

Вступ

Аналіз предметної області та постановка завдання

Огляд та обрання технологій для розробки

Опис реалізації програми

Висновки

Список використаної літератури

Дата видачі “____” _____ 2022 р.

Керівник _____

(підпис)

Завдання отримав _____

(підпис)

Календарний план виконання роботи

Тема: Web-застонук для розвитку музичних навичок

№	Назва етапу	Термін виконання	Примітка
1.	Отримання теми курсової	20.11.2021	
2.	Ознайомлення з предметною областю	Листопад 2021	
3.	Пошук та ознайомлення з корисною літературою	Грудень 2021 - січень 2022	
4.	Планування структури та архітектури практичної частини роботи	Січень 2022 – лютий 2022	
5.	Програмування застосунку	Лютий-квітень 2022	
6.	Написання текстової частини роботи	Квітень-травень 2022	
7.	Здача курсової роботи на перевірку	20.05.2022	

Студент Романенко М. О.

Керівник Борозенний С. О. “____” _____

ЗМІСТ

Перелік термінів та умовних позначень

Напівтон (semitone, half-step)	Атомарна одиниця відстані у висоті між двома звуками.
Ступінь, (т. щабель) (degree)	Термін, що позначає деякий звук в акорді чи ладі. 1 ступінь означає тоніку. Номери ступенів (1, 2, 3, 4, 5, 6, 7) відповідають висотам звуків у мажорному ладі. Відносно них позначаються інші ступені. Наприклад, якщо 3 означає звук, що знаходиться на відстані у 4 напівтони від тоніки(1), то b3 відповідає 3 напівтонам.
Октава (octave)	Відстань між двома однойменними звуками різної висоти (наприклад, від D до D). Складає 12 напівтонів. Номер октави позначає висоту ноти відносно інших нот.
Тоніка (root)	Головний звук у звуковисотній системі (акорд або лад).
Акорд (chord)	Сукупність трьох і більше нот різної висоти. Одна з цих нот є тонікою(root) акорду. Інші розглядаються відносно неї. Ноти акорду можуть звучати паралельно, а можуть послідовно (арпеджіо).
Лад (scale)	Сукупність звуків, впорядкованих відносно деякого основного звуку (тоніки).
b, бемоль (flat)	Означає, що звук, нотований цим символом, нижчий на півтон.
#, дієз (sharp)	Означає, що звук, нотований цим символом, вищий на півтон.
C, D, E, F, G, A, B	Латинські позначення нот, відповідають нотам до, ре, мі, фа, соль, ля, сі.
Аплікатура (fingering)	Спосіб взяти певний акорд на гітарі.

ВСТУП

Актуальність обраної теми

У світі сьогодення до вирішення будь-якої проблеми можна підійти з технічної точки зору. Вирішивши задачу один раз у загальному вигляді, можна застосувати алгоритм її розв’язку для отримання результату незалежно від конкретних вхідних даних. Одна з таких задач, з якою я зустрівся особисто - побудова музичних структур, таких як акорди, лади та аплікатури. Вивчаючи їх, я багато часу витрачав на їх побудову, мені не вистачало інструменту для їх швидкої візуалізації для подальшого аналізу. Я зрозумів, що їх узагальнену будову можна застосовувати для моментального відображення їх у будь-якому необхідному вигляді. Аналіз цих структур також можна автоматизувати, розробивши алгоритми побудови одного музичного об’єкту відносно іншого. Пошукавши схожі рішення в Інтернеті, я не був задоволений жодним із них: у них не було всіх акордів/ладів, що мене б цікавили, не було можливості пошуку взаємозв’язків між ними, дані у застосунку були статичними, не було налаштувань відображення, які були мені необхідні. Спираючись на це, я прийняв рішення самостійно розробити та в подальшому адмініструвати систему з усім необхідним функціоналом самостійно, а також надати її для вільного користування всім охочим.

Мета та завдання роботи

Метою даної роботи є розробка веб-застосунку для музик, що вирішував би задачі, з якими певною мірою стикається кожен музика, і повною мірою кожен гітарист, огляд та аналіз технологій сучасної веб-розробки на прикладі процесу проєктування та реалізації даного додатку.

Об’єкт дослідження

Об’єктом дослідження у даній роботі є шляхи полегшення процесу вивчення та розуміння музичної теорії за допомогою сучасних технологій.

Предмет дослідження

Предметом дослідження є застосування стеку технологій, що містить: бібліотеку для розробки користувацьких інтерфейсів React.js, фреймворк для

розробки веб-серверів Nest.js і документо-орієнтовану базу даних MongoDB, для побудови інтерактивного веб-застосунку, направленого на вирішення проблеми дослідження.

Використані технології

- Javascript – мова для розробки браузерної частини
- Typescript – надбудова над Javascript, мова для розробки сервера
- React.js – основа клієнтської частини
- Nest.js – основа серверної частина
- MongoDB – база даних
- Json Web Token – технологія аутентифікації
- Tone.js – JS-бібліотека для програвання та обробки звуку

РОЗДІЛ 1 Аналіз предметної області та постановка завдання

1.1 Постановка завдання

1.1.1 Функціонал, в якому існує потреба

При вивченні теорії музики, людина часто стикається з наступними задачами:

- Знайти новий акорд\лад для засвоєння
- Визначити його будову
- Знайти його ноти на музичному інструменті
- Будувати його відносно різних тонік
- Зіграти його ноти, ознайомитися з характером звучання
- Знайти взаємозв'язки між поточним акордом\ладом та іншими відомими акордами\ладами

І хоча самостійне виконання всіх цих кроків може слугувати вправою для тренування музичного мислення, цей підхід проблематично застосовувати на стадіях пошуку та ознайомлення з існуючими акордами\ладами та їх характером звучання, пошуком того звучання, яке припадає до душі. Особливу складність це викликає у початківців, які мало знайомі з нотами, ступенями та будовою базових та найбільш поширених музичних структур.

Розробивши рішення, яке дасть змогу користувачу у декілька кліків побудувати будь-яку музичну структуру і прослухати її звучання, можна суттєво полегшити процес ознайомлення людини з теорією музики, який є найскладнішим саме на перших етапах. Це зробило б вивчення музичної теорії набагато більш цікавим і зрозумілим, особливо для музик молодшого віку і(або) початківців.

Окрім цього, такий застосунок також може допомогти розвитку мислення гітариста, даючи зрозумілий візуальний образ положення музичних структур, нот та ступенів на грифі гітари. Це дало б можливість як краще засвоїти матеріал, так і самостійно знайти певні закономірності, повторювані структури,

патерни, які згодом стали б музИці у нагоді, поглибили його розуміння мУзики в цілому.

1.1.2 Портрет користувача

Користувачі системи - музиканти-початківці, зокрема діти, учні початкових класів муз. шкіл.

1.1.3 Вимоги до застосунку

Враховуючи вищезазначене, можна сформулювати наступний перелік вимог до застосунку:

- Надання функціоналу для вирішення описаних вище завдань
- Простий, зрозумілий інтерфейс, не перевантажений засобами управління, але з багатим функціоналом
- Можливість прослуховування кожної зображеної на екрані ноти
- Можливість розширювати застосунок новими даними
- Архітектура, що легко піддається розширенню, додаванню нових функцій
- Можливість доступу з будь-якої платформи
- Актуальність даних застосунку (наприклад, наявність аплікатур, які зручно брати і які використовуються на практиці)

1.2 Огляд існуючих рішень

1.2.1 Android-додаток “AllChords”

Даний застосунок надає можливості для перегляду аплікатур акордів та їх прослуховування. Акорди груповані за нотою, від якої вони побудовані (тонікою). Одна з варіацій відображується за замовчуванням, інші можна переглянути за допомогою контекстного меню [1].

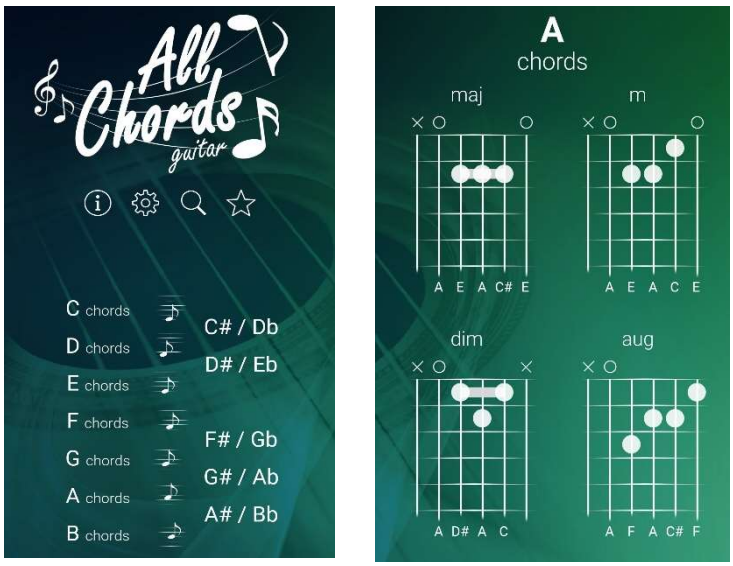


Рисунок 1.2.1.1. Інтерфейс застосунку AllChords

Переваги:

- Приємний і простий інтерфейс
- Приємні слуху звуки гітари
- Працює без підключення до Інтернету

Недоліки:

- Доступний лише на Android
- Відсутність ладів і всього пов’язаного з ними функціоналу
- Відсутність засобів відображення всіх нот акорду на грифі.
- Відсутність ступенів акорду, що суттєво зменшує цінність додатку для навчання саме загальної музичної теорії, а не простого вивчення того, як брати той чи інший акорд
- Акорди груповані не за загальною структурою (класом), а за кореневою нотою, що робить навігацію незручною
- Відсутні деякі корисні аплікатури, які вживаються на практиці.

1.2.2 Веб-додаток «Chords Database»

Цей веб-додаток містить у собі об’ємну базу акордів. При натисканні на певний акорд відображаються їх варіації, є можливість збереження генерованих зображень у форматах svg\png. Можна обирати між аплікатурами для укулеле та гітари, змінювати тоніку [2].

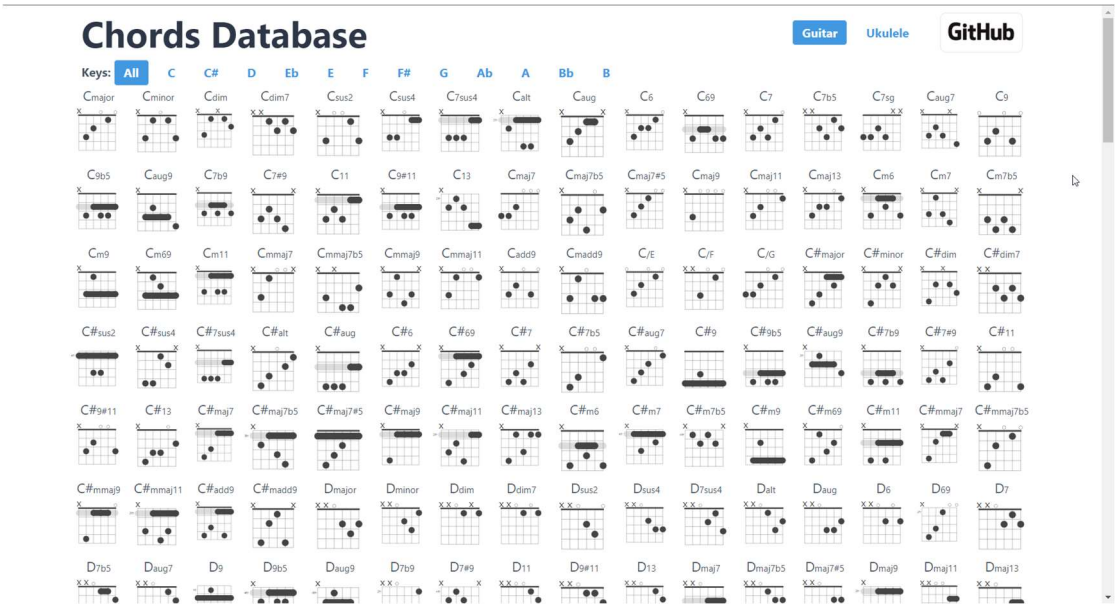


Рисунок 1.2.2.1 Інтерфейс застосунку Chords Database

Переваги:

- Об'ємна база з актуальною інформацією
- Можливість зберігати аплікатури у форматі svg\png
- Незалежність від платформи
- Підтримка гітари та укулеле

Недоліки:

- Відсутня робота з ладами та арпеджіо акордів
- Немає можливості прослуховування
- Немає позначень нот та ступенів

1.2.3 Android-додаток «Scales Learn»

Додаток зосереджений на роботі з ладами, містить можливості вільного перегляду ладів та міні-гра зі знаходженням звуків ладу на грифі гітари, має можливості перегляду ступенів та нот ладу [3].

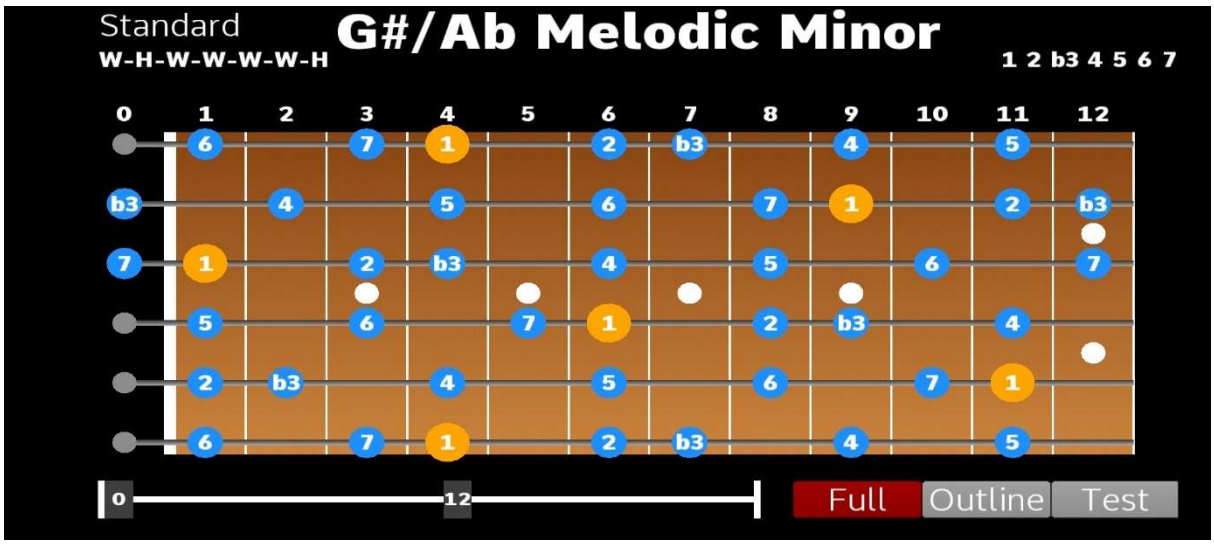


Рисунок 1.2.3.1 Інтерфейс застосунку Scales Learn

Переваги:

- Наявні різні режими відображення
- Не потрібний доступ до Інтернету
- Наявна міні-гра зі знаходження звуків ладу на гітарі

Недоліки:

- Відсутність акордів і пов’язаної з ними інформації
- Доступність лише на Android
- Звуки, що програються у програмі, неприємні і швидко втомлюють
- Немає інформації про взаємозв’язки між ладами
- Наявні лише близько десяти основних ладів

1.2.4 Веб-додаток «Chordbook»

Веб-додаток Chordbook містить інтерактивну гітару, на якій можна відображувати акорди і лади. Їх можна програвати назад і вперед, а також переглядати різні варіації їх розміщення на грифі гітари. Також можна шукати акорд у базі застосунку, розмістивши певним чином мітки на ладах гітари [4].

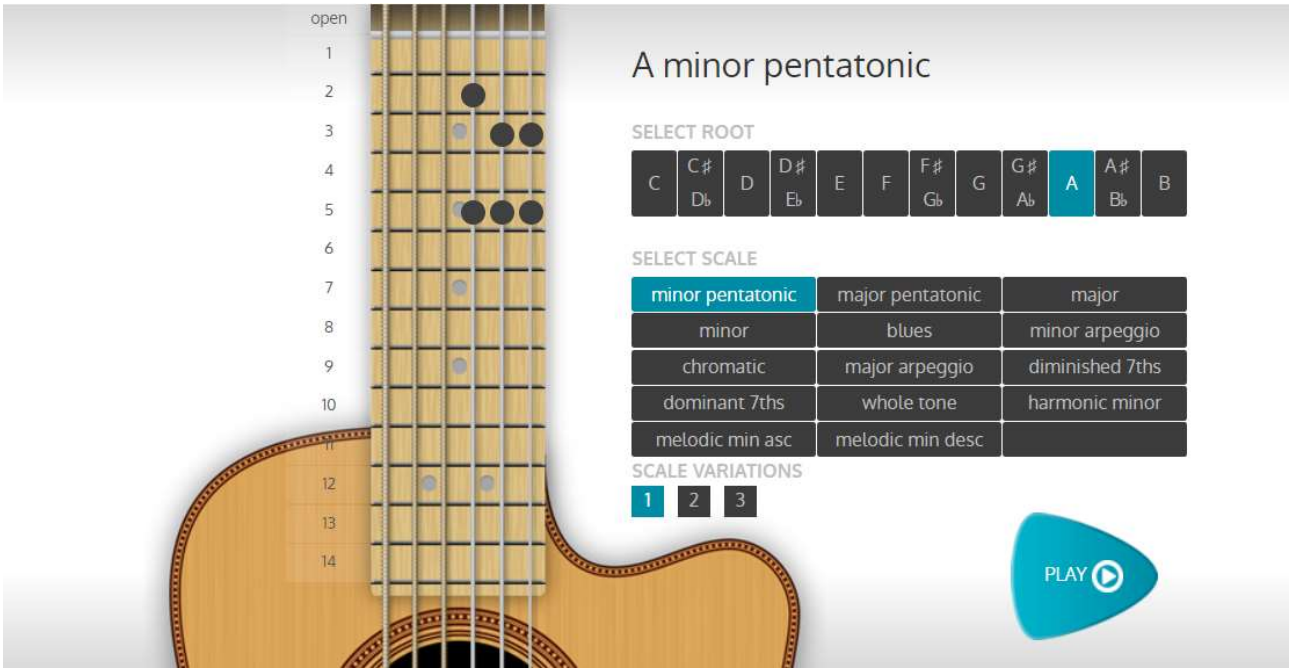


Рисунок 1.2.4.1 Інтерфейс застосунку Chordbook

Переваги:

- Кросплатформенність
- Є функція пошуку акорду
- Приємний звук

Недоліки:

- Обмежений набір ладів
- Немає відображення нот у ладів
- Немає відображення ступенів ладів та акордів
- Немає функції пошуку взаємозв’язків між акордами та ладами

1.3 Результат аналізу

У даній предметній області існують рішення, напрямлені на розв’язок поставленої задачі, але жодне з них повною мірою не відповідає як функціональним, так і нефункціональним вимогам, сформульованим у рамках даної роботи. Перевагою розробленого в рамках даної роботи рішення буде можливість шукати взаємозв’язки між акордами та ладами, а також зрозумілість інтерфейсу і можливість переглядати ноти і ступені будь-якої з музичних структур, розширювати базу даних застосунку новими акордами і

ладами. Окрім того, необхідно розробити застосунок з урахуванням його подальшого розвитку, додання нових функцій.

РОЗДІЛ 2 Огляд та обрання технологій для розробки

2.1 React.js

React.js – одне з найбільш популярних рішень для розробки клієнтської частини веб-застосувань. У центрі бібліотеки стоїть поняття компонентів – об’єктів, що енкапсулюють у собі набір елементів (HTML-тегів з надбудовою React), а також їх стан (state), тобто безпосередньо пов’язані з ними дані. Це дає можливість розбивати веб-сторінки на атомарні блоки, які можна параметризувати і перевикористовувати, збільшуючи таким чином ступінь зв’язності і зменшуючи зв’язаність системи.

Для розміщення компонентів на веб-сторінці та їх відображення використовується Virtual DOM – надбудова над стандартною Document Object Model браузера. За допомогою Virtual DOM бібліотека React може значно ефективніше маніпулювати веб-сторінкою, відслідковуючи зміни у стані компонентів та перемальовуючи у справжньому DOM лише ті елементи, які зазнали зміни [5].

Ще однією особливістю React є те, що він не задає жорсткої структури застосунку, зберігаючи свободу програмісту в організації логіки. React також не робить припущень з приводу технологій, поруч з якими він використовується.

React є прикладом застосування декларативного стилю програмування у розробці сучасних прикладних програм. Замість того, щоб вказувати послідовність інструкцій для приведення програми у певний стан, React пропонує програмісту напряму задати цей стан. Програма, написана за допомогою React, являє собою дерево компонентів, де кожен більш загальний складається з набору менших, а кореневим вузлом дерева буде компонент «увесь застосунок».

React є ідеальним вибором для розробки веб-застосунку для музик. Першою причиною цього є те, що візуальна структура застосунку дуже зручно описується в термінах компонентів React. Наприклад, компонент «гриф гітари» складається з 6 компонентів «струна», кожна з яких складається з N

компонентів «лад». Кожен з цих компонентів містить специфічну для свого рівня логіку – «гриф гітари» відповідає за розміщення струн, відображення загального фону, «струна» відповідає за розміщення ладів, графічне відображення струни, «лад» відповідає за відмалювання ладів, а також надсилання сигналів про те, що він був натиснутий. Другою причиною є спеціалізація React на динамічних змінах веб-сторінки. Високоефективна модель обробки змін веб-сторінки дозволить застосунку швидко реагувати, наприклад, на зміни параметрів відображення контенту, таких як коренева нота та застосовуваний символ альтерації (‘#’, ‘b’).

2.1.1 Управління станом React-застосунку

За умови відсутності додаткових засобів управління станом дані у React-застосунку передаються згори вниз, від батьківського компонента до дітей, через параметри (props). Таким чином, два компоненти можуть отримати доступ до одних і тих самих даних лише у тому випадку, якщо вище у дереві компонентів вони мають спільний компонент-батька. Таким чином, найпростішою технікою для надання двом компонентам доступу до загального стану є «підйом» цього стану у батьківський компонент. Такий підхід суттєво обмежений, адже зі збільшенням стану застосунку та ускладненням його структури необхідно буде підіймати спільний стан все вище і вище та виконувати дедалі більше передач його вниз через параметри. Через це виникає дублювання коду, одна зміна у коді програми тягне за собою велику кількість змін у різних компонентах, які не мають до неї ніякого відношення, ускладнюються читання та підтримка коду.

2.1.1.1 Redux

Найпопулярнішим рішенням даної проблеми у React-застосунках є використання бібліотеки Redux. Redux надає глобальне сховище даних та чітко регламентує доступ до стану застосунку і його зміни. Стан застосунку у будь-який момент часу являє собою незмінний об’єкт, що містить всі необхідні для роботи додатку дані. Він зберігається у об’єкті store, який надає інтерфейс для

перегляду стану та застосування до нього дій. При модифікації стану збережуваний об’єкт замінюється на новий екземпляр, який містить необхідні зміни. Зміни у стані застосунку описуються декларативним чином за допомогою об’єктів action. Action за загальноприйнятою конвенцією містить поле type, що позначає тип виконуваної дії, та дані, які потрібні для виконання цієї дії. Для створення об’єктів action застосовуються функції actionCreator, задача яких – створення об’єктів action з відповідним полем type та даними для виконання даної дії. Застосовуються зміни до стану застосунку за допомогою функцій reducer, які приймають на вхід поточний стан та дію і повертають новий екземпляр стану, утворений після виконання reducer-ом дії. Користувач бібліотеки ініціює зміни стану за допомогою виклику store.dispatch(), передаючи в параметри action. Store передає дію reducer-ам і зберігає новостворений екземпляр стану. Компоненти застосунку можуть підписуватись на певні частини стану і отримувати сповіщення про їх зміни [6].

Таким чином, Redux створює чітко відслідковуваний потік даних, гарантує актуальність стану у будь-який момент часу, дозволяє відокремити дані від компонентів, які покладаються на ці дані, забезпечує синхронізацію всіх компонентів програми з актуальним станом, полегшує процес зневадження(debug) застосунку. Це робить його майже обов’язковим вибором для великих проєктів. Проблемою Redux є необхідність у створенні великої кількості шаблонного коду, що може бути непотрібним ускладненням у проектах, менших за обсягом.

2.1.1.2 Context API

Ще один інструмент управління станом React-застосунку – Context API, який став частиною стабільної версії React починаючи з версії 16.3. Цей засіб дає змогу користуватися функціоналом, еквівалентним частині функціоналу Redux, ціною значно меншого обсягу написаного коду. Context API не здатен повністю замінити собою Redux, але є у нагоді у додатках, де існує потреба у глобальному стані, але немає необхідності у повному потенціалі Redux. Context

API дозволяє створювати об’єкти Context, та надавати до них доступ за допомогою компоненту Provider. Компоненти, що є дітьми Provider-а деякого контексту, можуть підписатися на цей контекст і використовувати значення дані, що зберігаються найближчим вгору по дереву компонентів Provider-ом.

Context API дає зручний спосіб отримання спільного стану для компонентів без написання великої кількості шаблонного коду, але, на відміну від Redux, не дає переваг у відслідковувані змін стану застосунку. Легкість у застосуванні робить його вдалим вибором для невеликих додатків. З урахуванням цього, саме Context API є найкращим вибором для реалізації рішення в рамках даної роботи.

2.2 React-Bootstrap

Ця бібліотека покладається на бібліотеку Bootstrap, що містить готові стилі та елементи функціоналу для побудови загально використовуваних елементів користувацьких інтерфейсів. Замість вказування стилів компонентів у вигляді css-класів, React-Bootstrap надає набір готових React-компонентів, що використовують відповідні стилі і які можна додатково кастомізувати через їх параметри [7]. Це суттєво впливає на читабельність коду, адже замість набору тегів з довгими значеннями властивості className розмітка з використанням React-Bootstrap складається з компонентів зі зрозумілими назвами, які відповідають назвам їх стилів.

2.3 Tone.js

Дана бібліотека використовується для відтворення звуків у браузері і спеціалізується на задаванні висоти цих звуків. За допомогою неї можна відтворити аудіофайл зі зміною у висоті звуку відповідно до заданої ноти. Також ця бібліотека містить інструментарій для обробки вихідного сигналу, наприклад, такими ефектами, як реверберація (ефект приміщення), еквалізація (зміна частотних характеристик сигналу) та ін. [8].

2.4 Nest.js

Nest.js – IOC-фреймворк для розробки веб-серверів на базі Node.js. В його основі лежить інверсія контролю (inversion of control, IOC) – принцип об'єктно-орієнтованого програмування, згідно з яким створення та управління програмними сутностями делегується фреймворку, на відміну від традиційного потоку управління, згідно з яким код програміста викликає код бібліотек, а не навпаки. Користувач IOC-фреймворку інтегрує написаний ним код у систему за допомогою передбачених точок розширення. Реалізацією принципу інверсії контролю в Nest.js є ін'єкція залежностей (dependency injection) – техніка передачі об'єктам тих об'єктів, від яких вони залежать, ззовні. IOC та DI використовуються для збільшення зв'язності (модульності) та зменшення зв'язаності програмних систем, що позитивно впливає на зрозумілість коду, легкість його підтримки та масштабовуваність системи.

Nest.js надає готову архітектуру «з коробки», центральною концепцією якої є модуль – логічно відокремлюваний структурний блок застосунку. Модулі складаються з provider-ів – об'єктів, що надають бізнес-логіку – та controller-ів – об'єктів, що обробляють зовнішні запити і викликають відповідну їм логіку з провайдерів. Модулі можуть експортувати свої складові та імпортувати інші модулі з метою використання їх експортованих складових. Внутрішні складові модуля зможуть використовувати експортовані складові імпортованого модуля за допомогою ін'єкції залежностей. Створенням модулів, контролерів, провайдерів, а також ін'єкцією екземплярів імпортованих об'єктів займається NestFactory під час запуску застосунку.

Ще однією важливою перевагою Nest є вбудована підтримка TypeScript, що дозволяє знаходити помилки у коді на ранніх етапах роботи над проектом, а також відкриває двері до експериментальних інструментів, таких як декоратори – аналог анотацій в Java – засіб декларування певних даних, пов'язаних з класом, методом або полем [9].

Nest js у своїй роботі покладається на рефлексію – процес, під час якого програмний код модифікує сам себе. Для цього фреймворк використовує пакет reflect-metadata та декоратори TypeScript. Під час збирання застосунку до програмних об'єктів додається увесь необхідний функціонал, потреба в наявності якого задекларована за допомогою декораторів. Таким чином, наприклад, можна вказати фреймворку використовувати функцію-обробник запиту для валідації певного ендпоінту. Даний підхід має назву «аспектно-орієнтоване програмування» [10]. Такий підхід дозволяє розбивати логіку не тільки за шарами (сервіс, контролер), а й повністю відокремлювати її за типом задачі, наприклад, відділити від коду контролера код валідації запиту та обробки помилок, декларуючи у контролері лише список обробників, які треба застосувати, відокремивши таким чином унікальний для даного контролера функціонал (наприклад, викликати метод певного сервісу) від загального (якщо сервіс надсилає NotFoundError, треба повернути 404), і додавати цей функціонал без прямої модифікації коду, до якого він додається. Це дозволяє суттєво підвищити чистоту коду, модульність, інтуїтивність розуміння структури проекту, перевикористовуваність компонентів.

Nest.js має об'ємну базу документації зі зрозумілою структурою, прикладами, інструкціями по використанню з іншими технологіями. Його архітектура дає можливість будувати легко масштабовувані проекти, а також пришвидшує процес розробки та позитивно впливає на чистоту і коректність коду. Саме тому цей фреймворк було обрано для розробки веб-сервера розробленого в рамках даної роботи застосунку.

2.5 JSON Web Token (JWT)

JWT – відкритий стандарт для створення токенів аутентифікації. Токен доступу складається з трьох частин – заголовку(header) у форматі JSON, у якому міститься інформація про тип алгоритму, використаного для підпису, і тип токена (має значення "JWT"), корисних даних(payload) у форматі JSON, у яких містяться термін придатності токена, ким він виданий і для кого, час

створення токена та ін. [11]. Третя частина містить підпис, який гарантує автентичність токену. Цей підпис перевіряється на сервері під час автентифікації користувача. JWT є одним з найпоширеніших сучасних стандартів автентифікації.

2.6 MongoDB

MongoDB – документо-орієнтована база даних. На відміну від реляційних баз даних, вона не нав’язує жорстко регламентовану модель даних і гнучкіше адаптується під потреби застосунку [12]. Документо-орієнтовані бази дають більше можливостей для зберігання даних і менше можливостей для підтримки цілісності даних та взаємозв’язків між ними. MongoDB – найкращий вибір для бази даних застосунку, розробленого в рамках даної роботи, адже дає можливість побудувати модель даних для предметної області, значно більш ефективну за реляційну, яка дозволить уникнути операцій з’єднання, без яких неможливо було б виконати навіть найпростіші запити, якщо дані були б збережені у реляційній базі даних. (Про це детальніше у ч.3).

РОЗДІЛ 3 Опис реалізації програми

3.1 Модель предметної області

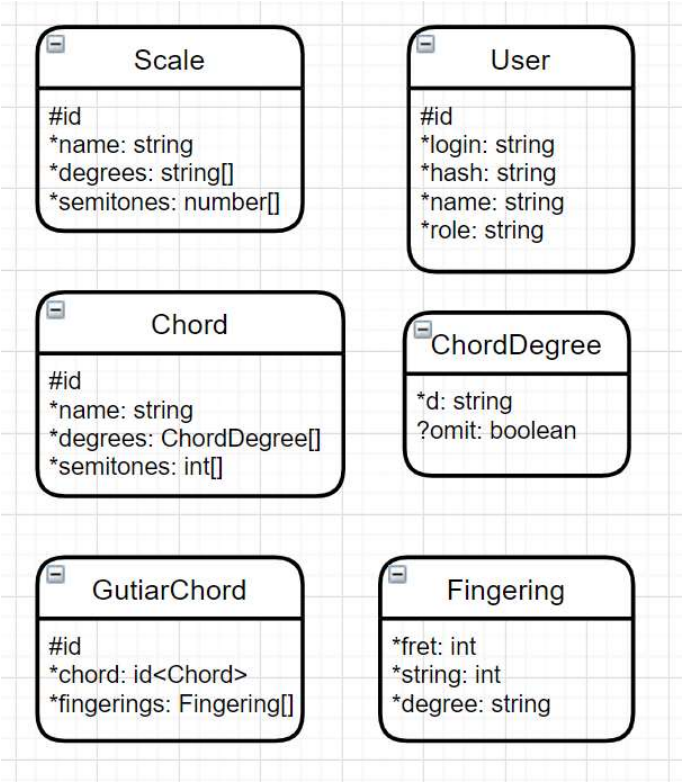


Рисунок 3.1.1 Схема даних

3.1.1 User

Сутність «User» має унікальний логін, хеш паролю, ім’я та роль, що приймає значення “admin” або “user”. Роль admin дозволяє створення та видалення музичних сутностей, user позначає зареєстрованого. Роль user додана з урахуванням подальшого розвитку застосунку, наразі права user нічим не відрізняються від прав незареєстрованих користувачів.

3.1.2 Scale

Scale – лад. Містить унікальне ім’я, список ступенів (degrees), заданих стрічками, список напівтонових відстаней (semitones) відносно певного тонального центру, відстань до якого дорівнює 0. Будь-який ступінь можна однозначно співставити з напівтоновою відстанню від центру, але напівтонову відстань зі ступенем – не можна. Лад задається списком ступенів, а застосунок сам співставляє його з напівтоновими відстанями. Є сенс зберігати обидва поля задля надання можливості виконувати запити по напівтоновим відстаням у БД.

Це потрібно у ситуаціях, коли треба знайти музичних об'єкт з ідентичним звучанням, але іншим представленням з точки зору теорії музики.

3.1.3 Chord

Позначає клас акордів, тобто структуру акорду в узагальненому представленні. Містить унікальне ім’я, список акордових ступенів, які аналогічні ступеням ладу за винятком того, що мають додаткове булеве поле omit, яке позначає, чи буде гітарний акорд, що не містить ноти, відповідної даному ступеню, вважатися правильним екземпляром даного класу акордів. Також містить список напівтонових відстаней, призначення якого аналогічне призначенню цього поля у сутності Scale.

3.1.4 GuitarChord

Позначає «екземпляр» акорду, тобто конкретний спосіб взяти акорд на гітарі. Містить ідентифікатор акорду, структурі якого відповідає. Містить також список об’єктів Fingering, кожен з яких має номер струни та взятого ладу, а також ступінь. Зберігання ступеню потрібно для того, щоб уникнути пошуку відповідностей між аплікатурою на грифі і ступенем, які потрібні при кожному використанні акорду. Встановлення цих відповідностей обов’язкове на етапі створення акорду з метою визначення його коректності, тож замість повторного встановлення цих відповідностей при кожному отриманні GuitarChord з БД можна зберегти їх. Враховуючи орієнтовну максимальну кількість сутностей у колекції (близько 300 шт.), можна зробити висновок про те, що витрати простору не будуть суттєвими.

3.2 База даних

Базу даних MongoDB розгорнуто в хмарі на спільному кластері за допомогою сервісу MongoDB Atlas.

MongoDB є високоефективним рішенням для реалізації даної моделі даних. З діаграми видно, що сутності «Scale», «Chord» і «GuitarChord» мають списки. Сутності Fingering та ChordDegree не можуть існувати без своїх сутностей-власників, а власники без них. Тому винесення їх в окремі таблиці,

як це було б реалізовано в реляційній СКБД, потягнуло б за собою непотрібні дорогі операції з’єднання при кожному зверненні до сутностей-власників. Зберігання допоміжних сутностей як піддокументів у MongoDB вирішує дану проблему.

3.3 Серверна частина

Сервер являє собою Rest API по сутностям, описаним вище, та JWT-автентифікацією.

3.3.1 Структура проекту

Проект згідно з архітектурою Nest розбито на модулі. Кожному REST-ресурсу відповідає модуль, також окремим модулем є модуль App – кореневий модуль, що відповідає за глобальні налаштування, і модуль Auth, що відповідає за автентифікацію.

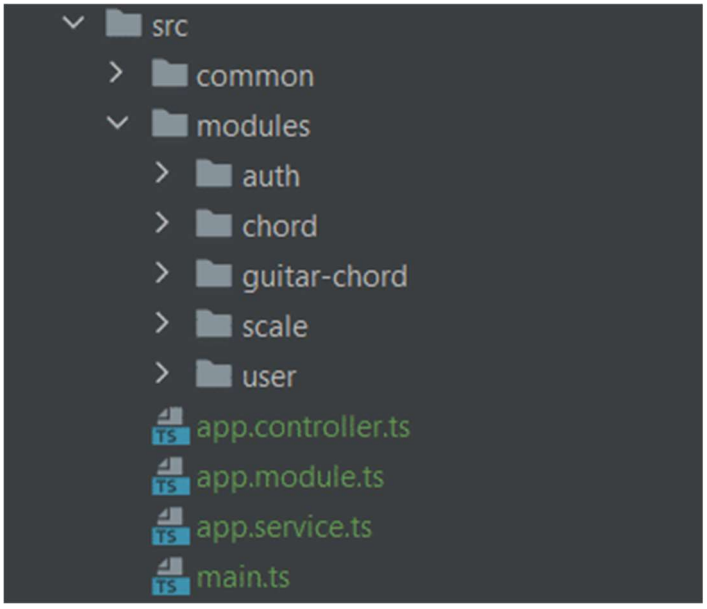


Рисунок 3.3.1.1 Структура модулів сервера

Загальні класи та утиліти, наприклад класи помилок, регулярні вирази для використання у валідації, глобальні обробники помилок і т.д. виносяться в окрему директорію common.

3.3.2 Початкові налаштування застосунку

```
async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.enableCors(); // to allow interactions with React
  app.use(helmet()); // middleware to protect from basic types of attacks
  app.useGlobalPipes(new ValidationPipe( options: { whitelist: true, transform: true }));
  await app.listen( port: 8080);
}
```

Рисунок 3.3.2.1 Запуск застосунку

Клас NestFactory збирає застосунок згідно з інструкціями з AppModule. Отриманий об’єкт app використовується для додаткових налаштувань. У даній функції встановлюється дозвіл на Cross-Origin-Resource-Sharing для роботи з React, додається проміжний обробник запитів Helmet, що фільтрує небезпечний вміст у вхідних запитах, а також додається проміжний обробник, що виконуватиме валідацію і трансформацію вхідних даних на базі валідаційних правил, задекларованих через TS-декоратори з пакету class-validator.

```
@Module( metadata: {
  imports: [
    ConfigModule.forRoot( options: { isGlobal: true }),
    TypegooseModule.forRoot(process.env.MONGODB_URI),
    ChordModule,
    GuitarChordModule,
    ScaleModule,
    AuthModule,
    UserModule,
  ],
  controllers: [AppController],
  providers: [
    AppService,
    {
      provide: APP_INTERCEPTOR,
      useClass: EntityAlreadyExistsInterceptor,
    },
    {
      provide: APP_INTERCEPTOR,
      useClass: NotFoundInterceptor,
    },
  ],
})
export class AppModule {}
```

Рисунок 3.3.2.2 AppModule

AppModule є кореневим модулем Nest-застосунку, він імпортує всі інші модулі застосунку, а також вбудовані службові модулі, такі як ConfigModule, що відповідає за парсинг .env файлів та надання доступу до змінних середовища, та TypegooseModule, що є розширенням Nest.js на базі пакетів Mongoose і Typegoose, що відповідає за роботу з MongoDB у Nest.js. Також у

даному модулі встановлюються глобальні перехоплювачів помилок, які автоматично виконуватимуться, коли в будь-якому контролері з’являтиметься помилка EntityAlreadyExistsError або NotFoundError. Ці перехоплювачі потрібні для того, щоб встановити відповідність між помилками бізнес-логіки, які виникають на рівні сервісів та не залежать від протоколів, що використовуються у контролерах, та HTTP-помилками.

3.3.3 UserModule

```
@Module( metadata: {
  imports: [TypegooseModule.forFeature( models: [User])],
  controllers: [UserController],
  providers: [UserService],
  exports: [UserService],
})
export class UserModule {}
```

Рисунок 3.3.3.1 UserModule

Декларація модуля користувачів проста – контролер, сервіс, який експортується назовні, та імпорт моделі (по суті DAO, Data Access Object – об’єкт, що надає функціонал для надсилання запитів у БД) сутності User, згенерованої через TypegooseModule.

```
export class User {
  _id: Types.ObjectId;

  @prop( options: { required: true })
  name: string;

  @prop( options: { required: true, unique: true })
  login: string;

  @prop( options: { required: true })
  hash: string;

  @prop( options: { required: true })
  role: Role;
}
```

Рисунок 3.3.3.2 user.ts

Клас User містить поля, анотовані декоратором @prop, який вказує TypegooseModule, яким чином генерувати модель для даного класу. Поле _id не

анотоване, адже воно створюється за замовчуванням, і додане у даний клас, як і всі інші класи сутностей, лише для доступності в статично типізованому коді TS.

Клас UserService містить створення, пошук і видалення користувача. Пароль користувача хешується за допомогою алгоритму bcrypt. Код класу є тривіальним.

3.3.4 Автентифікація

В AuthModule імпортується вбудований в Nest.js JwtModule, який відповідає за процес верифікації та підпису токенів. Налаштувати його синхронно неможливо, адже він має залежність – ConfigModule, експортований яким ConfigService надає метод get, за допомогою якого можна отримати змінну середовища JWT_SECRET для налаштування JwtModule. Для цього застосовано метод registerAsync, в параметр useFactory якого передається функція, через яку Nest зможе надати необхідну залежність за допомогою ін'єкції залежностей. Також AuthModule потребує UserModule для пошуку користувачів у БД. Строк придатності токена встановлюється у 10 хв.

```
@Module( metadata: {
  imports: [
    JwtModule.registerAsync( options: {
      useFactory: (configService: ConfigService) => ({
        secret: configService.get('JWT_SECRET'),
        signOptions: {
          expiresIn: '600s',
        },
      }),
      inject: [ConfigService],
    }),
    UserModule,
  ],
  controllers: [AuthController],
  providers: [AuthService, JwtStrategy],
})
export class AuthModule {}
```

Рисунок 3.3.4.1 AuthModule

AuthService містить логіку валідації користувача по логіну і паролю. Метод validateUser знаходить користувача в БД та порівнює пароль з хешем паролю з БД, створеним за допомогою алгоритму bcrypt. У випадку успішної перевірки користувача експортований з JwtModule jwtService, отриманий за допомогою ін'єкції залежностей, генерує токен доступу, який клієнтський код потім надсилатиме серверу через HTTP-заголовок Authorization.

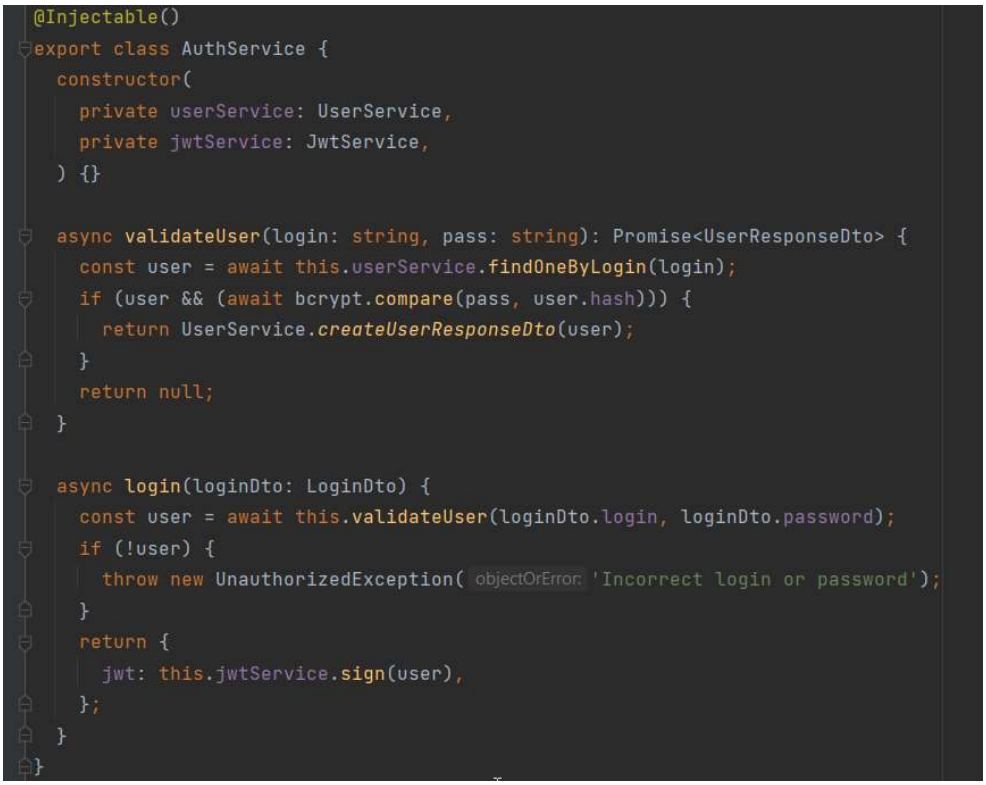


Рисунок 3.3.4.2 AuthService

AuthController містить POST-запит /auth/login, який викликає відповідну функцію сервісу.

Рисунок 3.3.4.3 AuthController

Клас JwtStrategy, розміщений у файлі jwt.strategy.ts, вказує, яким чином обробляти токен, що надсилається клієнтом. Nest самостійно знаходить цей клас і завантажує його, де він використовується JwtModule-м. Вказані опції – не пропускати прострочені токени, шукати токени у хедері Authorization у форматі Bearer myToken та секретний ключ для розпаковки токена. Метод validate викликається «під капотом» з метою користувацької валідації, але вбудованої валідації наразі достатньо, тому його реалізація тривіальна.

```
@Injectable()
export class JwtStrategy extends PassportStrategy(Strategy) {
  constructor() {
    super({
      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
      ignoreExpiration: false,
      secretOrKey: process.env.JWT_SECRET,
    });
  }

  async validate(payload: any) {
    return payload;
  }
}
```

Рисунок 3.3.4.4 JwtStrategy

Тепер можна створити JwtAuthGuard – проміжний обробник, який перевірятиме наявність валідного токену. Для цього треба створити клас-нащадок вбудованого класу AuthGuard.

```
@Injectable()
export class JwtAuthGuard extends AuthGuard( type: 'jwt') {}
```

Рисунок 3.3.4.5 JwtAuthGuard

Тепер можна у лаконічному декларативному стилі додавати функціонал автентифікації.

```
@UseGuards(JwtAuthGuard, AdminRoleGuard)
@Post()
create(@Body() createChordDto: CreateChordDto) {
  return this.chordService.create(createChordDto);
}
```

Рисунок 3.3.4.6 Endpoint з перевіркою токена адміна

Аналогічним чином можна реалізувати перевірку ролі. AdminRoleGuard перевіряє роль користувача, об’єкт з даними про якого дістається з об’єкту запиту. Інтерфейс CanActivate містить функцію, яка повертає булеве значення, що вказує, чи можна пропускати запит далі.

```
@Injectable()
export class AdminRoleGuard implements CanActivate {
  canActivate(context: ExecutionContext): boolean {
    const { user } = context.switchToHttp().getRequest();
    return user.role === Role.ADMIN;
  }
}
```

Рисунок 3.3.4.7 Перевірка прав адміна

3.3.5 Модулі ChordModule, GutiarChordModule, ScaleModule

Ці модулі містять CRUD-операції над сутностями БД. Кожен з них вказує TypegooseModule створити модель, що відповідає сутності, навколо якої побудований даний модуль. Операції створення та видалення сутностей доступні лише адміністраторам. Операція модифікації не імплементована, адже зміна у структурі акорду\ладу потягне за собою порушення цілісності даних, які спираються на них або спиратимуться в майбутньому.

При створенні акорду перевіряється, чи існує акорд з такою структурою у базі. У випадку існування сервіс кидає помилку, яку обробляє EntityAlreadyExistsInterceptor, доданий в AppModule як глобальний перехоплювач помилок. У протилежному випадку до сутності додається масив напівтонових відстаней і вона зберігається.

Рисунок 3.3.5.1 Створення акорду у ChordService

При створенні аплікатури гітарного акорду виконується додаткова валідація, для якої потрібен ChordService – перевірка, чи відповідають ноти, обрані на ладах гітари, структурі класу акордів, до якого належить аплікатура. (дод. 1).

Створення ладу не має специфічної логіки – лише перевірка, чи не існує в БД такий лад.

3.3.6 Патерн DTO для доступу до тіла або параметрів запиту у коді та їх валідації

Вхідні дані для запитів на створення сутностей енкапсулюються в об’єкти DTO (Data Transfer Object). За допомогою декоратора @Body можна вказати Nest.js розмістити дані з розпарсеного тіла запиту у даному об’єкті. Поля DTO анотовані декораторами валідації для перевірки коректності вхідних даних. Декларативний стиль декораторів дозволяє відокремити логіку валідації від бізнес логіки. Сам процес валідації виконується вбудованим об’єктом ValidationPipe, який було створено на етапі налаштування застосунку. Окрім вбудованих перевірок виконуються і користувацькі, наприклад IsValidChordDegreeList, який перевіряє, чи містить список ступенів, з яких

складається створюваний акорд, лише стрічки вигляду «1», «3», «7», «b2», «#9», та чи містить цей список 2 або більше ступенів з різними назвами, але однаковими напівтоновими відстанями, наприклад, #2 і b3, які обоє рівні 3 напівтонам, але мають різні назви залежно від теоретичного трактування акорду, та інших валідаторів.

```
export class ChordDegreeDto {
  @IsString()
  d: string;

  @IsOptional()
  @IsBoolean()
  omit?: boolean;
}

export class CreateChordDto {
  @IsString()
  @IsNotEmpty()
  name: string;

  @ArrayMinSize( min: 2)
  @ArrayMaxSize( max: 11)
  @ValidateNested( validationOptions: { each: true })
  @Type( typeFunction: () => ChordDegreeDto)
  @Validate(IsValidChordDegreeList)
  degrees: ChordDegreeDto[];
}
```

Рисунок 3.3.6.1 Створення акорду у ChordDegreeDto

3.4 Клієнтська частина

3.4.1 Структура проекту

Проект створено за допомогою утиліти create-react-app. Проект розбито на директорії components(містять розроблені графічні компоненти), css (стилі), services(взаємодія з API та інша бізнес-логіка, така як програвання звуків), common (містить загально використовуваний код, такий як об’єкти Context API

та користувацькі хуки, що дають змогу додавати зручно додавати повторюваний складний функціонал до компонентів).

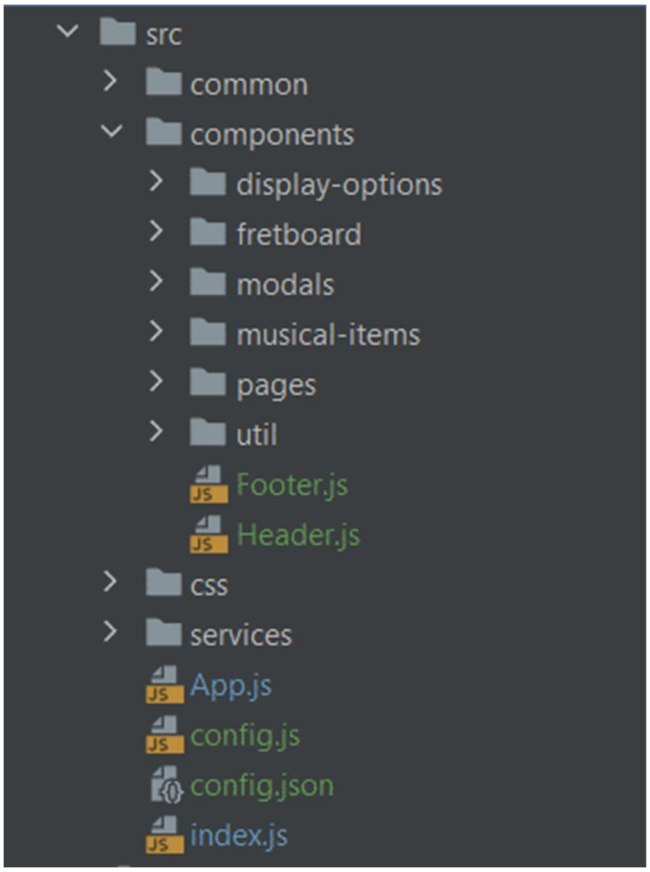


Рисунок 3.4.1.1 Структура проекту клієнтської частини

3.4.2 Запуск додатку

```
const root = ReactDOM.createRoot(document.getElementById( elementId: 'root'));
root.render(
    <React.StrictMode>
        <BrowserRouter>
            <ScrollToTop/>
            <App/>
        </BrowserRouter>

    </React.StrictMode>
);
```

Рисунок 3.4.2.1 Розміщення React компонентів на сторінці

У елементі з ідентифікатором root ReactDOM розміщує компонент App, що відповідає за відображувану частину застосунку, додаючи компоненти ScrollToTop для перенесення скролу вгору за замовчуванням під час переключення компонентів та BrowserRouter, що робить можливою навігацію засобами браузера без повторного створення всієї сторінки.

3.4.3 App

Компонент App містить загальні елементи інтерфейсу, такі як хедер і футер, та список компонентів, які відображуватимуться для різних URL. У цьому компоненті також виконуються запити до API для отримання даних про акорди і лади, ініціалізуються провайдери контекстів LoginContext та ContentContext, дані з яких потім використовуються компонентами, обгорнутими в них.

```
const App = () => {
    const [displayChord, setDisplayChord] = useState( initialState: null)
    const [displayScale, setDisplayScale] = useState( initialState: null)
    const [chords, setChords] = useState( initialState: null)
    const [scales, setScales] = useState( initialState: null)
    useEffect( effect: () => {
        getChords().then(setChords)
    }, deps: [])
    useEffect( effect: () => {
        getScales().then(setScales)
    }, deps: [])
    const contentContextValue = {displayChord, setDisplayChord,
        displayScale, setDisplayScale, chords, setChords, scales, setScales}
    return (
        <div className="app">
            <Header/>
            <ContentContext.Provider value={contentContextValue}>
                <LoginContext.Provider value={useLogin()}>
                    <Routes>
                        <Route path="/" element={<Home/>}/>
                        <Route path="/chords" element={<Chords/>}/>
                        <Route path="/chords/:id" element={<ChordDetails/>}/>
                        <Route path="/scales" element={<Scales/>}/>
                        <Route path="/scales/:id" element={<ScaleDetails/>}/>
                    </Routes>
                    <Footer/>
                </LoginContext.Provider>
            </ContentContext.Provider>
        </div>
    );
}
```

Рисунок 3.4.3.1 App

3.4.4 Автентифікація

Рисунок 3.4.4.1 Функція login

Auth-service містить функції login, logout, isLoggedIn. Login виконує POST-запит на ендпоінт /auth/login для отримання токену і зберігає його у localStorage. Logout видаляє токен, isLoggedIn перевіряє наявність і валідність токена.

Для ініціалізації LoginContext.Provider використовується хук useLogin, який створює стан логіну та прив’язує зміни цього стану до відповідних змін у сховищі з токеном.

```
const useLogin = () => {
    const [isLogged, setLogged] = useState(isLoggedIn())

    return {
        isLogged,
        logout: () => {
            logout()
            setLogged( value: false)
        },
        updateLogin: () => {
            setLogged(isLoggedIn())
        }
    }
}

export default useLogin
```

Рисунок 3.4.4.2 Хук useLogin

Кнопка входу захована у футері, адже наразі вхід потрібен лише адміністраторам. При натисканні з’являється проста форма входу

Рисунок 3.4.4.3 Форма для логіну

3.4.5 Сторінки ладів та акордів

Ці сторінки містять списки ладів та акордів відповідно(дод. 2). Для адміністратора також відображатиметься кнопка додавання акорду\ладу. При кліку на картку відбувається перехід на сторінку відповідного акорду\ладу.

3.4.6 Деталі перегляду акорду

На сторінці з деталями акорду(дод. 3) відображуються назва акорду (за замовчуванням він будується від ноти C), список його ступенів, нот та напівтонова структура. На першому грифі зображено ноту акорду по всьому грифу. При натисканні на Play програються послідовно ноти акорду у двох

октавах. Далі знаходяться аплікатури акордів – конкретні способи взяти акорд на гітарі. Їх можна прослухати у 2-х режимах – Strum, що зіграє ноти одним послідовним штрихом, та Pluck, що зіграє їх одночасно. Кількість ладів на картці автоматично підганяється під розмір аплікатури. Після цього йде секція із взаємозв'язками акорду. Першими представлені взаємозв'язки ладів з даним класом акордів, а потім показано, в яких ладах знаходиться конкретний акорд, побудований від заданої ноти.

3.4.7 Деталі перегляду ладу

На сторінці ладу(дод. 4) знаходиться інформація про лад. На грифі, підсвічені всі ноти з нього. При натисканні на кнопку Play вони програються послідовно вгору. Далі йдуть пов’язані лади, впорядковані за ступенями – ті, що можна побудувати від відповідного ступеня так, щоб всі їх ноти містились у ладі, який розглядається. Ще одна функція – пошук акордів які можна побудувати в межах даного ладу.

3.4.8 Компоненти Fretboard і FretboardCard

Fretboard (дод. 5) виконує відображення грифу з нотами. На вхід він приймає початковий лад на грифі, від якого відображати гриф, кількість ладів, а також функцію getDisplaySymbol, яка повертає символ, що треба відобразити. Таким чином гриф гітари відокремлюється від того, що зображується на ньому, і знає лише номери своїх ладів, що робить його більш перевикористовуваним.

FretboardCard обгортає гриф у картку, в якій він зображується на екрані. Даний компонент приймає на вхід тип відображення, і базуючись на цьому налаштовує гриф, розміщує відповідні кнопки. Є 3 типи вмісту грифу – арпеджіо акорду, лад, аплікатура акорду. Всі вони мають різні режими програвання звуків та різні кнопки програвання. Створення відповідного контенту грифу є завданням FretboardCard.

3.4.9 Параметри відображення

На сторінках детального перегляду в правому верхньому кутку є кнопка параметрів відображення. У панелі, що з'явиться при натисканні, можна змінювати кореневу ноту та спосіб відображення нот\ступенів на гітари, який має 3 опції – показувати ступені (degrees), показувати ноти зі знаком b (flats), показувати ноти зі знаком # (sharps). При натисканні на кнопку Apply React моментально перемальовує всі компоненти на сторінці, що залежать від даних налаштувань.

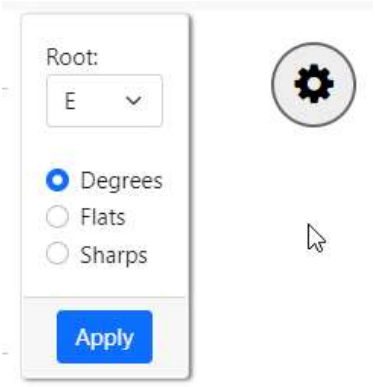


Рисунок 3.4.9.1 Меню параметрів відображення

Функціонал, пов’язаний з роботою цієї форми, додається в компоненти сторінок деталей за допомогою користувацького хука useDisplayOptions.

```
const useDisplayOptions = ({initRoot : number = 0, initAlteration : string = 'b', initMode : number = DisplayMode.DEGREES}) => {
    const [displayMode, setDisplayMode] = useState(initMode)
    const [alteration, setAlteration] = useState(initAlteration)
    const [root, setRoot] = useState(initRoot)
    const [isPanelOpen, setPanelOpen] = useState( initialState: false)
    const onDisplayOptionsChange = (values) => {
        setDisplayMode(values.displayMode)
        setAlteration(values.alteration)
        setRoot(values.root)
        setPanelOpen( value: false)
    }
    return {
        displayMode,
        setDisplayMode,
        alteration,
        setAlteration,
        root,
        setRoot,
        isPanelOpen,
        setPanelOpen,
        onDisplayOptionsChange
    }
}

export default useDisplayOptions
```

Рисунок 3.4.9.2 Хук useDisplayOptions

3.4.10 Бізнес-логіка

Бізнес логіка винесена у сервіси. Для надсилання запитів на сервер використовується вбудований Fetch API. Note-service містить логіку роботи з нотами – відображення символу ноти за числовим значенням, пошук ноти на грифі гітари, відображення всіх нот акорду\ладу у вигляді символів з відповідною альтерацією. Degree-service містить код для роботи зі ступенями – створити стрічкове представлення списку ступенів, валідації ступенів і т. д..

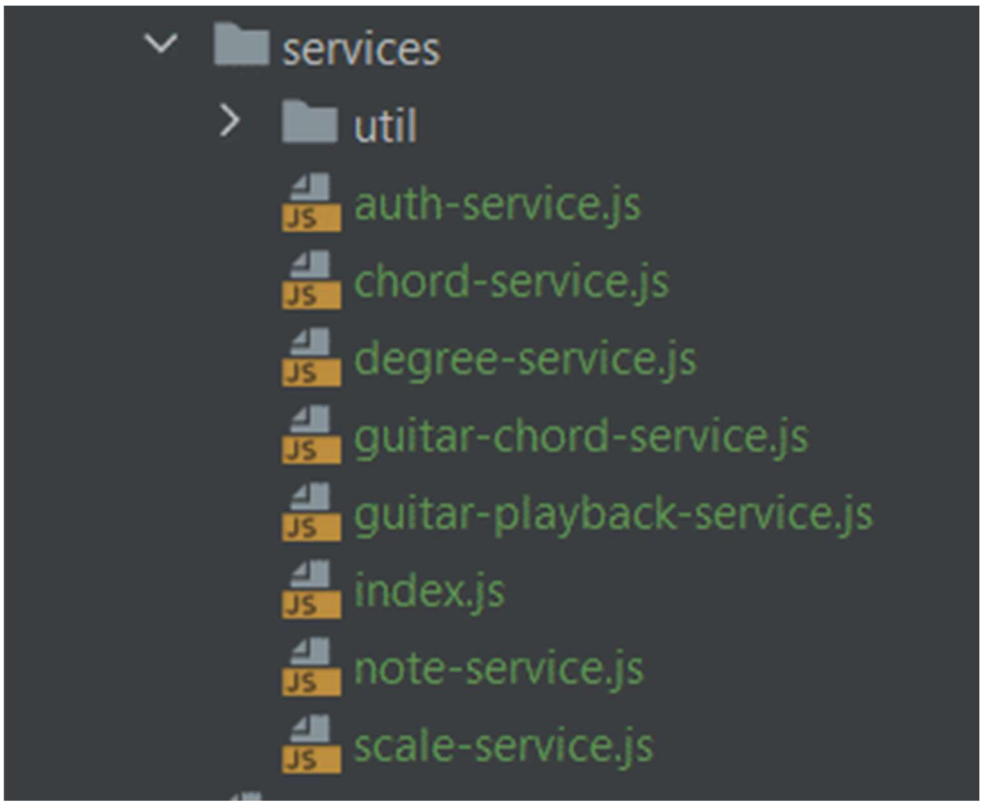


Рисунок 3.4.10.1 Сервіси

Chord-service містить CRUD-запити до сервера для роботи з акордами, а також логіку пошуку ладів, що містять певний акорд.

```
export const findScalesContainingSemitones = (scales, semitones) => {
    const res = []
    scales.forEach((scale) => {
        const pos = []; // this will contain degrees, on which provided chord can be built in a scale
        const scaleDegreesSet = new Set(scale.semitones);

        // check if chord semitones are present relatively to each semitone in the scale
        for (let i = 0; i < scale.semitones.length; ++i) {
            const scaleHasChord = semitones.every((d) =>
                scaleDegreesSet.has( value: (d + scale.semitones[i]) % 12),
            );

            if (scaleHasChord) {
                // add degree, on which chord is built in the scale
                pos.push(scale.degrees[i]);
            }
        }

        // if pos wasn't updated - scale doesn't have the chord
        if (pos.length) {
            res.push({...scale, pos})
        }
    })
    return res
}
```

Рисунок 3.4.10.2 Пошук ладів за акордом

Scale-service містить CRUD-запити до сервера для роботи з ладами, а також логіку пошуку акордів і ладів, що можна побудувати від кожного ступеня ладу.

```
export const findSemitonesInDegreeSet = (degrees, scaleOrChordData) => {
    const degreeMap = new Map() // key: degree, value: list of data that can be built from it
    const semitones = degrees.map(d => degreeToSemitones(d))
    const semitonesSet = new Set(semitones);

    scaleOrChordData.forEach((item) => {

        for (let i = 0; i < semitones.length; ++i) {
            const scaleHasChord = item.semitones.every((d) =>
                semitonesSet.has( value: (d + semitones[i]) % 12),
            );

            const degree = degrees[i]
            if (scaleHasChord) {
                // add degree, on which chord is built in the scale
                if (degreeMap.has(degree)) {
                    degreeMap.get(degree).push(item)
                } else {
                    degreeMap.set(degree, [item])
                }
            }
        }
    })

    return degreeMap
}
```

Рисунок 3.4.10.3 Пошук того, що можна побудувати від кожного ступеня ладу

Guitar-scale-service містить CRUD-запити для роботи з акордами, а також алгоритм побудови гітарних аплікатур від заданої кореневої ноти (на сервері аплікатури завжди зберігаються відносно кореневої ноти C для надання їх

уніфікованого вигляду). Всі ноти переносяться вліво по грифу (номери ладів зменшуються) якщо не виходять за межі грифу, або, якщо виходять - вправо.

```
const moveFret = (n, lowestFret, newRoot) =>
    (lowestFret + newRoot) % 12 >= lowestFret
        ? n + newRoot
        : n - (12 - newRoot);

export const transposeFingerings = (fingerings, newRoot) => {
    const transposedFingerings = []
    const initialMin = findMinAndMaxFrets(fingerings).min

    fingerings.forEach((f) => {
        transposedFingerings.push({...f, fret: moveFret(f.fret, initialMin, newRoot)})
    })

    return transposedFingerings
}
```

Рисунок 3.4.10.4 Побудова аплікатур відносно певної тоніки

За програвання звуків відповідає guitar-playback-service. У ньому створюється об’єкт Tone.Sampler, який приймає на вхід мапу аудіо файлів, що відповідають нотам різної висоти. Для цього не обов’язково передавати файли для кожної ноти. Tone.js автоматично змінить висоту звуку для програвання інших нот. Далі створюється об’єкт reverb – проміжний обробник звукового сигналу, який створює ефект приміщення. Вихідний сигнал гітари проходить через reverb і направляється на головний аудіовихід браузера.

```
const guitar = new Tone.Sampler( options: {
    urls: {
        A2: "A2.mp3",
        A3: "A3.mp3",
        A4: 'A4.mp3',
    },
    baseUrl: "/assets/sounds/"
})

const reverb = new Tone.Reverb( options: {decay: 0.5}).toDestination()

guitar.connect(reverb)
```

Рисунок 3.4.10.5 Налаштування Tone.js

Ноти для програвання звуку подаються у форматі C5, де C – назва ноти, 5 – номер октави, а також довжину ноти. Функція playSequence програє набір нот notes від певної початкової ноти root. Параметр octaveRepeats відповідає за

кількість повторень послідовності звуків у наступних октавах. Якщо, наприклад, на вхід приходить масив notes [0, 4, 7, 11], root 5 і ocataveRepeats 2, то програється послідовність F3 A3 C4 E4 F4 A4 C5 E5.

```
export async function playSequence(notes, root, octaveRepeats :number = 1, timeIntervalMs :number = 200) {
    const baseOctave = 3
    for (let i = 0; i < octaveRepeats; ++i) {
        for (let j = 0; j < notes.length; ++j) {
            const note = notes[j] + root
            const octavesAbove = Math.floor( x: note / 12)
            guitar.triggerAttackRelease(
                notes: getNoteSymbol( num: note % 12,  altSym: '#') + (i + baseOctave + octavesAbove),
                duration: "2n"
            )
            await wait(timeIntervalMs)
        }
    }
}
```

Рисунок 3.4.10.6 Програвання послідовності нот

3.4.11 Додавання ладів, акордів, аплікатур

Адміністратор має змогу додавати нові сутності у програму. Акорд задається у вигляді двох списків ступенів через кому – списку основних ступенів та списку тих ступенів зі списку вище, які можна пропускати. Лад задається одним списком ступенів, адже в ньому немає необов’язкових. Аплікатура задається списком із 6 розділених комою чисел або символів x, що відповідають струнам від 6 (найтовщої) до 1 (найтоншої). Символ x означає відсутність ноти на даній струні. (дод. 6)

Висновки

У ході виконання даної курсової роботи були досліджені сучасні підходи до розробки клієнтської та серверної частин веб-застосунків. Були проаналізовані підходи до управління станом React-додатків, шаблони проектування у програмах з використанням React та Nest, способи застосування декларативного стилю програмування в сучасних об’єктно-орієнтованих програмах (аспектно-орієнтоване програмування) з метою покращення організації коду, пришвидшення процесу розробки, досягнення сильної зв’язності та слабкої зв’язаності компонентів, вирішення повторюваних задач, таких як валідація, обробка помилок, аутентифікація і авторизація.

Результат виконання даної роботи – клієнт-серверний веб-застосунок, що надає інструментарій для побудови і візуалізації акордів і ладів, пошуку взаємозв’язків між ними. Під час розробки були застосовані результати аналізу технологій React та Nest. Дані технології є вдалим вибором для предметної області, що позитивно вплинуло на процес розробки. Отриманий додаток можна буде легко розширити новими функціями. Плануються подальші дії для розвитку застосунку – покращення UI\UX дизайну, додання мобільної версії веб-сайту, додання підтримки української мови, розгортання системи на хмарному хостингу, розширення функціоналу системи роботою з новими музичними інструментами.

Використані джерела

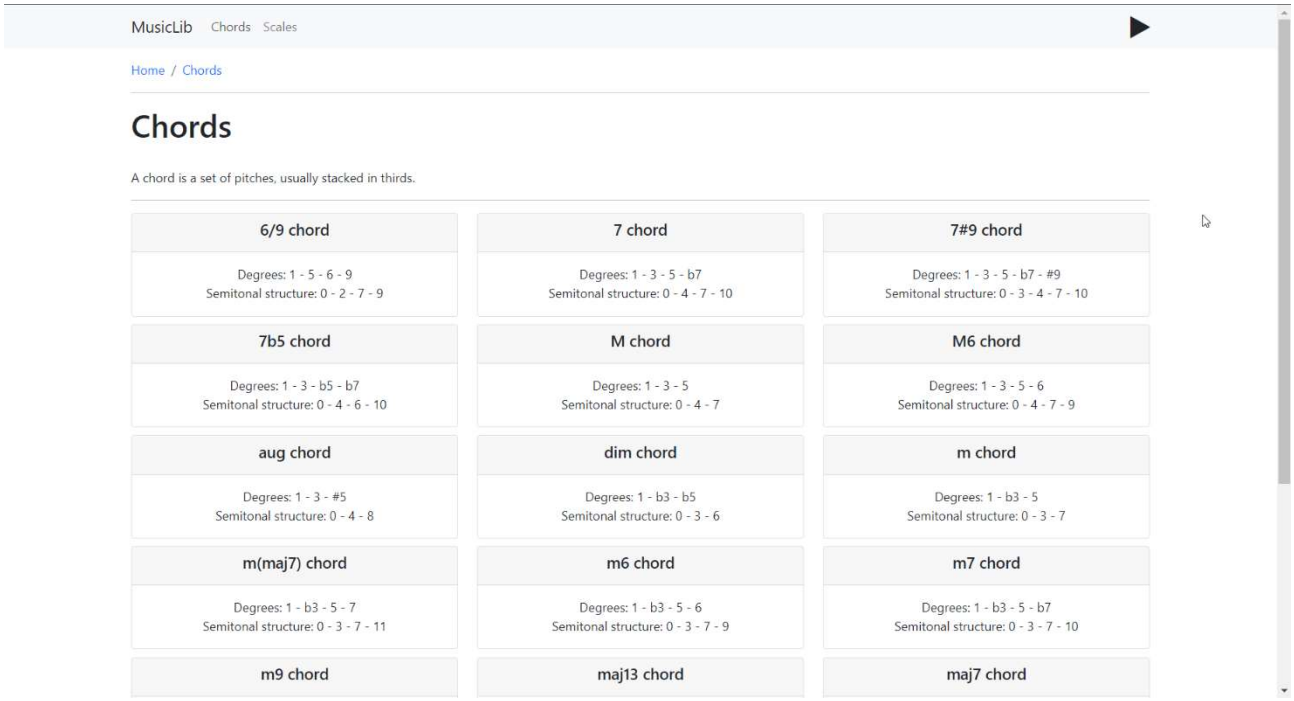
1. Сторінка застосунку All Chords у Google Play [Електронний ресурс] – Режим доступу до ресурсу: https://play.google.com/store/apps/details?id=com.mv2studio.allchodrs&hl=uk&gl=US
2. Сторінка застосунку Chord Database [Електронний ресурс] – Режим доступу до ресурсу: https://tombatossals.github.io/react-chords/
3. Сторінка застосунку ScalesLearn [Електронний ресурс] – Режим доступу до ресурсу: https://play.google.com/store/apps/details?id=com.completethink.scales.android&hl=uk&gl=US
4. Сторінка застосунку Chordbook [Електронний ресурс] – Режим доступу до ресурсу: https://chordbook.com/
5. Документація React.js [Електронний ресурс] – Режим доступу до ресурсу: https://uk.reactjs.org/docs/
6. Документація Redux [Електронний ресурс] – Режим доступу до ресурсу: https://redux.js.org/
7. Документація React-Bootstrap [Електронний ресурс] – Режим доступу до ресурсу: https://react-bootstrap.github.io/
8. Документація Tone.js [Електронний ресурс] – Режим доступу до ресурсу: https://tonejs.github.io/
9. Документація Nest.js [Електронний ресурс] – Режим доступу до ресурсу: https://nestjs.com/
10. An introduction to Aspect-Oriented Programming [Електронний ресурс] – Режим доступу до ресурсу: https://medium.com/@blueish/an-introduction-to-aspect-oriented-programming-5a2988f51ee2
11. Документація JWT [Електронний ресурс] – Режим доступу до ресурсу: https://jwt.io/introduction
12. Документація MongoDB [Електронний ресурс] – Режим доступу до ресурсу: https://www.mongodb.com/docs/

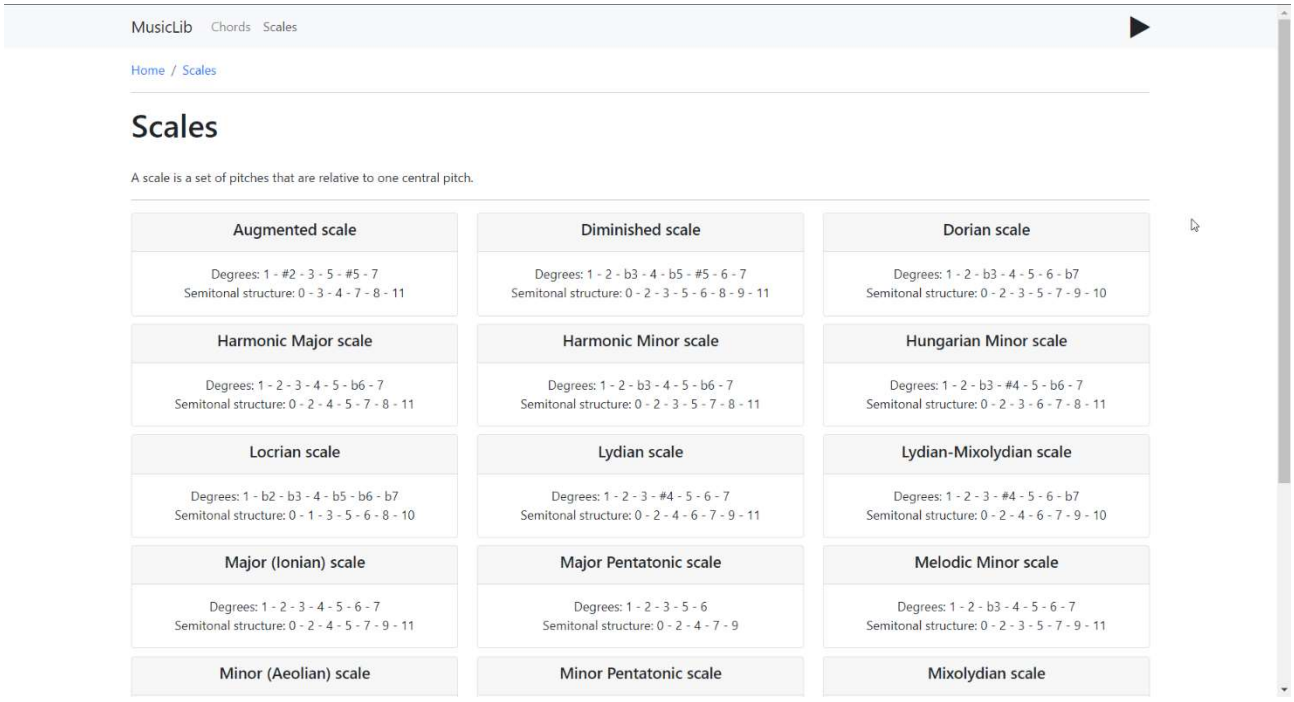
Додатки

Додаток 1 Створення аплікатури акорду

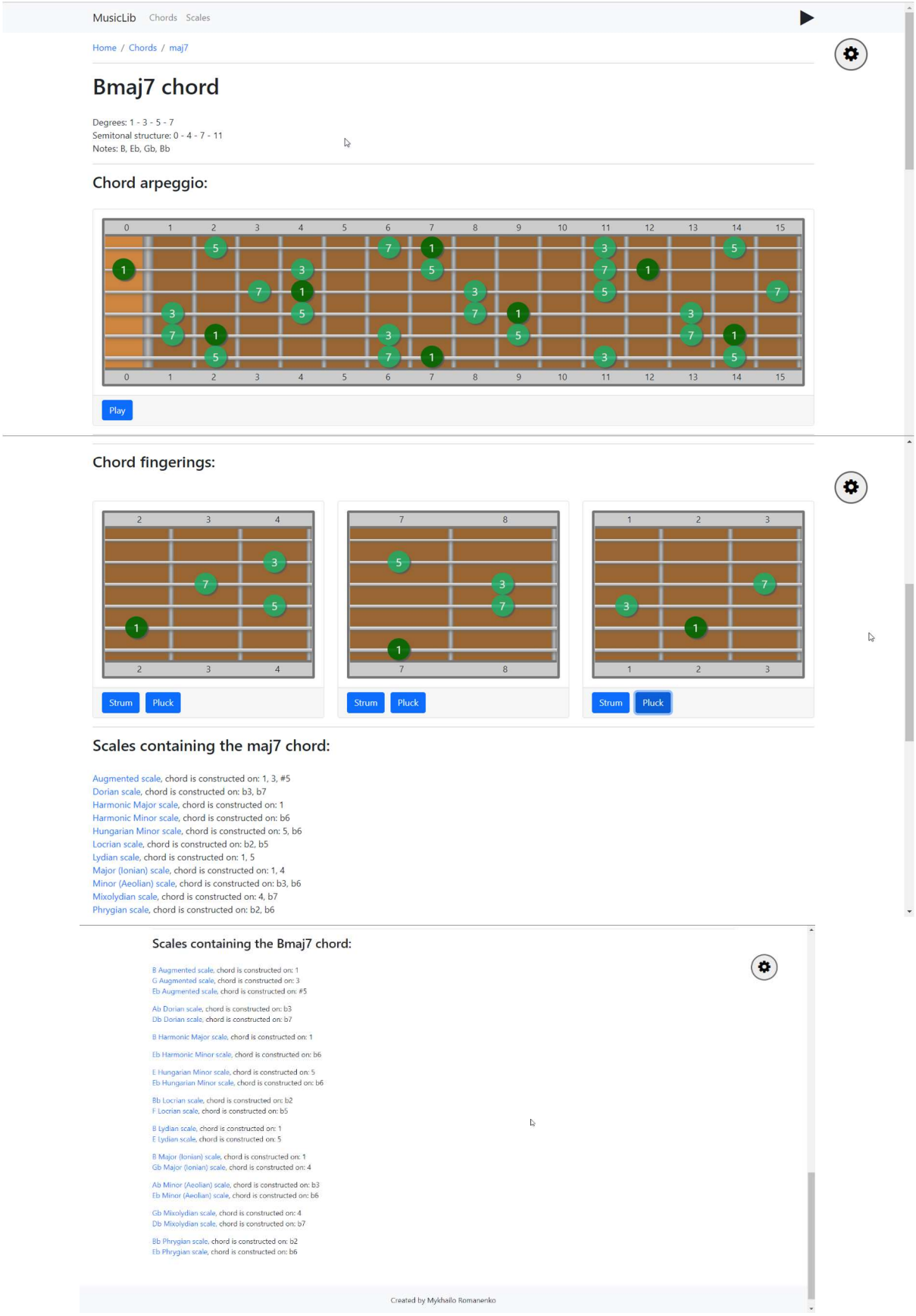
```
async create(createGuitarChordDto: CreateGuitarChordDto) {
  const semitonesFromFrets: number[] = this.getSemitonesFromFingerings(
    createGuitarChordDto.fingerings,
  );
  // find chord by provided id
  const providedChord: Chord = await this.chordService.findOneById(
    createGuitarChordDto.chord,
  );
  const fingeringsWithDegrees: Fingering[] = [];
  // check if chord structure matches fingerings (chord fingerings are in C)
  providedChord.degrees.forEach((degree: ChordDegree) => {
    const degreeSemitones = degreeToSemitones(degree.d);
    // if a degree is required and is absent -> error
    if (!degree.omit && !semitonesFromFrets.includes(degreeSemitones)) {
      throw new FingeringMismatchError( msg: 'Invalid fingerings.');
    }
    createGuitarChordDto.fingerings.forEach((f : FingeringDto ) => {
      if (
        degreeSemitones == DefaultGuitarFretboard.getNote(f.string, f.fret)
      ) {
        fingeringsWithDegrees.push({ ...f, degree: degree.d });
      }
    });
  });
  if (
    fingeringsWithDegrees.length !== createGuitarChordDto.fingerings.length
  ) {
    throw new FingeringMismatchError( msg: 'Invalid fingerings.');
  }
  fingeringsWithDegrees.sort( compareFn: (f1 : Fingering , f2 : Fingering ) => f1.string - f2.string);

  const existingGuitarChord = await this.guitarChordModel.findOne( filter: {
    fingerings: fingeringsWithDegrees,
  });

  if (
    existingGuitarChord &&
    existingGuitarChord.chord._id.equals(providedChord._id)
  ) {
    throw new EntityAlreadyExistsError( msg: `This fingering already exists`);
  }

  return this.guitarChordModel.create({
    ...createGuitarChordDto,
    fingerings: fingeringsWithDegrees,
  });
}
```

Додаток 2 Сторінки ладів та акордів

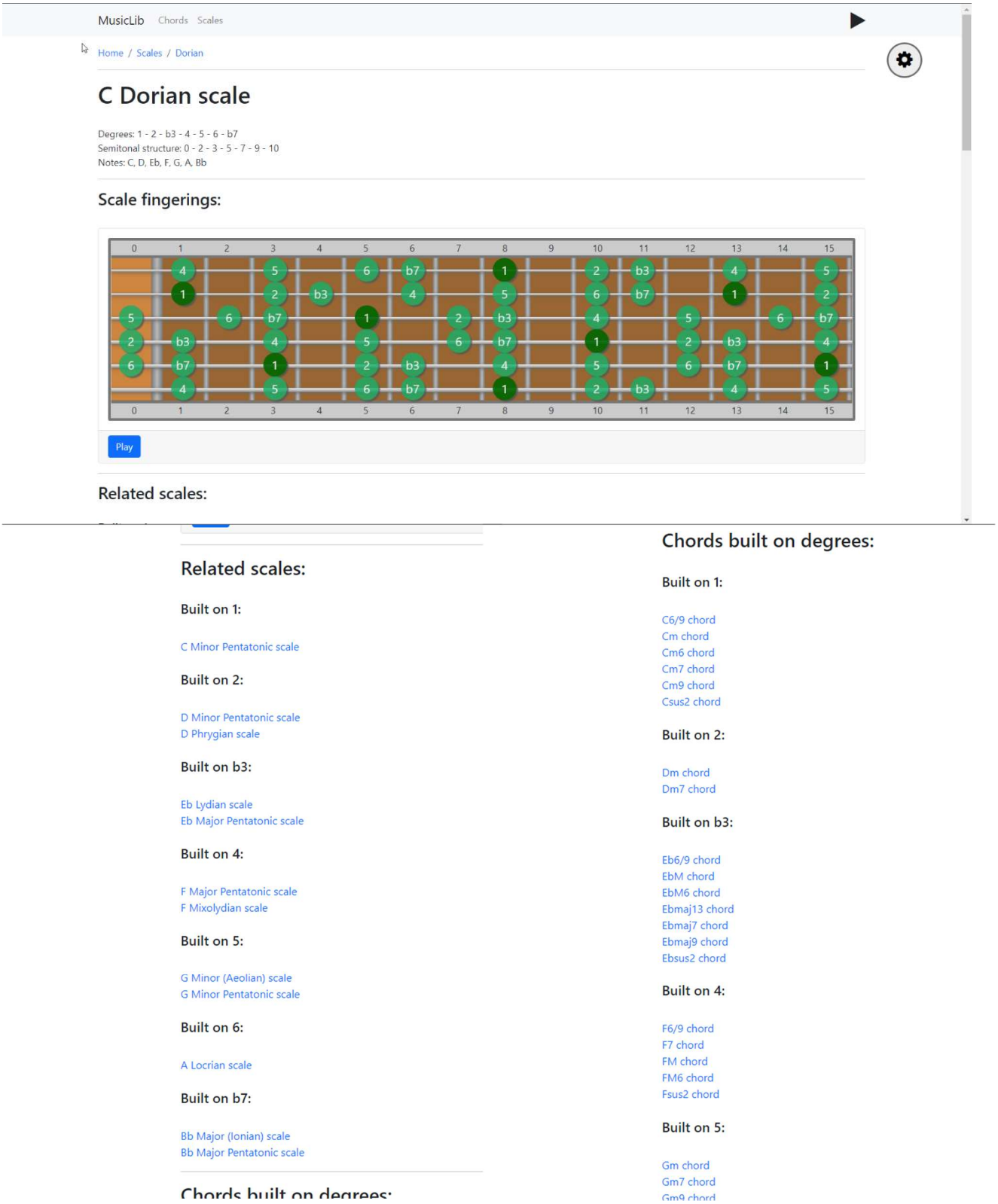




Додаток 3 Сторінка деталей акорду



Додаток 4 Сторінка деталей ладу



Додаток 5 Компонент Fretboard та його складові

```
const Fretboard = ({dimensions, getDisplaySymbol}) => {
    const strings = []
    for (let i = 1; i <= 6; ++i) {
        strings.push(
            <String key={i} strNum={i}
                    dimensions={dimensions}
                    getDisplaySymbol={getDisplaySymbol}/>
        )
    }
    const fretNumbers = []
    const {initFret, fretCount} = dimensions
    for (let i = initFret; i < initFret + fretCount; ++i) {
        fretNumbers.push(
            <div key={i} className={'fret' + (i === 0 ? ' zero-fret' : '') + ' no-guitar-styling'}>{i}</div>
        )
    }
    return (
        <div className="fretboard-wrapper">
            <div className="string-container">
                {fretNumbers}
            </div>
            <div className="fretboard">
                {strings}
            </div>
            <div className="string-container">
                {fretNumbers}
            </div>
        </div>
    )
}
const String = ({dimensions, getDisplaySymbol, strNum}) => {

    const frets = []

    for (let i = 0; i < dimensions.fretCount; ++i) {
        const fretNum = i + dimensions.initFret
        frets.push(<Fret key={fretNum} fretNum={fretNum} strNum={strNum} getDisplaySymbol={getDisplaySymbol}/>)
    }

    return (
        <>
            <div className="string-container">
                {frets}
                <div className="string-image" style={{height: `${3+strNum/2}px`}}/>
            </div>
        </>
    )
}
const Fret = ({fretNum, getDisplaySymbol, strNum}) => {
    const displaySymbol = getDisplaySymbol(strNum, fretNum)
    return (
        <div className={`fret ${fretNum === 0 ? " zero-fret" : ""}`}>
            {displaySymbol &&
                <div
                    className={'fret-highlighted' + (displaySymbol.isRoot ? ' root' : '')}
                    onClick={() => playSingleNote(getNoteWithOctave(strNum, fretNum))}
                >
                    <span className="note-symbol">{displaySymbol.sym}</span>
                </div>}
        </div>
    )
}
```

Додаток 6 Форми додання акорду та аплікатури