Київський національний університет імені Тараса Шевченка
Факультет інформаційних технологій
Кафедра програмних систем і технологій

УДК

На правах рукопису

ВИПУСКНА КВАЛІФІКАЦІЙНА БАКАЛАВРСЬКА РОБОТА

Тема: Розробка мобільного застосування для голосового написання програм
Спеціальність 121 “Інженерія програмного забезпечення”

ПОЯСНЮВАЛЬНА ЗАПИСКА

Студент ІПЗ-42
Олександр ДОБРОВОЛЬСЬКИЙ

Науковий керівник
Доцент Сергій ПОЛЯКОВ
(посада) (підпис) (розшифровка підпису) (дата)

Допускається до захисту
з питань нормоконтролю

_____
Тамара Чаповська
(підпис) (розшифровка підпису) (дата)

Завідувач кафедри доктор технічних наук ДТН, доцент Олексій БИЧКОВ
_____

2021

Київський національний університет імені Тараса Шевченка

Рішенням Екзаменаційної комісії
випускна кваліфікаційна робота студента

_____

захищена з оцінкою

_____

Голова Екзаменаційної комісії

професор, доктор техн. наук Вишнівський В.В
або
професор, доктор техн. наук Бондарчук А.П.

Факультет інформаційних технологій
Кафедра програмних систем і технологій
Спеціальність 121 “Інженерія програмного забезпечення”

ЗАТВЕРДЖУЮ:
Завідувач кафедри програмних систем і технологій Олексій БИЧКОВ
_____
„___” _____ 20___р.

ЗАВДАННЯ
НА ВИПУСКНУ КВАЛІФІКАЦІЙНУ БАКАЛАВРСЬКУ РОБОТУ
СТУДЕНТУ

_____
(прізвище, ім’я, по батькові)

1. Тема випускної кваліфікаційної бакалаврської роботи ”Розробка мобільного застосування для голосового написання програм”
затверджена наказом вищого навчального закладу від „__”_____20__р. №_____

2. Строк здачі студентом закінченої роботи _____
3. Вихідні дані до роботи _____
_____
_____
_____
_____
_____

4. Зміст пояснювальної записки (перелік питань, що їх належить розробити)
_____
_____
_____
_____
_____

5. Перелік графічного матеріалу (з точним забезпеченням обов’язкових креслень)
_____
_____
_____
_____

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання _____

Керівник _____ Сергій Поляков
(підпис) (розшифровка підпису)

Завдання прийняв до виконання _____ Олександр Добровольський
(підпис) (розшифровка підпису)

КАЛЕНДАРНИЙ ПЛАН

Номер і назва етапів бакалаврської роботи	Термін виконання етапів роботи	Примітка
Проектування	**09.03.2021**	виконано
Розробка мобільного додатку	**11.04.2021**	виконано
Розробка серверної частини	**15.04.2021**	виконано
Розробка веб клієнту	**26.04.2012**	виконано

Студент – бакалавр _____ **Олександр ДОБРОВОЛЬСЬКИЙ**
(підпис) (розшифровка підпису)

Керівник роботи _____**Сергій ПОЛЯКОВ**
(підпис) (розшифровка підпису)

АНОТАЦІЯ

Випускна кваліфікаційна бакалаврська робота: 39 с., 8 рис., 4 джерела.

Тема: Розробка мобільного застосування для голосового написання програм

Об'єкт дослідження: програма інтерпретації голосу в виконуваний код.

Мета роботи: розробка програмної системи що слугує для перетворення голосу в програмний код який можна виконувати за допомогою компілятора мови програмування JavaScript.

Предмет дослідження: технологія розпізнавання голосу та подальшого його представлення в якості виконуваного коду

Результати дослідження:

Одержана реалізація алгоритма розпізнавання голосу та подальшої його інтерпретації як виконуваного коду на мові програмування JavaScript.

АЛГОРИТМ, РОЗПІЗНАВАННЯ ГОЛОСУ, МОБІЛЬНИЙ ДОДАТОК, ІНТЕЛЕКТУАЛЬНА СИСТЕМА, СЕРВЕР, КРОСПЛАТФОРМЕНІСТЬ.

SUMMARY
Final qualifying bachelor's thesis: 39 pp., 8 figs., 4 sources.
Topic: Development of a mobile application for voice writing programs
Object of study: a program for interpreting voice into executable code.
Purpose: development of a software system used to convert voice into program code that can be executed using the JavaScript programming language compiler.
Subject of research: voice recognition technology and its subsequent presentation as executable code
Results of the research:
The implementation of the algorithm for voice recognition and its subsequent interpretation as executable code in the JavScript programming language is obtained.

ALGORITHM, VOICE RECOGNITION, MOBILE APPLICATION, INTELLECTUAL SYSTEM, SERVER, CROSS PLATFORM.

АННОТАЦИЯ
Выпускная квалификационная бакалаврская работа: 39 с., 8 рис., 4 источника.
Тема: Разработка мобильного приложения для голосового написания программ
Объект исследования: программа интерпретации голоса в исполняемый код.
Цель работы: разработка программной системы служащий для преобразования голоса в программный код который можно выполнять с помощью компилятора языка программирования JavaScript.
Предмет исследования: технология распознавания голоса и дальнейшего его представления в качестве исполняемого кода
Результаты исследования:
Полученная реализация алгоритма распознавания голоса и дальнейшей его интерпретации как исполняемого кода на языке программирования JavaScript.

АЛГОРИТМ, РАСПОЗНАВАНИЕ ГОЛОСА, МОБИЛЬНОЕ ПРИЛОЖЕНИЕ, ИНТЕЛЛЕКТУАЛЬНЫЕ СИСТЕМЫ, СЕРВЕР, КРОССПЛАТФОРМЕННОСТЬ.

ЗМІСТ

Стр.

Перелік умовних позначень, символів, одиниць, скорочень і термінів

БД	**-**	**база даних**
ЕОМ	**-**	**електронно-обчислювальна машина**
ІС	**-**	**інформаційна система**
ІТ	**-**	**інформаційні технології**
НДІ	**-**	**науково-дослідницький інститут**
ООП	**-**	**об’єктно-орієнтоване програмування**
ОС	**-**	**операційна система**
ПЗ	**-**	**програмне забезпечення**
СУБД	**-**	**система управління базами даних**
HTTP	**-**	**HyperText Transfer Protocol**
SSDP	**-**	**Simple Service Discovery Protocol**
TCP	**-**	**Transmission Control Protocol**
MVC	**-**	**Model View Controller**
API	**-**	**Application Program Interface**
DOM	**-**	**Document Object Model**
ORM	**-**	**Object-relational mapping**
IDE	**-**	**Integrated Development Environment**
MVVM	**-**	**Model-View-ViewModel**
HOC	**-**	**Higher Order Component**

ВСТУП

Актуальність роботи

Даний проект був розроблений для того, щоб вирішити декілька проблем, а саме, зробити можливим написання коду людям, що не мають змоги вручну друкувати код або зробити більш продуктивним його написання людям, що мають вади рухальної системи. Також дана програма може допомогти людям, що тільки почали вивчати програмування, адже проговорювання вголос сприяє кращому запам'ятовуванню. До того ж дана програма має перспективу, адже для багатьох людей саме диктування коду може бути просто набагато зручніше

Порівняння роботи з відомими розв’язаннями проблеми

При аналізі існуючих рішень було виявлено, що жодне з них не є цілком безкоштовним, до того ж жодне з них не має відкритого API та відкритого вихідного коду відповідно.

Мета і задачі дослідження

Метою бакалаврської роботи є розробка програмної системи що слугує для перетворення голосу в програмний код який можна виконувати за допомогою компілятора мови програмування JavaScript.

Досліджувана модель повинна інтерпрувати голос користувача в програмні команди за допомогою технології розпізнавання голосу, існує чіткий набір команд котрі може виконувати програма. Програма повинна чітко розуміти голос користувача та перекладати його команди в виконуваний потім код з мінімально допустимими похибками. Програма повинна мати зрозумілий та простий користувацький інтерфейс та працювати через мережу інтернет. Програма повинна представляти забезпечувати відповідний рівень комфортного користування, також програма повинна мати API для можливого її підключення та роботи зі сторонніми клієнтами.

Об’єктом дослідження є програма інтерпретації голосу в виконуваний код.

Предметом дослідження технологія розпізнавання голосу та подальшого його представлення в якості виконуваного коду

Методи дослідження

Для втілення мети в реальність використовуються різні технології розробки програмного забезпечення серед яких є технології розробки мобільних додатків для всіх сучасних мобільних платформ, технології розробки веб-додатків та технології розробки серверної частини системи.

Наукова новизна отриманих результатів

При розробці системи було використано новий підхід а саме коли мікрофон та безпосередньо модуль розпізнавання голосу знаходить на смартфоні користувача що спрощує користування системою та надає декілька нових можливостей для використання, при аналізі конкурентів виявлено що практично жоден з них не є цілком безкоштовним, до того ж жоден з них не має кодової бази з відкритим доступом - це відкриває безліч нових можливостей для підтримки та особистого використання системи.

Практичне значення одержаних результатів

Одержана реалізація алгоритма розпізнавання голосу та подальшої його інтерпретації як виконуваного коду на мові програмування JavScript.

Особистий внесок студента

Основним результатом є:

1. Запропонована автором модель системи де пристроєм для запису являється безпосередньо смартфон користувача;

2. Реалізація всіх модулів системи, приведення їх в робочий стан як функціонуюче програмне забезпечення, забезпечення потрібного рівня доступності для подальшої підтримки кодової бази, забезпечення простого та зрозумілого API для легкої замінюваності клієнту на будь який інший зручний для користувача клієнт.

Структура та обсяг роботи

Робота викладена на 39 сторінках друкованого тексту, який складається із вступу, трьох розділів, висновків, списку використаних джерел. Робота містить 8 рисунків.

РОЗДІЛ 1

СКЛАДОВІ СИСТЕМИ

1.1 Опис складових системи

Реалізація системи складається з трьох основних розділів (програм). Що комунікують між собою за допомогу інтернет протоколу HTTP. Деякі з цих складових відповідають за основну логіку та обчислення системи. Деякі являються швидкозамінюваними та практично не несуть в собі ключової логіки або обчислень необхідних для роботи системи, тобто відіграють роль суто відображення даних. Основні складові системи відображені нище:

1. Мобільний додаток для розпізнавання голосу - містить в собі модуль розпізнавання голосу та основну логіку обробки інформації, являється незамінним для роботи системи.
2. Сервер - шлюз для комунікації основної частини системи (тобто мобільного додатку) та клієнта, виконує логіку зберігання та передачі даних між іншими складовими системи, являється частково замінюваним (так як можливо замінити деякі його компоненти, наприклад базу даних).
3. Клієнтський додаток - веб сторінка для перегляду або експорту результатів обробки даних. Ця складова системи являється “виводом” даних для користувача, за допомогою неї користувач може переглянути та експортувати результати роботи програми (в подальшому редагувати або навіть працювати з ними в якості компіляції та запуску продиктованого коду).

1.2 Архітектура взаємодії макро-елементів системи

Як було сказано вище ці три елемента системи спілкуються та взаємодіють між собою за допомогою HTTP протоколу.

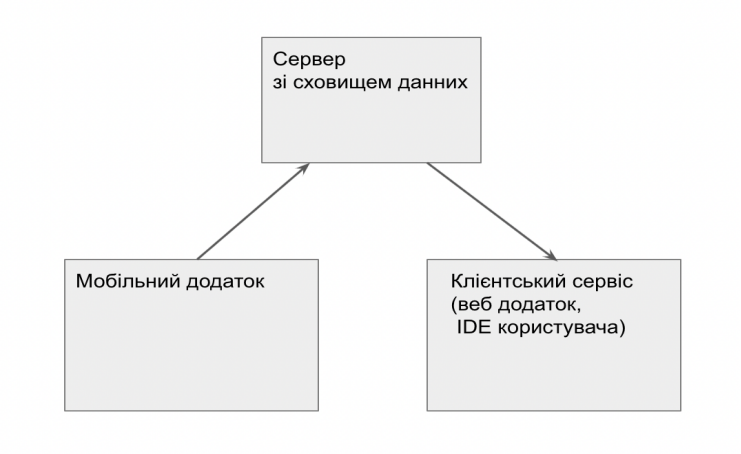

Рис. 1.1 Архітектура взаємодії макро-елементів системи

HTTP функціонує як протокол запиту-відповіді в обчислювальній моделі клієнт-сервер. Наприклад, веб-браузер може бути клієнтом, а програма, що працює на комп'ютері, що розміщує веб-сайт, може бути сервером. Клієнт надсилає повідомлення із запитом HTTP на сервер. Сервер, який надає такі ресурси, як файли HTML та інший вміст, або виконує інші функції від імені клієнта, повертає клієнту відповідне повідомлення. Відповідь містить інформацію про стан завершення запиту, а також може містити запитуваний вміст у його тілі повідомлення.

Веб-браузер є прикладом агента користувача (UA). Інші типи користувальницьких агентів включають програмне забезпечення для індексування, що використовується постачальниками послуг пошуку (веб-сканери), голосові браузери, мобільні програми та інше програмне забезпечення, яке здійснює доступ, використання або відображення веб-вмісту.

HTTP розроблений, щоб дозволити проміжним елементам мережі покращити або забезпечити зв'язок між клієнтами та серверами. Веб-сайти з високим трафіком часто отримують вигоду від серверів веб-кеш-пам’яті, які доставляють вміст від імені вихідних серверів для покращення часу відгуку. Веб-браузери кешують раніше доступні веб-ресурси та, якщо це

можливо, використовують їх повторно, щоб зменшити мережевий трафік. Проксі-сервери HTTP на межі приватної мережі можуть полегшити спілкування для клієнтів без глобальної маршрутизуваної адреси, передаючи повідомлення зовнішнім серверам.

HTTP - це протокол прикладного рівня, розроблений в рамках набору протоколів Інтернету. Його визначення передбачає базовий та надійний протокол транспортного рівня, і зазвичай використовується Протокол управління передачею (TCP). Однак HTTP можна адаптувати для використання ненадійних протоколів, таких як User Datagram Protocol (UDP), наприклад, в HTTPU і Simple Service Discovery Protocol (SSDP).

Ресурси HTTP ідентифікуються і розміщуються в мережі за допомогою уніфікованих локаторів ресурсів (URL-адрес), використовуючи схеми уніфікованих ідентифікаторів ресурсів (URI) http і https. Як визначено в RFC 3986, URI кодуються як гіперпосилання в документах HTML, щоб утворити взаємопов'язані гіпертекстові документи.

HTTP / 1.1 - це версія початкового HTTP (HTTP / 1.0). У HTTP / 1.0 для кожного запиту ресурсу встановлюється окреме підключення до одного і того ж сервера. HTTP / 1.1 може повторно використовувати з'єднання кілька разів для завантаження зображень, сценаріїв, таблиць стилів тощо після доставки сторінки. Тому комунікації HTTP / 1.1 мають менші затримки, оскільки встановлення TCP-з'єднань представляє значні накладні витрати.

РОЗДІЛ 2
МОБІЛЬНИЙ ДОДАТОК

2.1 Роль мобільного додатку

Основною частиною системи являється кросплатформений мобільний додаток для операційних систем IOS та Android, розроблений за допомогою фреймворку React Native. Додаток містить в собі основну логіку розпізнавання голосу та подальногї трансформації тексту в код для написання програм на мові JavaScript. React Native поєднує найкращі частини власної розробки з React, найкращою у своєму класі бібліотекою JavaScript для побудови користувальницьких інтерфейсів.

2.2 Технології мобільного додатку

Ви можете використовувати React Native сьогодні у своїх існуючих проектах Android та iOS або створити абсолютно нову програму з нуля.

Примітиви React відображаються в інтерфейсі платформи, тобто ваш додаток використовує ті самі API рідної платформи, що й інші програми.

Багато платформ, одна React. Створюйте специфічні для платформи версії компонентів, щоб одна кодова база могла обмінюватися кодом на різних платформах. За допомогою React Native одна команда може підтримувати дві платформи та використовувати спільну технологію - React.

React Native дозволяє створювати справді рідні програми та не загрожує досвіду користувачів. Він надає базовий набір вбудованих компонентних агностичних компонентів, таких як View, Text та Image, які безпосередньо відображаються на власних будівельних блоках інтерфейсу платформи.

Компоненти React обгортають наявний власний код та взаємодіють із власними API через декларативну парадигму інтерфейсу користувача React та JavaScript. Це дозволяє розробляти власні програми для цілих нових команд розробників і може дозволити існуючим рідним командам працювати набагато швидше.

У 2018 році React Native мав 2-е місце за кількістю авторів для будь-якого сховища в GitHub. Сьогодні React Native підтримується внесками приватних осіб та компаній по всьому світу, включаючи Callstack, Expo, Infinite Red, Microsoft та Software Mansion.

Наше співтовариство завжди постачає нові захоплюючі проекти та досліджує платформи за межами Android та iOS із такими репозиторіями, як React Native Windows, React Native macOS та React Native Web.

Принципи роботи React Native практично ідентичні React, за винятком того, що React Native не маніпулює DOM через віртуальний DOM. Він працює у фоновому процесі (який інтерпретує написаний розробниками

JavaScript) безпосередньо на кінцевому пристрої та взаємодіє з власною платформою через серіалізовані дані через асинхронний та пакетний міст.

Компоненти React обгортають наявний власний код та взаємодіють із власними API через декларативну парадигму інтерфейсу користувача React та JavaScript. Це дозволяє розробляти власні програми для цілих нових команд розробників і може дозволити існуючим рідним командам працювати набагато швидше.

Хоча стилі React Native мають подібний синтаксис до CSS, він не використовує HTML або CSS. Натомість повідомлення з потоку JavaScript використовуються для маніпулювання власними поданнями. React Native також дозволяє розробникам писати власний код такими мовами, як Java або Kotlin для Android та Objective-C або Swift для iOS, що робить його ще більш гнучким.

Для розпізнавання голосу в мобільному додатку було використану бібліотеку react-native-voice що працює завдяки вбудованим в IOS та Android модулям розпізнавання голосу.

Для операційної системи IOS використовується вбудований в операційну систему Speech Framevork. Підтримка диктування клавіатури використовує розпізнавання мови для перекладу аудіовмісту в текст. Цей фреймворк забезпечує подібну поведінку, за винятком того, що ви можете використовувати його без присутності клавіатури. Наприклад, ви можете використовувати розпізнавання мови, щоб розпізнавати словесні команди або обробляти текстові диктанти в інших частинах програми.

Ви можете виконувати розпізнавання мови багатьма мовами, але кожен об’єкт SFSpeechRecognizer працює на одній мові. Розпізнавання мовлення на пристрої доступне для деяких мов, але система також розраховує на сервери Apple для розпізнавання мови. Завжди припускайте, що для розпізнавання мови потрібне підключення до мережі.

Для операційної системи Android модуль розпізнавання голосу використовує движок Google Speech-to-Text. Запит синхронного

розпізнавання API перетворення тексту в текст є найпростішим методом для розпізнавання мовних аудіоданих. Мова в текст може обробляти до 1 хвилини мовних звукових даних, надісланих у синхронному запиті. Після обробки мови в текст і розпізнавання всього звуку, він повертає відповідь.

Синхронний запит блокує, що означає, що функція перетворення мови в текст повинна повернути відповідь перед обробкою наступного запиту. Функція перетворення мови в текст зазвичай обробляє звук швидше, ніж у реальному часі, обробляючи в середньому 30 секунд звуку за 15 секунд. У випадках низької якості звуку ваш запит на розпізнавання може зайняти значно більше часу.

Усі запити на синхронне розпізнавання API мовлення до тексту повинні містити поле конфігурації розпізнавання мови (типу RecognitionConfig). RecognitionConfig містить наступні підполя:

кодування - (обов’язково) визначає схему кодування поданого звуку (типу AudioEncoding). Якщо у вас є вибір з кодеків, віддайте перевагу кодуванню без втрат, такому як FLAC або LINEAR16 для найкращої роботи. (Докладніше див. У розділі Кодування звуку.) Поле кодування є необов’язковим для файлів FLAC та WAV, де кодування включено у заголовок файлу.
sampleRateHertz - (обов’язково) визначає частоту дискретизації (у герцах) поданого звуку. (Докладніше про частоту дискретизації див. Нижче Частота дискретизації.) Поле sampleRateHertz є необов’язковим для файлів FLAC та WAV, де частота дискретизації включена в заголовок файлу.
languageCode - (обов’язково) містить мову + регіон / регіон для використання для розпізнавання мови поданого аудіо. Код мови повинен бути ідентифікатором BCP-47. Зверніть увагу, що мовні коди зазвичай складаються з тегів основної мови та підтегів вторинної області для позначення діалектів (наприклад, 'en' для англійської та 'US' для США у наведеному вище прикладі.) (Список підтримуваних мов див. У розділі Підтримувані Мови.)
maxAlternatives - (необов’язково, за замовчуванням 1) вказує кількість альтернативних транскрипцій, які слід надати у відповіді. За замовчуванням API перетворення мови в текст забезпечує одну основну

транскрипцію. Якщо ви хочете оцінити різні альтернативи, встановіть maxAlternatives на більш високе значення. Зверніть увагу, що функція перетворення тексту в текст поверне альтернативи лише в тому випадку, якщо розпізнавач визначить альтернативи достатньо якісними; загалом, альтернативи більше підходять для запитів у режимі реального часу, що вимагають зворотного зв'язку з користувачем (наприклад, голосових команд), а тому більше підходять для запитів на розпізнавання потоку.
profanityFilter - (необов'язково) вказує, чи слід фільтрувати нецензурні слова або фрази. Слова, відфільтровані, будуть містити першу літеру та зірочки для решти символів (наприклад, f ***). Фільтр нецензурної лексики діє на окремих словах, він не виявляє образливих або образливих мов, що є фразою або комбінацією слів.
speechContext - (необов'язково) містить додаткову контекстну інформацію для обробки цього аудіо. Контекст містить таке підполе:
фрази - містить перелік слів і фраз, що дають підказки для завдання розпізнавання мови. Для отримання додаткової інформації див. Інформацію про контекст мовлення.
Аудіо подається до мови для перетворення тексту через параметр звуку типу RecognitionAudio. Звукове поле містить одне з наступних підполів:

вміст містить звук для оцінки, вбудований у запит. Для отримання додаткової інформації див. Розділ Вбудовування аудіовмісту нижче. Аудіо, передане безпосередньо в цьому полі, триває не більше 1 хвилини.
uri містить URI, що вказує на аудіовміст. Файл не можна стискати (наприклад, gzip). Наразі це поле повинно містити URI Google Cloud Storage (формату gs: // ім'я сегмента / шлях_до_аудіо_файлу). Див. Передавання аудіодовідки за допомогою URI нижче.)
Більше інформації про ці параметри запиту та відповіді наведено нижче.

Ставки вибірки
Ви вказуєте частоту дискретизації вашого аудіо в полі sampleRateHertz конфігурації запиту, і вона повинна відповідати частоті дискретизації відповідного аудіовмісту або потоку. Частота дискретизації від 8000 Гц до 48000 Гц підтримується функцією перетворення тексту в текст. Частоту дискретизації для файлу FLAC або WAV можна визначити із заголовка файлу, а не з поля sampleRateHertz.

Якщо у вас є вибір при кодуванні вихідного матеріалу, захопіть звук із частотою дискретизації 16000 Гц. Значення нижче цього можуть погіршити точність розпізнавання мови, а більш високі рівні не мають помітного впливу на якість розпізнавання мови.

Однак, якщо ваші аудіодані вже записані з існуючою частотою дискретизації, відмінною від 16000 Гц, не передискретизуйте звук до 16000 Гц. Наприклад, більшість застарілих аудіосигналів телефонії використовують частоту дискретизації 8000 Гц, що може дати менш точні результати. Якщо вам потрібно використовувати такий звук, надайте звук у Speech API із рідною частотою дискретизації.

2.3 Архітектура мобільного додатку

Основною архітектурою мобільного додатку для взаємодії з сервером є Flux паттерн, що приймає дані від сервера як REST API та обробляє дані за допомогою відповідної патерну бібліотеки redux та багатьох допоміжних для неї бібліотек як redux-saga, redux-persist та інші.

Flux - це архітектура додатків, яку Facebook використовує для створення веб-додатків на стороні клієнта. Він доповнює компонуючі компоненти подання React, використовуючи односпрямований потік даних. Це скоріше шаблон, а не формальний фреймворк, і ви можете негайно почати використовувати Flux без великої кількості нового коду.

Додатки Flux мають три основні частини: диспетчер, stores та подання (компоненти React). Їх не слід плутати з Model-View-Controller. Контролери існують у програмі Flux, але вони є поданнями контролерів - представленнями, які часто знаходяться у верхній частині ієрархії та отримують дані з магазинів і передають ці дані своїм дітям. Крім того, для підтримки семантичного API, що описує всі можливі зміни в програмі, використовуються творці дій - методи допоміжних засобів диспетчера. Може бути корисно розглядати їх як четверту частину циклу оновлення Flux.

Потік уникає MVC на користь односпрямованого потоку даних. Коли користувач взаємодіє з поданням React, подання розповсюджує дію через центральний диспетчер у різні сховища, що містять дані програми та бізнес-логіку, яка оновлює всі представлення даних, які зазнають впливу. Це особливо добре працює з декларативним стилем програмування React, який дозволяє магазину надсилати оновлення, не вказуючи, як переносити подання між станами.

Спочатку ми вирішили правильно обробляти похідні дані: наприклад, ми хотіли показати кількість непрочитаних повідомлень для потоків повідомлень, тоді як інший вигляд показав список потоків, з непрочитаними. З MVC це було важко впоратись - позначення одного потоку як прочитаного оновить модель потоку, а потім також потрібно

оновити модель непрочитаного підрахунку. Ці залежності та каскадні оновлення часто трапляються у великому додатку MVC, що призводить до заплутаного переплетення потоку даних та непередбачуваних результатів.

Контроль інвертується з магазинами: магазини приймають оновлення та узгоджують їх за необхідністю, а не залежно від чогось зовнішнього, щоб послідовно оновлювати свої дані. Ніщо за межами магазину не має жодного уявлення про те, як він управляє даними для свого домену, допомагаючи чітко відокремити проблеми. Магазини не мають прямих методів встановлення, таких як setAsRead (), а натомість мають лише єдиний спосіб потрапляння нових даних у свій автономний світ - зворотний виклик, який вони реєструють у диспетчера.

Дані в додатку Flux рухаються в одному напрямку:

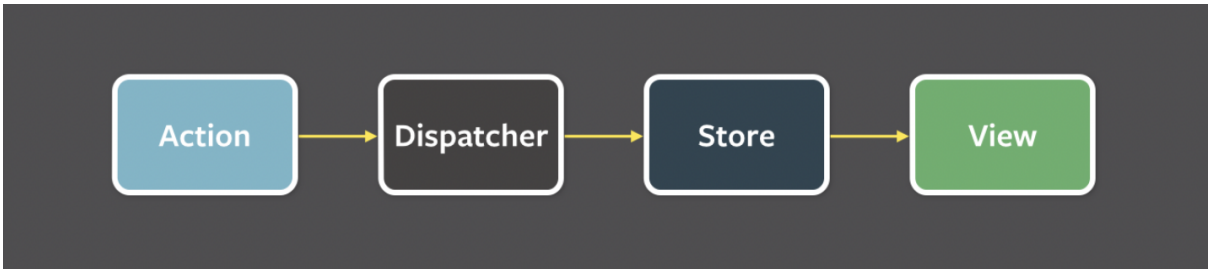

Рис. 2.1 Flux паттерн (без Action)

Односпрямований потік даних є центральним для шаблону Flux, і наведена діаграма повинна бути основною розумовою моделлю для програміста Flux. Диспетчер, сховища та подання є незалежними вузлами з різними входами та виходами. Дії - це прості об'єкти, що містять нові дані та властивість ідентифікуючого типу.

Погляди можуть спричинити розповсюдження нової дії через систему у відповідь на взаємодію користувачів:

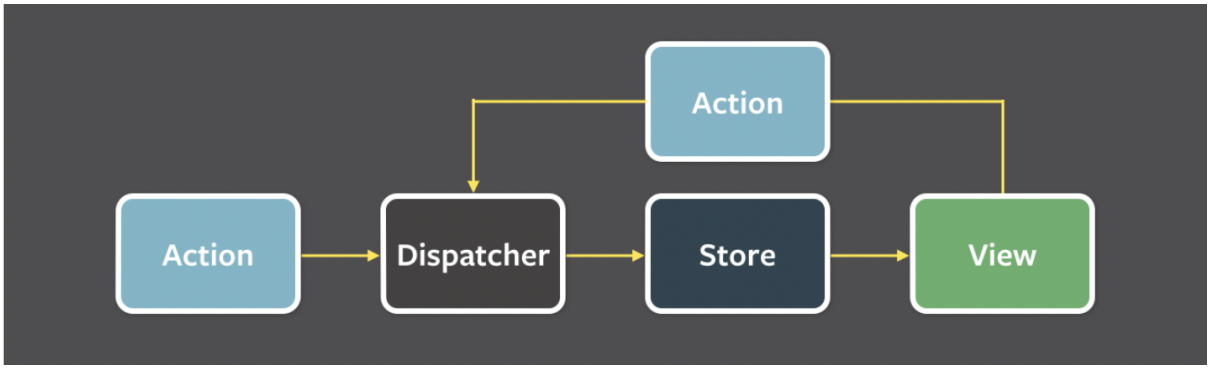

Рис. 2.2 Flux паттерн (з Action)

Усі дані проходять через диспетчер як центральний вузол. Дії надаються диспетчеру за допомогою методу творця дій і найчастіше походять від взаємодії користувача з представленнями. Потім диспетчер викликає зворотні виклики, які магазини зареєстрували в ньому, надсилаючи дії у всі магазини. У межах зареєстрованих зворотних викликів магазини реагують на будь-які дії, що стосуються стану, який вони підтримують. Потім магазини видають подію зміни, щоб попереджати подання контролера про те, що відбулася зміна рівня даних. Перегляди контролерів прослуховують ці події та отримують дані з магазинів у обробнику подій. Погляди контролера викликають власний метод setState (), викликаючи повторний візуалізацію себе та всіх своїх нащадків у дереві компонентів.

Ця структура дозволяє легко міркувати про наш додаток таким чином, що нагадує функціональне реактивне програмування, а точніше програмування потоку даних або програмування на основі потоку, коли дані протікають через додаток в одному напрямку - немає двох- спосіб прив'язки. Стан програми підтримується лише в магазинах, що дозволяє різним частинам програми залишатися сильно розв’язаними. Там, де залежності все ж виникають між магазинами, вони зберігаються в суворій ієрархії, синхронними оновленнями керує диспетчер.

Ми виявили, що двосторонні прив'язки даних призводять до каскадного оновлення, коли зміна одного об'єкта призводить до зміни іншого, що також може спричинити більше оновлень. У міру зростання додатків ці каскадні оновлення дуже ускладнили прогнозування того, що зміниться в результаті взаємодії одного користувача. Коли оновлення можуть

змінювати дані лише за один раунд, система в цілому стає більш передбачуваною.

Давайте розглянемо різні частини Flux зблизька. Гарне місце для початку - диспетчер.

Диспетчер - це центральний центр, який управляє всім потоком даних у програмі Flux. По суті, це реєстр зворотних дзвінків у магазини і не має власного реального інтелекту - це простий механізм розподілу дій між магазинами. Кожен магазин реєструється і забезпечує зворотний дзвінок. Коли творець дії надає диспетчеру нову дію, усі магазини у програмі отримують дію через зворотні виклики в реєстрі.

У міру зростання додатка диспетчер стає більш важливим, оскільки його можна використовувати для управління залежностями між сховищами, викликаючи зареєстровані зворотні виклики в певному порядку. Магазини можуть декларативно чекати, поки інші магазини закінчать оновлення, а потім відповідно оновлюватись.

Той самий диспетчер, який Facebook використовує у виробництві, доступний через npm, Bower або GitHub.

Stores містять стан програми та логіку. Їх роль дещо схожа на модель у традиційному MVC, але вони керують станом багатьох об'єктів - вони не представляють жодного запису даних, як це роблять моделі ORM. Вони не є однаковими з колекціями Backbone. Більше, ніж просто управління колекцією об’єктів у стилі ORM, магазини управляють станом програми для певного домену в програмі.

Наприклад, у редакторі відеороликів Lookback Video Editor використовувався TimeStore, який відстежував позицію часу відтворення та стан відтворення. З іншого боку, той же додаток ImageStore відстежував колекцію зображень. TodoStore у нашому прикладі TodoMVC схожий тим, що керує колекцією завдань. Store демонструє характеристики як колекції моделей, так і одномісної моделі логічного домену.

Як зазначалося вище, магазин реєструється у диспетчера та надає йому зворотний дзвінок. Цей зворотний виклик отримує дію як параметр. У межах зареєстрованого зворотного виклику магазину для витлумачення дії та надання відповідних гачків внутрішнім методам магазину використовується оператор switch, який базується на типі дії. Це дозволяє дію призвести до оновлення стану магазину через диспетчера. Після оновлення магазинів вони транслюють подію, в якій заявляють, що їхній стан змінився, тому подання можуть запитати новий стан та оновити себе.

React надає види перегляду, що складаються та вільно переглядаються, потрібні для рівня перегляду. Близько до верху вкладеної ієрархії подань особливий вид подання прослуховує події, які транслюються магазинами, від яких це залежить. Ми називаємо це видом-контролером, оскільки він забезпечує код клею для отримання даних із сховищ і передачі цих даних по ланцюжку своїх нащадків. У нас може бути один із цих подань контролера, який регулює будь-який значний розділ сторінки.

Коли він отримує подію з магазину, він спочатку запитує нові дані, які йому потрібні, за допомогою методів загального отримання магазинів. Потім він викликає власні методи setState () або forceUpdate (), викликаючи запуск його методу render () та методу render () усіх його нащадків.

Ми часто передаємо весь стан магазину по ланцюжку переглядів в одному об’єкті, дозволяючи різним нащадкам використовувати те, що їм потрібно. На додаток до збереження поведінки, подібної контролеру, у верхній частині ієрархії, і, таким чином, підтримуючи наші подання нащадків якомога функціональнішими чистими, передача всього стану сховища в один об’єкт також має наслідком зменшення кількості реквізитів нам потрібно керувати.

Іноді нам може знадобитися додати додаткові подання контролера глибше в ієрархії, щоб компоненти були простими. Це може допомогти нам краще інкапсулювати розділ ієрархії, пов'язаний з певним доменом даних. Однак пам’ятайте, що погляди контролерів, глибші в ієрархії, можуть порушити особливий потік даних, вводячи нову, потенційно суперечливу точку входу для потоку даних. Приймаючи рішення щодо додавання глибокого подання

контролера, збалансуйте приріст простих компонентів проти складності кількох оновлень даних, що надходять в ієрархію в різних точках. Ці багаторазові оновлення даних можуть призвести до непарних ефектів, оскільки метод візуалізації React неодноразово викликається оновленнями з різних подань контролера, що потенційно збільшує труднощі налагодження.

Диспетчер пропонує метод, який дозволяє нам запускати відправлення до магазинів та включати корисний набір даних, який ми називаємо дією. Створення дії може бути перетворено на семантичний допоміжний метод, який надсилає дію диспетчеру. Наприклад, ми можемо захотіти змінити текст об'єкта справ у програмі списку справ. Ми створили б дію з підписом функції, як updateText (todoId, newText) у нашому модулі TodoActions. Цей метод може бути використаний в обробниках подій наших поглядів, тому ми можемо викликати його у відповідь на взаємодію користувача. Цей метод створення дії також додає тип до дії, так що коли інтерпретується дія в магазині, він може відповісти належним чином. У нашому прикладі цей тип може бути названий приблизно так: TODO_UPDATE_TEXT.

Дії також можуть надходити з інших місць, наприклад із сервера. Це трапляється, наприклад, під час ініціалізації даних. Це також може трапитися, коли сервер повертає код помилки або коли сервер має оновлення для надання додатку.

2.4 Алгоритм розпізнавання голосу

Прихована марковська модель при розпізнаванні мови розставляє фонеми в потрібному порядку, використовуючи статистичні ймовірності. Для цього використовується три різні шари.

У першому шарі модель повинна перевірити рівень звуку, і ймовірність того, що виявлена нею фонема є правильною. Як зазначалося раніше, варіація фонем залежить від кількох різних факторів, таких як акценти, каденція, емоції, стать тощо.

У другому шарі модель перевіряє фонеми, які знаходяться поруч, і ймовірність того, що вони повинні бути поруч. Наприклад, якщо у вас звук "st", то, швидше за все, буде такий голосний, як "a". Менш імовірно або навіть неможливо, щоб фонема “n” наслідувала “st” фонему - принаймні англійською мовою.

Нарешті, у третьому шарі модель перевіряє рівень слова. Тобто, чи мають значення слова поруч одне з одним. Це робиться шляхом перевірки ймовірності того, що вони повинні бути поруч. Наприклад, він перевірить, чи у фразі занадто багато чи мало дієслів. Він також перевіряє прислівники, підмети та кілька інших складових речення.

Модель перевіряє та перевіряє всі ймовірності того, що найімовірніший текст було вимовлено. Ви можете переконатися в цьому в режимі реального часу, коли диктуєте помічнику телефону. Ви можете помітити, що слова на початку вашої фрази починають змінюватися, коли система намагається зрозуміти, що ви говорите.

Ця модель чудово підходить для послідовного характеру мовлення. Однак він не є гнучким. Крім того, існує така велика різноманітність фонем та їх потенційних комбінацій, що йому ще потрібно пройти довгий шлях, перш ніж його можна буде вважати ідеальним.

Внутрішня робота штучної нейронної мережі базується на тому, як працює людський мозок. Нейронна мережа - це мережа вузлів, які будуються за

допомогою вхідного шару, прихованого шару, що складається з безлічі різних шарів, і вихідного шару.

Всі з'єднання мають різну вагу, і лише інформація, яка досягла певного порогу, надсилається на наступний вузол. Якщо вузол повинен вибрати між двома входами, він вибирає вхід вузла, з яким він має найсильніший зв’язок. У деяких системах він також може приймати обидва входи і придумати коефіцієнт.

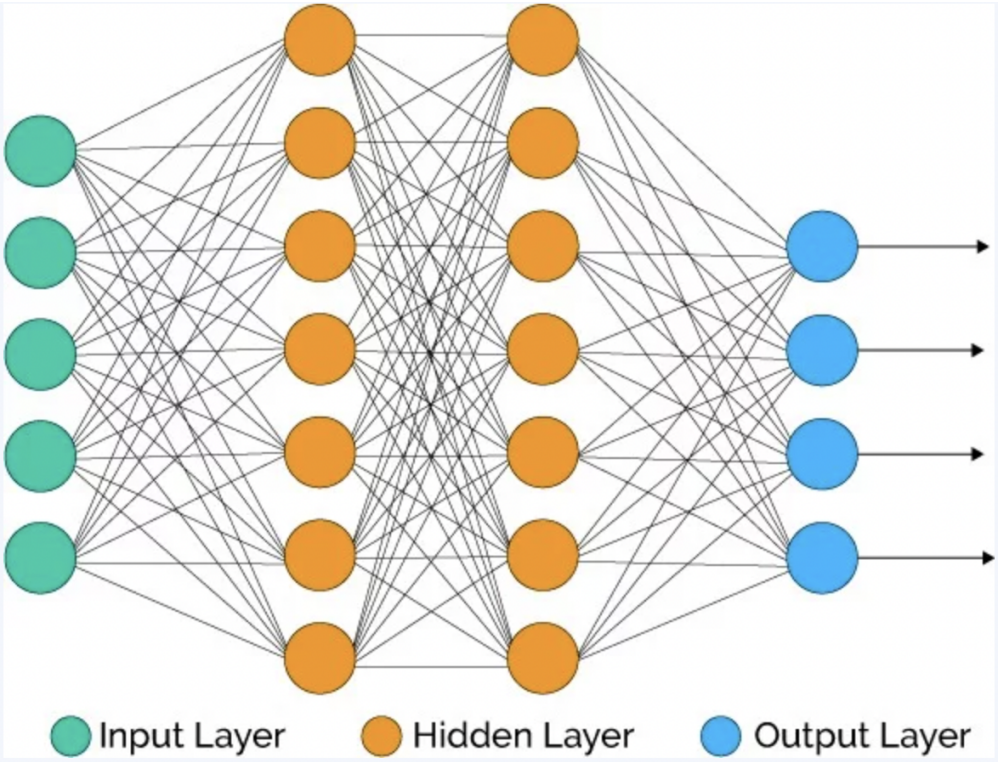

Рис. 1.2 Модель нейронної сеті

Перевага нейронних мереж полягає в тому, що вони гнучкі і тому можуть змінюватися з часом. Це означає, що нейронну мережу потрібно тренувати, оскільки всі різні з'єднання спочатку мають однакову вагу. Вхідні дані подаються в нейронну мережу, і бажаний вихідний результат вказується. Потім нейронна мережа робить своє і виходить з певним результатом, який не є таким, як бажаний результат, оскільки потрібна додаткова підготовка. Ця різниця є помилкою. Нейронна мережа розуміє, що є помилка, і тому починає адаптуватися, щоб зменшити помилку. Щоб нейронна мережа постійно вдосконалювалась та усувала помилку, вона потребує значного вкладу.

Ця вимога до великої кількості вхідних даних до того, як вона стане досконалою, є одним із мінусів нейронних мереж при розпізнаванні мови. Інший мінус полягає в тому, що він погано підходить для послідовного характеру мовлення, але плюсом є гнучкість, яка також охоплює різновиди фонем. Тому він може виявити унікальність акцентів, емоцій, віку, статі тощо.

Слабкі сторони нейронних мереж пом'якшуються сильними сторонами прихованої моделі Маркова і навпаки. Ось чому прихована модель Маркова та нейронні мережі використовуються разом у програмах розпізнавання мови.

РОЗДІЛ 3

СЕРВЕР (REST API)

3.1 Роль серверу

Сервер в даній системі відіграє роль зберігання та транспортування даних, він отримує опрацьовані дані з мобільного додатку та з його допомогою доступ до цих даних може мати будь який клієнт системи (будь то веб додаток чи клієнтська IDE). Сервер представлено в вигляді класичної REST API, хоча в подальшому його може бути модифікувати для використання web sockets що робить комунікацію між ним та клієнтами більш зручною та швидкою для користувача.

3.2 Архітектура серверу

Сервер написано на мові програмування JavaScript за допомогою досить популярного фреймворка express. Основна складова його архітектури це класичний MVC паттерн, однак в якості представлення використовується окремий клієнт та сервер набуває вигляду REST API.

MVC розшифровується як Model - View - Controller. Це парадигма. Це дозволяє кодеру виділяти різні компоненти програми та легше їх оновлювати. MVC дозволяє програмісту створювати бар'єри для організації коду, що дозволяє програмісту розділити функціональність.

Модель - це те, чим є ваша програма (а не як вона відображається)
Перегляд - службовці контролера, загальні об’єкти багаторазового інтерфейсу, які контролер використовує для виконання своєї роботи. Зрештою, як модель відображається на екрані. Більшість об’єктів у поданні (кнопки, повзунки, текст)
Контролер - як ваша модель представляється користувачеві (UI Logic); Платформа специфічна
В кінцевому підсумку, архітектура MVC - це управління комунікацією між «таборами».
Контролер має необмежений зв’язок
Модель і Погляд ніколи не повинні говорити одне з одним.
Три (3) основні способи перегляду подання з контролером
(1) Цільова дія - Контролер передає дію виду, від якого він зацікавлений слухати (тобто кнопку, повзунок тощо) Коли користувальницький інтерфейс торкається, дія надсилається цілі, яка розміщується на Контролер.
(2) Делегування - контролер надсилає повідомлення і стає делегатом подання із заявою
Делегат встановлюється за допомогою протоколу (тобто він сліпий для класу).
(3) Протокол джерела даних -
Завданням контролера є інтерпретація даних для подання.
В одній програмі може бути багато MVC, кожна з яких взаємодіє між собою.

Як випливає з назви, він складається з трьох основних частин. Традиційний шаблон дизайну програмного забезпечення працює за шаблоном "Вхід - Процес - Вивід", тоді як MVC працює як "Контролер-Модель - Вид". З появою моделі MVC створення додатків враховує різні аспекти окремо. Ці аспекти заявки:

- Логіка інтерфейсу користувача
- Логіка введення
- Бізнес-логіка

Але між цими аспектами або елементами існує вільна зв'язок. Згідно з цією моделлю, кожен елемент повинен існувати в додатку, але не тісно пов’язаний або взаємопов’язаний. Логіка користувальницького інтерфейсу стосується подання або інтерфейсу програми. Логіка введення має справу з контролером. Нарешті, бізнес-логіка має справу з моделлю програми. Цей нещільно зв'язаний елемент допомагає розробникам вирішувати ускладнення при розробці будь-якого додатка. Це допомагає користувачам одночасно фокусуватись на одному конкретному елементі реалізації. Візьміть сценарій, коли ви працюєте з бізнесом; Ви можете вирішити це і зосередитись на побудові бізнес-логіки, не залежачи від логіки подання.

MVC був представлений доктором Тригве Ренскаугом мовою програмування Smalltalk-76, коли він відвідав Дослідницький центр Xerox Palo Alto (PARC) у середині 1970 року. Пізніше реалізація стала популярною в інших версіях Small-Talk. Потім, у 1988 році, статті в "Журналі об'єктних технологій (JOT)" представляють загальну картину MVC як добре прийняту концепцію.

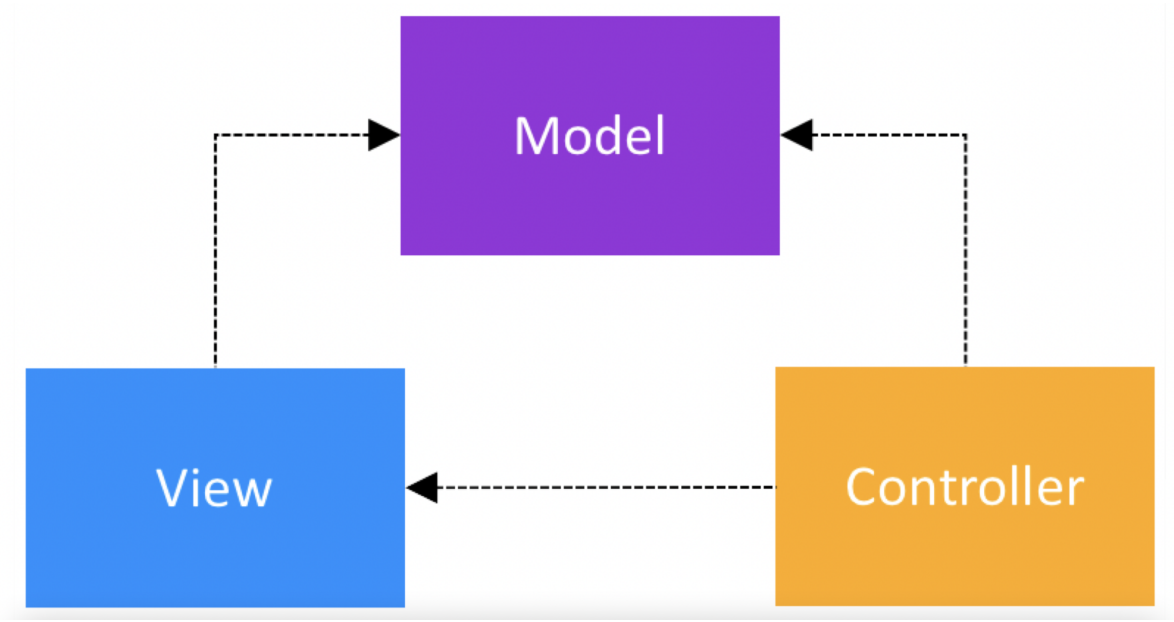

Рис. 2.3 MVC паттерн

REST API (також відомий як RESTful API) - це інтерфейс програмування програм (API або веб-API), який відповідає обмеженням архітектурного стилю REST і дозволяє взаємодіяти з веб-службами RESTful. REST позначає репрезентативну передачу стану і був створений інформатиком Роєм Філдінгом.

API - це набір визначень та протоколів для побудови та інтеграції прикладного програмного забезпечення. Іноді це називають контрактом між постачальником інформації та користувачем інформації - встановлення контенту, що вимагається від споживача (дзвінок), та контенту, який вимагає виробник (відповідь). Наприклад, конструкція API для метеорологічної служби може визначати, що користувач надає поштовий індекс, а виробник відповідає двоскладовою відповіддю, перша - висока температура, а друга - низька.

Іншими словами, якщо ви хочете взаємодіяти з комп'ютером або системою для отримання інформації або виконання функції, API допомагає вам повідомляти, що ви хочете, цій системі, щоб вона могла зрозуміти та виконати запит.

Ви можете уявити API як посередника між користувачами або клієнтами та ресурсами або веб-службами, які вони хочуть отримати. Це також спосіб

для організації обмінюватися ресурсами та інформацією, зберігаючи при цьому безпеку, контроль та автентифікацію - визначаючи, хто до чого отримує доступ.

Ще однією перевагою API є те, що вам не потрібно знати особливості кешування - як отримується ваш ресурс або звідки він береться.

REST - це набір архітектурних обмежень, а не протокол або стандарт. Розробники API можуть реалізувати REST різними способами.

Коли запит клієнта робиться через RESTful API, він передає подання стану ресурсу запитувачу або кінцевій точці. Ця інформація, або подання, передається в одному з декількох форматів через HTTP: JSON (нотація об'єктів Javascript), HTML, XLT, Python, PHP або звичайний текст. JSON - найпопулярніша мова програмування, оскільки, незважаючи на свою назву, вона є мовно-агностичною, а також доступною для читання як людьми, так і машинами.

Ще щось, про що слід пам’ятати: заголовки та параметри також важливі для методів HTTP запиту HTTP-запиту RESTful API, оскільки вони містять важливу інформацію про ідентифікатори метаданих запиту, авторизації, єдиного ідентифікатора ресурсу (URI), кешування, файлів cookie та більше. Існують заголовки запитів та заголовки відповідей, кожен із яких має власну інформацію про з'єднання HTTP та коди стану.

РОЗДІЛ 4
КЛІЄНТСЬКИЙ ВЕБ-ДОДАТОК

4.1 Роль веб-додатку

В даній системі веб додаток відіграє найбільш не значну роль та може бути легко замінений десктоп додатком або плагіном для IDE користувача, за допомогою веб-додатку користувач отримує доступ до вихідного коду та може його скомпілювати та виконати в браузері. Веб додаток спілкується з сервером за допомогою API що останній предоставляє. Веб додаток написаний на мові програмування JavaScript та за допомогою найбільш популярного в цьому середовищі фреймворка React.

4.2 Архітектура веб-додатку

Зараз ReactJs широко використовується бібліотекою View у всьому світі. Тому стає дуже важливим застосовувати найкращі архітектурні практики, щоб зробити код багаторазовим, ремонтопридатним, а також це допомогло б поліпшити читабельність коду.
Однак, оскільки React опікується лише шаром перегляду програми, він не застосовує жодної конкретної архітектури (наприклад, MVC або MVVM). Це може ускладнити підтримку вашої кодової бази в міру зростання проекту React.
У цій статті ви дізнаєтеся, які найкращі архітектурні практики слід застосовувати до коду під час використання ReactJS для створення багаторазової бібліотеки.

Спочатку стиль та код наших компонентів були розділені. Усі стилі жили у спільному файлі CSS (ми використовуємо SCSS для попередньої обробки). Фактичний компонент (у даному випадку FilterSlider) був відокремлений від стилів.

Для розробника React дуже важливо знати, як використовувати компонент, PureComponent та функціональний компонент без стану.
По-перше, давайте перевіряємо функціональний компонент без стану.

Функціональні компоненти без громадянства - один із найпоширеніших типів компонентів у вашому арсеналі. Вони надають нам гарний і стислий спосіб створення компонентів, які не використовують будь-який стан, посилання або методи життєвого циклу.
Ідея функціонального компонента без стану полягає в тому, що він не має стану і є просто функцією. Отже, чудовим у цьому є те, що ви визначаєте свій компонент як постійну функцію, яка повертає деякі дані.
Простими словами, функціональні компоненти без стану - це лише функції, які повертають JSX.

Остання версія React принесла нам гачки React, які дозволять нам констатувати, ефекти та посилання на функціональні компоненти без необхідності перетворювати їх на компоненти класу.

Зазвичай, коли компонент отримує в ньому новий реквізит, React повторно відтворює цей компонент. Але іноді компонент отримує новий реквізит, який насправді не змінився, але React все одно ініціює повторний рендер.
Використання PureComponent допоможе вам запобігти цьому марному рендерингу. Наприклад, якщо властивість є рядком або логічним значенням, і воно змінюється, PureComponent збирається це розпізнати, але якщо властивість в об'єкті змінюється, PureComponent не збирається викликати повторне відображення.
То як ви дізнаєтесь, коли React ініціює непотрібний рендеринг? Ви можете перевірити цей дивовижний пакет React під назвою “Why Did You Update”. Цей пакет сповістить вас у консолі, коли відбудеться потенційно непотрібне рендеринг.

Іноді доводиться гарантувати, що компонент React відображається лише тоді, коли користувач увійшов у вашу програму. Спочатку ви будете робити деякі перевірки стану розумності у своєму методі візуалізації, поки не виявите, що багато повторюєтеся. Під час вашої місії ВИСУХИТИ цей код, ви рано чи пізно знайдете компоненти вищого порядку, які допоможуть вам витягти та абстрагувати певні проблеми компонента. З точки зору розробки програмного забезпечення, компоненти вищого порядку є формою шаблону декоратора. Компонент вищого порядку (HOC) - це в основному функція, яка отримує компонент React як параметр і повертає інший компонент React. Погляньте на наступний приклад:

```
export default function requiresAuth(WrappedComponent) {
  class AuthenticatedComponent extends Component {
    static propTypes = {
      user: PropTypes.object,
      dispatch: PropTypes.func.isRequired
    };

    componentDidMount() {
      this._checkAndRedirect();
    }

    componentDidUpdate() {
      this._checkAndRedirect();
    }

    _checkAndRedirect() {
      const { dispatch, user } = this.props;

      if (!user) {
        dispatch(push('/signin'));
      }
    }

    render() {
      return (
        <div className="authenticated">
          { this.props.user ? <WrappedComponent {...this.props} /> :
null }
        </div>
      );
    }
  }
```

Рис. 2.4 Приклад компоненту вищого порядку

Функція requireAuth отримує компонент (WrappedComponent) як параметр, який буде прикрашений потрібною функціональністю. Усередині цієї функції клас AuthenticatedComponent відображає цей компонент і додає функціональність, щоб перевірити, чи присутній користувач, інакше перенаправляючи на сторінку входу. Нарешті, цей компонент підключається до магазину Redux і повертається. Redux корисний у цьому прикладі, але не є абсолютно необхідним.

Створення рядка таблиці, що складається, не є дуже простим завданням. Як відображається кнопка згортання? Як ми покажемо дітей, коли стіл не згорнутий? Я знаю, що з JSX 2.0 все стало набагато простіше, оскільки ви можете повернути масив замість одного тегу, але я розкладу цей приклад,

оскільки це ілюструє хороший варіант використання функції як дочірній шаблон. Уявіть наступну таблицю:

```
import React, { Component } from "react";

export default class Table extends Component {
  render() {
    return (
      <table>
        <thead>
          <tr>
            <th>Just a table</th>
          </tr>
        </thead>
        {this.props.children}
      </table>
    );
  }
}
```

Рис. 2.5 Приклад використання функції як дочірній шаблон 1

Ви просто передаєте функцію в дитячому віці, яка отримує виклик під час відтворення функції компонента. Можливо, ви також бачили цю техніку, яку називають «зворотним викликом зворотного виклику», або в особливих випадках, як «візуалізаційною опорою».

Як бачите, існує властивість render, яка є функцією, що викликається під час процесу візуалізації. Функція, що викликається всередині неї, отримує повний стан як параметр і повертає JSX. Тепер подивіться на наступне використання:

```
<Fetch
  url="https://api.github.com/users/imgly/repos"
  render={({ data, isLoading }) => (
    <div>
      <h2>img.ly repos</h2>
      {isLoading && <h2>Loading...</h2>}

      <ul>
        {data.length > 0 && data.map(repo => (
          <li key={repo.id}>
            {repo.full_name}
          </li>
        ))}
      </ul>
    </div>
  )} />
```

Рис. 2.6 Приклад використання функції як дочірній шаблон 2

Як бачите, параметри даних і isLoading деструктуруються з об'єкта стану і можуть використовуватись для управління реакцією JSX. У цьому випадку, залишити обіцянку не виконано, відображається загальний виклад "Завантаження". Через вас, які частини стану переходять до візуалізації, і як ви використовуєте їх у своєму інтерфейсі користувачів. Загалом, це дуже потужний механізм вилучення загальної поведінки. Описаний вище шаблон "Функція як діти" в основному є тим самим шаблоном, де влада є дітьми.

У використанні методів стану чи життєвого циклу в компонентах немає нічого “поганого”. Як і будь-яку потужну функцію, їх слід використовувати в помірних кількостях, але ми не маємо наміру їх видаляти. Навпаки, ми вважаємо, що вони є невід’ємними частинами того, що робить React корисним. Ми можемо дозволити більш функціональні моделі в майбутньому, але як місцевий стан, так і методи життєвого циклу будуть частиною цієї моделі.

Компоненти часто описуються як "просто функції", але, на наш погляд, вони повинні бути більшими, ніж корисні. У React компоненти описують будь-яку складну поведінку, і це включає візуалізацію, життєвий цикл та стан. Деякі зовнішні бібліотеки, такі як Relay, доповнюють компоненти з іншими обов'язками, такими як опис залежностей даних. Цілком можливо, що ці ідеї також можуть повернутись у React в якійсь формі.

ВИСНОВКИ

В ході виконання роботі була створена система когда дозволяє писати програмний код (програмне забезпечення) на мові програмування JavaScript за допомогою голосу використовуючи мобільний телефон та веб клієнт (в подальшому будь який інший можливий клієнт).

Було вивчено та проаналізовано різні модулі для розпізнавання голосу та існуючі програми-конкуренти. Після аналізу та на основі отриманих висновків було написано три основні модулі системи.

- Мобільний додаток
- Сервер-шлюз
- Веб-додаток клієнт

Даний проект був розроблений для того, щоб вирішити декілька проблем, а саме, зробити можливим написання коду людям, що не мають змоги вручну друкувати код або зробити більш продуктивним його написання людям, що мають вади рухальної системи.

До того ж дана програма має перспективу, адже для багатьох людей саме диктування коду може бути просто набагато зручніше.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Mozila developer - https://developer.mozilla.org/
2. React Native docs - https://reactnative.dev/
3. Medium - https://medium.com/
4. Flux Architecture - https://facebook.github.io/flux