**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ**

**НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ**

**Кафедра комп’ютеризованих систем управління**

ДОПУСТИТИ ДО ЗАХИСТУ

Завідувач кафедри

Литвиненко О.Є

“\_\_\_\_”\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_2020 р.

**ДИПЛОМНА РОБОТА**

**(пояснювальна записка)**

**випускника освітньоГО СТУПЕНЯ «магістр»**

**Тема:** Програмний модуль створення кросплатформених додатків на основі технологій *React/React Native*

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Виконавець:** Мурий Є.В

**Керівник:** Кучерява О. М.

**Нормоконтролер:** Тупота Є.В.

**НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ**

Факультет Кібербезпеки, комп’ютерної та програмної інженерії

Кафедра комп’ютеризованих систем управління

Спеціальність 123 «Комп’ютерна інженерія»

(шифр, найменування)

Освітньо-професійна програма «Системне програмування»

ЗАТВЕРДЖУЮ

Завідувач кафедри

Литвиненко О.Є

“\_\_\_\_”\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_2020 р.

**ЗАВДАННЯ**

**на виконання дипломної роботи**

Мурого Євгенія Владиславовича

(прізвище, ім'я, по батькові випускника в родовому відмінку)

**1. Тема дипломної роботи:**  «Програмний модуль створення кросплатформених додатків на основі технологій *React/React Native*»

затверджена наказом ректора від «27» серпня 2019 р. № 1808/ст

**2. Термін виконання роботи:** з 14 жовтня 2019 р. по 09 лютого 2020 р.

**3. Вхідні дані до роботи:** механізм логування дій користувачів; Використання об’єктної архітектури для розробки модуля створення кросплатформених додатків

**4. Зміст пояснювальної записки:**

1) Огляд основних методів створення кросплатформених додатків

2) Використані технології при розробці модуля

3) Розробка модуля створення кросплатформених додатків

4) Використання додатку клієнтом

**5. Перелік обов’язкового графічного (ілюстрованого) матеріалу:**

1) Взаємодія з *API*

2) Архітектура *React Native*

3) Головне вікно *web*-версії

4) Головний екран *native* версії

**6. Календарний план-графік**

|  |  |  |  |
| --- | --- | --- | --- |
| №  пор. | Завдання | Термін  виконання | Відмітка про  виконання |
| 1 | Провести аналіз літератури за темою дипломної роботи | 14.10.2019 – 29.10.2019 |  |
| 2 | Написати розділ «Огляд основних методів створення кросплатформених додатків» | 30.10.2019 – 12.11.2019 |  |
| 3 | Провести аналіз необхідних компонентів для реалізації кросплатформного додатку. | 13.11.2019 – 26.11.2019 |  |
| 4 | Написати розділ «Розробка модуля створення кросплатформених додатків» | 27.11.2019 – 03.12.2019 |  |
| 5 | Написати розділ «Розробка кросплатформного додатку» | 08.01.2020 – 14.01.2020 |  |
| 6 | Написати розділ «Використання додатку клієнтом» | 15.01.2020 – 21.01.2020 |  |
| 7 | Оформити пояснювальну записку | 22.01.2020 – 27.01.2020 |  |
| 8 | Оформити графічний (ілюстрований) матеріал | 28.01.2020 – 30.01.2020 |  |

**7. Дата видачі завдання:** “ 14 “ жовтня 2019 р.

Керівник дипломної роботи Кучерява О. М.

(підпис керівника) (П.І.Б.)

Завдання прийняв до виконання Мурий Є. В.

(підпис студента) (П.І.Б.)

РЕФЕРАТ

Пояснювальна записка до дипломної роботи «Програмний модуль створення кросплатформених додатків на основі технологій *React/React Native*»: 89 сторінок, 12 рисунків, 30 використаних джерел, 1 додаток.

МОДУЛЬ, КРОСПЛАТФОРМЕНИЙ, МОБІЛЬНИЙ ДОДАТОК, *REACT*, *REACT NATIVE*, *API.*

Об’єкт дипломної роботи – кросплатформені додатки.

Предмет дипломної роботи – модуль створення кросплатформених додатків.

Мета дипломної роботи – розробка модулю для створення кросплатформених додатків.

Методи дослідження – мова програмування *JavaScript,* середовище розробки *WebStorm, API MovieDB,* фреймворки *React* та *React Native.*

В дипломній роботі спроектовано та розроблено програмний модуль для створення кросплатформених додатків з використанням технологій *React* та *React* *Native*, який дозволяє створювати мобільні, та веб-додатки, використовуючи спільну кодову базу. На базі модулю було спроектовано кросплатформений додаток для роботи з *API MovieDB****.***

ЗМІСТ

[ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ ТА СКОРОЧЕНЬ 7](#_Toc2032990489)

[ВСТУП 8](#_Toc1055692575)

[РОЗДІЛ 1 ОГЛЯД ОСНОВНИХ МЕТОДІВ СТВОРЕННЯ КРОСПЛАТФОРМЕНИХ ДОДАТКІВ 13](#_Toc1501612239)

[1.1. Кросплатформеність 13](#_Toc927653849)

[1.2. Основні кросплатформені фреймворки 15](#_Toc1154361758)

[1.3. Особливості розробки ПЗ за допомогою *React* 25](#_Toc765645223)

[1.4. Висновки до розділу 30](#_Toc409439109)

[РОЗДІЛ 2 ВИКОРИСТАНІ ТЕХНОЛОГІЇ ПРИ РОЗРОБЦІ МОДУЛЯ 31](#_Toc734059471)

[2.1. *MovieDB API* 31](#_Toc983469023)

[2.2. Мова запитів *SQL* 36](#_Toc1233999813)

[2.4. *React Native* 42](#_Toc1414980017)

[2.5. *Yoga Engine* 44](#_Toc1207583541)

[2.6. *React* 47](#_Toc1469274333)

[2.7. *Expo* 49](#_Toc1765631756)

[2.8. *React Native Bridge* 51](#_Toc1600637)

[2.9. Висновки до розділу 52](#_Toc1354781175)

[РОЗДІЛ 3 РОЗРОБКА МОДУЛЯ СТВОРЕННЯ КРОСПЛАТФОРМЕНИХ](#_Toc673840683)

[ДОДАТКІВ 53](#_Toc1503212876)

[3.1. Постановка задачі 53](#_Toc1503212876)

[3.2. Серверна частина та *API* додатку 57](#_Toc361341058)

[3.3. Клієнтська частина додатку 65](#_Toc179817217)

[3.4. Висновки до розділу 72](#_Toc1163286240)

[РОЗДІЛ 4 ВИКОРИСТАННЯ КРОСПЛАТФОРМЕНОГО ДОДАТКУ 74](#_Toc1185427361)

[4.1. Вимоги до апаратно-програмного забезпечення 74](#_Toc1380858471)

[4.2. Запуск додатку на виконання 76](#_Toc815974407)

[4.3. Робота користувача з додатком 78](#_Toc465960953)

[4.4. Висновки до розділу 83](#_Toc1146634213)

[ВИСНОВКИ 84](#_Toc2049974220)

[СПИСОК БІБЛІОГРАФІЧНИХ ПОСИЛАНЬ, ВИКОРИСТАНИХ ДЖЕРЕЛ 88](#_Toc542657703)

[ДОДАТОК А 90](#_Toc542657703)

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ ТА СКОРОЧЕНЬ

*REST (Representational state transfer) –* архітектурний стиль взаємодії додатка із сервером.

*API (Application programming interface) –* інтерфейс для взаємодії клієнта та сервера призначений для використання розробниками.

*UI (User interface)* – графічне відображення даних для клієнта, або дизайн додатка.

*ПЗ* – програмне забезпечення

# ВСТУП

**Актуальність теми.** З 2012 року було зроблено великий прорив у розробці кросплатформених додатків. Мобільні платформи набирають все більшої популярності, тому сучасне програмне забезпечення (ПЗ) повинно підтримуватись не лише стаціонарними комп’ютерами, а також і мобільними пристроями, такими як планшети, смартфони та інші. Але не всі механізми створення кросплатформених додатків оптимальні, так як існує не тільки багато пристроїв, а також багато різних платформ, на яких працюють ці пристрої, тому й досі існує проблема створення гнучких рішень в сфері кросплатформених програмних додатків. Спроби створити універсальну базу коду та кросплатформені додатки були зроблені ще на початку 90-х років ХХ-го століття, але технології та пристрої настільки швидко змінювались, що будь-які модулі для створення кросплатформених додатків застарівали менше ніж за півроку.

Весь минулий період розвитку комп’ютерних наук підготував підґрунтя для розвитку кросплатформених програмних додатків. Найважливіші досягнення – створення кросплатформеної та не типізованої мови програмування *JavaScript*, за допомогою якої вдалося створити універсальні фреймворки для взаємодії між платформами.

У свій час дослідницькі групи пророкували, що в 2019 році виробники зможуть продати «удвічі більше» смартфонів, ніж у 2018, але виробники перевищили цей показник втричі. Смартфонами прийнято вважати апарат з досить складною операційною системою: наприклад *Android*, *iOS*, *Windows* *Mobile* або *Linux*.

Видавці газет і журналів звертають свій погляд на смартфони і планшетні комп’ютери, адже користувачі цих пристроїв переглядають найбільше контенту. Відповідно до збільшення кількості смартфонів та планшетів, відчутний жвавий розвиток зі створення додатків до них, які дозволяють розширити можливості мобільного пристрою: виходити в Інтернет, переглядати новини, погоду та багато іншого. В основному додатки доступні для завантаження та встановлення на популярних *online*-маркетах. Деякі програми безкоштовні, в той час як інші мають ціну. Як правило, вони завантажуються для цільового пристрою, як *iPhone*, *BlackBerry*, *Android* або *Windows* *Phone*. Сьогодні додатки до мобільних телефонів – це величезний ринок. Перед розробниками постає питання вибору технології для створення додатків до більшості мобільних платформ. Але на сьогоднішній день не існує чіткої рекомендації щодо розробок кросплатформених додатків і не структуровані та не наведені пропозиції з поєднання різних технологій для створення додатків. Через це перед розробниками постає досить цікаве питання: яку технологію та для якої платформи слід використовувати?

Вихід з цієї ситуації потрібно шукати у застосуванні сучасних, досить широко поширених та швидких, технологіях кросплатформеної розробки для створення зручного модулю, який дозволить перевикористовувати компоненти та функції.

Досягти ефективної роботи модулю, можна за допомогою такого інструменту як *React* *Native*. Також автоматизувати в цій системі вдосконалені процеси, методи обробки даних та часткового створення дизайну, що не були включені до інших кросплатформених систем.

При створенні кросплатформених додатків від програміста вимагається виконання певних процесів, починаючи від створення адаптивних інтерфейсів, до оптимізації додатку під певні платформи.

Модуль створення кросплатформених додатків пришвидшує процеси створення, тестування та оптимізації, що в свою чергу дозволяє розробникам покращити якість ПЗ та швидкість його написання.

В даній дипломній роботі розроблено модуль, що реалізує сучасні кросплатформені технології, та за його допомогою створено програмний додаток для відображення даних із *MovieDB*.

Кількість пристроїв, які повинні підтримуватись кросплатформеним додатком сягає сотень пристроїв, які мають різне апаратне та програмне забезпечення. Модуль працює із будь-якими пристроями, на будь-яких системах, так як логіка розподілена на мобільно-специфічні додатки та веб-додаток, таким чином, навіть якщо платформи *Android* та *iOS* не підтримуються пристроєм, але пристрій має програмне забезпечення для перегляду веб-сторінок, то список пристроїв, які будуть підтримувати таке ПЗ розширюється і охоплює будь-який сучасний пристрій, будь то комп’ютер, планшет чи смартфон.

**Мета і завдання виконання дипломної роботи.** Основною метою дипломної роботи є розроблення програмного модулю створення кросплатформених додатків для зручного і швидкого процесу розробки, тестування та кросплатформеного використання додатків.

Робота з модулем буде полягати у створенні *web-*додатку та мобільного додатку, який будується на основі дворівневої клієнт-серверної архітектури, та за допомогою фреймфорків *React* та *React Native*, що дасть змогу додатку працювати не тільки на мобільних пристроях, а і на будь-яких пристроях, які мають програмне забезпечення для перегляду веб-сторінок. Розробити архітектуру та алгоритми програми, які будуть повністю відповідати вимогам і швидко та якісно виконувати поставлені задачі. Обрати середовище розробки, мови програмування, програмні інструменти та інші засоби для розробки програмного продукту.

Розробити програмний кросплатформений додаток, за допомогою якого буде надано можливість обробляти дані із *MovieDB*, забезпечити зручний інтерфейс для взаємодії користувача з системою, створити необхідний функціонал.

**Об’єкт і предмет дослідження.** Об’єктом дослідження даної роботи є кросплатформені додатки. Предметом даної роботи є модуль створення кросплатформених додатків.

**Методи дослідження.** В ході розробки модуля створення кросплатформених додатків використовуються певні методи об’єктно-орієнтованого програмування за допомогою мови *JavaScript.* Програмне виконання буде відбуватись в середовищі *React Native*. Дане середовище надає широкий інструментарій для розробки веб*-*додатків, та мобільних додатків в тому числі, для створення кросплатформених додатків, ствоерння *web*-інтерфейсів, роботи із базами даних.

Метод *API* *MovieDB* буде використано у вигляді мобільного та веб-додатку за допомогою технології *React.* Для збільшення швидкості роботи додатку буде використано кешування даних за допомогою інструменту *navigator.* Також буде використано загальноприйняті шаблони проектування та підходи до розробки системи, популярні допоміжні бібліотеки для кешування даних, що пришвидшують видачу та обробку даних системою, бібліотеки для кросплатформених додатків.

**Наукова новизна отриманих результатів.** В ході виконання дипломної роботи отримано модуль створення кросплатформених додатків та створений за його допомогою кросплатформений додаток. Завдяки розробленому функціоналу модуля стало можливим швидке створення кросплатформених додатків. Система також надає функціонал для перегляду оброблених даних за допомогою *API* *MovieDB* та формування вибірок необхідних даних за певними параметрами, що задаються користувачем.

Механізми, що створено для роботи з користувачем, забезпечують простоту та гнучкість роботи, максимально простий та зрозумілий інтерфейс роботи з модулем на рівні користувача, а процеси створення кросплатформених функціональних елементів повністю автоматизовані. За допомогою розробленого програмного засобу відбувається робота з базою даних з використанням будь-яких пристроїв. Додаток дає змогу користувачу здійснювати пошук по базі даних та вибірку даних за допомогою системи фільтрації.

**Практичне значення отриманих результатів.** Розроблений модуль розроблявся для створення кросплатформених додатків. За допомогою модулю було розроблено додаток, який працює з даними із *MovieDB,* використовуючи *API*. Також додаток може використовуватись будь-якими пристроями, під управлінням будь-якого програмного забезпечення, якщо у використовуваного пристрою є доступ до веб-браузеру.Модуль частково автоматизує створення кросплатформених додатків, в результаті чого будуть отримуватися кращі результати по швидкості створення кросплатформених рішень. Здійснюватиметься первинний аналіз потрібних для використання платформ і на основі цієї інформації створюються потрібні для створення додатку модулі та встановлюються кросплатформені бібліотеки. Також автоматизація процесу створення кросплатформених додатків частково виключить людський фактор з загального процесу створення ПЗ.

Розроблений в ході виконання дипломної роботи модуль успішно використовується в комерційних цілях. За допомогою модуля значно покращилась якість кросплатформених додатків та швидкість їх створення, зменшився проміжок часу, за який створювалися окремі додатки для кожної платформи. Показник часу зменшився більше ніж в 3 рази.

**Апробація отриманих результатів**. Основні ідеї, та результати, що ввійшли в дипломну роботу, зокрема створення кросплатформених додатків за допомогою модуля, пройшли апробацію на міжнародній науково-технічній конференції «Інтелектуальні технології лінгвістичного аналізу» (Київ, 2019) [1].

# РОЗДІЛ 1 ОГЛЯД ОСНОВНИХ МЕТОДІВ СТВОРЕННЯ КРОСПЛАТФОРМЕНИХ ДОДАТКІВ

## 1.1. Кросплатформеність

Кросплатформеність – здатність програмного забезпечення працювати більш, ніж на одній апаратній платформі або операційній системі.

Забезпечується завдяки використанню високорівневих мов програмування, середовищ розробки і виконання, що підтримують умовну компіляцію, компоновку і виконання коду для різних платформ.

Типовим прикладом є програмне забезпечення, призначене для роботи в операційних системах *Linux* і *Windows* одночасно або програмне забезпечення призначене для використання на мобільних пристроях та одночасно на повноцінних комп’ютерах. Здійснюється реалізацією одних і тих же функцій для кожної операційної системи або платформи окремо. Однак в деяких випадках функції повторюють функціонал.

Кросплатформеними можна назвати більшість сучасних високорівневих мов програмування. Наприклад, *C, С ++, Java* – кросплатформені мови на рівні компіляції, тобто для цих мов є компілятори під різні платформи. Це дозволяє – при належній якості коду – не переписувати основний двигун програми, змінюються лише особливі системозалежні частини. Не менш важливі для платформ є стандартизовані бібліотеки середовища виконання. Зокрема, стандартом стала бібліотека мови *С*. З великих кроссплатформенних бібліотек – *Qt, GTK +, FLTK, STL, Boost, OpenGL, SDL, OpenAL, OpenCL.* Зараз найбільшої популярності набуває мова програмування JavaScript, адже вона дозволяє створювати програмне забезпечення майже під будь-які платформи та апаратні платформи.

Кросплатформені додатки класифікують за різними ознаками, проте головними є класифікації за кількістю підтримуваних пристроїв та платформ, а також за мовою програмування, за допомогою якої було розроблено додаток [2].

Cі++ – мова програмування високого рівня з підтримкою кількох парадигм програмування: об'єктно-орієнтованої, узагальненої та процедурної. Розроблена Б'ярном Страуструпом в 1979 року та початково отримала назву «Сі з класами». Згодом Страуструп перейменував мову на Cі++ у 1983 р. Базується на мові Сі.

У 1990-х роках Сі++ стала однією з найуживаніших мов програмування загального призначення. Мову використовують для системного програмування, розробки програмного забезпечення, написання драйверів, потужних серверних та клієнтських програм, а також для розробки розважальних програм, наприклад, відеоігор. Сі++ суттєво вплинула на інші популярні сьогодні мови програмування.

*Java* – об'єктно-орієнтована мова програмування, випущена 1995 року, як основний компонент платформи *Java*. В офіційній реалізації *Java* програми компілюються у байт-код, який при виконанні інтерпретується віртуальною машиною для конкретної платформи. Мова значно запозичила синтаксис із Cі і Cі++. Зокрема, взято за основу об'єктну модель Сі++, проте її модифіковано. Усунуто можливість появи деяких конфліктних ситуацій, що могли виникнути через помилки програміста та полегшено сам процес розробки об'єктно-орієнтованих програм.

Передусім *Java* розроблялась як платформо-незалежна мова, тому вона має менше низькорівневих можливостей для роботи з апаратним забезпеченням, що в порівнянні, наприклад, з Cі++ зменшує швидкість роботи програм.

*JavaScript­* – динамічна, об'єктно-орієнтована прототипна мова програмування. Реалізація стандарту *ECMAScript*. Найчастіше використовується для створення сценаріїв веб-сторінок, що надає можливість на стороні клієнта взаємодіяти з користувачем, керувати браузером, асинхронно обмінюватися даними з сервером, змінювати структуру та зовнішній вигляд веб-сторінки. *JavaScript* класифікують як прототипну (підмножина об'єктно-орієнтованої), скриптову мову програмування з динамічною типізацією. Окрім прототипної, *JavaScript* також частково підтримує інші парадигми програмування (імперативну та частково функціональну) і деякі відповідні архітектурні властивості, зокрема: динамічна та слабка типізація, автоматичне керування пам'яттю, прототипне наслідування, функції як об'єкти першого класу [3].

## 1.2. Основні кросплатформені фреймворки

Для створення кросплатформених додатків потрібна не лише мова програмування, а й спеціальні фреймворки, у яких зібрані методології створення ПЗ і які дають можливість створити самостійний модуль для створення кросплатформених додатків. Основними фреймворками на сьогоднішній день є: *Cordova*, *Flutter* та *React Native.* У кожного з цих фреймворків є свої переваги та недоліки, але найбільш гнучким у використанні за рахунок використання мови програмування *JavaScript* залишається *React Native*, за допомогою якого і буде побудовано модуль для створення кросплатформених додатків.

### 1.2.1. *Cordova*

*Apache Cordova* — фреймворк для створення мобільних застосунків, що продовжує розвиток платформи *PhoneGap*, після передачі проекту компанією *Adobe* в руки фонду *Apache*. Одночасно компанія *Adobe* представляє заснований на єдиній кодовій базі з *Apache Cordova* продукт *PhoneGap*, функціонально ідентичний до *Apache Cordova*. У руки *Adobe* проект потрапив восени 2011 після поглинання компанії *Nitobi*, після чого було прийнято рішення розвивати проект у максимально відкритому вигляді з залученням додаткових учасників. Для реалізації даного плану, майже відразу після поглинання *Nitobi* код *PhoneGap* був переданий фонду *Apache*, який прийняв проект під свою опіку під ім'ям *Apache Cordova* [4].

Розрахунок *Adobe* виявився правильним і до розробки *PhoneGap* приєдналося кілька відомих компаній, серед яких *IBM*, *Sony*, *Google*, *Microsoft*, *Blackberry* і *HP*. *Apache Cordova* дозволяє створювати універсальні мобільні застосунки, що працюють на різних мобільних платформах, з використанням стандартних веб-технологій. Використання *Apache* *Cordova* дозволяє створювати застосунки, що функціонують на широкому спектрі мобільних платформ, включаючи *Tizen*, *webOS*, *Android*, *Apple* *iOS*, *Blackberry*, *Samsung* *Bada* і *Windows* *Phone*.

Процес установки і зовнішній вигляд таких застосунків нічим не відрізняється від програм, створених з використанням рідного для кожної мобільної платформи інструментарію. Через спеціальний прошарок підтримується використання специфічних для кожної платформи функцій, а також доступ до *API* для взаємодії з обладнанням, телефонним стеком, адресною книгою, звуковою підсистемою, камерою і іншими компонентами мобільних платформ. Через створення плагінів підтримується додавання додаткової функціональності і розширення *API*. Технології програмного каркасу *Apache Cordova* використовує *Ubuntu*. Починаючи з версії 3.0 можна використовувати два основних робочих процеси для створення мобільних додатків. Хоча ви можете використовувати будь-який робочий процес для виконання однієї і тієї ж задачі, кожен з цих шляхів має свої переваги.

Кросплатформний робочий процес дає можливість додатку запускатися на максимально можливій кількості мобільних платформ, з мінімальними потребами для платформо-специфічної розробки. Цей робочий процес формується для *cordova*, також відомої як *Cordova CLI*, який був введений починаючи з *Cordova* 3.0.

*CLI* – це високо-рівневий інструмент який дозволяє будувати проекти для якомога більшої кількості платформ одночасно, абстрагуючи якомога більше функціональності низько-рівневих скриптів. *CLI* копіює загальний набір *web*-ресурсв в підкаталоги для кожної мобільної платформи, робить будь-які необхідні зміни в конфігурацію для кожної платформи, запускає сценарії збірки для створення виконуваних файлів програми. *CLI* також надає загальний інтерфейс для застосування плагінів для вашого застосування.

Платформо-орієнтований процес розробки використовується для побудови програми для однієї платформи, де буде необхідно вносити модифікації на низькому рівні. Цей процес використовується, наприклад, якщо ви хочете щоб ваш додаток поєднував власні компоненти з веб-орієнтованими компонентами *Cordova*. Роботу додатку, який розроблено за допомогою *Cordova* зображено на рис. 1.1.

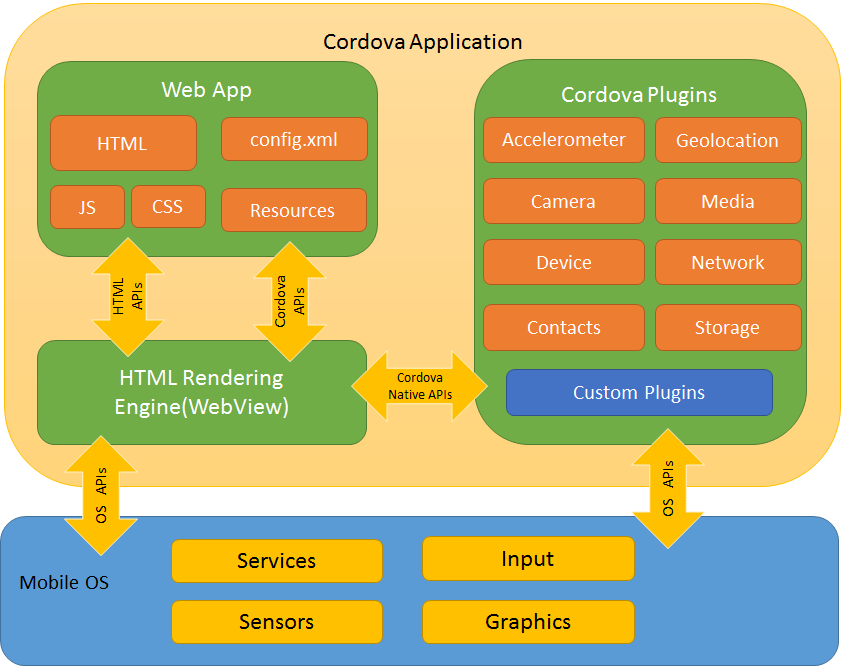


Рис. 1.1. Робота додатку за допомогою *Cordova*

Як правило цей робочий процес використовується, якщо необхідно змінити проект за допомогою *SDK*. Цей робочий процес спирається на набір скриптів нижчого рівня, які пристосовані для кожної підтримуваної платформи і окрему утиліту Plugman, яка дозволяє застосовувати плагіни індивідуально до обраної платформи. Хоча ви можете використовувати цей робочий процес для створення кросплатформних додатків, це як правило складніше, тому що відсутність утиліт більш високого рівня означає окремі цикли збирання та модифікації плагінів для кожної платформи. Проте цей робочий процес дає вам більш широкий доступ до варіантів розробки, що надаються кожним *SDK*, важливих для складних гібридних додатків.

Додатки *Cordova* покладаються на загальний файл *config.xml*, який містить інформацію про програму та визначає параметри, що впливають на те як додаток працює, такі як, чи реагує воно на зміну орієнтації пристрою. Цей файл відповідає специфікації *W3C*.

Сам додаток реалізований як веб-сторінка, за замовчуванням локальний файл під назвою *index.html*, який посилається на будь-який *CSS*, *JavaScript*, зображення, файли мультимедіа або інші ресурси необхідні для його запуску. Додаток виконує код, як *WebView* в межах оболонки додаткуї. *WebView* з підтримкою *Cordova* може представляти додатки і призначений для користувача інтерфейс. На деяких платформах вона також може бути компонентом у великих, гібридних додатках, які об'єднують *WebView* з іншими компонентами додатку. Роботу *Cordova* із *WebView* зображено на рис. 1.2.

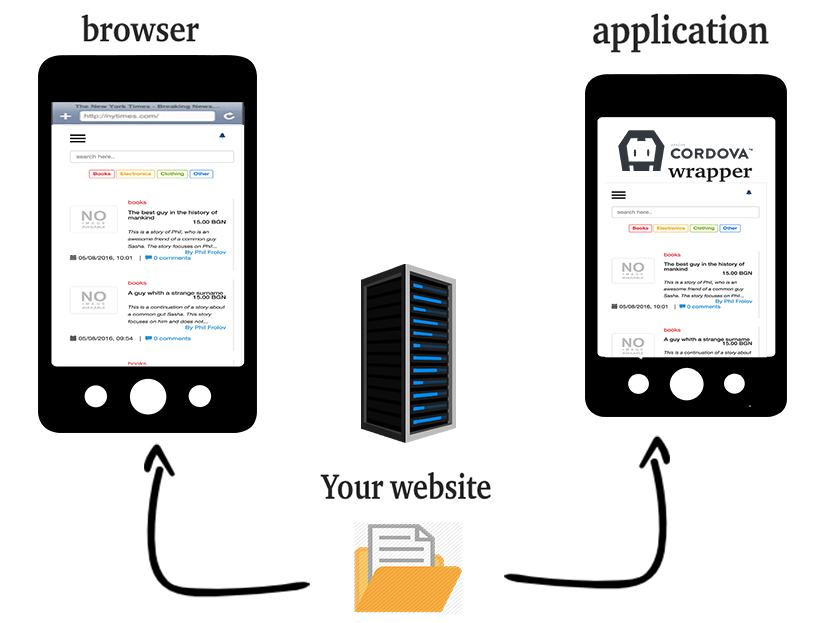


Рис. 1.2. Робота *Cordova* із *WevView*

Інтерфейс плагіна доступний для *Cordova* і інших компонентів, для взаємодії один з одним. Це дозволяє викликати код на мові платформи з *JavaScript*. В ідеалі на декількох платформах пристроїв узгоджуються *JavaScript* *API*, щоб машинний код не відрізнявся. Починаючи з версії 3.0 плагіни потребують прив'язки до стандартних *API* інтерфейсів пристрою. Сторонні плагіни надають додаткові прив'язки для функцій не обов'язково доступних на всіх платформах. Можна знайти ці сторонні плагіни в реєстрі плагінів і використовувати їх в своєму додатку.

### 1.2.2. *Flutter*

*Flutter* – це програмний каркас із відкритим кодом, для створення додатків для платформ *Android* та *iOS*, а також на веб-додатків, розроблений компанією *Google*. Він є основним способом створення додатків для *Google Fuchsia*. Весь графічний інтерфейс *Google Fuchsia* створено за допомогою *Flutter*.

*Flutter* використовує розроблену *Google* мову *Dart*. Якщо ви раніше використовували *Java*, синтаксис *Dart* буде теж знайомим, оскільки вони дуже схожі. Але, крім синтаксису, *Dart* в іншому відрізняється від *Java*. Однією з найкорисніших функцій є функція підтримки асинхронних операцій. Не лише *Dart* забезпечує таку підтримку, але він робить її дуже простою. Найімовірніше, ви будете використовувати її у всіх ваших програмах *Flutter*, якщо виконуєте асинхронні операції або інші часовитратні операції, такі як запит до бази даних. Без асинхронних операцій будь-які часовитратні операції призведуть до зависання програми під час їхнього виконання. Щоб цього уникнути, *Dart* надає нам ключові слова *async* та *await*, які дають змогу нашій програмі продовжувати виконання, очікуючи завершення цих часовитратних операцій [5].

Більшість додатків сьогодні залежать від певних даних, які потрібно десь зберігати, щоб вони могли відображатися й можна було працювати з ними пізніше. Важливо пам’ятати про це, коли хочете створювати програми з новим *SDK*, як *Flutter*.

Неймовірно просто створити програми як для початківців, так і для експертів, але ця простота жодним чином не дорівнює відсутності якості. Бібліотеки також доступні, щоб ви могли працювати з базою даних за вашим вибором. Використовуючи бібліотеку *sqflite*, можемо швидко запускати й працювати з базою даних *SQLite*. А завдяки синглтонам можемо отримати доступ до бази даних і звертатися до неї практично з будь-якого місця без потреби щоразу відновлювати об’єкт. Після отримання даних із бази можна перетворити їх на об’єкт за допомогою моделі. Або, якщо хочете зберегти об’єкт у базі даних, ви можете перетворити його в *JSON* за допомогою тієї самої моделі. Ці дані не будуть корисними без можливості їх відображення користувачеві. Тому у *Flutter* є такі віджети, як *FutureBuilder* або *StreamBuilder*.

Особливості мови *Dart*:

* звичний і простий для вивчення синтаксис, природний для програмістів на *JavaScript* та *Java;*
* забезпечення швидкого запуску і високої продуктивності для всіх сучасних веб-браузерів і різних типів оточень, від портативних пристроїв до потужних серверів;
* можливість визначення класів і інтерфейсів, що дозволяють використовувати інкапсуляцію і повторно використовувати існуючі методи і дані;
* необов'язкове вказування типів, використовувати чи ні статичні типи вирішує розробник;
* вказування типів дозволяє спростити зневадження і виявлення помилок, робить код яснішим і читаним, спрощує його доопрацювання та аналіз сторонніми розробниками. Серед підтримуваних типів: різні види хешів, масивів і списків, черги, числові і рядкові типи, типи для визначення дати і часу, регулярні вирази (*RegExp*). Можливо створення своїх типів;
* для організації паралельного виконання пропонується використовувати класи з атрибутом *isolate*, код яких виконується повністю в ізольованому просторі в окремій області пам'яті, взаємодіючи з основним процесом через відправку повідомлень;
* підтримка використання бібліотек, що спрощують підтримку і зневадження великих веб-проектів. Сторонні реалізації функцій можуть підключатися у вигляді поділюваних бібліотек. Застосунки можна розбити на частини і доручити розробку кожної з частин окремій команді програмістів;
* набір готових інструментів для підтримки розробки мовою *Dart*, включаючи реалізацію засобів динамічної розробки та зневадження з виправленням коду на льоту;
* можливість створювати однорідні системи, що охоплюють як клієнтську, так і серверну частину. Використання однієї мови та інструментарію для клієнтських і серверних компонентів спрощує процес кодування і позбавляє від постійної зміни контексту.

Мова має схожий на *Java* синтаксис, не вимагає явного визначення типів і може використовуватися для створення серверних і клієнтських застосунків. Для запуску всередині браузера код мовою *Dart* може бути перетворений в *JavaScript*-подання або запущений безпосередньо під управлінням спеціального *JavaScript*-інтерпретатора *Dartboard*. Підтримується вбудовування коду мовою *Dart* в *HTML*-сторінки.

На стороні сервера застосунок на мові *Dart* може бути виконаний всередині спеціальної віртуальної машини, яка забезпечує продуктивність виконання близьку до компільованих в машинний код мов. Віртуальну машину *Dart* планують інтегрувати в майбутні версії браузера *Chrome*, що дозволить виконувати застосунки мовою *Dart* *без* компіляції в *JavaScript*.

Мова підходить як для розробки одним програмістом невеликих скриптів без жорсткої структури, так і для створення високо-масштабованих великих модульних проектів, підтримуваних великим колективом з потребою більш явної типізації для того, щоб уникнути плутанини і помилок.

При цьому явне задання типів не обов'язкове, наприклад, можна почати розробку без вказання типів, а надалі при необхідності додати їх. Код *Dart* завжди виконується тільки в рамках одного потоку, для організації паралельного виконання пропонується використовувати класи з атрибутом *isolate*. У кожному скрипті використовується власний простір імен, для використання зовнішніх об'єктів, функцій або змінних слід їх явно імпортувати.

Всі змінні, початково, діють тільки в межах поточного скрипта і не експортуються глобально. Для спрощення розробки мовою *Dart* поставляється *SDK*, який включає в себе компілятор *dart2js*, віртуальну машину *Dart* *VM*, пакетний менеджер *pub*, статичний аналізатор коду *dart\_analyzer*, і набір бібліотек. Для виконання і відлагодження застосунків на мові *Dart*, без компіляції в *JavaScript*, поширюється *Dartium* – складання браузера *Chromium* з інтегрованою віртуальною машиною *Dart VM*. До складу *SDK* входить *Dart* – спеціалізоване інтегроване середовище розробки мовою *Dart*. *Dart Editor* з одного боку досить легковаговий і не вимогливий до ресурсів, а з іншого підтримує такі елементи сучасних *IDE*, як рефакторинг, автодоповнення коду, перехід на визначення, контекстні підказки, містить вбудований відлагоджувач і аналізатор коду. Плагіни з підтримкою *Dart* також підготовлені для *IntelliJ IDEA*, *WebStorm*, *Eclipse*, *Emacs* і *Vim*. Додаткові пакунки з бібліотеками і утилітами поширюються через репозиторій *pub*, який станом на весну 2015 налічує понад півтори тисячі пакунків, у тому числі фреймворки для розробки веб-застосунків *AngularDart* і *polymer.dart*.

### 1.2.3. *React Native*

*React Native* – це мобільний фреймворк, який дозволяє створювати додатки, використовуючи лише *JavaScript*. Однак, на відміну від інших гібридних мобільних технологій, ви не створюєте мобільний веб-додаток (веб додаток загорнений у рідний контейнер). Ваша база коду *JavaScript* компілюється в мобільний додаток, який не відрізнятиметься від *iOS* додатка, побудованого з використанням *Objective-C* або ж *Android* додатка з використанням *Java* [6].

Це означає, що *React Native* надає переваги як для нативних, так і для гібридних мобільних додатків. На даний момент існує достатньо фреймворків, які використовують *JavaScript* для створення *iOS* та *Android* додатків, таких як *PhoneGap* або *Titanium*. Проте *React Native* має перед ними ряд переваг:

* на відміну від *PhoneGap*, в *React Native* логіка додатка пишеться і працює на *JavaScript*, в той час як його інтерфейс залишається повністю нативним. Таким чином не потрібно ніяких компромісів, характерних для *HTML5*;
* на відміну від *Titanium*, *React* вводить новий оригінальний і вкрай ефективний підхід до створення призначених для користувача інтерфейсів. Якщо говорити коротко, *UI* додатки виражається як функція поточного стану програми. Ключова особливість *React Native* в тому, що його розробники мають намір принести модель програмування *React* в сферу розробки мобільних додатків. Якщо у вас є досвід створення програмного забезпечення на *Objective-C* або *Swift*, ви напевно не зрадієте ідеї переходу на *JavaScript*. Але разом з тим, другий пункт явно мав зацікавити *Swift*-розробників. Безсумнівно, працюючи зі *Swift*, вам доводилося вивчати багато нових і більш ефективних способів шифрування алгоритмів, а також методик, що сприяють перетворенню і незмінності. Проте, спосіб побудови *UI* тут дуже схожий на той, що використовується при роботі з *Objective-C*: він теж грунтується на *UIKit* і є імперативним. *React* за рахунок таких новітніх понять як *Virtual DOM* і узгодженню переносить функціональне програмування на шар призначений для інтерфейсу користувача;
* єдина кодова база для *Android* і *iOS* *React Native* дозволяє писати код один раз і запускати як на *Android*, так і на *iOS*. Кодова база буде однаковою для обох платформ, тому вам не доведеться витрачати гроші і час на два проекти розробки. Оскільки *React Native* має відкритий вихідний код і сприяє повторному використанню, ви можете повторно використовувати компоненти на будь-якому рівні, не переписуючи його і не виконуючи компіляцію додатка;
* можливість повторного використання. Тепер поговоримо про можливості повторного використання. *React Native* приносить блоки, які складаються з повторно використовуваних компонентів. Майже всі компоненти *Android* і *iOS* мають свої аналоги в *React Native*. Всі ці компоненти компілюються безпосередньо в нативні додатки і, таким чином, дозволяють розробникам підтримувати зовнішній вигляд платформи. Раніше в гібридних додатках було неможливо отримати структуру, специфічну для компонента. Але з *React Native* ця структура дозволяє розробникам створювати додатки, використовуючи гнучкі веб-методології. З іншого боку, *React Native* також дозволяє розробникам додавати компоненти *Native* в код своїх існуючих додатків. Їм не доведеться кодувати з нуля, щоб перетворити існуюче мобільний додаток в додаток *React Native*;
* продуктивність. *React Native* скорочує цикл розробки, дозволяє розробляти додатки дуже швидко і дозволяє доставляти додатки максимально швидко. Він використовує бібліотеку *ReactJS UI*, розроблену *Facebook* для призначених для користувача інтерфейсів. У той час, як нативні додатки працюють на центральному процесорі, *React Native* використовує графічний процесор. Він поставляється з унікальною функцією під назвою перезавантаження, яка дозволяє переглядати останні зміни коду відразу, розбиваючи екран на дві частини. Перший показує код, а другий показує результат коду в кадрі мобільного телефону. В цілому, за допомогою *React Native* компанії можуть скоротити витрати на розробку майже на 50% без шкоди для якості або продуктивності;
* орієнтація на призначений для користувача інтерфейс. *React Native* повністю орієнтований на призначений для користувача інтерфейс. Декларативний *API* значно спрощує розуміння ваших вимог і прогнозування призначеного для користувача інтерфейсу. Декларативний стиль дозволяє вам контролювати потік і стан в вашому додатку. Вам більше не доведеться турбуватися про деталі реалізації;
* сумісність зі сторонніми плагінами. Додавання апаратних можливостей пристрою в додаток є одним з найбільш поширених вимог в наші дні. На відміну від функцій *WebView*, *React Native* дозволяє безпосередньо зв'язати плагін з власним модулем через інфраструктуру. Це призводить до більш плавного запуску додатка, більш швидкого завантаження та менших вимог до пам'яті.

В цілому, додатки написані за допомогою *React Native* працюють з такою ж швидкістю, як і нативні додатки, які написані мовою платформи, на якій використовується код написаний за допомогою фреймворку.

З максимальною швидкістю *React Native* працює при використанній його із платформою *iOS*, але і при використанні інших платформ швидкість залишається в рамках норми. Швидкість завантаження одного і того ж додатку під різними платформами представлена в таблиці 1.1.

Таблиця 1.1

Швидкість завантаження додатку

|  |  |  |
| --- | --- | --- |
| Платформа | Завантаження, с | Максимальна кількість кадрів, с |
| *Android* | 3,2 | 80 |
| *iOS* | 1,8 | 100 |
| *iOS Emulator* | 2,5 | 60 |
| *Android Emulator* | 4,1 | 60 |
| *Web* | 3,2 | 60 |

Як можна побачити із таблиці, найшвидші показники *React Native* показує при роботі з платформою *iOS*, через найбільш оптимізовані процеси виконання *JavaScript* коду. *Android* працює більш повільно, так як виникають конфлікти при портуванні *JavaScript* інструкцій у код *Java*.

Кількість кадрів за секунду – це величина з якою відбувається оновлення контенту на екрані мобільного пристрою, чи комп’ютеру. Зазвичай мінімальна величина з якою можна переглядати контент дорівнює 60 кадрам на секунду. У випадку з *iOS*, *React Native* показує 100 кадрів на секунду, що перевищує норму і дає можливість створювати продуктивні додатки, які працюють не гірше, ніж нативні.

## 1.3. Особливості розробки ПЗ за допомогою *React*

### 1.3.1. Мова *Javascript*

З початку *JavaScript* було створено для того, щоб додати інтерактивності та «оживити» сайт, але ця мова програмування давно вже вийшла за рамки того, для чого була створена. Динамічна мова програмування, яка від самого початку носила назву *Mocha*, була написана групою програмістів на чолі з Брендоном Ейком у 1995 році. Мову було створено за 10 днів для компанії *Netscape*, яка потребувала скриптову мову для свого браузера з однойменною назвою. З *Mocha* мову було перейменовао у *LiveScript*, а потім вже у *JavaScript* через високу популярність у той час мови *Java*.

Статистика відразу кількох шанованих ресурсів, таких як *GitHub*, *StackOverflow* та інших, стверджують, що *JavaScript* є мовою програмування, яка швидше за все набирає популярність, а результати опитувань розробників по всьому світові й взагалі ставлять *JavaScript* на перше місце серед найпопулярніших мов програмування.

*JavaScript* було вигадано як мову для браузерів, але сьогодні на ній можна створювати сервер, розробляти веб-додатки, мобільні додатки, ігри та багато іншого. Якщо раніше *JavaScript* входив у обов’язковий список вмінь для *front end* розробки, то зараз він впевнено відчуває себе й на *back end*. Найчастіше *JavaScript* використовують у поєднанні з принципами *SOLID* архітектури.

*SOLID* – це абревіатура для п'яти принципів об'єктно-орієнтованого дизайну, описаних Робертом Мартіном.

Принципи *SOLID*:

* принцип єдиної відповідальності (*Single responsibility*);
* принцип відкритості-закритості (*Open-closed*);
* принцип підстановки лісков (*Liskov substitution*);
* принцип поділу інтерфейсу (*Interface segregation*);
* принцип інверсії залежностей (*Dependency inversion*).

Принципи *SOLID* часто обговорюються в рамках класичних, об'єктно-орієнтованих мов. У той же час, *JavaScript* є мовою, заснованою на прототипах, які суміщають в собі концепції як об'єктно-орієнтованої, так і функціональної парадигм програмування.

Об'єкт повинен мати пов'язаний набір поведінки, що відноситься до однієї відповідальності. Зміна цієї відповідальності повинна призводити до зміни об'єкта. Коли об'єкт має кілька відповідальностей, його зміна, пов'язане з однією з відповідальностей, може побічно змінити логіку, пов'язану з іншого відповідальністю. Поділ відповідальностей як раз і робить код більш стійким до змін.

Стереотипні ролі об'єктів є одним з підходів, що допомагають в поділі спарених відповідальностей, заснований на стереотипах ролей об'єкта. Стереотипні ролі об'єктів – це набір загальних ролей, які зазвичай зустрічаються в об'єктно-орієнтованої архітектури.

Хоча *JavaScript* і може нагадувати *Java* методами роботи з об’єктами, відмінностей дуже багато.

*JavaScript* наслідує синтаксис більшості виразів *Java*, також наслідуються угоди про іменування та основні конструкції керування перебігом виконання сценарію. Це і стало причиною того, що свого часу назву мови змінили з *LiveScript* на *JavаScript*. На відміну від системи часу коміляції декларативно побудованих класів у *Java*, мова

*JavaScript* підтримує систему часу виконання, що ґрунтується на невеликій кількості типів даних, представлених числовим типом, логічними, та рядковими значеннями.

*JavaScript* має об'єктну модель, що спирається на прототипи, замість більш поширеної, побудованої на класах, об'єктної моделі, властивої мові *Java*. Модель на базі прототипів надає можливість динамічного спадкування; тобто успадковане може бути різним для різних об'єктів. Також у *JavaScript* реалізована підтримка функцій без будь-яких особливих декларативних вимог. Функції можуть бути властивостями об'єктів, функціями, що виконуються, та слабко типізованими методами.

### 1.3.2. Перевикористання компонентів

Компоненти дозволяють розділити інтерфейс користувача на незалежні частини, придатні до повторного використання, і сприймати їх як такі, що функціонують окремо один від одного.

Компоненти можуть посилатися на інші компоненти під час виведення. Це дозволяє нам використовувати одну і ту ж абстракцію компонентів для будь-якого рівня деталізації. Кнопка, форма, діалогове вікно, екран. У *React* додатках всі вони зазвичай виражаються як компоненти.

Розбиття компонентів може здатися спочатку невдячною роботою. Проте, у великих додатках така велика кількість багаторазових компонентів є дуже корисною. Суть в тому, що якщо частина вашого інтерфейсу використовується кілька разів, або сама собою досить складна, краще винести її в окремий компонент [7].

У кожного з компонентів є свій життєвий цикл, який включає в себе методи ініціалізації, оновлення та припинення роботи компоненту, схема роботи життєвого цикла відображена на рис. 1.3.

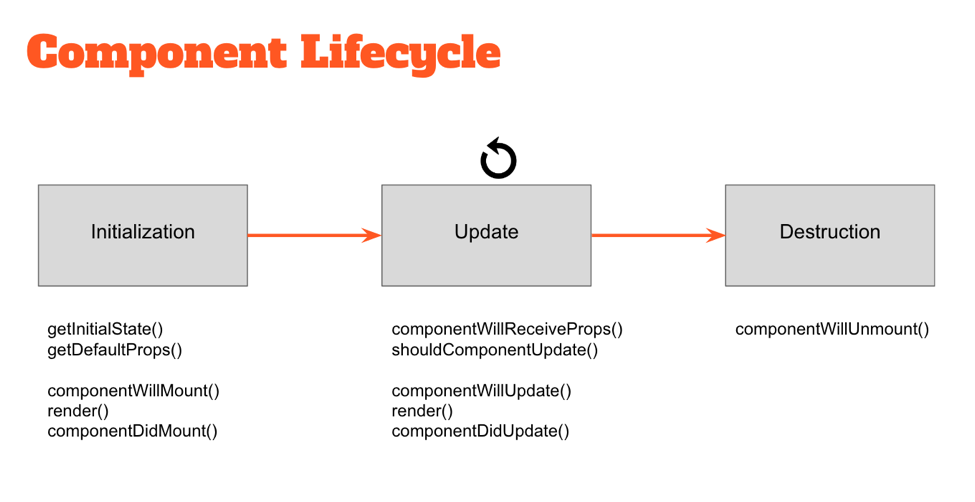


Рис. 1.3. Життєвий цикл компоненту

Незалежно від того як ви оголосите компонент як функцію чи клас, він ніколи не повинен змінювати свої власні параметри. Такі функції називаються чистими, оскільки вони завжди повертають один і той же результат для тих же аргументів.

### 1.3.3. *React Hooks*

Хуки – це функції, за допомогою яких ви можете зачепитися за стан та методи життєвого циклу *React* з функціональних компонентів. Хуки не працюють всередині класів, вони дають вам можливість використовувати *React* без класів [8].

*React* містить кілька вбудованих хуків, таких як *useState*. Також можна створювати власні хуки, щоб повторно використовувати їх в інших своїх компонентах. За допомогою хука ефекту *useEffect* можна виконувати побічні ефекти із функціонального компонента. Він виконує таку ж саму роль, що і *componentDidMount*, *componentDidUpdate* та *componentWillUnmount* у *React* класах, об’єднавши їх в єдиний *API*.

Коли ви викликаєте *useEffect*, *React* отримує вказівку запустити вашу функцію з ефектом після того, як він відправив зміни у *DOM*. Оскільки ефекти оголошуються всередині компонентів, то у них є доступ до пропсів та стану. За замовчуванням, *React* запускає ефекти після кожного рендеру, включаючи перший рендер.

У разі потреби, ви можете повернути з ефекта функцію, яка вказує ефекту, як виконати за собою скидання. Наприклад, цей компонент використовує ефект, щоб підписатися на статус друга в мережі, і виконує скидання, відписуючись від нього. Хуки є звичайними функціями *JavaScript*, але вони накладають два додаткових правила.

Хуки слід викликати тільки на верхньому рівні. Не викликайте хуки всередині циклів, умов або вкладених функцій. Хуки слід викликати тільки з функціональних *React*-компонентів. Не викликайте хуки із звичайних функцій *JavaScript*.

За допомогою користувацьких хуків це завдання вирішується без додавання непотрібних компонентів у ваше дерево. Але при використанні цих шаблонів проектування, розробник вимушений реструктурувати компоненти, через що код стає заплутаним та важко зрозумілим. Якщо поглянути у *React Devtools* на типовий застосунок написаний на *React*, з великою імовірністю там буде ситуація, коли компоненти обгортаються у шари провайдерів, споживачів, компонентів вищого порядку, рендер-пропів та інших абстракцій [9].

Незважаючи на те, що їх можливо відфільтрувати у *Devtools*, стає зрозуміло, що необхідно створити кращий спосіб повторно використовувати логіку навколо стану. За допомогою хуків ви можете виокремити логіку стану з компонента, щоб протестувати її або повторно використати. З хуками зʼявляється можливість перевикористання логіки з відслідкуванням стану без зміни ієрархії ваших компонентів.

## 1.4. Висновки до розділу

В розділі було розглянуто основні поняття кросплатформеної розробки, а також технології, які використовуються для досягнення кросплатформеності. Описано основні кросплатформені фреймворки. Виконано порівняльну характеристику кросплатформених фреймворків, а також вказано їх переваги та недоліки перед іншими системами. Також було розглянуто характерні мови програмування, які використовуються для розробки кросплатформених додатків.

Описано основні фреймворки та обрано мову програмування для розробки модуля створення кросплатформених додатків. Було описано основні концепції фреймворку *React*, його взаємодії з компонентами, методи життєвого циклу кожного із компонентів. Описано основні принципи роботи фреймворку *React Native*, обґрунтовано вибір цього фреймворку та описано особливості виконання коду ПЗ, написаного за допомогою *React Native* під різними платформами та пристроями.

# РОЗДІЛ 2 ВИКОРИСТАНІ ТЕХНОЛОГІЇ ПРИ РОЗРОБЦІ МОДУЛЯ

## 2.1. *MovieDB API*

Мета будь якої *API* – надання даних у зручному вигляді для клієнта, яким в даному випадку є кросплатформене ПЗ. *API* дозволяє отримувати дані із бази даних шляхом послання зручних запитів.

В загальному *API* – це набір готових класів, процедур, функцій, структур і констант, що надаються додатком (бібліотекою, сервісом) для використання в зовнішніх програмних продуктах.

Зазвичай при зверненні до веб *API* використовуються запити *HTTP*. Стандартними методами *HTTP* запитів є:

* *GET* – для отримання або читання даних. Запити з використанням цього методу можуть тільки отримувати дані;
* *HEAD* – запрошує ресурс так само, як і метод *GET*, але без тіла відповіді;
* *PUT* – зазвичай використовується для оновлення ресурсу, шляхом заміни даних запиту;
* *POST* – зазвичай використовується для створення нового ресурсу або використовується для відправки сутностей до певного ресурсу;
* *DELETE* – видаляє дані;
* *OPTIONS* – використовується для опису параметрів з’єднання з ресурсом;
* *PATCH* – використовується для часткової зміни ресурсу.

Сервер може посилати різні коди у відповідь на запити користувачів. Це можуть бути коди помилок або просто коди, що інформують користувачів про стан сервера

Найбільш відомі коди – *OK* *Success* (успішна відповідь від серверу), *Error* (проблеми на стороні сервера). Про те, які коди повертати в тій чи іншій ситуації, вирішують розробники самих *API*.

### 2.1.1. Автентифікація

Зазвичай для використання *API* потрібен спеціальний ключ, за допомогою якого сервер дізнається про користувача. *API* ключ може бути відсутнім або надаватися за запитом (наприклад, після реєстрації на сайті).

При використанні *MovieDB* для використання *API* потрібно отримати спеціальний ключ, який надається клієнту після авторизації додатку.

*API* складається з декількох дублюючих один одного частин, що зберігаються на різних обчислювальних машинах мережі. Робота з такою *API* відбувається за допомогою системи керування розподіленою базою даних (СКРБД). За способом доступу, *API* використовують клієнт-серверну архітектуру.

Клієнт – сервер. В даній концепції мається на увазі, що окрім зберігання централізованої бази даних центральна машина (сервер бази даних) повинна забезпечувати виконання основного об’єму обробки даних. Запит на дані, що видаються клієнтом, породжує пошук та вивід даних на сервері. Вивід даних транспортується через мережу від сервера до клієнта. Специфікою архітектури клієнт – сервер є використання мови запитів *SQL*. Завдяки даній концепції, *API MovieDB* працює дуже швидко і надає змогу не тільки отримувати дані з сервера, а також оброблювати їх безпосередньо на сервері.

Один із способів автентифікації в інформаційній системі полягає у попередній ідентифікації на основі користувацького ідентифікатора і пароля – певної конфіденційної інформації, знання якої передбачає володіння певним ресурсом в мережі. Отримавши введений користувачем логін і пароль, комп'ютер порівнює їх зі значенням, яке зберігається в спеціальній захищеній базі даних і, у випадку успішної автентифікації проводить авторизацію з подальшим допуском користувача до роботи в системі.

### 2.1.2. Пошукові запити і *Query*

Поняття *API* тісно пов’язано з такими поняттями, як пошукові запити, або *query*. Пошуковий запит – елементарна одиниця з функціонального набору для роботи з *API*, яка відповідає набору даних, які будуть отримані після успішного запиту.

Для опису пошукового запиту використовують наступні характеристики:

* метод запиту;
* тип запиту;
* пошукові ключові слова;
* формат запиту.

Метод запиту – один із методів для роботи з *API*. Тип запиту – один із типів дозволених для використання з веб *API*.

Пошукові ключові слова – сукупність ідентифікаторів, по яким відбувається пошук.

В структурі запиту вказуються поля, значення яких є важливими для пошуку, але також важливо вказати і формат в якому буде переданий запит. Найбільш популярний формат даних, який використовується у веб-мережах – це формат *JSON*.

### 2.1.3. Підтримка *JSON*

*JSON* – формат обміну даними, що використовується як альтернатива *XML* та є на третину компактнішим від нього. Широко використовується в складних програмах, які часто обмінюються даними між браузером та веб сервером (технологія *AJAX*). Підтримує чотири примітивних типи даних (*String, Number, Boolean, Null*) та два комплекстних (*Object* – фігурні дужки, *Array* – квадратні дужки).

### 2.1.4. Сортування результатів

Одні і ті ж дані можуть групуватися у відношення різними способами, тобто можлива організація різних наборів відношень взаємопов’язаних інформаційних об’єктів. Групування атрибутів у відношеннях повинно бути раціональним, тобто виключати дублювання даних і спрощувати процедуру їх обробки і оновлення.

Процес упорядкування записів у базі даних називається сортуванням. Порядок сортування записів визначається конкретним завданням. Під час сортування змінюється порядок проходження записів у базі даних чи таблиці. У такий спосіб відбувається зміна бази даних.

*API MovieDB* надає зручні методи для сортування даних. Сортування за зростанням передбачає такий порядок:

* числа;
* текст;
* логічні значення;
* значення помилок;
* порожні комірки.

Сортування за убуванням відбувається в зворотному порядку. Винятком є порожні комірки, що завжди розташовуються в кінці списку. Найшвидше і найпростіше сортування за убуванням чи зростанням проводиться за допомогою відповідних методів *API*.

Під час сортування за зростанням текстові дані впорядковуються за абеткою від А до Я. Числові дані впорядковуються за зростанням значень від мінімального до максимального. Дати впорядковуються від найбільш ранньої дати до найбільш пізньої. У разі вибору сортування за убуванням порядок сортування змінюється на протилежний.

Зазвичай для сортування використовується декілька стандартних алгоритмів з різною ефективністю. На практиці елементи, що впорядковуються, рідко бувають просто числами. Набагато частіше, кожен такий елемент є записом. В кожному записі є ключ, по якому власне і здійснюється впорядкування, в той же час є й інша супутня інформація.

Алгоритм сортування на практиці має бути реалізований так, щоб разом з ключами переміщати і супутню інформацію. Якщо кожен запис містить супутню інформацію великого обсягу, то з метою звести до мінімуму переписування великих обсягів інформації, впорядкування відбувається не у самому масиві елементів, а в масиві вказівників на елементи.

Сам метод сортування не залежить від того, чи впорядковуються тільки числа, чи також і супутня інформація, тому при описі алгоритмів для простоти припускають, що елементи є числами.

### 2.1.5. Фільтрація результатів

Фільтрація списку дає можливість знаходити й відбирати для обробки частину записів у списку, таблиці, базі даних. У відфільтрованому списку виводяться на екран лише ті рядки, що містять певне значення чи відповідають певним критеріям. При цьому інші рядки виявляються схованими.

*API MovieDB* надає багато методів для фільтрації даних. Кінцева вибірка даних може бути відфільтрована будь-яким чином, з використанням абсолютно усіх полів вибірки. Зазвичай фільтрація поділяється на два типи:

* один до багатьох. Фільтрація відбувається з використанням лише одного поля із вибірки даних;
* багато до багатьох. Такий спосіб фільтрації припускає застосування фільтрів по декільком полям із вибірки даних, що в свою чергу ще називають мультифільтром.

## 2.2. Мова запитів *SQL*

*SQL* – це діалогова мова програмування для здійснення запиту і внесення змін до бази даних, а також управління базами даних. Багато баз даних підтримує *SQL* з розширеннями до стандартної мови. Ядро *SQL* формує командна мова, яка дозволяє здійснювати пошук, вставку, обновлення, і вилучення даних, використовуючи систему управління і адміністративні функції. *SQL* також включає *CLI* для доступу і управління базами даних дистанційно.

Різновидів операторів *SQL* є велика кількість, але основні команди майже однакові.

Основні оператори, які використовуються з *SQL*:

* рівність*;*
* нерівність*;*
* менше, більше, менше рівно, більше рівно*;*
* всі значення що знаходяться між даними двома включно*;*
* порівняння з шаблоном*;*
* приймає одне з перелічених значень.

*Movie DB* була спроектована за СКБД *Microsoft SQL Server*.

Мова запитів *SQL* поділяється на три структурні підрозділи:

* *DDL (Data Definition Language);*
* *DML (*[*Data Manipulation Language*](https://ru.wikipedia.org/wiki/DML)*);*
* *DCL (*[*Data Control Language*](https://ru.wikipedia.org/wiki/Data_Control_Language)*).*

### 2.2.1. Види запитів

*Data Definition Language (DDL)* (мова опису баз даних) – це сімейство комп'ютерних мов, які використовуються в комп'ютерних програмах для опису структури [баз *даних*](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=ru&ie=UTF8&prev=_t&rurl=translate.google.com.ua&sl=ru&sp=nmt4&tl=uk&u=https://ru.wikipedia.org/wiki/%25D0%2591%25D0%25B0%25D0%25B7%25D0%25B0_%25D0%25B4%25D0%25B0%25D0%25BD%25D0%25BD%25D1%258B%25D1%2585&usg=ALkJrhh3bcW-zcXyFVKjRhTPjo6RXgRK4w)*.*

Популярною мовою *DDL* є [*SQL*](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=ru&ie=UTF8&prev=_t&rurl=translate.google.com.ua&sl=ru&sp=nmt4&tl=uk&u=https://ru.wikipedia.org/wiki/SQL&usg=ALkJrhhBGr0zEVZm5ISnjn5GNA0sXZnCIQ), яка використовується для отримання і маніпулювання даними в [СКБД](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=ru&ie=UTF8&prev=_t&rurl=translate.google.com.ua&sl=ru&sp=nmt4&tl=uk&u=https://ru.wikipedia.org/wiki/%25D0%25A0%25D0%25A1%25D0%25A3%25D0%2591%25D0%2594&usg=ALkJrhi1aKZKkGg10uQtq2C3WahPoXgcFQ) , і об’єднує в собі елементи *DDL*, [*DML*](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=ru&ie=UTF8&prev=_t&rurl=translate.google.com.ua&sl=ru&sp=nmt4&tl=uk&u=https://ru.wikipedia.org/wiki/DML&usg=ALkJrhghEyRbEzYPR3Pp0F6Egv94HltVjQ) і [*DCL*](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=ru&ie=UTF8&prev=_t&rurl=translate.google.com.ua&sl=ru&sp=nmt4&tl=uk&u=https://ru.wikipedia.org/wiki/Data_Control_Language&usg=ALkJrhgfvfY7tZE24Ied37ISanosm4oLjA).

Функції мов *DDL* визначаються першим словом в запиті. У випадку з *SQL* це *CREATE*, *ALTER*, *DROP*.

Запит «*CREATE*» використовується для створення бази даних, таблиць, індексу, розрізу даних або збереженої процедури.

Запит «*ALTER*» використовується для зміни існуючого об'єкта бази даних (таблиці, індексу, розрізу даних або збереженої процедури) або самої бази даних.

Запит «*DROP*» використовується для видалення існуючого об'єкта бази даних (таблиці, індексу, розрізу даних або збереженої процедури) або самої бази даних.

В *DDL* існують поняття первинного і зовнішнього ключа, які здійснюють дотримання цілісності даних. Команди «первинний ключ» *PRIMARY KEY*, «зовнішній ключ» *FOREIGN KEY* включаються в запити «*CREATE TABLE*», «*ALTER TABLE*».

Мови *DDL* можуть істотно відрізнятися у різних розробників СКБД. Існує розроблений ряд стандартів *SQL*, встановлений [*ISO*](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=ru&ie=UTF8&prev=_t&rurl=translate.google.com.ua&sl=ru&sp=nmt4&tl=uk&u=https://ru.wikipedia.org/wiki/ISO&usg=ALkJrhiPnjyrnn3yFMg9E6ig1-OJH4Dk5A)*/*[*IEC*](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=ru&ie=UTF8&prev=_t&rurl=translate.google.com.ua&sl=ru&sp=nmt4&tl=uk&u=https://ru.wikipedia.org/wiki/IEC&usg=ALkJrhgHcxE1Xnh-_6hREax6mtZHdaULtw)(*SQL*–89, [*SQL*–92](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=ru&ie=UTF8&prev=_t&rurl=translate.google.com.ua&sl=ru&sp=nmt4&tl=uk&u=https://ru.wikipedia.org/wiki/SQL-92&usg=ALkJrhhM0s9Z3jlqgaZF3Kjhmr1sHx9l3g), [*SQL*: 1999](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=ru&ie=UTF8&prev=_t&rurl=translate.google.com.ua&sl=ru&sp=nmt4&tl=uk&u=https://ru.wikipedia.org/wiki/SQL:1999&usg=ALkJrhiMYc53tAT2IC0e7jfNi45YZrLNzA), [*SQL*: 2003](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=ru&ie=UTF8&prev=_t&rurl=translate.google.com.ua&sl=ru&sp=nmt4&tl=uk&u=https://ru.wikipedia.org/wiki/SQL:2003&usg=ALkJrhgWIYgEmLGgUCWRWshRv_m7-m4p9w), [*SQL*: 2008](https://translate.googleusercontent.com/translate_c?act=url&depth=1&hl=ru&ie=UTF8&prev=_t&rurl=translate.google.com.ua&sl=ru&sp=nmt4&tl=uk&u=https://ru.wikipedia.org/wiki/SQL:2008&usg=ALkJrhhSDNHF8Iva3iZFCMtGXCjxfRqsaQ)), але виробники СКБД пропонують свої власні мови і, не підтримують стандарт повністю.

## 2.3. Статичний аналіз типів *Flow*

*Flow* – це статичний аналізатор коду і набір синтаксичних конструкцій, для прямої вказівки типу змінної. *Flow* вміє обчислювати тип змінної, без внесення змін до коду (на відміну від *TypeScript*) що дозволяє почати використовувати його вже зараз в будь-якому проекті. Також є можливість самостійно вказувати типи в стилі *TypeScript*.

У аналізатора *Flow* є три режими:

* не перевіряти нічого (за замовчуванням);
* перевірка без використання анотацій;
* суворі вказівки типу змінних.

Інструменти для статичної типізації, такі як *Flow* або *TypeScript*, дозволяють відловлювати більшу частину помилок ще до виконання коду. Крім того, вони істотно поліпшують процеси розробки, додаючи автодоповнення і інші можливості. Для додатків з великою кодовою базою рекомендується використовувати *Flow* або *TypeScript*.

*Flow* доповнює *JavaScript* власним синтаксисом для вказівки типів, яка не підтримується браузерами. Для того, щоб код працював, потрібно переконатися в тому, що анотації *Flow* коректно видаляються з скомпільованого *JavaScript*.

Для аналізу помилок і видачі спливаючих підказок компілятор *Flow* використовує файли оголошень. Вони містять у собі всю інформацію про типи, які використовуються в конкретній бібліотеці. У свою чергу це дозволяє нам використовувати *JavaScript* бібліотеки в проекті спільно з *Flow*.

Існує два основних способи отримання файлів оголошень:

* *Bundled* – бібліотека встановлюється разом з власним файлом оголошень. Це прекрасний варіант для нас, так як все, що нам потрібно - встановити потрібний пакет. Щоб перевірити, чи є у бібліотеки файл оголошень, здійснюється пошук по індексу;
* *DefinitelyTyped* - це значний репозиторій файлів оголошень. Наприклад, *React* встановлюється без власного файлу оголошення - замість цього ми встановлюємо його окремо;

### 2.3.1. Система типізації

У мові *JavaScript* існує система типізації даних і хоча мова динамічно типізовано, можна виділити основні типи даних, які використовуються.

При створенні змінної, для її назви існує три обмеження:

* не можна використовувати в якості назви змінної зарезервовані слова;
* назви змінних не можуть починатись з цифри;
* ім'я може складатись з: букв, цифр та символів.

Мова *JS* містить шість типів даних:

* *Undefined* (відсутні дані);
* *Null* (нульовий);
* *Boolean* (логічний);
* *String* (рядковий);
* *Number* (числовий);
* *Object* (об'єктний).

Всі типи даних, крім об'єктів є примітивними. Оператор *typeof* повертає тип даних для значення у вигляді рядка. У всіх випадках оператор *typeof* буде повертати вірне значення зі списку крім *null*. У випадку з *null* оператор *typeof* поверне *object*. Це баг мови *JavaScript*. Незважаючи на те, що при перевірці *typeof*, *null* поверне *object*, при перетворенні в тип *boolean* *null* поверне *false*.

Масиви також мають тип даних *object*, тому що фактично масиви і є об'єкти з тією лише різницею, що властивості масиву є нумерований індексами. Якщо порівнювати зі строго типізованими мовами програмування, в *JavaScript* масиви всього лише контейнери для будь-яких типів значень. Це можуть бути значення від рядка, до числа, об'єкта, або навіть іншого масиву.

В *JavaScript* у змінних немає типу, тип є у значень. Змінні ж можуть містити будь-яке значення в будь-який час. В одному випадку це може бути число, в іншому, це може бути рядок і т.д.

### 2.3.2. Анотації типів

Різні операції з пошуку елементів *DOM* повертають список *DOM* елементів, які є масивоподібною колекцією значень з нумерованими індексами. Тому, є можливість конвертації масивоподібного значення в справжній масив і застосування методів масиву до них. Інший приклад – коли функція надає аргументи у вигляді масивоподібного об'єкта, для доступу до аргументів, як до списку значень.

Є поширена думка, що рядки всього лише масиви з символів. Важливо розуміти, що рядки і масиви символів - не одне й те саме. Рядки мають поверхневу схожість з масивами, бо для них працюють властивість .*length*, методи .*indexOf* та .*concat* . Однак, рядки в *JavaScript* незмінні, тоді як масиви можуть змінюватися. Осередки масиву мають числові індекси (як ми зазвичай і очікуємо), але хитрість в тому що масиви також є об'єктами і можуть мати властивість у вигляді рядка, яка не буде враховуватися при підрахунку розміру масиву в властивості .*length*

Для рядка немає методів, які змінюють його вміст на місці, метод краще створить новий рядок і поверне його. Можна позичати методи масивів які не вимагають зміни рядка за місцем або конвертують рядок в масив кожен раз коли це знадобиться.

В *JavaScript* є тільки один числовий тип даних, який називається *number*. Цей тип включає в себе цілі числа і десяткові дроби. В *JavaScript* ціле число – число, яке не має дробової частини.

Оскільки числові значення можуть бути представлені з обгорткою, об'єктом *Number*, з числовими значеннями можна застосовувати методи вбудовані в *Number.prototype*. Зовсім необов'язково використовувати змінні для зберігання числових значень, щоб потім застосувати до них методи. Можна застосувати метод прямо до числового літералу, правда, у випадку з цілими числами потрібно або використовувати дужки, або ставити дві точки, або пробіл тому одна точка відразу після литерала буде інтерпретована як частина числа.

Для невизначених значень в *JavaScript* є два типи даних, які називаються *null* і *undefined*. Часто вважається що вони взаємозамінні, але є нюанси. Незалежно від того, як визначені ці значення, *null* є спеціальним словом, а не ідентифікатором, і його не можна використовувати для призначення змінної. При цьому *undefined* на жаль ідентифікатор. У *non-strict mode* можна створити глобальну змінну з ім'ям *undefined* і присвоїти їй своє значення. У *strict mode* можна зробити те ж саме з локальною змінною. Інший спосіб отримати *undefined* полягає в тому, щоб використовувати оператор *void*. Його потрібно використовувати якщо замість результату виразу доречніше використовувати *undefined*.

Не число (*NaN*) – будь-яка математична операція, в якій один з операндів не є числом, або не може бути інтерпретований як число в результаті нічого очікувати до виконання і в цьому випадку ви отримаєте значення *not a number* або *NaN*. Але, в даному випадку, не число, не зовсім вірне визначення, тому що оператор *typeof* для *NaN* поверне *number*.

*NaN* сигналізує про стан помилки при виконанні дії з числом. Стан помилки по суті говорить, що була спроба виконати математичну операцію в результаті якої було неможливо отримати число. Якщо є необхідність перевірити змінну на *NaN* нічого не вийде, тому *NaN* не дорівнює навіть самому собі, тому для перевірки на *NaN* використовується *isNaN*.

*Infinity* (нескінченність) виходить при перевищенні *Number.MAX\_VALUE* Наприклад, при діленні числа на 0 або у разі перевищення максимальних чисел. Відповідно *Infinity* виходить при розподілі негативного числа на 0. Не можна повернутися від *Infinity* до цілого числа. При розподілі реального числа на *Infinity* отримаємо 0. При розподілі *Infinity* на *Infinity* отримаємо *NaN*.

### 2.3.2. Створення інтерфейсів

В об'єктно-орієнтованому програмуванні протокол або інтерфейс є звичайним засобом для незв'язаних об'єктів спілкуватися один з одним. Це визначення методів та цінностей, з яким об'єкти погоджуються для співпраці.

Наприклад, у *Java*, де протоколи називаються інтерфейсами, інтерфейс *Comparable* визначає метод *compareTo*, який реалізовані класи повинні виконувати. Це означає, що окремий метод сортування, наприклад, може сортувати будь-який об'єкт, який реалізує інтерфейс *Comparable*, без необхідності знати будь-що про внутрішню природу класу.

Функціональне програмування та розподілені мови програмування також мають поняття протоколу, значення якого тонко відрізняється (тобто специфікація дозволеного обміну повідомленнями, акцент на обмін, а не на повідомлення). Ця різниця обумовлена дещо різними припущеннями функціонального програмування та парадигмами об'єктно-орієнтованого програмування.

## 2.4. *React Native*

При використанні *React Native* є чудова можливість використовувати емулятори мобільних пристроїв. Запуск програми на емуляторі, або віртуальному пристрої *Expo CLI* дозволяє запускати додаток *React Native* на фізичному пристрої без створення середовища розробки.

Після налаштування емуляторів можна запускати додаток на віртуальному пристрої *Android*, запустивши *npm run android* або на *iOS Simulator*, запустивши *npm run ios.*

Оскільки ви не створюєте сереовища розробки під час використання *Expo* для створення проекту, неможливо включати власні модулі за межами *API React Native* та компонентів, доступних у клієнтському додатку *Expo.* Якщо до додатку з великою ймовірністю будуть додаватися інші модулі, використання *Expo* може дещо обмежити його потенціал, тож потрібно використовувати вбудовані засоби розробки для *React Native*.

*React Native* схожий на *React*, але він використовує нативні компоненти замість веб-компонентів як будівельні блоки. Отже, щоб зрозуміти основну структуру програми *React Native*, потрібно зрозуміти деякі основні концепції *React*, наприклад, *JSX*, компоненти, стан та реквізит. Створення простого додатку за допомогою *React Native* можна розглянути на цьому прикладі:

*import React, { Component } from 'react';*

*import { Text, View } from 'react-native';*

*export default class HelloWorldApp extends Component {*

*render() {*

*return (*

*<View style={{ flex: 1, justifyContent: "center", alignItems: "center" }}>*

*<Text>Hello, world!</Text>*

*</View>*

*);*

*}*

*}*

В цьому фрагменті коду відбувається імпорт основних методів із *React*, які дають змогу створювати та відображати компоненти, такі динамічні імпорти стали можливими через використання останніх стандартів *JavaScript*, таких як *ES6.*

Наступним кроком імпортуються спеціальні нативні модулі, які відображають текст, а також блоки із даними. Цей імпорт відбувається напряму із фреймворку *React Native*, так як усі елементи, які використовуються є специфічними.

Після цього експортується клас, в якому для відображення використовуються усі імпортовані раніше елементи для того, щоб відобразити на екрані смартфону текст.

## 2.5. *Yoga Engine*

*Yoga Engine –* комп'ютерна програма, спеціальний внутрішній сервіс, який використовується у *React Native* для побудови нативних компонентів.

З допомогою спеціальної, *Yoga Engine* дозволяє перетворювати файли зі стилями у набір координат, який буде оброблений будь-яким пристроєм. Запит до бази *Yoga Engine* може виглядати як в наступному прикладі:

*render() {*

*return (*

*<View style={styles.container}>*

*<AppHeader />*

*<View style={styles.listItems}>*

*{isLoading ? (*

*<ActivityIndicator size="large" color="#0000ff" />*

*) : (*

*<AnimatedFlatList*

*data={movies}*

*keyExtractor={item => String(item.id)}*

*renderItem={({ item, index }) => (*

*/>*

*)}*

*</View>*

*<View style={styles.paginationButtons}>*

*<View>*

*<Button*

*icon={{ name: 'ban', type: 'font-awesome' }}*

*style={styles.paginationButton}*

*onPress={onClearFilters}*

*title="Clear Filters"*

*buttonStyle={{*

*borderRadius: 20, width: 130,*

*}}*

*/>*

*</View>*

*<Button*

*icon={{ name: 'arrow-left', type: 'font-awesome' }}*

*style={styles.paginationButton}*

*onPress={prevPage}*

*title="Previous"*

*disabled={page === 1}*

*buttonStyle={{*

*borderRadius: 20, width: 130,*

*}}*

*/>*

*<Button*

*icon={{ name: 'arrow-right', type: 'font-awesome' }}*

*style={styles.paginationButton}*

*onPress={nextPage}*

*title="Next"*

*buttonStyle={{*

*borderRadius: 20, width: 130,*

*}}*

*/>*

*</View>*

*</View>*

*);*

В результаті виконання такого запиту, назад від *Yoga Engine* пристрій отримає набір даних такого вигляду:

*View(x:120; y:200; z:310)*

*AppHeader(x:180; y:150; z:210)*

*AnimatedFlatList(x:140; y:300; z:250)*

*View(x:120; y:200; z:310)*

*Button(x:150; y:220; z:150)*

*Button(x:160; y:260; z:180)*

*AnimatedFlatList(x:120; y:200; z:310)*

*View(x:120; y:200; z:310)*

За допомогою цієї системи координат пристрій зможе побудувати реальні елементи на своєму екрані. Управління розміткою елементів в інтерфейсі це важлива частина розробки. Причому робиться вона на кожній платформі по різному.

В *iOS*, в основному, через *Auto Layout*, в *Android*, контейнерами або нещодавно з'явилися *ConstraintLayout*. *Yoga*, окрема программа, дозволяє створювати розмітку швидко і відразу для всіх платформ.*Yoga* написана на мові *C* для поліпшення швидкодії і на сьогодні її можна використовувати в *Java*, *Objective-C*, і *C #*.

*Yoga* вже давно використовується в *React Native*, *Components for Android* і *Oculus*, деякі екрани в *Instagram* теж переводяться на *Yoga* і ведеться робота по впровадженню фреймворка в *ComponentKit*.

В цілому, саме завдяки унікальним можливостям *Yoga*, зараз є можливість створювати кромплатформені додатки. Потужності *Yoga* вистачає для того, щоб відображати графічні компоненти на швидкості, яка складає не менше ніж 60 кадрів на секунду. Завдяки цим властивостям, зрозуміти, використовується *Yoga* чи ні, майже неможливо.

## 2.6. *React*

*React –* це *JavaScript* фреймворк для створення інтерфейсів користувача, який використовує синтаксис *JSX* для відображення компонентів. Надає величезну кількість існтрументів для роботи з мобільними додатками, веб-додатками, потужними платформами, веб-запитами, мультимедійним функціоналом, а також анімацією.

*React* використовує той факт, що логіка виводу пов’язана з іншою логікою інтерфейсу користувача, з тим, як обробляються події, як змінюється стан з часом і як дані готуються для рендерингу. Замість того, щоб штучно відокремити технології, розмістивши розмітку і логіку в окремих файлах, *React* розділяє відповідальність між вільно зв’язаними одиницями, що містять обидві технології і називаються компонентами. *React* не вимагає використання *JSX*, але більшість людей цінують його за візуальну допомогу при роботі з інтерфейсом користувача в коді *JavaScript*. Він також дозволяє *React* показати зрозуміліші повідомлення про помилки та попередження.

Типічний код з використанням *JSX*:

*const element = <div tabIndex="0"></div>;*

Більшість *React* додатків мають власні розбиті за бандлами файли використовуючи такі інструменти, як *Webpack*, *Rollup* або *Browserify*. Бандлінг – це процес імпортування файлів та об’єднання їх в один файл – бандл (модуль). Цей бандл може потім бути включений до веб-сторінки для завантаження всього додатку одночасно.

Розбиття по бандлам – це прекрасно, але з темпом росту вашого додатку, бандли теж зростають. Це особливо помітно, якщо було встановлено та використано великі сторонні бібліотеки. Потрібно стежити за кодом, який потрапляє у бандл для того, щоб в один момент не зробити цей бандл настільки великим, що для завантаження додатку знадобиться чимало часу.

Щоб уникнути розростання бандла, варто почати розділяти бандл. Поділ коду - це функціонал, який підтримується такими бандлерами, як *Webpack*, *Rollup* та *Browserify*, що можуть створювати декілька бандлів та завантажувати їх у разі потреби. Цей інструмент може створити кілька модулів з одного, які можна динамічно завантажувати під час виконання вашого основного бандлу.

Розбиття коду додатку може допомогти поступово завантажити тільки те, що необхідно користувачеві в цей момент. Це може значно покращити продуктивність вашого додатку. Хоча кількість коду додатку не була скорочена, уникнуто завантаження коду, який може ніколи не знадобитись користувачеві та скорочено об’єм коду, що необхідний на початку завантаження додатку.

Найкращий спосіб розбиття коду досягається використанням *import*. В той момент, коли *Webpack* стикається з таким синтаксисом, він автоматично починає розбивати код вашого додатку. Якщо при створенні додатку використовувався *Create React App*, це вже налаштовано отже можна одразу почати користуватися цим. Це також підтримується *у Next.js* за замовчуванням. Конфігурація *Webpack* повинна мати вигляд, схожий на цей. Під час використання *Babel*, треба пересвідчитись в тому, що *Babel* може парсити синтаксис динамічних *import*, виключаючи можливість його перетворення. Для цього знадобиться *babel-plugin-syntax-dynamic-import*.

Рішення стосовно того, де саме вводити розбиття коду може бути доволі складним. Потрібно переконатися, обрані місця, які рівномірно розділять бандли, але в жодному разі не порушуватимуть роботу користувача. Вдалим місцем для початку може бути файл з маршрутами додатку. Приклад розбиття бандлів використовуючи бібліотеку *React Router* за допомогою *React.lazy*:

*import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';*

*import React, { Suspense, lazy } from 'react';*

*const Home = lazy(() => import('./routes/Home'));*

*const About = lazy(() => import('./routes/About'));*

*const App = () => (*

*<Router>*

*<Suspense fallback={<div>Завантаження...</div>}>*

*<Switch>*

*<Route exact path="/" component={Home}/>*

*<Route path="/about" component={About}/>*

*</Switch>*

*</Suspense>*

*</Router>*

*);*

Наразі *React.lazy* підтримує тількі експорти за замовчуванням. Якщо модуль, який буде імпортований, використовує іменовані експорти, можна створити проміжний модуль, який повторно експортуватиме його за замовчуванням. Це гарантує, що *tree shaking* продовжить працювати та компоненти, які не будуть використовуватись не підвантажаться.

## 2.7. *Expo*

*Expo –* це набір інструментів, які спрощують створення *React Native* додатків. З *Expo* розробники можуть створювати *React Native* додатки, не відчуваючи при цьому проблем з установкою і налаштуванням залежностей програмного забезпечення таких як *Android Studio*, *Xcode* і інших інструментів, які необхідні для розробки і запуску *React Native* додатків.

*Expo* виступає у ролі фреймворку для *React Native* додатків, які дуже швидко розвиваються. Це як *Laravel* або *Symphony* для *PHP*-розробників, або *Ruby on Rails* для *Ruby* розробників. *Expo* надає поверхневий шар команд *React Native API*, щоб ними було зручніше користуватися і керувати. Він також надає інструменти, які спрощують початкове створення і тестування *React Native* додатків. І нарешті він надає компоненти для користувача інтерфейсу і сервіси, які зазвичай доступні тільки при встановленні сторонніх нативних компонентів *React*. Всі вони доступні через *Expo SDK*.

Деякі обмеження *Expo*:

* *Expo* програми не підтримують фонове виконання коду. Це означає, що ви не можете, наприклад, запустити код, який відслідковує зміни, коли додаток закрито;
* *Expo* додатки обмежені нативними *API*, які підтримує *Expo SDK*. Це означає, що якщо ваш додаток має досить специфічні завдання, як наприклад обмін даними з *Bluetooth* пристроєм, єдиний варіант для реалізації такої функціональності - це використання простого *React Native*, або написання власного коду за допомогою бібліотеки *ExpoKit*;
* *Expo* прив'язує вас до використання його інструментів. Це означає, що ви не можете просто встановити і використовувати більшу частину відмінних інструментів, доступних для розробки *React Native*, таких як інструменти командного рядка, допоміжні бібліотеки і фреймворки користувацького інтерфейсу. Але хороша новина в тому, що *Expo SDK* сумісний з простими бібліотеками *React Native*, тому ви не будете мати ніяких проблем, якщо створите додаток з *Expo*;
* автономні виконувані файли додатків *Expo* можуть бути побудовані тільки при наявності підключення до мережі. *Expo* надає інструменти командного рядка під назвою *Exp*. Це дозволяє розробникам запускати процес компіляції проекту на серверах *Expo*. Після завершення збирання буде доступна *URL* з посиланням для скачування файлу *.apk* або *.ipa*.

Навіть з урахуванням цих обмежень важливо мати на увазі, що *Expo* є повністю функціональним фреймворком з величезною підтримкою часто використовуваних *Android* і *iOS* інтерфейсів *API*. Це означає, що він забезпечить розробника основним функціоналом, який зазвичай потрібен для додатків. Так що для реалізації нативного функціоналу найчастіше вистачить можливостей *Expo.Transitions.*

Для розробки модулю створення кросплатформених додатків було обрано саме фреймворк *Expo*, так як він підтримується сторонніми розробниками і таким чином стає найбільш вигідним рішенням для створення модулю, який постійно буде оновлюватись.

## 2.8*. React Native Bridge*

*React Native Bridge* – це інструмент за допомогою якого і стала можливою побудова кросплатформених додатків. За допомогою *Bridge* додатки *React Native* мають можливість виконувати інструкції в нативному потоці і використовуючи одну і ту саму кодову базу, будувати відображення для будь-яких платформ.

*Bridge* виконує роль прошарку між нативними модулями, а саме кодом на *Java* та *Objective-C,* та кодом *JavaScript*. Все це виконується в абсолютно різних потоках і *Bridge* тут служить у ролі перекладача інструкцій з однієї мови програмування на іншу. Також важливою частиною *Bridge* є описаний вище *Yoga Engine*, який будує відображення компонентів і передає їх у виді координат у нативні потоки.

Схему роботи *Bridge* зображено на рис. 2.1.

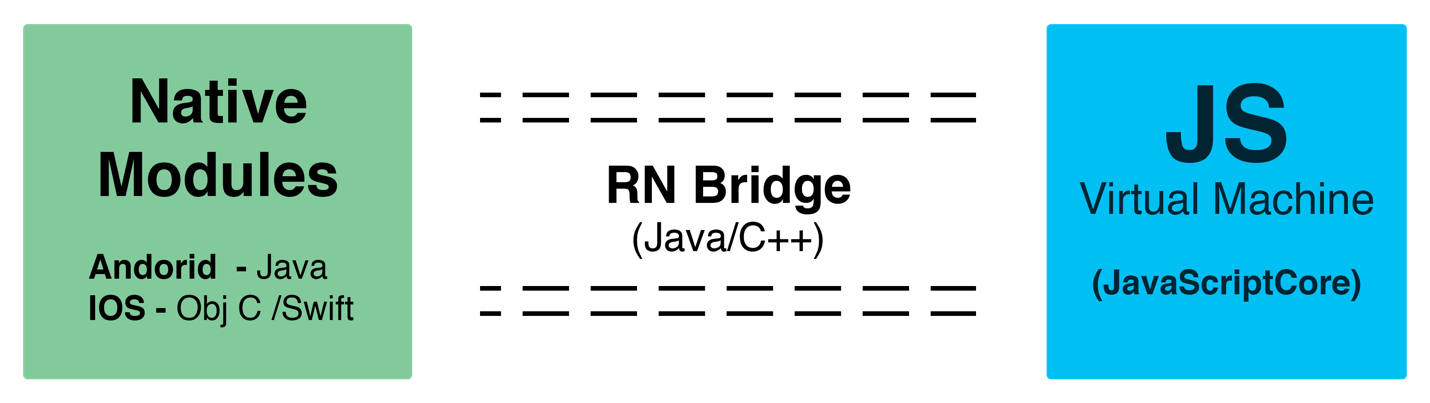


Рис. 2.1. Робота *React Native* *Bridge*

Для реалізації швидкодіючих модулів, які повинні використовувати повні можливості пристрою, доречно створювати нативні модулі и зв’язувати їх з додатками створеними за допомогою *React Native.*

Приклад такого модулю:

*RCT\_EXPORT\_METHOD(logTheObject:(NSDictionary\*) map)*

*{*

*NSString \*value = map[@"key1"];*

*NSLog(@"[NativeModule] %@", value);*

*}*

## 2.9. Висновки до розділу

В розділі було розглянуто поняття *API*, автентифікації, сортування та фільтрації даних, структурні елементи *API*, поняття пошукового запиту, нормалізацію даних за допомогою формату даних *JSON*. Також було описано мову структурних запитів *SQL* для проектування бази даних, таблиць та зв’язків між ними, розглянуто мову *DDL*.

Описано роботу фреймворку *React*, розробленого для створення віртуального *DOM* елементів та швидких веб-додатків.

Розглянуто поняття типів даних, їх використання в мові програмування *JavaScript*. Також було описано систему статичної типізації даних *Flow*, яка дозволяє суворо типізувати *JavaScript* код та зменшити кількість помилок у коді. Описано процес створення інтерфейсів та їх використання.

Описано роботу *React Native* та використання допоміжного фреймворку *Yoga Engine*, для побудови компонентів і відображення їх на екрані будь-якого пристрою.

Розглянуто набір інструментів для швидкої розробки додатків *React Native,* *Expo*, що дозволяє створювати додатки у готовому середовищі розробки, без використання зайвого функціоналу.

Описано роботу *React Native Bridge* – основного інструменту, за допомогою якого код, написаний мовою *JavaScript,* може буте скомпільований та результати виконання цього коду відображено на реальних пристроях, які працюють під управлінням мов програмування *Java* та *Objective-C.*

# РОЗДІЛ 3 РОЗРОБКА МОДУЛЯ СТВОРЕННЯ КРОСПЛАТФОРМЕНИХ ДОДАТКІВ

## 3.1. Постановка задачі

### 3.1.1. Призначення та ціль модуля

Модуль створення кросплатформених додатків повинен реалізовувати основний функціонал по створенню та адаптації додатків під будь-які пристрої та платформи, підвищувати швидкість та якість створення кросплатформених додатків та їх підтримку після створення.

Кросплатформений додаток, що створюється за допомогою даного модулю повинен містити в собі такий функціонал:

* підтримка різних платформ;
* автентифікаційний модуль;
* отримання даних з серверу;
* модуль для зберігання паролів;
* окремі таблиці під кожен параметр;
* список груп на які поділяються вхідні дані;
* фільтрацію та сортування списків;
* список логінів та паролів зареєстрованих користувачів для перевірки сервером та виводу за запитом сервера.

*API* повинна виконувати такі функції:

* виконання запиту клієнта на додавання, редагування та вивід інформації;
* виконання запиту клієнта на отримання інформації для формування списку обраних;
* виконання запиту клієнта на автентифікацію, авторизацію та на зберігання паролю та логіну користувача;
* збереження даних клієнта та обробка їх на стороні серверу, зберігання обраних списків, даних фільтрації та сортування;
* виконання запиту від клієнта на редагування вмісту бази даних *API*;
* виконання запиту клієнта на отримання інформації про статус системи;
* виконання запиту клієнта на отримання необхідної для інформації для розділу фільтрації.

Цілями використання *API* стали:

* збереження даних різних структурних типів;
* забезпечення структурної цілісності даних;
* забезпечення зручності доступу до інформації користувача;
* полегшення фільтрації та сортування даних;
* швидкий пошук даних за запитом клієнта;
* забезпечення раціональної організації даних;

### 3.1.2. Вимоги до компонентів модулю

Модуль створення кросплатформених додатків повинен відповідати таким вимогам:

Структурованість. Головна вимога – модуль повинен бути сформований за єдиним принципом: за компонентами, абстракціями даних, платформно-специфічним функціоналом.

Зручність у користуванні. Модулем буде зручно користуватися, якщо він буде складений у вигляді окремих підмодулів зі зрозумілими вимогами до побудови додатків. Візуальне сприйняття компонентів полегшить роботу. Список кросплатформених компонентів повинен бути зв’язаним списком у корені модулю. Це заощадить час на пошуки потрібного компонента або кросплатформених рішень.

Максимальна повнота інформації. Інформація, що міститься в модулі створення кросплатформених додатків, повинна обов'язково включати такі інформаційні елементи: документація, сторонні бібліотеки, компоненти, стильові файли.

Регулярне поновлення. Модуль створення кросплатформених додатків необхідно постійно поновлювати. Усі бібліотеки та фреймворки, за допомогою яких побудовано модуль, повинні відповідати останнім стандартам. Оновлення бібліотек та фреймворків рекомендовано провадити по мірі того як виходять їх нові стабільні версії.

Регулярне поповнення. На відміну від оновлення фреймворків та бібліотек, конкретних нормативів щодо цього процесу не існує. У залежності від обставин до модулю можна вносити нові бібліотеки та фреймворки, які розширюють його функціонал практично щодня. Слід лише зберігати початкову структуру модулю, або розробляти нову, якщо цього вимагатимуть нововнесені фреймворки та бібліотеки.

### 3.1.3. Вимоги до серверної частини додатку

*API*, яке може обробляти запити клієнта і працює з кількома варіантами попередньо встановленого програмного забезпечення на серверній стороні:

* *SQL* (рекомендується) версії 4.4 і вище*;*
* *IIS (Internet Information Server)* (можливе встановлення) – робота продукту можлива з *IIS 5, IIS 6 і IIS 7 і 7.5.* Потрібно додаткове налаштування для коректної роботи з продуктом.
* *Elastic Search (*можливе встановлення*) –* продукт протестований для спільної роботи з веб-сервером для розумного пошуку.
* *Java.* Для роботи продукту потрібна наявність *Java* версії не нижче *8.0.х.* Використання версії *Java* нижче вказаної також можливе так як мова надає можливість зворотньої сумісності синтаксису але будуть виникати помилки в середовищі розробки які супроводжуватимуться підсвічування і відображенням повідомлень про помилку що суттєво знизить зручність роботи, а також вплине на коректність виконання програми;
* *Java XML –* використовується для роботи системи оновлень. Бібліотека за замовчуванням включена в стандартному встановленні *Java.* Для версії під всі ОС вбудована підтримка*.*

### 3.1.4. Вимоги до клієнтської частини додатку

* *React Native;*
* *JavaScript;*
* *React –* конструктор відображень для пришвидшення побудови *HTML сторінки;*
* *Expo –* технологія використовується для побудови кросплатформених додатків. Завдяки цій технології побудова і підтримка кросплатформених додатків стає гнучкішою*;*
* *Yoga Engine –* фреймворк для побудови графічного інтерфейсу користувача*;*
* *React Native Bridge –* прошарок за допомогою якого є можливість створювати кросплатформені додатки і використовувати їх на мобільних пристроях*;*
* *Flow –* набір засобів для статичної типізації *JavaScript;*
* *SOLID –* архітектурний принцип, за допомогою якого розробляється набір компонентів та архітектура додатку*;*
* *REST -* додаток використовує стандарт *REST* для організації обміну даними додатку з серверами *API;*
* підтримка різних платформ – програмний продукт розроблений з урахуванням підтримки найбільш поширених мобільних та веб платформ. Кодова база оптимізована для роботи з ними. Відображення користувацької частини додатку не залежить від типу пристрою*.*

## 3.2. Серверна частина та *API* додатку

Для створення бази даних та її об’єктів на стороні *API* використано *DDL* оператор мови *SQL* – *CREATE*.

Стандарт *SQL*–*2011* визначає оператор *CREATE* у варіантах:

* *CREATE ASSERTION –* створення твердження;
* *CREATE CHARACTER SET –* створення набору символів;
* *CREATE COLLATION –* створення набору правил сортування символів;
* *CREATE DOMAIN –* створення домену (користувацького типу даних стовпця);
* *CREATE SCHEMA* – створення схеми (іменованої групи об’єктів);
* *CREATE TABLE –* створення таблиці бази даних;
* *CREATE TRANSLATION –* створення правил перетворення з одного набору символів в інший;
* *CREATE VIEW –* створення розрізу даних;
* *CREATE DATABASE –* створення бази даних.

База даних працює із стандартним *REST* протоколом для зв’язку з клієнтом, саме цей стандарт надає можливість стандартизувати усі запити до серверу та відповіді клієнту.

*{*

*"poster\_path": "/IfB9hy4JH1eH6HEfIgIGORXi5h.jpg",*

*"adult": false,*

*"overview": "Jack Reacher must uncover the truth behind a major government conspiracy in order to clear his name. On the run as a fugitive from the law, Reacher uncovers a potential secret from his past that could change his life forever.",*

*"release\_date": "2016-10-19",*

*"genre\_ids": [*

*53,*

*28,*

*80,*

*18,*

*9648*

*],*

*"id": 343611,*

*"original\_title": "Jack Reacher: Never Go Back",*

*"original\_language": "en",*

*"title": "Jack Reacher: Never Go Back",*

*"backdrop\_path": "/4ynQYtSEuU5hyipcGkfD6ncwtwz.jpg",*

*"popularity": 26.818468,*

*"vote\_count": 201,*

*"video": false,*

*"vote\_average": 4.19*

*}*

За допомогою стандартизованого вигляду відповіді від серверу, можна легко конфігурувати запити.

### 3.2.1. Створення системи автентифікації

Будь-який додаток, який працює з користувачами передбачає наявність модулю для автентифікації. Модуль для автентифікації складається з коритувацького інтерфейсу для вводу даних, серверної частини, яка працює з користувацькими даними і на основі введених користувачем паролю та логіну, надає клієнту спеціальний ключ, з яким клієнт може робити запити до *API* та отримувати доступну для нього інформацію із бази даних.

База даних для модулю автентифікації складається з декількох таблиць, в яких в зашифрованому вигляді зберігаються користувацькі дані. Однією із головних складових частин автентифікаційного модулю є робота із ключами для запитів, які поділяються на декілька типів. Ключ може бути аксессором та мати обмежену тривалість використання, яка задається сервером. Також ключ може бути ключом оновлення і його основна функція отримати новий ключ аксессор після того, як тривалість використання аксессора закінчиться. У ключа оновлення також є тривалість використання, але вона значно більша ніж у ключа аксессора. Якщо і ключ оновлення більше не валідний, треба використовувати логін та пароль користувача для того, щоб знову отримати ключі доступу.

Для створення автентифікаційного модуля було використано спеціальний класс *AUTHENTICATE.*

*return (*

*<View style={styles.container}>*

*<View style={styles.form}>*

*<View>*

*<FormLabel>Login</FormLabel>*

*<FormInput*

*placeholder="User"*

*value={username}*

*onChangeText={(inputValue) => {*

*onChangeInput({ name: 'username', value: inputValue });*

*}} />*

*{errors.username && (*

*<FormValidationMessage>{errors.username}</FormValidationMessage>*

*)}*

*</View>*

*<View>*

*<FormLabel>Password</FormLabel>*

*<FormInput*

*placeholder="User"*

*value={password}*

*onChangeText={(inputValue) => {*

*onChangeInput({ name: 'password', value: inputValue });*

*}}*

*/>*

*{errors.password && (*

*<FormValidationMessage>{errors.password}</FormValidationMessage>*

*)}*

*</View>*

*</View>*

*</View>*

*);*

За допомогою оператора *View* створено користувацьке відображення форми для заповнення автентифікаційних даних.

Для автентифікації потрібно указати: *username* – логін користувача, *password* – пароль користувача.

За допомогою *FormValidationMessage* створено розумну валідацію помилок на клієнтській стороні, яка передбачає валідацію логіну та паролю. Також за допомогою оператора *onChangeText* дані були записані у систему.

### 3.2.2. Використання методів *API*

Для зручної роботи з методами *API* було створено допоміжний класс, який формує запити від клієнта до бази даних.

*import queryString from "query-string";*

*export const API\_URL = "https://api.themoviedb.org/3";*

*export const API\_KEY\_3 = "bfa75e91c5cda1d779e8278820b69bd8";*

*export const API\_KEY\_4 =*

*"eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJhdWQiOiJiZmE3NWU5MWM1Y2RhMWQ3NzllODI3ODgyMGI2OWJkOCIsInN1YiI6IjViZjNlYWQ2OTI1MTQxNWNjNDA4ZDM5YyIsInNjb3BlcyI6WyJhcGlfcmVhZCJdLCJ2ZXJzaW9uIjoxfQ.DbtCepBrnTouEdMwZknPyD04iUceWQmStD1vy28Trq8";*

*export const fetchApi = (url, options = {}) => {*

*return new Promise((resolve, reject) => {*

*fetch(url, options)*

*.then(response => {*

*if (response.status < 400) {*

*return response.json();*

*} else {*

*console.log("ERROR");*

*throw response;*

*}*

*})*

*.then(data => {*

*resolve(data);*

*})*

*.catch(response => {*

*response.json().then(error => {*

*reject(error);*

*});*

*});*

*});*

*};*

*export default class CallApi {*

*static get(url, options = {}) {*

*const { params = {} } = options;*

*const queryStringParams = {*

*api\_key: API\_KEY\_3,*

*...params*

*};*

*return fetchApi(*

*`${API\_URL}${url}?${queryString.stringify(queryStringParams)}`,*

*{*

*mode: "cors",*

*headers: {*

*"Content-type": "application/json"*

*}*

*}*

*);*

*}*

*static post(url, options = {}) {*

*const { params = {}, body = {} } = options;*

*const queryStringParams = {*

*api\_key: API\_KEY\_3,*

*...params*

*};*

*return fetchApi(*

*`${API\_URL}${url}?${queryString.stringify(queryStringParams)}`,*

*{*

*method: "POST",*

*mode: "cors",*

*headers: {*

*"Content-type": "application/json"*

*},*

*body: JSON.stringify(body)*

*}*

*);*

*}*

*static delete(url, options = {}) {*

*const { params = {}, body = {} } = options;*

*const queryStringParams = {*

*api\_key: API\_KEY\_3,*

*...params*

*};*

*return fetchApi(*

*`${API\_URL}${url}?${queryString.stringify(queryStringParams)}`,*

*{*

*method: "DELETE",*

*mode: "cors",*

*headers: {*

*"Content-type": "application/json"*

*},*

*body: JSON.stringify(body)*

*}*

*);*

*}*

Наведений вище скрипт формує новий запит з клієнтської сторони до серверу та автоматично оброблює помилки, які можуть виникнути у ході роботи асинхронного запиту. Метод *fetch* приймає першим аргументом посилання, а другим приймає додаткові параметри, за допомогою яких і виконується кастомізація запиту.

Метод *catch* оброблює помилки, які можуть виникати в результаті асинхронного запиту. Метод виводить помилки на екран, що допомагає зрозуміти їх сутність.

### 3.2.3. Створення методів для роботи із фільтрацією даних

Для представлення роботи з фільтрацією та сортуванням даних *API* надає спеціальний набір опціональних параметрів, які можна передати разом із посиланням використовуючи метод *fetch*. За допомогою цих параметрів на стороні серверву можно налагодити автоматичну фільтрацію та сортування елементів.

За допомогою технологій *React* можна створитимодель для відображення відсортованих та відфільтрованих даних проекту, що складається з підкомпонентів, у вигляді списків з елементами та перемикачів, за допомогою яких відбувається фільтрація та сортування даних.

Створені відображення даних та перемикачів представлені у вигляді *JSX* –коду.

*return (*

*<View>*

*<RNPickerSelect*

*style={{ height: 5, width: 200 }}*

*itemStyle={{*

*color: 'blue',*

*fontSize: 17,*

*}}*

*onValueChange={this.handleChangeSortBy}*

*value={filters.sort\_by}*

*items={optionsSortBy}*

*style={{ ...pickerSelectStyles }}*

*/>*

*</View>*

*);*

За допомогою *RNPickerSelect* створюється перемикач, який дає користувачу можливість вибирати фільтри зі списку, а також сортувати вхідні дані.

### 3.3. Клієнтська частина додатку

Клієнтська частина додатку складається із декількох підмодулів, частина з яких є функціональними. Функціональні модулі виконують запити до *API*, отримують дані, організовують їх у потрібному і зрозумілому для *React* порядку та записують їх у стан системи. Модулі, які не несуть ніякого функціонального впилву, є репрезентативними та виконують функції відображення даних, та взаємодії з користувачем. Прикладом функціонального модуля, є модуль для автентифікації, який виконує роботу пов’язану з реєстрацією та авторизацією користувача до системи. Прикладом репрезентативного модулю можно назвати список, який можно гортати за допомогою жестів, чи відображення перемикачів для фільтрації та сортування данхи. Клієнтська частина додатку реалізує кросплатформеність та дозволяє встановлювати або запускати додаток за допомогою будь-якої платформи чи операційної системи.

В цілому, клієнтська частина реалізує два модулі, модуль обробки даних користувача, та візуалізації даних отриманих за допомогою *API*.

Модуль обробки даних є основним. Він відповідає за обробку користувацьких даних. Модуль реалізований у вигляді окремого класу, у якого є набір методів для роботи з даними користувача.

Модуль візуалізації відповідає за представлення даних користувачу у вигляді відображень, де користувач може взаємодіяти з інформацією, сортувати та фільтрувати дані, використовуючи перемикачі фільтрації.

### 3.3.1. Кросплатформені модулі

Усі модулі, які входять до клієнтської частини додатку реалізовані за допомогою модулю створення кросплатформених додатків і є кросплатформеними.

Кожен з модулів, які входять до додатку має свій власний набір функціоналу, потрібний для роботи під різними платформами і операційними системами:

*import React from "react";*

*import Favorite from "../Movies/Favorite";*

*import WatchList from "../Movies/WatchList";*

*import { Link } from "react-router-dom";*

*export default class MovieItem extends React.Component {*

*render() {*

*const { item } = this.props;*

*return (*

*<div className="card" style={{ width: "100%" }}>*

*<img*

*className="card-img-top card-img--height"*

*src={`https://image.tmdb.org/t/p/w500${item.backdrop\_path ||*

*item.poster\_path}`}*

*alt=""*

*/>*

*<div className="card-body">*

*<Link className="card-title" to={`/movie/${item.id}/detail`}>*

*{item.title}*

*</Link>*

*<div className="card-text">Рейтинг: {item.vote\_average}</div>*

*<Favorite item={item} />*

*<WatchList item={item} />*

*</div>*

*</div>*

*);*

*}*

*}*

Функція виконується та повертає відображення у вигляді класичної веб-сторінки, так як модуль для створення кросплатформених додатків визначив, що це єдиний спосіб відображення даних для даної системи та платформи. Далі, якщо потрібно запустити додаток на мобільному пристрої та відобразити ті ж самі дані, *JSX* код буде підлаштовуватися під інші платформи.

*import React from 'react';*

*import {*

*Card, Button, Icon,*

*} from 'react-native-elements';*

*import {*

*View, Image, StyleSheet, Dimensions, Animated,*

*} from 'react-native';*

*const SCREEN\_WIDTH = Dimensions.get('window').width;*

*class MovieItem extends React.Component {*

*render() {*

*const { item } = this.props;*

*return (*

*<View*

*style={styles.constainer}*

*>*

*<Animated.View*

*style={this.props.style}*

*>*

*<Card*

*title={item.title}*

*borderRadius={20}*

*>*

*<Image*

*style={styles.image}*

*resizeMode="cover"*

*source={{ uri: `https://image.tmdb.org/t/p/w500${item.backdrop\_path || item.poster\_path}` }}*

*/>*

*<Button*

*icon={<Icon name="visibility" color="#ffffff" />}*

*backgroundColor="#03A9F4"*

*buttonStyle={{*

*borderRadius: 10,*

*}}*

*title="Детально"*

*/>*

*</Card>*

*</Animated.View>*

*</View>*

*); }}*

*export default MovieItem;*

*const styles = StyleSheet.create({*

*constainer: {*

*width: SCREEN\_WIDTH,*

*},*

*image: {*

*width: '100%',*

*height: 400,*

*marginBottom: 20, },*

*});*

Тут виконується та ж сама функція, але тепер із використанням *React Native* для відображення на мобільних пристроях. Додаток автоматично підлаштовується під вимоги платформи та операційної системи.

### 3.3.2. Клієнтський *UI* інтерфейс

Клієнтський *UI* інтерфейс додатку побудований з використанням кросплатформених модулів, тож усі графічні елементи перевикористовуються, кодова база не змінюється.

Основна функція *render(),* викликається при вході користувачем в кожен окремий компонент. Функція при першому вході виводить користувачу дані, які були описані у форматі *JSX*, у зрозумілому користувачу вигляді, а саме у вигляді графічного інтерфейсу додатку. Інтерфейс відображає ключові екрани додатку, а саме екрани з функціями фільтрації, сортування, відображення даних із навігацією по сторінкам, модулі автентифікації та авторизації.

*import React from 'react';*

*import { Button } from 'react-native-elements';*

*import { inject, observer } from 'mobx-react';*

*import {*

*StyleSheet, ActivityIndicator, View, Dimensions, Animated, PanResponder, FlatList,*

*} from 'react-native';*

*import MovieItem from './MovieItem';*

*import AppHeader from '../../shared/AppHeader';*

*const AnimatedFlatList = Animated.createAnimatedComponent(FlatList);*

*const SCREEN\_WIDTH = Dimensions.get('window').width;*

*const x = new Animated.Value(0);*

*const transitionAnimation = index => ({*

*@inject('moviesPageStore', 'userStore')*

*@observer*

*class MoviesScreen extends React.Component {*

*componentDidMount() {*

*this.props.moviesPageStore.getMovies();*

*this.props.userStore.getUserFromStore();*

*}*

*render()*

*return (*

*<View style={styles.container}>*

*<AppHeader />*

*<View style={styles.listItems}>*

*{isLoading ? (*

*<ActivityIndicator size="large" color="#0000ff" />*

*) : (*

*<AnimatedFlatList*

*style={{ width: SCREEN\_WIDTH }}*

*horizontal*

*showsHorizontalScrollIndicator={false}*

*pagingEnabled*

*scrollEventThrottle={16}*

*onScroll={Animated.event(*

*[{ nativeEvent: { contentOffset: { x } } }],*

*{ useNativeDriver: true },*

*)}*

*data={movies}*

*/>*

*)}*

*</View>*

*<View style={styles.paginationButtons}>*

*<View>*

*<Button*

*icon={{ name: 'ban', type: 'font-awesome' }}*

*style={styles.paginationButton}*

*onPress={onClearFilters}*

*title="Clear Filters"*

*buttonStyle={{*

*borderRadius: 20, width: 130,*

*}}*

*/>*

*</View>*

*<Button*

*icon={{ name: 'arrow-left', type: 'font-awesome' }}*

*style={styles.paginationButton}*

*onPress={prevPage}*

*title="Previous"*

*disabled={page === 1}*

*buttonStyle={{*

*borderRadius: 20, width: 130,*

*}}*

*const styles = StyleSheet.create*

*container: {*

*flex: 1,*

*alignItems: 'center',*

*justifyContent: 'center',*

*marginTop: 20,*

*},*

*listItems: {*

*flex: 6,*

*width: '100%',*

*margin: 10,*

*},*

*paginationButtons: {*

*flexDirection: 'row',*

*justifyContent: 'space-around',*

*width: '100%',*

*marginTop: -10,*

*height: 50}*

*});*

## 3.6. Висновки до розділу

В третьому розділі, згідно з розглянутими матеріалами стосовно кросплатформених додатків, мов програмування, інструментів створення кросплатформених функцій, було сформульовано відповідні вимоги до *API*, бази даних додатка, куди буде заноситись декодована інформація про користувачів. Сформульовано вимоги щодо серверної частини та обрано базові технології розробки, що забезпечать швидкість та надійність обробки користувацьких даних. Сформульовано основні вимоги до клієнтської частини. Описано технологію *React Native* на основі *Expo*, що дає змогу виконувати додаток під будь-якою платфоромю і операційною системою.

Також було розроблено автентифікаційний модуль, що призначений для збереження всіх даних користувачів, оброблених системою, та на прикладі описано оператори та графічний інтерфейс, що дають змогу автентифікуватися до додатку та робити запити до серверної частини *API*.

В розділі наведено опис всіх основних методів, за допомогою яких реалізована кросплатформність. До таких методів відносяться методи для створення кросплатформених компонентів за допомогою *JSX*, оптимізація додатку для роботи з будь-якими операційними системами та платформами.

Також було наведено основні фрагменти коду за допомогою яких відбувається автентифікація, відображення основних елементів та списків, представлення перемикачів для фільтрації та сортування даних, форма для вводу даних користувача та надання прав доступу до *API*, задання зв’язків між клієнтом і сервером.

Було наведено фрагменти коду методів *API*, які відповідають за зв’язок клієнту із сервером та полегшують роботу із запитами, приймаючи опціональні параметри у вигляді аргументів.

# РОЗДІЛ 4 ВИКОРИСТАННЯ КРОСПЛАТФОРМЕНОГО ДОДАТКУ

## 4.1. Вимоги до апаратно-програмного забезпечення

Вимоги до апаратно програмного забезпечення – це опис апаратної та програмної платформ, що забезпечують роботу та підтримку додатку, опис інтерфейсів комунікації з іншими пристроями, операційні вимоги, що забезпечать організацію роботи додатку та збереження даних на сервері.

Вимоги для функціонування додатку діляться на дві категорії:

* локальні вимоги – рекомендований мінімальний набір апаратного та програмного забезпечення для функціонування на пристроях розробника додатку для його коректного та швидкого функціонування, для зручності додавання та розширення функціоналу. Для кросплатформених додатків локальні вимоги включають в себе не тільки персональні комп’ютери, а й мобільні пристрої;
* мережеві вимоги – рекомендований мінімальний набір апаратного та програмного забезпечення для функціонування серверної частини, куди буде надходити великий потік даних, а також буде відбуватись обробка запитів клієнта з подальшою вибіркою даних із БД.

Зазвичай мережеві вимоги до апаратного забезпечення відрізняються від локальних, так як додаток на сервері працює з великою кількістю користувацьких підключень та великою кількістю вхідної інформації для обробки, але у нашому випадку, створений додаток використовує *API* для роботи з даними, яке працює з надпотужними віддаленими серверами.

Локальні вимоги в свою чергу поділяються на два типи:

* персональні комп’ютери;
* мобільні пристрої;

Вимоги до локального апаратного забезпечення ПК:

* процесор *Intel Cele*ron – з тактовою частотою 2.2 *ГГц* (мінімум 4 ядра);
* оперативна пам'ять не менше 2 *Гб* (за умови що ПК цілком буде виділено під роботу із додатком, у іншому випадку об’єм має бути збільшений до 4 *Гб* для комофортної роботи користувача), з них на браузер має бути виділено мінімум 1 *Гб*;
* вільне місце на жорсткому диску - не менше 10 *Гб*, з них має бути виділено мінімум 500 *МБ* на користувача додатку та мінімум 2 *Гб* на збереження даних які можуть буду збережені браузером;
* доступ до мережі Інтернет.

Вимоги до апаратного забезпечення мобільного пристрою:

* будь-який мобільний процесор з тактовою частотою 1 - 2.5 *ГГц*;
* оперативна пам'ять не менше 2 *Гб*;
* вільне місце на пристрої – не менше 2 *Гб*, з них 500 *Мб* для даних додатку та 1 *Гб* для файлів які можуть бути збережені у процесі роботи;
* кольоровий графічний дисплей (розподільча здатність 1024 х 768);
* сенсорні органи управління;
* доступ до мережі Інтернет.

Вимоги до програмного забезпечення для функціонування додатку для ПК та мобільних пристроїв відрізняються.

Вимоги до програмного забезпечення для ПК:

* будь-який браузер, який підтримує відображення веб-сторінок з версією *JavaScript ES*3;
* *Windows*, *MacOS* чи *Linux* будь-якої версії;
* *SQL Server* – для роботи додатка та користувача з базою даних, система використовує об’єкт програмного підключення до бази даних, який налаштовується глобально для всієї системи, після чого буде можливим виклик цього об’єкту та використання його функціоналу локально в методах і класах системи для виконання операцій над даними. Користувачу повинен бути відкритий доступ до бази даних тільки за допомогою системи підключення *API*, через *ssh* підключення, тільки для перегляду структури бази даних та інформації. Всі інші операції по зміні структури бази даних та додавання, редагування записів повинні виконуватися тільки за допомогою спеціального програмного забезпечення;
* *Javascript Engine –* повинен бути сконфігурований вбудований до операційних систем елемент для запуску додатків, які використовують *JavaScript* та *React*, порт 9080 вільний для побудови додатку.

До програмних вимог серверу ще додаються операційні вимоги, що забезпечують захист роботи серверу та даних, що використовуються.

Вимоги до програмного забезпечення для мобільних пристроїв:

* операційна система *iOS* чи *Android*;
* доступ до *AppStore* чи *Google* *Playmarket* для встановлення, використання та оновлення додатку;
* доступ до *LocalStorage* мобільних пристроїв для збереження автентифікаційних даних.

## 4.2. Запуск додатку на виконання

*React* та *React Native* за допомогою яких описано додаток, запускають два різні *JavaScript-*сценарії на виконання.

Перший сценарій призначений для визначення необроблених файлів, їх вибору, компіляції коду та створення спеціальної збірки, яка називається бандлер. Сценарій запускається за допомогою *Native* команди на пристрої при старті додатку кожну та виконує необхідні маніпуляції з даними та збіркою бандлеру.

*JS Bundler –* класичний мініфікатор (додаток до компілятору), що використовує код, написаний на мові зрозумілій розробнику, мініфікує його, зменшуючи розмір інструкцій. Такий код зрозумілий лише компілятору і не може бути прочитаний розробником, але він значно швидший та його можна зібрати в спеціальний бандлер, який буде виконуватися *JavaScript* двигуном.

Файли, які будуть зібрані в бандл, розташовані в каталозі */usr/app/src* або */usr/app/node-modules*. Конфігураційні файли знаходяться в каталозі */usr/app/app.json.* Редагувати їх вручну не рекомендується, для цього використовують команду *expo config*. Файли *package.json*, використовувані для управління додатком і його залежностями, розташовуються в каталозі */usr/app/package.json*. Також файли можуть бути переміщені в будь який зручний користувачу каталог, для цього потрібно буде додати шлях файлу *package* до конфігурації *app.json*.

Базови файл конфігураці *app.json* файлу виглядає як показано на рис. 4.1.

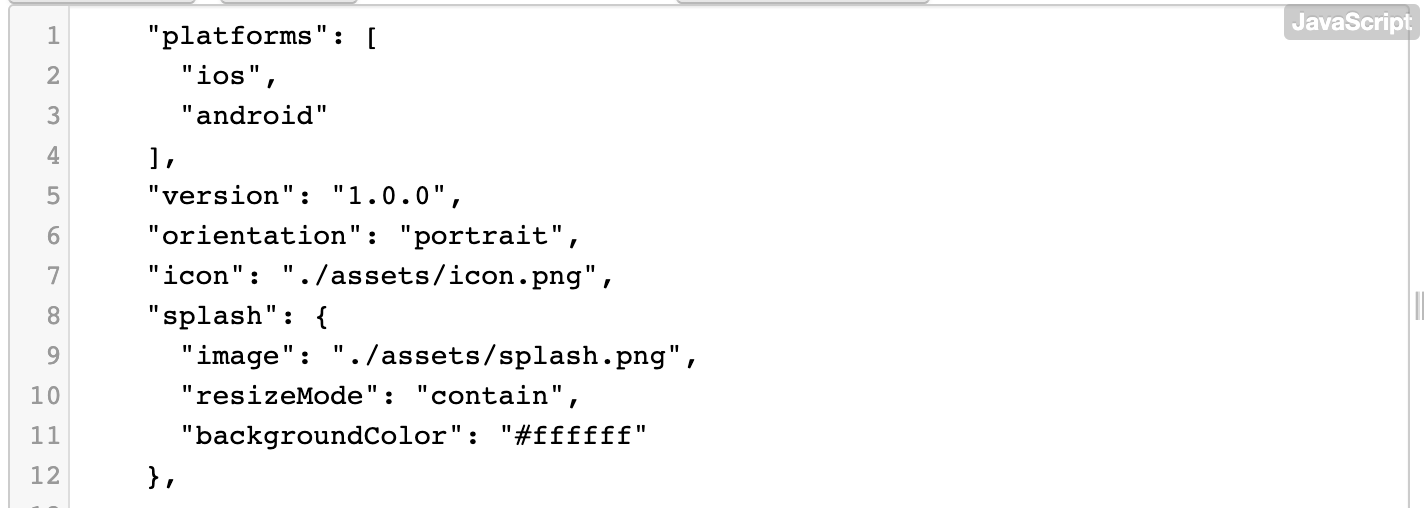


Рис. 4.1. Конфігурація *app.json* файлу

Кожен користувач додатку має свій файл *package.json*, в якому описано, в який час і які залежності включати до додатку від імені цього користувача. Для редагування файлу *package.json* використовується спеціальний редактор коду, який підтримує файли *JSON* формату, що дозволяє не переривати процес підключення залежностейна час редагування. Файл *package.json* складається з об’єктів, поділених пробілами або табуляторами. Перший об’єкт задає основні налаштування додатку, такі як шлях до файлу, з якого починається збірка бандлеру, скрипти для запуску *iOS* або *android* версій. Об’єкт із залежностями додатку містить у собі інформацію про всі залежності у вигляді назв бібліотек та їх версій. У потрібний момент часу бібліотеки підключаються до додатку та виконують свої функції.

Другий сценарій служить для взаємодії з клієнтом на стороні пристрою, керує запуском першого сценарію та обробкою запитів користувача до додатку та відображенням вибраної користувачем інформації на екрані.

Клієнт завантажує додаток за допомогою *Expo* у випадку, якщо він використовує мобільний пристрій, або запускає його за допомогою браузеру та входу на , якщо використовується ПК за допомогою входу на домен *muryj-movie-app.herokuapp.com.*

## 4.3. Робота користувача з додатком

Користувачу при переході на домен *muryj-movie-app.herokuapp.com* відкривається головне вікно додатку з усіма елементами управління, зображено на рис. 4.2.

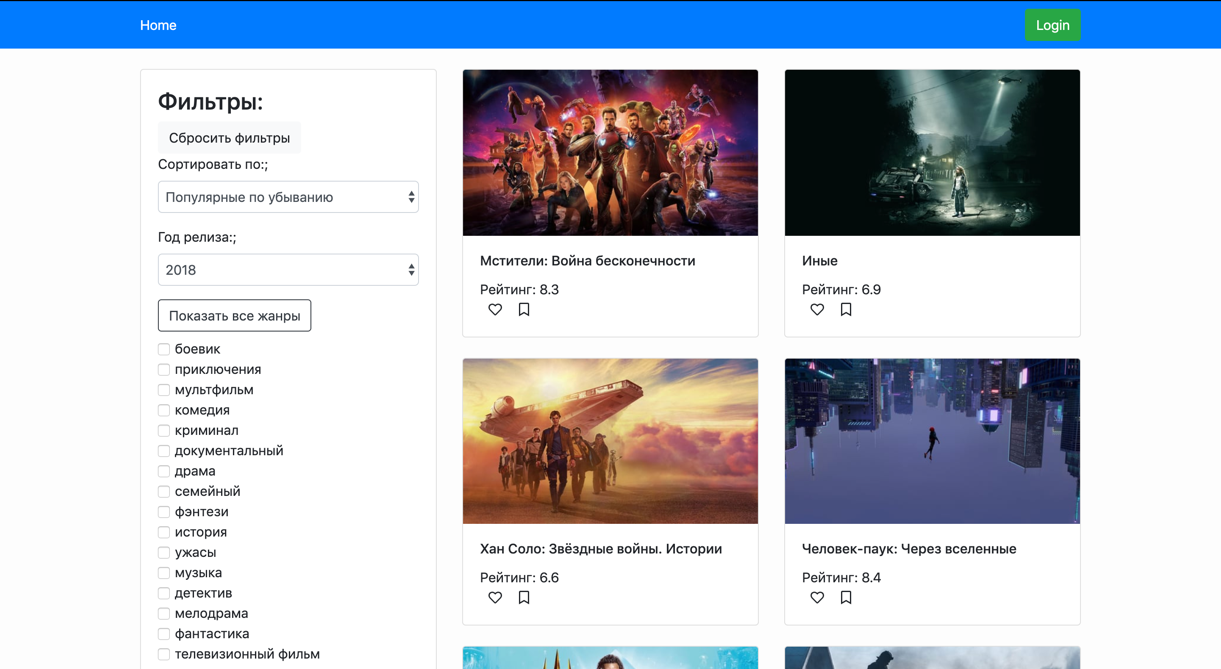


Рис. 4.2. Головне вікно додатку

Нові користувачі додаються через адміністратора системи, за допомогою *MovieDB* *API,* права доступу, додавання та видалення користувачів здійснює тільки адміністратор системи, у звичайного користувача прав доступу до адміністрування немає з міркувань безпеки та збереження режимної інформації.

Після успішного створення аккаунту, користувач може авторизуватися використовуючи форму для авторизації, яку зображено на рис 4.3.

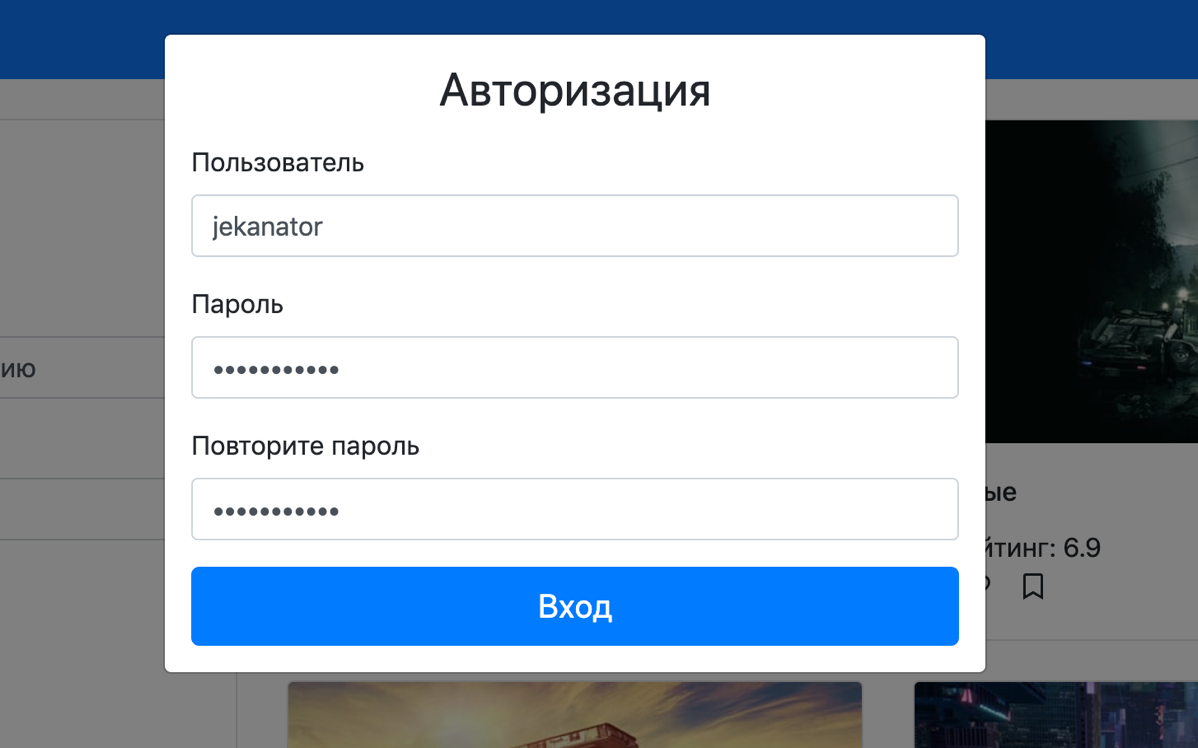


Рис. 4.3 Форма авторизації

Користувач обирає метод сортування, згідно якому буде здійснюватися вибірка даних з серверу. Після чого задає рік за який буде показано інформацію. Після цього користувач має можливість обрати жанри для відображення на головному екрані та здійснювати навігацію по-сторінково. На даний момент доступно 500 сторінок із інформацією для кожного року. Навігація по сторінкам дозволяє не відображати та не завантажувати одразу усі дані, тим самим оптимізує роботу додатку. Також створений функціонал для того, щоб скинути фільтрацію по жанрам та сортування за допомогою спеціальних кнопок.

Далі користувачу надається можливість обрати улюблені фільми та додати їх до обраних. Такий функціонал працює тільки для зареєстрованих та авторизованих користувачів. Усі обрані та улюблені фільми зберігаються у базі даних, у той час, як фільтрація та сортування даних є опціональними параметрами та будуть скинуті при оновленні веб-сторінки. Інформація про авторизованого користувача зберігається у зашифрованому сховищі мобільного пристрою, чи браузера. Після налаштування фільтрів та сортування, користувачу виведеться на екран набір даних згідно з його обраними критеріями, де будуть відмічені його улюблені та обрані фільми, це зображено на рис. 4.4.

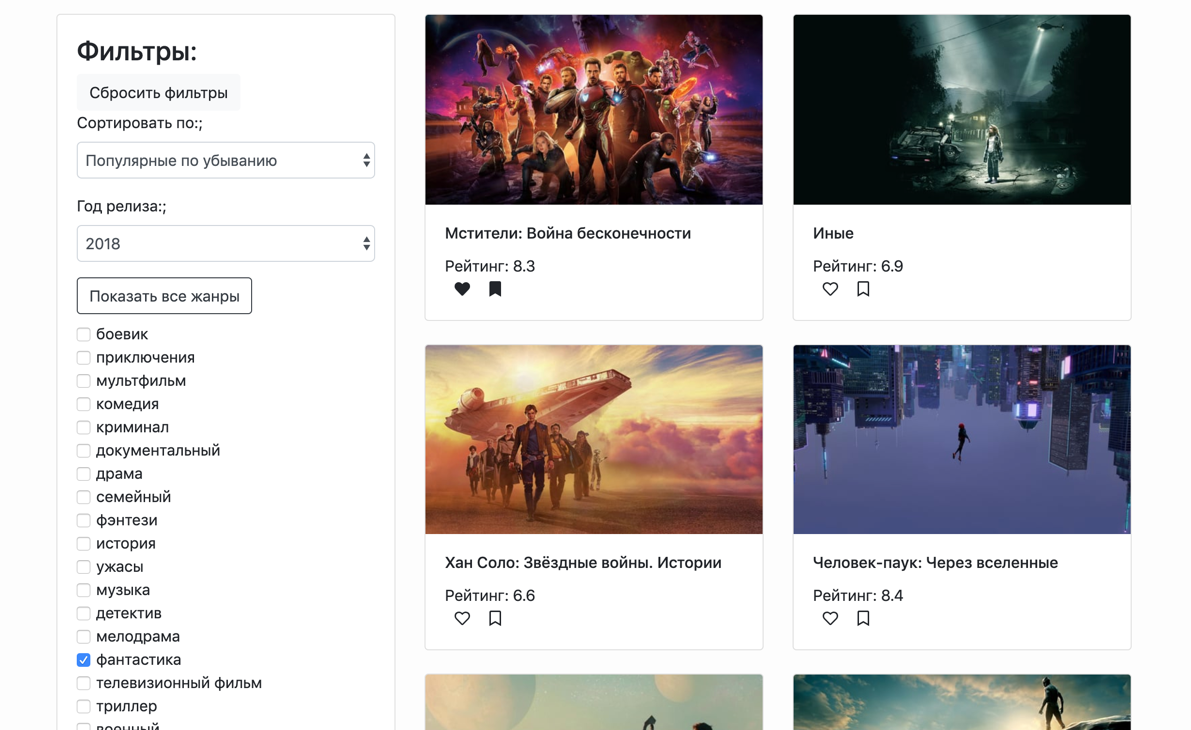


Рис. 4.4. Вибірка відфільтрованих та відсортованих даних

З рисунку 4.4 видно, що користувачем було обрано сортування по рейтингу спадання та 2018 року релізу. Обрані фантастичні жанри фільмів. За допомогою зручного графічного інтерфейсу, наданого *MovieDB,* можна управляти своїм аккаунтом та подивитися списки улюблених та обраних фільмів.

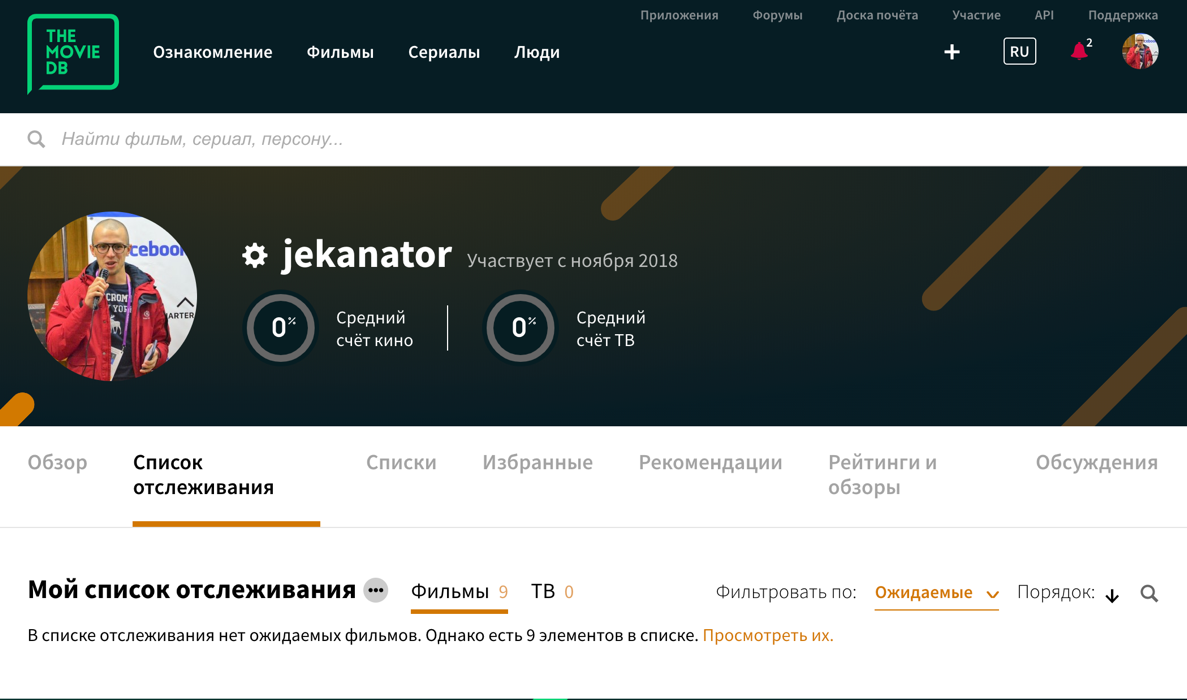


Рис. 4.5. Інтерфейс користувача *MovieDB*

Фільтр, який користувач налаштовував при вході в системи для першої вибірки, не зберігається при оновленні сторінки, але надається змога редагувати обрані та улюблені фільми, так як вони будуть збережені у базі даних навіть після виходу користувача із системи. При наступному вході в систему та авторизації користувача, параметри фільтрації та сортування будуть ініціалізовані знову, а обрані та улюблені фільми будуть автоматично виведені по раніше обраним параметрам. Обрані фільми також можна передивитися за допомогою *MovieDB,* що дуже зручно для розробників.

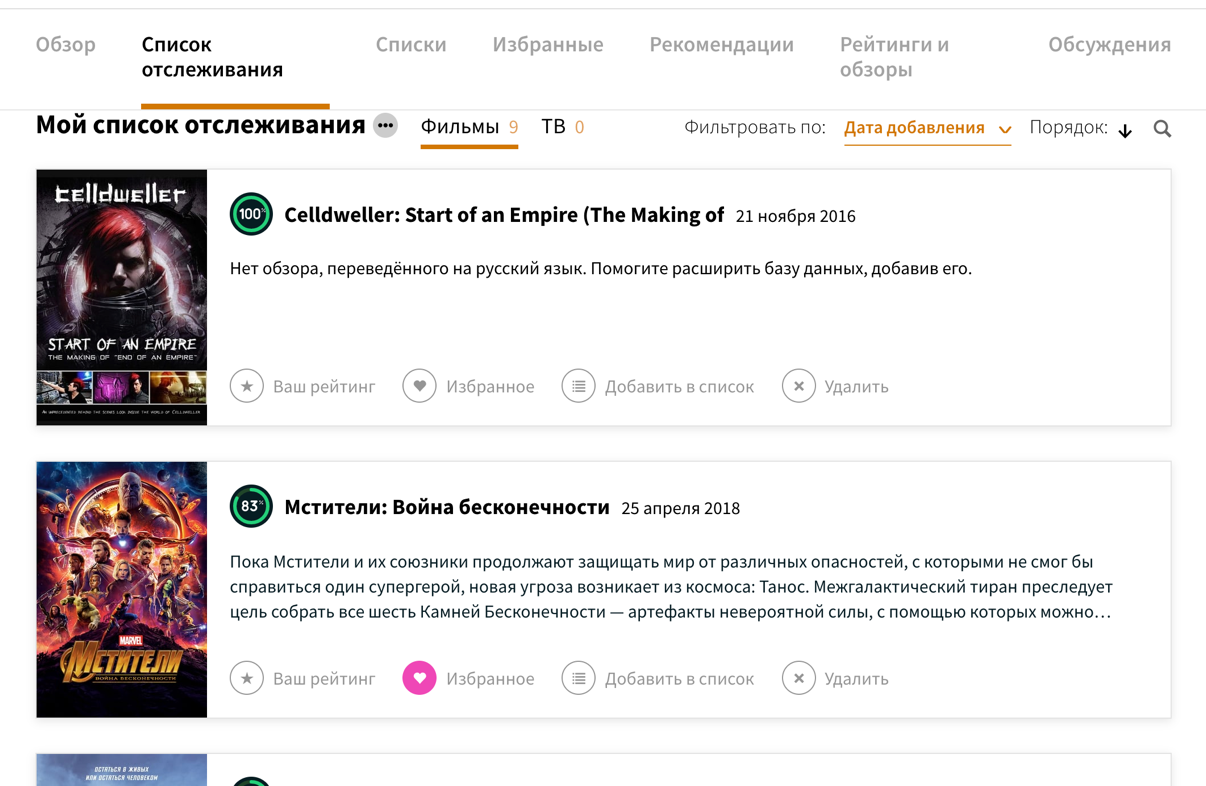


Рис. 4.6. Список обраних фільмів відображений за допомогою *MovieDB*

Якщо користувачем було обрано фільм та додано його до улюблених, дані будуть синхронізовано і кросплатформена версія додатку, яка використовується на мобільному пристрої також автоматично отримає серед даних параметр з ідентифікаторами обраних та улюблених фільмів, з рис. 4.7 видно, що користувачем було обрано деякі фільми, в результаті чого при виводі на екран мобільного пристрою даних за такими ж фільмами, вони автоматично отримали потрібну інформацію.

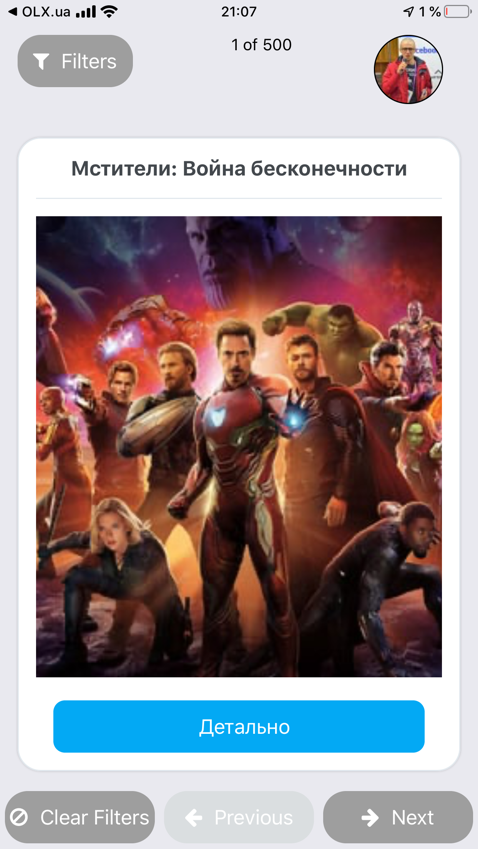


Рис. 4.7. Вибірка даних з мобільного пристрою

Також на мобільному пристрої кросплатформено продубльована вся логіка роботи із додатком, наприклад на рис. 4.8 можна побачити систему фільтрації та сортування даних на мобільному пристрої.

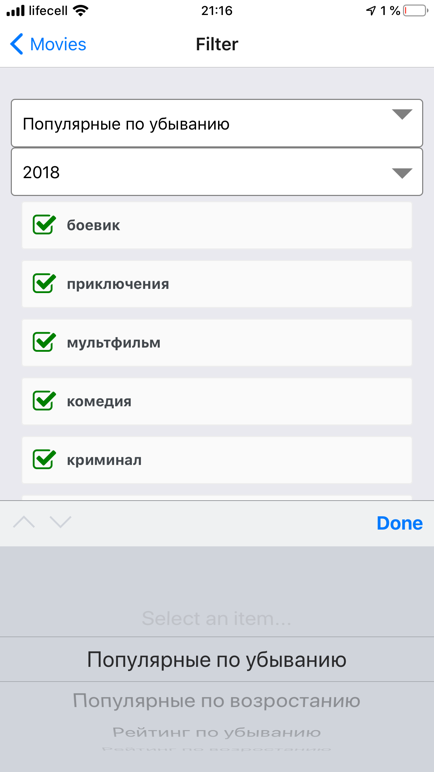


Рис. 4.8. Система фільтрації та сортування даних на мобільному пристрої

## 4.4. Висновки до розділу

В четвертому розділі було описано апаратно-програмні вимоги до серверу та пристроїв, на яких буде виконуватися додаток. Вимоги до серверу, на якому буде розміщуватись дані та *API* для роботи з ними, були сформовані з огляду на велику кількість вхідних даних для обробки додатком та велику кількість користувачів, що будуть підключатись до системи. Вимоги до локальних пристроїв були сформовані з огляду на те, що це кросплатформений додаток і для роботи з ним може бути використаний будь-який ПК, а також мобільні пристрої, буде використовуватися невелика кількість вхідних пакетів з даними та буде тільки одне підключення користувача яке не потребує великої кількості ресурсів. Було взято до уваги базу даних додатку, база даних на сервері значно більша ніж база тестових даних на локальному пристрої розробника, тому вимоги до локальної машини набагато менші ніж вимоги до серверу на якому буде розміщуватись система.

Сформовано вимоги до програмних засобів, що використовуються системою. Для серверу та пристрою, на якому буде запускатися додаток, вони відрізняються, так як система потребує програмні засоби залежно від середовища виконання, об’ємів даних та кількості підключень користувачів.

Також було описано роботу системи з файлами управління залежностями, що використовуються додатком, а саме робота за бібліотеками через *package.json*, що дає змогу запускати скрипти сторонніх бібліотек через певний проміжок часу, коли вони потрібні, наведено приклад *package.json* файлу для запуску скриптів від імені користувача.

Було описано роботу користувача з додатком, а саме вхід користувача до системи, адміністрування доступу користувачів до системи, авторизацію користувачів в системі. Було наведено приклади роботи користувача з функціоналом фільтрації та сортування додатку для формування вибірок з даними. Описано функціонал для додавання фільмів до обраних та улюблених.

# ВИСНОВКИ

В ході виконання дипломної роботи було спроектовано та розроблено програмний модуль створення кросплатформених додатків на основі технологій *React* та *React Native*, за допомогою якого було створено кросплатформеий додаток, який функціонує на будь-яких комп’ютерах та мобільних пристроях.

Даний модуль був розроблений за допомогою програмних засобів *JavaScript* та фреймворків *React* та *React Native,* що надає зручний функціонал для розробки веб додатків та мобільних додатків, що в свою чергу надає функціонал для створення кросплатформених додатків. Технології *MovieDB API,* яка дозволяє представити базу даних в об’єктній формі,*,* яка дає змогу працювати з базами даних. Представлення бази даних в об’єктній формі полегшує роботу з даними за рахунок того, що об’єкти бази даних представлені у вигляді класів і звертатися до них можна так само, як до звичайних *JavaScript* об’єктів.

Даний модуль надає можливості автоматизації створення кросплатформених додатків. Завдяки тому, що реалізується кросплатформена архітектура, в основі якої лежить клієнт-серверна парадигма, створення кросплатформених додатків стає набагато швидшим, а доступ до серверної частини можна отримати з будь-якого пристрою який має підключення до мережі.

В першому розділі розглянуто теоретичні відомості, що стосуються кросплатформених додатків, поняття кросплатформеності, основні фреймворки, за допомогою яких можна реалізувати кросплатформеність та методи створення кросплатформених додатків. Реалізація кросплатформеного функціоналу за допомогою фреймворків *React* та *React Native.* Розглянуто поняття перевикористання компонентів та *React Hooks*. Особливості розробки ПЗ за допомогою *React*, а саме завдяки чому існує можливість створювати швидкодіючі застосунки , які в той же час будуть кросплатформеними. Було розглянуто характерні для мови *JavaScript* особливості, фактори за якими визначається швидкодія додатків. Розглянуто та описано основні методи та принципи, за якими будуються фреймворки, за допомогою яких можуть буду створені кросплатформені додатки. Було описано основні архітектурні принципи *Cordova*, *Flutter* та *React Native*. За допомогою цих фреймворків нативні дані записуються до файлів, після чого обробляються системою та компіляторами і формуються в збірку для використання програмними додатками, що реалізують кросплатформену архітектуру.

В другому розділі розглянуто засоби, що були використані при розробці модуля створення кросплатформених додатків для подальшої розробки додатків. Описано *MovieDB* та *API* за допомогою яких відбувається доступ до бази даних. Розглянуто основні методи роботи з *API*, а саме можливість автентифікації, та принципи її роботи, підтримка пошукових запитів і *Query* за допомогою яких відбувається вибірка даних із бази. Розглянуто формат універсальний формат передачі даних по мережі, який має назву *JSON*, за допомогою якого відбувається передача усіх мережевих запитів з клієнта на сервер. Розглянуто надані *API* методи фільтрації та сортування результатів, реалізовані на сервері за допомогою окремих логічних модулів. Описано мову запитів *SQL* та середовище розробки, що використовується при проектуванні баз даних. Розглянуто основні можливості *SQL* та типи запитів, які є стандартними для цієї мови.

Статичний аналіз типів даних *Flow,* що розроблено спеціально для мови програмування *JavaScript*, представляє собою набір методів для співставлення між типами, їх властивостями та даними, які записуються у змінні. За допомогою статичного аналізу типів, а саме чіткої системи типізації, є можливість створити строго типізувати інтерфейси та створити анотації типів, в результаті чого, отримати повністю передбачуваний та типізований код, написаний за допомогою динамічно-типізованої мови програмування *JavaScript*.

*React*, фреймворк за допомогою якого можна будувати швидкі та кросплатформені застосунки. Код написаний з використання фреймворку *React*, може бути перевикористаний у інших подібних фреймворках. Розглянуто оптимізаційні процеси, які відбуваються у фреймворці *React*, швидкодію компонентів та методи їх прискорення.

*React Native,* являє собою фреймворк за набором класів та методів для створення кросплатформених програмних додатків. За допомогою цього фреймворку, можна перевикористати код, написаний за допомогою фреймворку *React* та запускати його на мобільних пристроях під управлінням *iOS* та *Android*. Розглянуто *Yoga* *Engine*, за допомогою якого відображення створені з використанням *JSX* перетворюються у нативні відображення та завдяки використанню спеціальної системи координат, розташовуються на екрані.

*Expo,* набір інструментів, які спрощують створення *React Native* додатків. За допомогою *Expo* розробники можуть створювати *React Native* додатки, не відчуваючи при цьому проблем з установкою і налаштуванням залежностей програмного забезпечення таких як *Android Studio*, *Xcode* і інших інструментів, які необхідні для розробки і запуску *React Native* додатків.

Розглянуто інструмент *React Native Bridge,* за допомогою якого і стала можливою побудова кросплатформених додатків. За допомогою *Bridge* додатки *React Native* мають можливість виконувати інструкції в нативному потоці і використовуючи одну і ту саму кодову базу, будувати відображення за допомогою *Yoga* для будь-яких платформ.

В третьому розділі було описано розробку модуля створення кросплатформених додатків, алгоритми роботи серверної частини додатку та використання *API*.

В четвертому розділі було розглянуто вимоги до апаратно-програмного забезпечення, роботу користувача з додатком, запуск додатку на виконання, та внутрішню організацію скриптів додатку для роботи зі сторонніми бібліотеками та створенням бандлу, який запускає нативну частину додатку.

Розглянуто сценарії, які запускають додаток на виконання, перший з яких, призначений для визначення необроблених файлів, їх вибору, компіляції коду та створення спеціальної збірки, яка називається бандлер, а також другий сценарій, який служить для взаємодії з клієнтом на стороні пристрою, керує запуском першого сценарія та обробкою запитів користувача до додатку та відображенням вибраної користувачем інформації на екрані.

Розглянуто архітектуру додатку та його конфігураційних файлів, які будуть зібрані у бандл та використані для упрвління додатком, було наведено приклади таких файлів та каталогів у яких вони розташовані.

В ході вивчення теоретичних матеріалів для розробки системи було сформовано основні вимоги до бази даних, а саме простота використання, забезпечення цілісності даних, швидкість роботи з даними. Вимогами до серверної частини є швидкість обробки даних, швидкість обробки клієнтських запитів, простота роботи з консольними скриптами, що в результаті зведено до запуску одного скрипта за допомогою *cron* системи. Сформовано основні вимоги до клієнтської часини.

В ході розробки системи було визначено основні технології та підходи, що були використані при розробці системи. Основними з яких є фреймворки *React* та *React Native*, та *MovieDB API.*

Було розроблено кросплатформений додаток, що запускається на мобільних пристроях за допомогою *Expo* та на ПК за допомогою веб-браузеру та виконує обробку вхідних даних, що постійно надходять з бази даних. Розроблено клієнтську частину для взаємодії з користувачем, обробки запитів користувача на вибірки необхідних даних, фільтрації або сортування.

Основні результати дипломної роботи:

1. Сформульовано переваги та основні проблеми крослатформених додатків, визначено особливості фреймворків для їх створення.

2. Визначено основні реалізації кросплатформеності, що використовують сучасні технологічні рішення.

3. Розроблено основні шляхи щодо використання кросплатформеності у додатках створених за допомогою *React* та *React Native*.

4. Розроблено модуль створення кросплатформених додатків, у якому використовується *React* та *React Native* для реалізації.

# СПИСОК БІБЛІОГРАФІЧНИХ ПОСИЛАНЬ ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Мурий Є.В.  Аналіз лінгвістичних даних у мобільних додатках. Міжнародна науково-технічна конференція «Інтелектуальні технології лінгвістичного аналізу». Тези доповідей. – К.: НАУ, 2019. – С.14.
2. Дарси Л., Коннер Ш. Разработка приложений для *Android*-устройств. Т. 1: Базовые принципы, Л.: Диалектика, 2014. – 402 с.
3. Эспозито Д. Разработка современных веб-приложений: анализ предметных областей и технологий. Л.: Диалектика-Вильямс, 2017. – 464 сВитвицкий Г.Н. Зональность климата Земли. М.: Мир, 1980. – 305 с.
4. *Wargo J. Apache Cordova API Cookbook*. Л.: *Gardner Books*, 2017. – 328 с.
5. *Windmill E. Flutter in Action* Л.: *Manning Publications*, 2019. – 368 с.
6. *Abshak P., Abshishek N. React Native for Mobile Development (2nd ed.)* Л.: *Apress*, 2017. – 648 с.
7. Бэнкс А. *React* и *Redux*. Функциональная веб-разработка. Руководство. – Л.: Питер, 2017. – 755 с.
8. Тиленс М. *React* в действии. Л.: Питер, 2017. – 648 с.
9. Мардан А. *React* быстро. Веб-приложения на *React*, *JSX*, *Redux* и *GraphQL*. – Л.: *Manning*, 2019. – 528 с.
10. Браун Э., Изучаем *JavaScript*: руководство по созданию современных веб-сайтов. 3-е издание. – Л.: Диалектика, 2019. – 368 с.
11. Симпсон К. {Вы не знаете *JS*} Типы и грамматические конструкции. – Л.: Питер, 2016. – 240 с.
12. Стефанов С. *JavaScript* шаблоны проектирования. – Л.: Символ, 2011. – 262 с.
13. Гамма Э. Хелм Р., Джонсон Р., Влиссидес С., Паттерны проектировнаия в обьектно-ориентированном программировании. – Л.: Питер, 2001. – 348 с.
14. *Zakas C. Principles of object-oriented JavaScript*. – Л.: *Allite*, 2019. – 92 с.
15. Дейт К. Введение в системы баз данных. – С. Пб.:Вильямс, 2000. – 354 c.
16. ГОСТ 2.106–96 ЕСКД “Текстовые документы”.
17. ДСТУ 3008–95 “Документація. Звіти у сфері науки і техніки. Структура і правила оформлення”.
18. Слободян О. Положення про дипломні роботи (проекти) випускників Національного авіаційного університету. Затверджено наказом ректора від 14.12.2017 № 178/од.
19. Тоу Д. Настройка *SQL*; СПб: Питер, 2009. – 539 c.
20. Хаббард, Дж. Автоматизированное проектирование баз данных; М.: Мир, 2011. – 453 c.
21. W*eb Performance Optimization with webpack. Google Developers, 2018*.
22. Наумов А.Н.; Вендров А.М.; Иванов В.К. и др. Системы управления базами данных; М.: Финансы и статистика, 2010. – 352 c.
23. Аткинсон, Леон *MySQL*. Библиотека профессионала; М.: Вильямс, 2010. 624 c.
24. *S. Klein. React Native* – *Pinnacle*.: *Appres*, 2017. – 280.
25. Таненбаум Э, Уэзеролл Д. Компьютерные сети. Питер, 2012. – 960.
26. *React documentation* – *online* документація з використання ресурсів фреймфорку *React* – Електрон. дан. – Режим доступу: *http://ru.reactjs.org/docs/getting-started/*, вільний.
27. *React Native documentation* – *online* документація з використання фреймворку *React Native*– Електрон. дан. – Режим доступу: *https://facebook.github.io/react-native/docs/getting-started*, вільний.
28. *Expo documentation* – *online* документація– Електрон. дан. – Режим доступу: *https://www.docs.expo.io/versions/latest*, вільний.
29. *MovieDB* – *online* документація– Електрон. дан. – Режим доступу: *https://developers.themoviedb.org/*, вільний.
30. *Flow*– *online* документація– Електрон. дан. – Режим доступу: *https://flow.org/*, вільний.