МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

на тему: **«РОЗРОБКА КРОСПЛАТФОРМНОГО ПРОГРАМНОГО ПРОДУКТУ ДЛЯ ЗБИРАННЯ ДАНИХ ОБЛАДНАННЯ»**

Виконав: студент 2 курсу, групи 8.1218

спеціальності 121 інженерія програмного забезпечення
(шифр і назва спеціальності)

освітньої програми програмна інженерія
(назва освітньої програми)

Є.Д. Петренко
(ініціали та прізвище)

Керівник доцент кафедри програмної інженерії, доцент, к.т.н. Чопоров С.В.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Рецензент завідувач кафедри комп'ютерних наук, доцент, к.т.н. Борю С.Ю.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя – 2020

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра програмної інженерії

Рівень вищої освіти магістр

Спеціальність 121 інженерія програмного забезпечення
(шифр і назва)

Освітня програма програмна інженерія

ЗАТВЕРДЖУЮ

Завідувач кафедри програмної інженерії, к.ф.-м.н., доцент
_____ Лісняк А.О.
(підпис)

« 29 » травня 2019 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Петренку Євгенію Дмитровичу
(прізвище, ім'я та по-батькові)

1. Тема роботи (проекту) Розробка кросплатформного програмного продукту для збирання даних обладнання

керівник роботи (проекту) Чопоров Сергій Вікторович, к.т.н., доцент
(прізвище, ім'я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 29 » травня 2019 року № 811-с

2. Строк подання студентом роботи 27.12.2019

3. Вихідні дані до роботи 1. Постановка задачі.
2. Перелік літератури.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Постановка задачі.
2. Основні теоретичні відомості.
3. Реалізація продукту.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
Презентація

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання 29.05.2019

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	27.05.2019-09.07.2019	
2.	Збір вихідних даних.	09.07.2019-15.09.2019	
3.	Обробка методичних та теоретичних	15.09.2019-03.10.2019	
	джерел.		
4.	Розробка першого та другого розділу.	03.10.2019-05.11.2019	
5.	Розробка третього розділу.	05.11.2019-01.12.2019	
6.	Оформлення та нормоконтроль	01.12.2019-27.12.2019	
	кваліфікаційної роботи.		
7.	Захист кваліфікаційної роботи.	15.01.2020	

Студент _____ Є.Д. Петренко
(підпис) (ініціали та прізвище)

Керівник роботи _____ С.В. Чопоров
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

Нормоконтролер _____ О.В. Кудін
(підпис) (ініціали та прізвище)

РЕФЕРАТ

Кваліфікаційна робота магістра «Розробка кросплатформного програмного продукту для збирання даних обладнання»: 79 с., 38 рис., 14 джерел, 4 додатка.

МОБІЛЬНИЙ ДОДАТОК, КРОСПЛАТФОРМНІСТЬ, РОБОТА В РЕЖИМІ ОФЛАЙН, СИНХРОНІЗАЦІЯ ДАНИХ, REACT NATIVE.

Об’єкт дослідження – мова Javascript, фреймворк React Native, бібліотека MobX для керування станом додатка, NoSQL база даних PouchDB для підтримки офлайн режиму.

Мета роботи: створення кросплатформного продукту для збирання даних обладнання, призначеного для сучасних смартфонів с можливістю роботи в офлайн режимі.

Метод дослідження – вивчення специфікації та документації вибраних засобів, використання їх в процесі розробки.

У якості головного інструменту було обрано React Native фреймворк, який дозволяє створювати кросплатформні додатки для мобільних операційних систем Android та iOS, з використанням мультипарадигмальної мови програмування Javascript. Також було додано бібліотеку MobX для зручного та масштабованого керування станом додатка, а для підтримки роботи в режимі офлайн було додано NoSQL базу даних PouchDB. У ході розробки було створено кросплатформний мобільний додаток, який дозволяє збирати дані обладнання в будь-якій точці планети, лише за умови наявності смартфона, та синхронізувати ці дані з віддаленим сервером, як тільки з’являється доступ до мережі Інтернет.

Перевагами додатка є простота у використанні, можливість використання на значній більшості сучасних смартфонів, а також робота в автономному режимі.

SUMMARY

Master's Qualifying Thesis "Development of Cross-platform Software for Equipment Data Collection": 79 pages, 38 figures, 14 sources, 4 applications.

MOBILE APP, CROSS-PLATFORM, OFFLINE WORK, DATA SYNCHRONIZATION, REACT NATIVE.

The object of the study is Javascript, React Native framework, MobX library for application state management, NoSQL PouchDB database for offline support.

The aim of the study is to create a cross-platform product for data collection of equipment designed for modern smartphones with offline capability.

The method of research is to study the specifications and documentation of the selected tools, to use them in the development process.

As the main development tool, the React Native framework was used, which allows you to create cross-platform applications for mobile operating systems Android and iOS, using the multi-paradigmatic Javascript programming language. Also, a MobX library was added for easy and scalable application state management, and a NoSQL PouchDB database was added to support offline work. A cross-platform mobile application was created during development to collect equipment data from anywhere on the planet, only with a smartphone, and to synchronize this data with a remote server as soon as access to the Internet will be available.

The advantages of the application are ease of use, the ability to use on the vast majority of modern smartphones, as well as work offline.

ЗМІСТ

ВСТУП

На даний час стало дуже популярним використання смартфонів, планшетів, смартгодинників. А також виріс попит на створення різноманітних додатків для них. Кожного дня комп'ютери заміняють більш зручні та компактні мобільні засоби. Тому зараз є дуже актуальним створення якісних додатків, які можуть покрити та поліпшити всі аспекти життя користувача. Велика кількість компаній, яка раніше могла відмовитися від використання сучасних засобів, зараз відчуває велику потребу в цих засобах. Тому було вирішено написати мобільний додаток для компанії, яка займається встановленням конвеєрів та обладнанням для них. Цей додаток дозволить користувачу заповнювати дані цього обладнання в смартфон і при наявності інтернету автоматично відправлятиме ці дані на віддалений сервер. Таким чином користувач зможе працювати незалежно від місця знаходження.

Актуальність створюваного продукту полягає в наступному:

а) кросплатформність;

б) підтримка режиму роботи в офлайн;

в) великий попит на мобільні додатки;

г) відкритий код;

д) безкоштовний.

Мета роботи полягає в створенні кросплатформного додатку, який буде праювати без з'єднання з мережою Інтернет та матиме зручний інтерфейс і в кінцевому підсумку завдяки своїм функціональним можливостями вирішить всі проблеми, з якими стикається компанія-замовник.

Для того щоб досягти поставленої мети, необхідно використати всі можливі сучасні засоби для розробки мобільних додатків. Розробити правильну архітектуру, оптимізувати додаток та протестувати його перед випуском.

Щоб виконати всі поставлені задачі у якості основних об'єктів для дослідження було обрано такі сучасні засоби: React Native, MobX та PouchDB.

React Native дозволяє розробити додаток відразу для двох популярних мобільних операційних систем – Android та iOS. Використання React Native вирішує дві проблеми: скорочує час розробки та дозволяє створити додаток з єдиною кодовою базою.

MobX – це бібліотека для керування станом додатка. Вона заснова на принципах реактивного програмування та не обмежує в можливостях планування та розробки архітектури додатка.

PouchDB – засіб для роботи з локальною базою даних. Його використання значно полегшує процес синхронізації даних з серврером і в деякій мірі вирішує питання нормалізації даних.

Методом дослідження перерахованих засобів є вивчення документації та використання цих засобів в процесі розробки.

1 ОГЛЯД ТЕХНОЛОГІЙ

1.1 Вирішення проблеми

Створення кросплатформного мобільного додатка з можливістю роботи в режимі офлайн є проблемою, яку потрібно вирішити в даній роботі. Сьогодні існує багато рішень для розробки кросплатформних додатків. Серед цього різноманіття, було обрано фреймворк React Native. Його головною особливістю є повна безкоштовність без навяності Pro пакетів, тобто весь функціонал доступний без зайвих платежів. Для контролю стану додатка було обрано бібліотеку MobX, а для роботи з базою даних в мобільному пристрої було обрано PouchDB.

1.1.1 Огляд фреймворку React Native

React Native – це JavaScript фреймворк для написання кросплатформних мобільних додатків для iOS та Android. Він базується на React, бібліотеці JavaScript Facebook для створення інтерфейсів користувача, але замість націлення на браузер, він націлений на мобільні платформи [1].

Подібно до React для веб, програми React Native записуються за допомогою суміші JavaScript та XML-націльної розмітки, відомої як JSX. Потім, "міст" React Native викликає вбудовані API візуалізації в Objective-C (для iOS) або Java (для Android). Таким чином, додаток відображатиметься за допомогою реальних компонентів мобільного інтерфейсу, а не веб-компонентів, і буде виглядати як будь-який інший мобільний додаток. React Native також має відкриті інтерфейси JavaScript для API платформ, тому додатки React Native можуть отримувати доступ до функцій платформи, таких як камера телефону чи місцезнаходження користувача.

На даний момент React Native підтримує iOS та Android, і він має можливість розширитись і на майбутніх платформах.

Той факт, що React Native фактично відображається за допомогою стандартних API візуалізації своєї хост-платформи, дозволяє йому виділятися з більшості існуючих методів розвитку платформних додатків, таких як Cordova або Ionic. Існуючі методи написання мобільних додатків за допомогою комбінацій JavaScript, HTML та CSS зазвичай відображаються за допомогою веб-,браузеру. Хоча такий підхід може працювати, він також має недоліки, особливо щодо ефективності. Крім того, вони зазвичай не мають доступу до нативних елементів інтерфейсу хост-платформи. Коли ці фреймворки намагаються імітувати нативні елементи інтерфейсу, результат зазвичай «відчувається» лише трохи; реверсивна інженерія всіх тонких речей, таких як анімація, вимагає величезних зусиль, і вони можуть швидко застаріти.

На відміну від цього, React Native фактично переводить розмітку на реальні, нативні елементи інтерфейсу, використовуючи існуючі засоби візуалізації компонентів на будь-якій платформі, з якою ви працюєте. Крім того, React працює окремо від основного потоку інтерфейсу, тому написана програма може підтримувати високу продуктивність без шкоди для можливостей. Цикл оновлення в React Native такий же, як і в React: коли властивості або стан компоненту змінюються, React Native повторно надає перемальовує цей компонент. Основна відмінність React Native від React у веб полягає в тому, що React Native це робить, використовуючи бібліотеки інтерфейсу користувача на своїй хост-платформі, а не використовуючи розмітку HTML та CSS [1].

Робота з React Native може різко скоротити ресурси, необхідні для створення мобільних додатків. Будь-який розробник, який вміє писати код React, може орієнтуватися на iOS та Android, і все з тим самим набором навичок. React Native видаляє необхідність розробників «під платформу».

Але не весь код, крос-платформений, і залежно від того, яка функціональність потрібна на певній платформі, можна час від часу

занурюватися в Objective-C або Java. Але повторно використовувати код на платформах напрочуд легко з React Native.

React Native дозволяє веб-розробникам створювати надійні мобільні додатки, використовуючи наявні знання JavaScript. Він пропонує більш швидку мобільну розробку та більш ефективний обмін кодом через iOS та Android, не приносячи шкоди досвіду та якості додатків кінцевому користувачеві.

Ідея писати мобільні додатки на JavaScript видається трохи дивною. Як можливо використовувати React в мобільному середовищі? Щоб зрозуміти технічні підґрунтя React Native, спершу потрібно згадати одну з особливостей React – віртуальний DOM [1].

Віртуальний DOM виступає як прошарок між описом розробника про те, як компонент повинен виглядати, і тим, що потрібно зробити щоб цей компонент так виглядав. Щоб візуалізувати інтерактивні користувальницькі інтерфейси у веб-браузері, розробники повинні редагувати DOM або об’єктну модель документа. Це дорогий крок, і надмірне записування до DOM суттєво впливає на продуктивність. Замість того, щоб безпосередньо відображати зміни на сторінці, React обчислює необхідні зміни, використовуючи вбудовану в пам'ять версію DOM і робить мінімально необхідну кількість змін в ньому, для того щоб відобразити всі зміни. На рисунку 1.1 показано, як це працює.

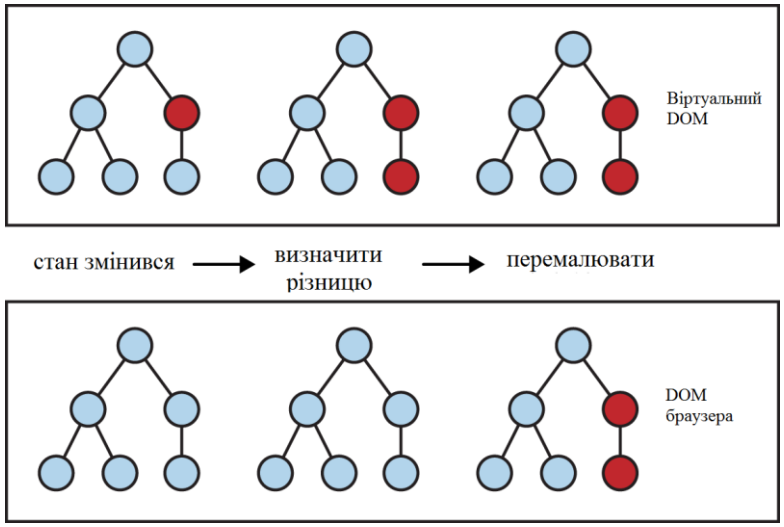

Рисунок 1.1 – Порівняння виконання обчислень у віртуальному DOM та DOM веб-браузера

У контексті React для веб більшість розробників вважають, що віртуальний DOM більш оптимізований та продуктивний. Віртуальний DOM, безумовно, має перевагу в продуктивності, але його реальний потенціал полягає у силі його абстракції. Розміщення чистого шару абстракції між кодом розробника та фактичним візуалізацією відкриває масу цікавих можливостей. Це й дозволяє візуалізувати один і той же код під різні платформи.

На рисунку 1.2 показано як працює React Native. Замість того, щоб візалізувати в DOM браузера, React Native викликає API Objective-C для візуалізації компонентів iOS або Java API для надання компонентів Android. Це відрізняє React Native від інших варіантів розробки додатків між платформами, які часто закінчуються на відображенні веб-браузера.

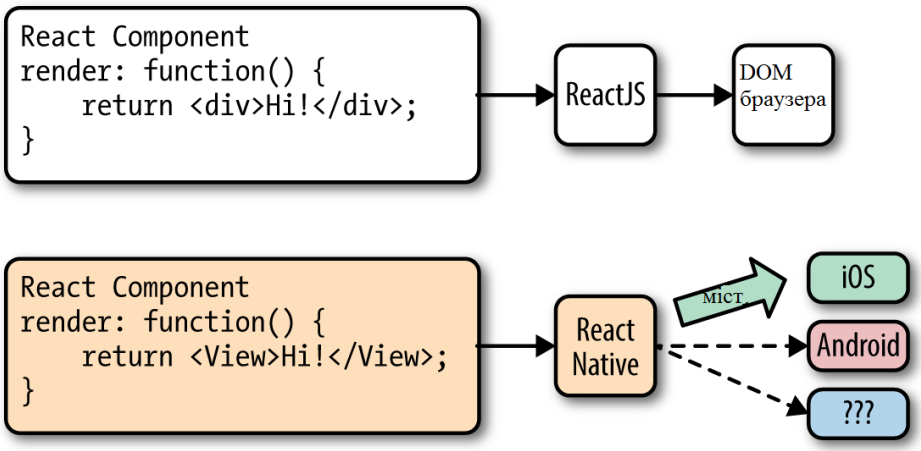

Рисунок 1.2 – React Native може візалізувати під різні платформи

Це все можливо через "міст", який надає React – інтерфейс до власних компонентів хост-платформи. Компоненти React повертають розмітку зі своєї функції візуалізації, яка описує, як вони повинні виглядати. В React для веб це перекладається безпосередньо на DOM веб-браузера. Для React Native ця розмітка перекладається на хост-платформу [1].

На даний момент React Native підтримує iOS та Android. Через прошарок абстракції, який надає віртуальний DOM, React Native також можна націлити на інші платформи, але для цього потрібно написати власний «міст».

Розглянемо плюси та мінуси React Native.

Плюси:

а) кросплатформність – спочатку цей фреймворк підтримував лише розробки під iOS. Але React Native почав набирати популярність та успіх, а тому Facebook згодом вирішив забезпечити підтримку платформи Android. Таким чином, програми на React Native можна розробити за допомогою використання єдиної бази коду для обох платформ одночасно, що ще більше підвищило її популярність;

б) висока продуктивність – React Native додатки компілюються в нативно написаний код, що дозволяє йому не тільки працювати в обох операційних системах(Android та iOS), але також і однаково функціонувати;

в) швидка розробка – можливість перевикористання компонентів під обидві патформи значно скорочує час розробки.

Мінуси:

а) деякі компоненти недоступні на Android чи iOS;

б) іноді потрібно писати нативний код на Java або Objective-C і в цьому разі, потрібно вміти писати код під обидві платформи.

1.1.2 Особливості мови програмування JavaScript

Для розробки додатка на React Native, використовується мова JavaScript, розглянемо її більш детальніше.

JavaScript є крос-платформенною, об'єктно-орієнтованою мовою сценаріїв, що використовується для створення інтерактивних веб-сторінок (наприклад, складні анімації, кнопки, які можна натискати, спливаючі меню тощо). Існують також більш розвинені серверні версії JavaScript, такі як Node.JS, які дозволяють додавати більше функціональних можливостей на веб-сайт, ніж просто завантаження файлів (наприклад, співпраця в режимі реального часу між декількома комп'ютерами). У середовищі хосту (наприклад,

веб-браузері) JavaScript можна підключити до об'єктів свого середовища, щоб забезпечити програмний контроль над ними [4].

JavaScript містить стандартну бібліотеку об'єктів, таких як Array, Date та Math, а також основний набір елементів мови, таких як оператори, керуючі структури та твердження. Основний JavaScript може бути розширений для різних цілей, доповнюючи його додатковими об'єктами; наприклад [3]:

а) JavaScript на стороні клієнта розширює основну мову, постачаючи об'єкти для керування браузером та його об'єктною моделлю документів (DOM). Наприклад, розширення на стороні клієнта дозволяють додатку розміщувати елементи у формі HTML та реагувати на подію користувачів, такі як клацання миші, введення форми та навігація по сторінках;

б) серверний код JavaScript розширює основну мову, постачаючи об'єкти, що мають відношення до запуску JavaScript на сервері. Наприклад, розширення на стороні сервера дозволяють програмі спілкуватися з базою даних, забезпечувати безперервність інформації від одного виклику до іншої програми або здійснювати маніпуляції файлами на сервері.

Це означає, що в браузері JavaScript може змінити спосіб перегляду веб-сторінки (DOM). І, так само, JavaScript Node.js на сервері може відповідати на власні запити з коду, написаного у браузері.

JavaScript стандартизовано в Ecma International – європейська асоціація стандартизації інформаційно-комунікаційних систем (ECMA раніше була абревіатурою для європейської асоціації виробників комп'ютерів) для надання стандартизованої, міжнародної мови програмування на основі JavaScript. Ця стандартизована версія JavaScript, що називається ECMAScript, поводиться однаково в усіх програмах, що підтримують стандарт. Компанії можуть використовувати відкриту стандартну мову для розробки їх реалізації JavaScript. Стандарт ECMAScript задокументований у специфікації ECMA-262:

а) JavaScript – це легка інтерпретована мова програмування;

б) призначений для створення мережевих програм;

в) доповнює до та інтегрована з Java;

г) доповнює до та інтегрована з HTML;

д) відкрита;

е) крос-платформена.

Переваги використання JavaScript:

а) менше взаємодії з сервером – можна перевірити вхід користувача перед надсиланням сторінки на сервер. Це заощаджує трафік сервера, що означає меншу навантаження на сервер;

б) негайні відгуки відвідувачів – не доведеться чекати перезавантаження сторінки, щоб побачити, чи вони забули щось увійти;

в) підвищена інтерактивність – можна створювати інтерфейси, які реагують, коли користувач наводить на них мишу або активує їх за допомогою клавіатури;

г) більш гнучкі інтерфейси – можна використовувати JavaScript для включення таких елементів, як Drag and Drop компонентів і слайдерів, щоб надати відвідувачам сайту багатий інтерфейс.

JavaScript не дуже повноцінна мова програмування. Вона не має таких важливих функцій як:

а) JavaScript на стороні клієнта не дозволяє читати чи записувати файли. Це зберігається з міркувань безпеки;

б) JavaScript не може бути використаний для мережевих програм, оскільки підтримки такого функціоналу немає;

в) JavaScript не має багатопотокового або багатопроцесорного можливостей.

Однією з головних переваг JavaScript є те, що вона не потребує дорогих інструментів розробки. Можна використовувати простого текстовий редактора, такий як Блокнот. Оскільки це інтерпретована мова в контексті веб-браузера, навіть не потрібно купувати компілятор [5].

Різні постачальники випускають дуже гарні інструменти редагування JavaScript. Деякі з них перераховані нижче.

Microsoft FrontPage – Microsoft розробила популярний редактор HTML під назвою FrontPage. FrontPage також надає веб-розробникам кілька інструментів JavaScript, які допомагають створювати інтерактивні веб-сайти.

Macromedia Dreamweaver MX – це дуже популярний редактор HTML та JavaScript серед розробників. Він надає кілька зручних попередньо скомпонованих JavaScript-компонентів, добре інтегрується з базами даних та відповідає новим стандартам, таким як XHTML і XML.

Macromedia HomeSite 5 – це добре відомий редактор HTML та JavaScript від Macromedia, який можна використовувати для ефективного управління персональними веб-сайтами.

В заключення про JavaScript [3]:

а) спочатку JavaScript був створений як мова, доступна лише для браузера, але зараз вона використовується і в багатьох інших середовищах;

б) на даний момент JavaScript має унікальну позицію як найпоширенішу мову браузера з повною інтеграцією з HTML / CSS;

в) існує багато мов, які "перетворюються" на JavaScript і надають певні функції.

1.1.3 Огляд бібліотеки для керування станом MobX

Стан – це серце кожної програми і немає більш швидкого способу створення некерованого додатки, як відсутність консистентності стану. Або стан, яке неузгоджено з локальними змінними навколо. Тому безліч рішень з управління станом намагаються обмежити способи, якими можна його змінювати, наприклад зробити стан незмінним. Але це породжує нові проблеми, дані потребують нормалізації, немає гарантії посилальної цілісності і стає майже неможливо використовувати такі потужні концепти як прототипи.

MobX дозволяє зробити управління станом знову простим, повернувшись до кореня проблеми: він унеможливлює інконсістентность стану. Стратегія

досягнення цього досить проста: переконається що, все що може бути вийнято зі стану, буде вийнято. Автоматично.

Концептуально MobX обробляє додаток як електронну таблицю [6].

По-перше, є стан додатка. Графи об'єктів, масивів, примітивів, посилань які формують модель додатка.

По-друге є похідні. Зазвичай, це будь-яке значення, яке може бути обчислено автоматично з даних стану додатка.

Реакції дуже схожі на похідні. Основна відмінність: вони не повертають значення, але запускаються автоматично, щоб виконати якусь роботу. Зазвичай це пов'язано з введенням і виведенням даних. Вони перевіряють, що DOM оновився або мережеві запити виконалися вчасно.

Нарешті, є дії. Дії це все, що змінює стан. MobX простежить, щоб всі зміни в стані додатки, викликані діями, автоматично обробились усіма похідними і реакціями. Синхронно і без перешкод.

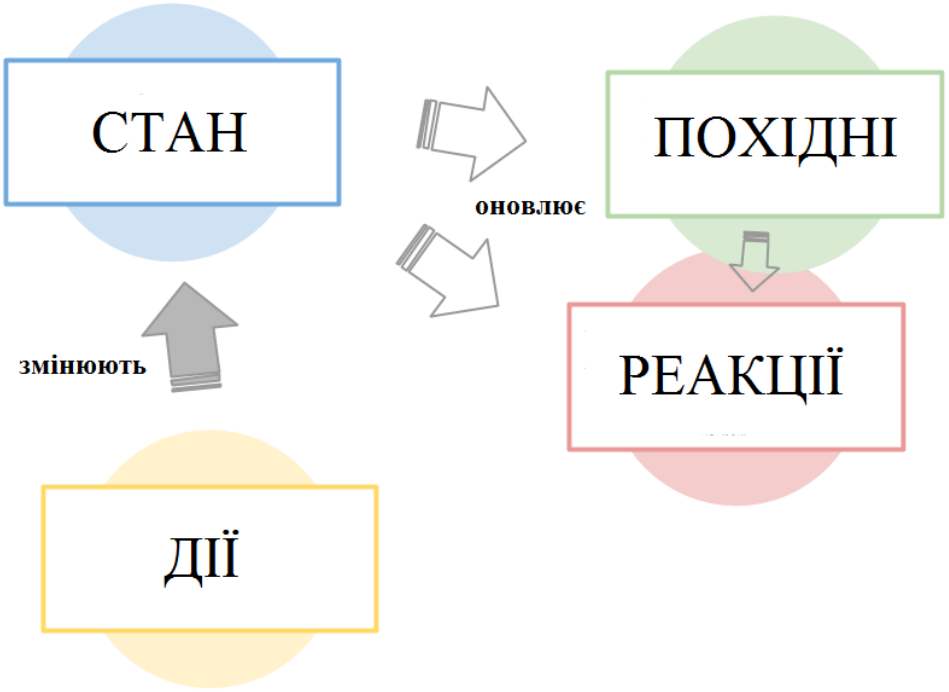

Рисунок 1.3 – Схема роботи MobX

MobX часто асоціюється як альтернатива Redux. Але це просто бібліотека, для вирішення певної проблеми а не архітектура або контейнер

стану, яким є Redux. Використовувати MobX означає використання контролерів, диспетчерів, дій, супервізорів або будь-якої іншої форми управління потоком даних, це все веде до того, що архітектуру потрібно проектувати власноруч. Але ця можливість дозволяє не залежати від одної архітектури, як це відбувається з Redux, і таким чином дозволяє проектувати більш специфічні додатки.

React і MobX разом – це потужне поєднання. React візуалізує стан програми, надаючи механізми для перекладу його на дерево компонентів, що візуалізуються. MobX надає механізм зберігання та оновлення стану програми, який потім використовує React.

І React, і MobX забезпечують оптимальні та унікальні рішення загальних проблем у розробці додатків. React забезпечує механізми оптимальної візуалізації користувальницького інтерфейсу за допомогою віртуального DOM, що зменшує кількість дорогих мутацій DOM. MobX забезпечує механізми оптимальної синхронізації стану програми з компонентами React, використовуючи графік стану реактивної віртуальної залежності, який оновлюється лише тоді, коли це суворо потрібно [6].

Переваги використання MobX:

а) це проста, дуже масштабована та ненав’язлива бібліотека управління станом;

б) з MobX не потрібно нормалізувати дані. Це робить бібліотеку дуже придатною для дуже складних моделей доменів;

в) оскільки дані не потрібно нормалізувати, а MobX автоматично відстежує відносини між станом та похідними, то на виході маємо референтну цілісність;

г) MobX будує графік усіх похідних у програмі, щоб знайти найменшу кількість повторних обчислень, необхідних для запобігання несвіжості. «Вивести все» може здатися дорогим, але MobX будує графік віртуальної деривації, щоб мінімізувати кількість перерахунків, необхідних для підтримки синхронізації зі станом;

д) MobX працює з простими структурами JavaScript. Завдяки своїй ненав’язливості, він працює з більшістю бібліотек JavaScript «із коробки», не потребуючи спеціальних додатків.

1.1.4 Огляд NoSQL бази даних PouchDB

На сьогодні більшість програм потребує постійного підключення до Інтернету. Втрата даних або повільні мережі стали проблемою, з якою стикається велика кількість розробників. Отже, питання полягає в тому, чи можна досягти досвіду в режимі офлайн, використовуючи React Native? Чи можемо ми зберігати дані локально? Чи можемо ми синхронізувати дані між кількома пристроями? Що стосується того, щоб дозволити користувачам виходити в офлайн, не втрачаючи своїх даних, PouchDB пропонує деякі дивовижні функції, які допоможуть в вирішенні цього питання.

PouchDB дозволяє програмам локально зберігати дані в режимі офлайн, а потім синхронізувати їх з будь-якими віддаленими серверами, коли програма знову в мережі, зберігаючи дані користувача в синхронізації незалежно не від чого [14].

PouchDB це NoSQL база даних. NoSQL – це база даних, яка забезпечує механізм зберігання та пошуку даних. Ці дані моделюються іншими засобами, ніж табличні співвідношення, що використовуються у реляційних базах даних. Бази даних NoSQL використовуються у веб-додатках для даних для роботи в реальному часі та для великих даних, а їх використання з часом збільшується. Системи NoSQL також іноді називають не тільки SQL, щоб підкреслити той факт, що вони можуть підтримувати SQL-подібні мови запитів.

База даних NoSQL включає простоту проектування, більш просте горизонтальне масштабування для кластерів машин і більш чіткий контроль за доступністю. Структури даних, що використовуються в базах даних NoSQL, відрізняються від тих, які використовуються за замовчуванням у реляційних базах даних, що робить деякі операції швидшими в NoSQL. Придатність даної

бази даних NoSQL залежить від проблеми, яку вона повинна вирішити. Структури даних, що використовуються в базах даних NoSQL, іноді також розглядаються як більш гнучкі, ніж таблиці реляційних баз даних [14].

Багато NoSQL сховищ роблять компромісну послідовність на користь доступності та швидкості до розділів. Перешкодами для більш широкого прийняття NoSQL сховищ є використання мов запитів низького рівня, відсутність стандартизованих інтерфейсів та величезні попередні інвестиції в існуючі реляційні бази даних.

Більшість баз даних NoSQL пропонують концепцію можливої послідовності, при якій зміни бази даних поширюються на всі вузли, тому запити для даних можуть не повернути оновлені дані негайно або можуть призвести до неточного зчитування даних, що є проблемою, відомою як несвіжі зчитування. Також деякі системи NoSQL можуть демонструвати втрачені записи та інші форми втрати даних. Деякі системи NoSQL надають такі поняття, як реєстрація заздалегідь запису, щоб уникнути втрати даних. Для розподіленої обробки транзакцій у кількох базах даних узгодженість даних є ще більшою проблемою. Це важко як для NoSQL, так і для реляційних баз даних. Навіть поточні реляційні бази даних не допускають референтних обмежень цілісності для проміжних баз даних.

Є багато переваг роботи з базами даних NoSQL. Основні переваги - висока масштабованість і висока доступність:

а) висока масштабованість – база даних NoSQL використовує шардинг для горизонтального масштабування. Розбиття даних та розміщення їх на декількох машинах таким чином, щоб порядок збереження даних посилювався. Вертикальне масштабування означає додавання більше ресурсів до існуючої машини, тоді як горизонтальне масштабування означає додавання більшої кількості машин для обробки даних. Вертикальне масштабування не так просто здійснити, але горизонтальне масштабування легко здійснити. NoSQL може обробляти величезну кількість даних завдяки масштабованості, оскільки з

ростом кількості даних NoSQL масштабується, щоб ефективно обробляти ці дані;

б) висока доступність – функція автоматичної реплікації в базах даних NoSQL робить її високодоступною, оскільки в разі будь-якого збою дані реплікуються до попереднього послідовного стану.

NoSQL має такі недоліки:

а) вузький фокус – бази даних NoSQL мають дуже вузьку спрямованість, оскільки вони в основному розроблені для зберігання, але це забезпечує дуже мало функціональності. Реляційні бази даних - кращий вибір у галузі управління транзакціями, ніж NoSQL;

б) відкрите джерело – NoSQL – це база даних з відкритим кодом. Наразі ще немає надійного стандарту для NoSQL. Іншими словами, дві системи баз даних, ймовірно, нерівні;

в) завдання управління – мета великих інструментів даних – зробити управління великим обсягом даних максимально простим. Але це не так просто. Управління даними в NoSQL набагато складніше, ніж в реляційній базі даних;

г) великий розмір документа – деякі системи баз даних, зберігають дані у форматі JSON. Що означає, що документів досить великі), а описові назви ключових слів насправді шкодять, оскільки вони збільшують розмір документа.

1.1.5 Огляд допоміжних бібліотек

Для більш швидкої розробки графічного інтерфесу користувача, було обрано бібліотеку Shoutem UI. Це набір стильованих компонентів, що дозволяє створювати прекрасні програми React Native для iOS та Android. Всі компоненти створені як для компонування, так і для налаштування. Кожен компонент має заздалегідь визначений стиль, сумісний з рештою інтерфейсу

Shoutem, що дає змогу створювати складні компоненти, які чудово виглядають без необхідності вручну визначати складні стилі.

Також були застосовані наступні бібліотеки:

а) Lodash – для спрощення обробки даних у JavaScript;

б) React Router – для керування переходами між вікнами додатка, та збереження стану цих переходів;

в) Moment – для спрощення роботи з часом та датами;

г) React Native Camera – для можливості робити фото та записувати відео;

д) React Native Background Timer – для моживості виконання задач в фоні, таких як синхронизація даних.

1.2 Порівняння обраних засобів з альтернативними

1.2.1 Порівняння React Native з Ionic

Вибір фреймворку був зроблений на користь React Native, але існує різноманіття фреймворків для розробки кросплатформених мобільних додатків. Далі буде порівняно React Native з другим по пополуярності фрефмворком Ionic.

Ionic Framework: краща продуктивність та елегантний дизайн.

Ionic є стандартною основою для розвитку гібридних мобільних додатків. Це дозволяє веб-розробникам створювати додатки для більшості платформ, використовуючи єдину кодову базу. Він використовує веб-технології, такі як HTML, CSS, JavaScript та такі платформи, як PhoneGap/Cordova, щоб забезпечити подібний досвід. Це один з широко популярних фреймворків для створення багатофункціональних сучасних та елегантних додатків.

React Native Framework: Створюйте «нативні» додатки за допомогою React.

Розроблений спільнотою Facebook, React Native - це кросплатформний фреймворк, заснована на технології JavaScript. Це дозволяє розробникам створювати вдосконалені та подібні до програми додатки, використовуючи єдину базу коду. Метою цього фреймворкує створення першокласного нативного досвіду застосування за допомогою JavaScript та React, а також за допомогою нативних засобів iOS та Android розробки.

Технологічні розбіжності.

Ionic пропонує повний SDK для створення гібридних додатків, тоді як мета React Native – використовувати теорію «навчись один раз, пиши де завгодно». React Native сприймає звичну поведінку та стандарти платформ, а отже, дає безперебійний досвід. Тут перевага полягає в тому, що він дає повністю фокусуватися на користувальницьких інтерфейсах, реалізованих за допомогою нативних компонентів інтерфейсу.

Якщо говорити про Ionic, він охоплює багато класичних веб-технологій для створення багатофункціональних, багатоплатформних додатків з мінімальною базою коду. Крім того, він побудований на вершині Angular, на відміну від React Native, який побудовано на вершині React.

Отже, якщо вже є навички роботи з Angular, буде не важко починати з Ionic. Він також має багато вбудованих компонентів, які роблять розробку простішою, швидшою та плавнішою.

Продуктивність.

React є більш стабільним, а також ідеально підходить для масштабних застосувань. Він покликаний забезпечити високу ефективність та чуйність.

З Ionic іноді можуть виникнути проблеми з продуктивністю, оскільки існує багато зворотних викликів до рідного коду, а також необхідно завантажити різні плагіни для доступу до нативних функцій. Ці плагіни містять багато зайвого коду, але як правило, потрібно лише декілька функцій.

Плагіни та спільнота.

Існує величезна кількість плагінів, доступних для React Native. Крім того, React Native має широке співтовариство, яке дуже допомагає знаходити

важливу інформацію для вирішення проблем. Також є плагін для використання плагінів PhoneGap/Cordova, які допомагають отримати доступ до багатьох готових плагінів з інших спільнот.

У Ionic також існує співтовариство, яка може допомогти у вирішенні проблем. Для отримання нативних функцій використовується багато плагінів PhoneGap/Cordova. Але все ж таки Cordova плагіни можуть в основному використовувати браузерне відображення, тому виникає спірне питання щодо нативності деякий реалізацій.

Платформи.

Обидва фреймворки підтримують Android або iOS.

Обидва фреймворки мають свої переваги та недоліки, допомогають одразу створити кросплатформений додаток з мінімальними зусиллями, але зважаючи на нативну реалізацію React Native, та кращу продуктивність і можливості, було обране саме його.

1.2.2 Порівняння MobX з Redux

Mobx, як і Redux є популярним вибором, коли справа стосується керування станом додатка. Давайте оглянемо деякі основні відмінності Mobx від Redux:

а) у Mobx зберігаються денормовані дані. У Redux в основному зберігаються лише нормалізовані дані;

б) Mobx в основному використовує спостережувачі для зберігання даних. Redux в основному використовував об’єкт Javascript для зберігання даних;

в) у Mobx оновлення можна робити автоматично за допомогою спостережуваного атрибута. У Redux оновлення потрібно контролювати вручну;

г) у Mobx стани можуть бути перезаписані, які також називаються нечистими станами, оскільки стан можна оновити просто новими значеннями.

У Redux цей стан називають чистим станом, оскільки стани лише для читання, і їх неможливо просто переписати. Це означає, що він використовує незмінний стан;

д) Mobx простіший у навчанні та має просту криву навчання. Якщо хтось знає концепцію ООП, навчитися Mobx для розробників JavaScript легко. Redux важко вивчити, оскільки він має круту криву навчання. Він дотримується парадигми функціонального програмування, яка вимагає багато зусиль, щоб зрозуміти речі;

е) у Mobx багато вбудованої абстракції, що призводить до меншої кількості коду. У Redux менше абстрагування і потрібно написати більше коду;

ж) Mobx в основному використовується для розробки програми швидко і за менший час. Програми, розроблені Redux, зазвичай потребують часу через її складність;

з) Mobx є менш ретельним. Redux є більш ретельним.

Основними причинами вибору MobX стали коротший час розробки та незалежність від FLUX-архітектури, яка є основою Redux.

1.3 Технічне завдання

Темою роботи є «Розробка кросплатформного програмного продукту для збирання даних обладнання». В якості замовника програмного продукту виступає компанія, яка займається монтуванням різноманітного обладнання власного виробництва.

У цієї компаніє є власний єдиний сервер, на якому зберігаются всі дані по обладнанню, а також є сайт, який працює з цим сервером і дозволяє виконувати наступні дії:

а) редагувати дані по обладнанню;

б) додавати нові дані;

в) імпортувати дані зі старих систем;

г) імпортувати дані зі вручну створених звітів;

д) створювати звіти;

е) контролювати робочих, які встановлюють обладнання;

ж) створювати проекти по встановленню та обслуговуванню обладнання;

з) назначати робітників, менеджерів, супервізорів на ці проекти.

Перевагою створенної системи є те, шо вона вже доступна майже для всіх сучасних пристроїв. І кожен, хто має доступ, а саме працює в цій компанії, може зайти та відрегувати дані по проекту, над котрим він працює або по проекту, яким він керує.

Але виникає одна проблема, не кожен замовник обладнання знаходиться на території з хорошим покритям Інтернету. А іноді навіть немає можливості подзвонити з тих місць. Тому компанія вирішила створити додаток, який би допоміг вирішити цю проблему, та полегшив роботу тим, кто встановлює обладнання. Адже, якщо не має доступу до Інтернету, тоді працівник повинен записувати всі дані по обладненю на папері, або в телефон. Щоб потім, коли повернеться з виклику, перенести ці дані у систему. Це накладається ще більше, коли потрібно встановити обладнання декільком замовникам.

Такий підхід створює багато незручностей, тому що папери можно втратити або невірно записати номер частин обладнання і тоді потрібно буде повертатися знову та записувати всі дані ще раз.

Сам додаток повинен працювати лише з частиною функціонала існуючої системи. А саме повинен дати можливість заповнювати форми по обладнанню, а потім автоматично, як з’явиться Інтернет, відправляти ці дані на сервер. Це можуть бути будь-які дані, як текстові дані, так і медіа файли. А також для деяких полів форм він повинен зберігати списки номерів обладнання, щоб користувач міг обрати їх, а не вводити вручну. Все це зменшить час на завповнення даних, а також значно зменшить кількість помилок при заповнюванні.

Виходячи із зазначеного вище, додаток повинен мати наступний функціонал:

а) працювати в онлайн та в офлайн режимах;

б) в офлайн режимі повинен відображати лише ті проекти, які взято в роботу;

в) давати доступ до системи лише визначеному списку користувачів;

г) мати можливість пошуку даних;

д) мати можливість фотографувати та записувати відео;

е) зберігати списки даних з інформацією про типи обладнання, щоб дати можливість користувачу обирати ці дані з випадаючого списку, а не вводити вручну;

ж) мати можливість клонувати вже заповнені форми, щоб скоротити час при заповнені наступної схожої форми.

2 ПРОЕКТУВАННЯ ПРОГРАМНОГО ПРОДУКТУ

2.1 Проектування додатка з погляду користувача

Для того щоб мати повне представлення того, як повинен працювати додаток, необхідно уявити себе користувачем. Продумати всі можливі нюанси і помилки, які можуть виникнути в ході роботи. Правильно обробляти їх, щоб було зрозуміло що на даний момент відбувається. Щоб користувач міг виконати певні дії для усунення помилок.

Основні критерії якості мобільного додатка [2]:

а) дизайн – додаток повинен забезпечувати користувальницький інтерфейс і інтуїтивно зрозуміле взаємодію з користувачем;

б) продуктивність і стабільність – додаток повинен функціонувати на всіх сенсорних пристроях, для яких розроблюється;

в) контент – коректне відображення всіх елементів;

г) швидкість відгуку – при використанні функціоналу програми-відкриття меню, натискання кнопки, пошук, користувач повинен отримувати очікувану реакцію (відгук) від додатка;

д) позаштатні ситуації – додаток має адекватно реагувати на позаштатні ситуації (при потрапляння в кишеню без блокування екрану, натискання на кілька пунктів одночасно і т.д.) Необхідно перевірити: відсутність порожніх екранів; одночасне натискання на всі клавіші; жести непередбачений функціоналом;

е) переривання роботи програми – додаток має адекватно реагувати на переривання роботи під час дзвінка або вхідного повідомлення;

ж) мультиплатформеність – додаток має правильно відображатися на різних типах пристроїв. При переходах від горизонтального до вертикального положення. Має правильно робити на різних пристроях і операційних системах.

На діаграмі (див. рис. 2.1) зображено взаємодію користувача з додатком, його основний потік роботи.

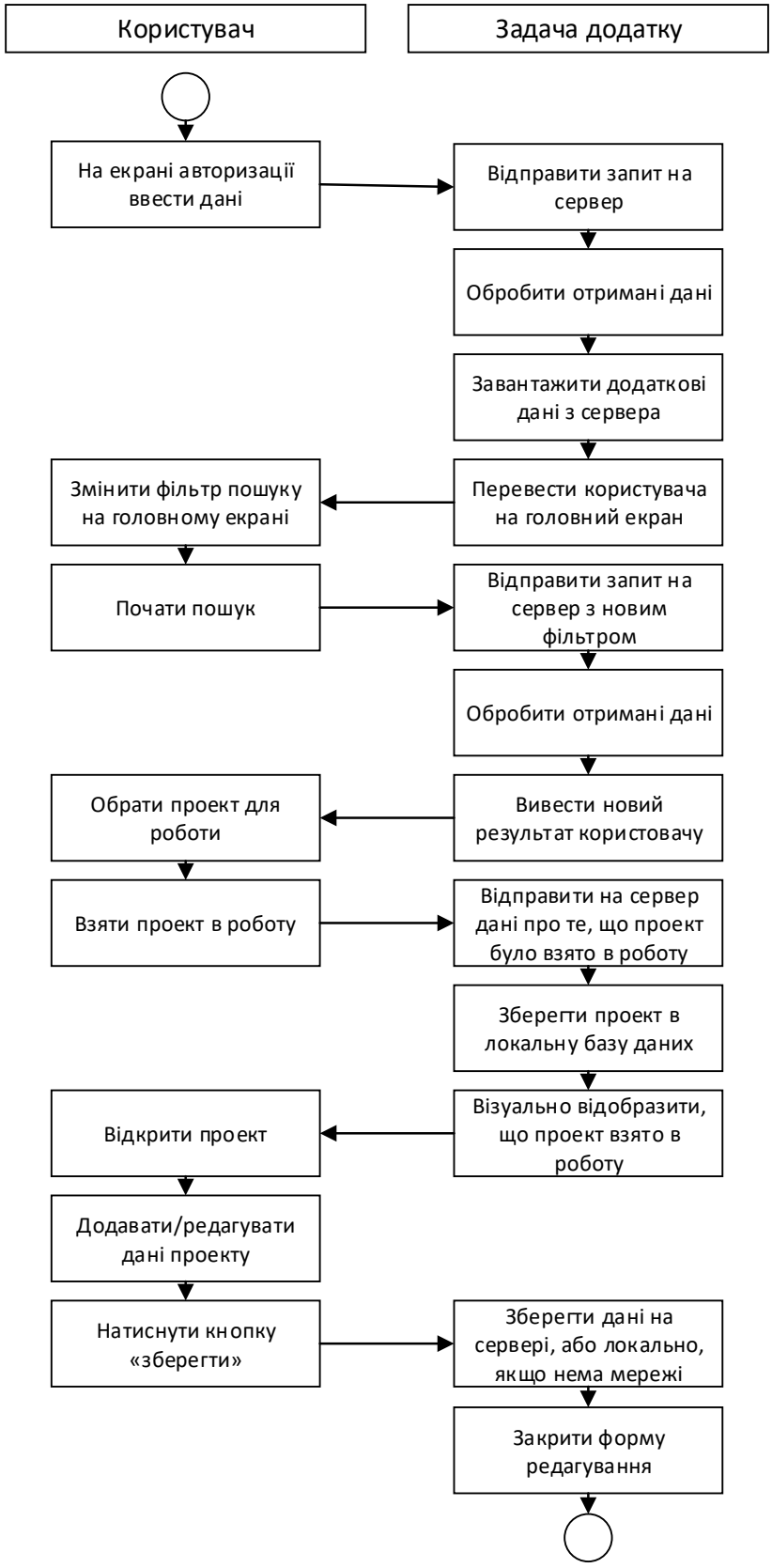

Рисунок 2.1 – Діаграма діяльності користувача

Далі розглянемо діаграми взаємодії при утворенні користувачем помилок при заповнені даних форм (див. рис. 2.2) та авторизаціїї (див. рис. 2.3).

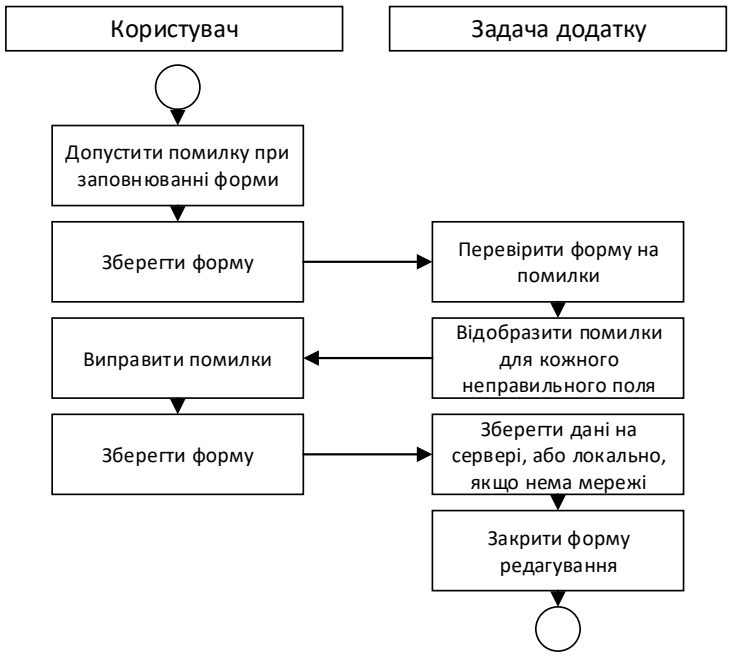

Рисунок 2.2 – Діаграма діяльності користувача при утворені ним помилок

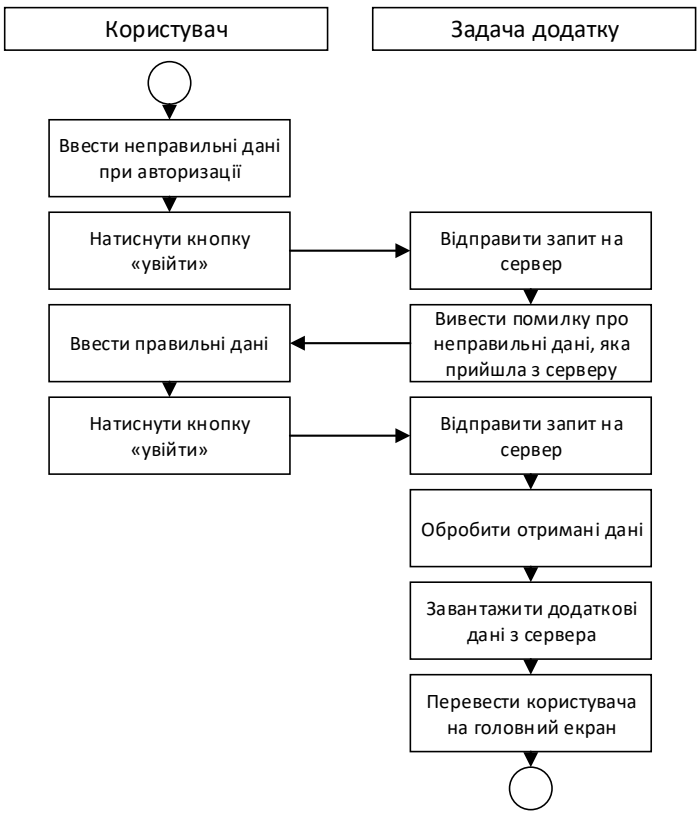

Рисунок 2.3 – Діаграма діяльності користувача при утворені ним помилок при авторизації

Також важливим є зовнішній вигляд додатка. Для найкращого користувальницького досвіду було взято мокапи, зроблені дизайнером. Мокап – це шаблон за яким потрібно робити користувальницький інтерфейс. Всі мокапи зберігаються в сервісі InVision.

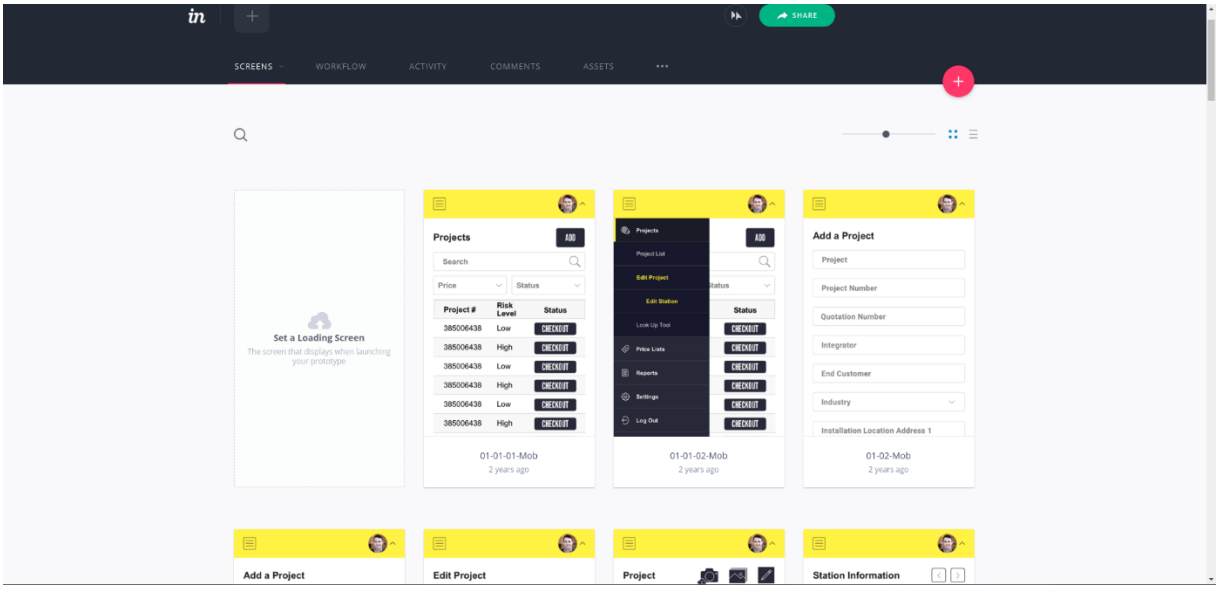

Рисунок 2.4 – Мокапи додатка

Головною переваго цього сервісу є можливість інтеракції з інтерфейсом. Можна переходити на інші мокапи, натиснувши кнопку на мокапі або побачити випадаючий список, якщо такий є.

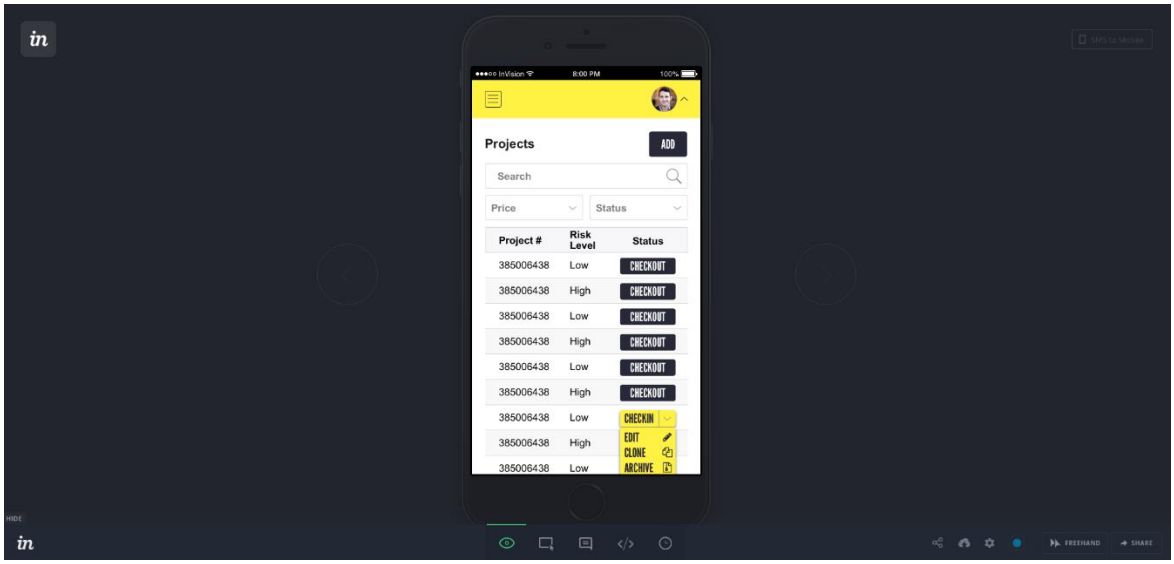

Рисунок 2.5 – Мокап в режимі інтеракції

2.2 Архітектура програмного продукту

2.2.1 Планування архітектури бізнес-логіки додатка

Архітектура додатка буде побудована відштовхуючись від того, як додаток повинен поводити себе при взаємодії с користувачем, а також від того, що потрібно реалізувати синхронізацію даних.

Основою проекту є частина коду, сценарій, який керує логікою всього додатка. Він повинен враховувати різні ситуації при кожному збереженні або считуванні даних. Цей сценарій повинен включати в себе набір файлів, кожний з яких має реалізовувати свій функціонал. Також повинен бути головний файл сценарію, котрий буде відповідати за загальний стан додатка, а також безпосередньо або побічно буде взаємодіяти з іншими сценаріями.

Головний файл сценарію повинен виконувати наступні функції:

а) бути вхідною точкою, а саме завантажувати необхідні дані, які потрібні для правильної роботи додатка – це дані про користувача та списки даних з інформацією про типи обладнання;

б) стежити за статусом з’єднання з мережою Інтернет;

в) запускати синхронізацію даних;

г) керувати процесом авторизації користувача.

Для обміну даних з сервером потрібен файл сценарію, який буде контролювати весь потік запитів на сервер, обробляти помилки, нормалізуючи їх та буде предсталяти зручний програмний інтерфейс для спрощення і узагальнення цього процесу. Так як сервер використовує Swagger для визначення своєї специфікації, файл сценарію повинен також використовувати цю специфікацію. А головний файл сценарію в свою чергу повинен завантажувати специфікацію сервера.

Для роботи з локальною базою даних на випадок відсутності доступу до мережі Інтернет потрібно реаліувати окремий сценарій. Він має виконувати всі

операціїї зчитування, запису та видалення даних. У випадку, коли потрібно обробити декілька записів, повинен використовувати «масові» виклики.

Окремим сценарієм повинна бути реалізована синхронізація. Цей сценарій має визначити, які дані потрібно синхронізувати, а які ні. Спочатку він має синхронізувати видалені об'єкти, потім змінені, а останнім кроком повинен оновити проекти, які працівник взяв до роботи. При оновлені проектів необхідно розуміти, яку частину можна оновити, а яку ні. Для цього краще підійде окремий допоміжний сценарій, який саме буде виконувати цю роботу.

Щоб нормалізувати дані для сценаріїв роботи з базою та синхронізації повині бути використовані моделі даних. Це набір об'єктів, поділених за типом. Кожен тип об'єктів повинен мати однакові поля, це значно полегчить роботу з даними.

Прошарком між базою даних та моделями повинен виступати сценарій, який реалізує всі базові операції читання та записування цих моделей в базу. Кожна модель повина наслідувати цей сценарій, і якщо необхіно, переписувати реалізацію базового сценарію. Тобто кожна модель являє собою окрему одиницю колекції, яка зберігається в базі.

Кожна модель може зберігатися в списку. Список – це об’єкт, зав’язаний на моделі, він виконує функції пошуку та зберігання набору моделей. Він повинен визначати, викликаючи головний сценарій та перевіряючи наявність доступу до мережі Інтернет, який пошук потрібно зробити: звернутися на сервер або знайти в локальній базі даних. Також, якщо в локальній базі даних наявні змінені, але ще не синхронизовані об’єкти, він повинен визначити, який об’єкт новіший і відобразити його.

Щоб для кожної моделі не перевіряти наявність мережі та не визначати «свіжість» об’єктів, для списків також потрібно створити прошарок, на подобу до прошарка, який повинен бути використаний в моделях. Таким чином кожний список моделей має також наслідувати цей проміжний сценарій.

Деякі форми в додатку необхідно заповнювати покроково. Щоб контролювати взаємодією між кроками, також потрібно реалізувати окремий

сценарій. Він має містити в собі інформацію про наступний, поточний та попередній крок.

Тому що дані в додатку в основному представлені списками, то при оновлювлені обє'кту, потрібно оновити його в списках. Для цього потріно створити механізм, який буде синхронізувати списки, а саме додавати нові елементи, оновлювати існуючи, видаляти непотрібні. Цей сценарій не взаємодіє з базою даних, та не відправляє запити на сервер, але схожий на головний сценарій та існує протягом всієї роботи додатка.

Також потрібно створити сценарій, який буде перевіряти чи було оновлено додаток. У разі, якщо додаток було оновлено, цей сценарій має видалити всі дані, які можуть спричинити конфлікти при синхронізації з сервером. Якщо додаток було встановлено вперше, цей сценарій не повинен нічого робити, а лише записати в локальне сховище інформацію про поточну версію. Також, він повинен оновити цю версію, при першому запуску після оновлення додатка. Таким чином, він кожного разу, при старті додатка, буде звіряти версію додатка зі збереженою версією, і на основі цього порівняння визначатиме наступні дії.

Виходячі з написаного вище, доречним буде, якщо головний сценарій, сценарії для синхронізації, роботи з базою, взаємодії з сервером та сценарій роботи з списками будуть реалізовувати патерн сінглтон.

На додачу до патерна сінглтон, сценарій роботи з списками має реалізовувати патерн наглядача.

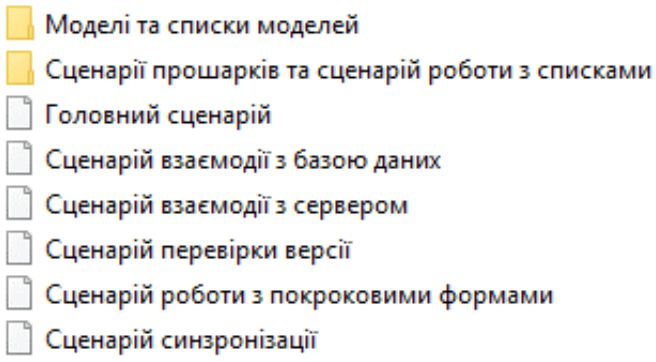

Рисунок 2.6 – Візуалізація архітектури

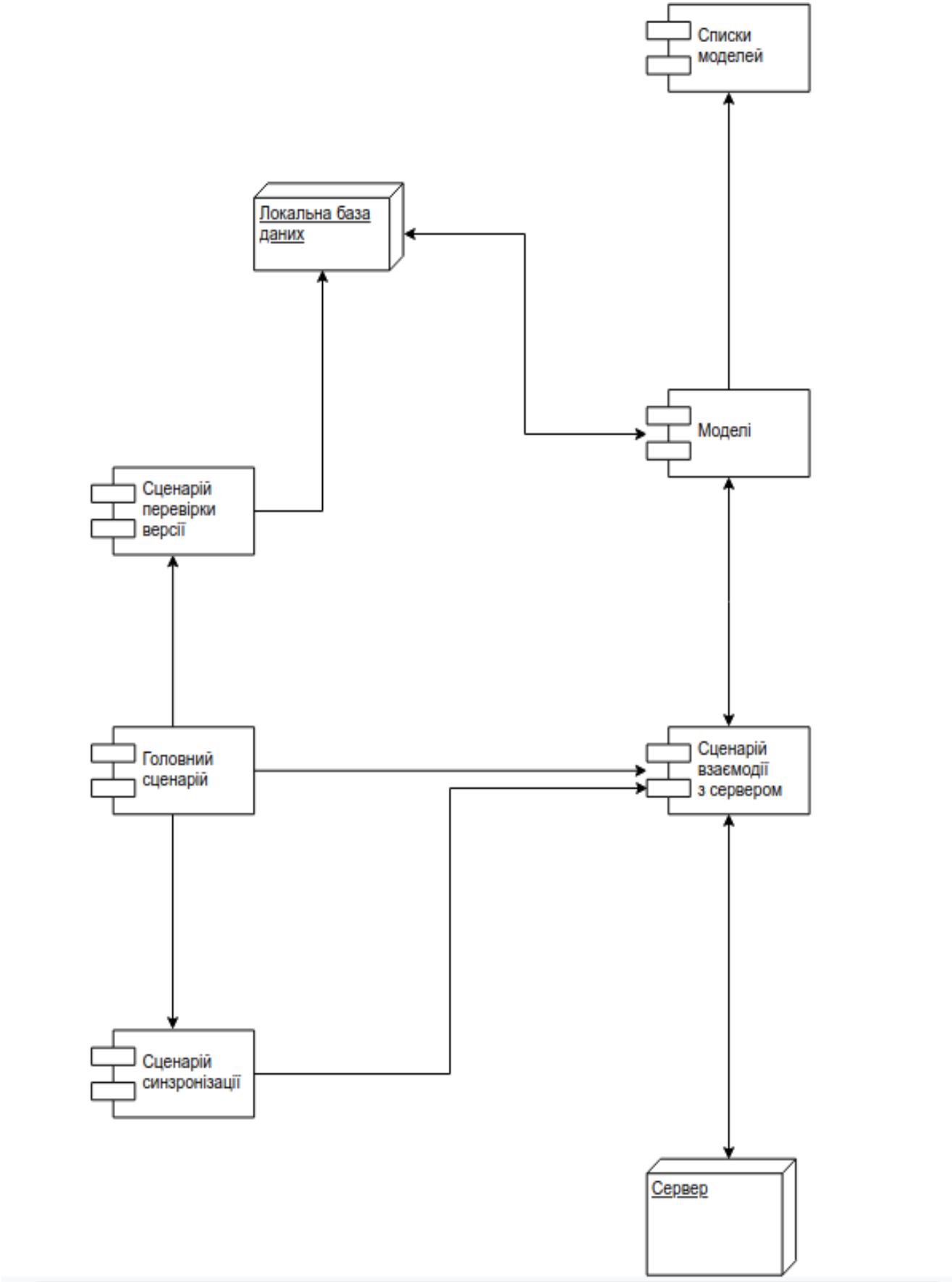

Рисунок 2.7 – Діаграма компонентів додатка

На рисунку 2.8 зображено алгоритм подій при авторизації користувача за умов наявності та відсутності з’єднання з мережою Інтернет.

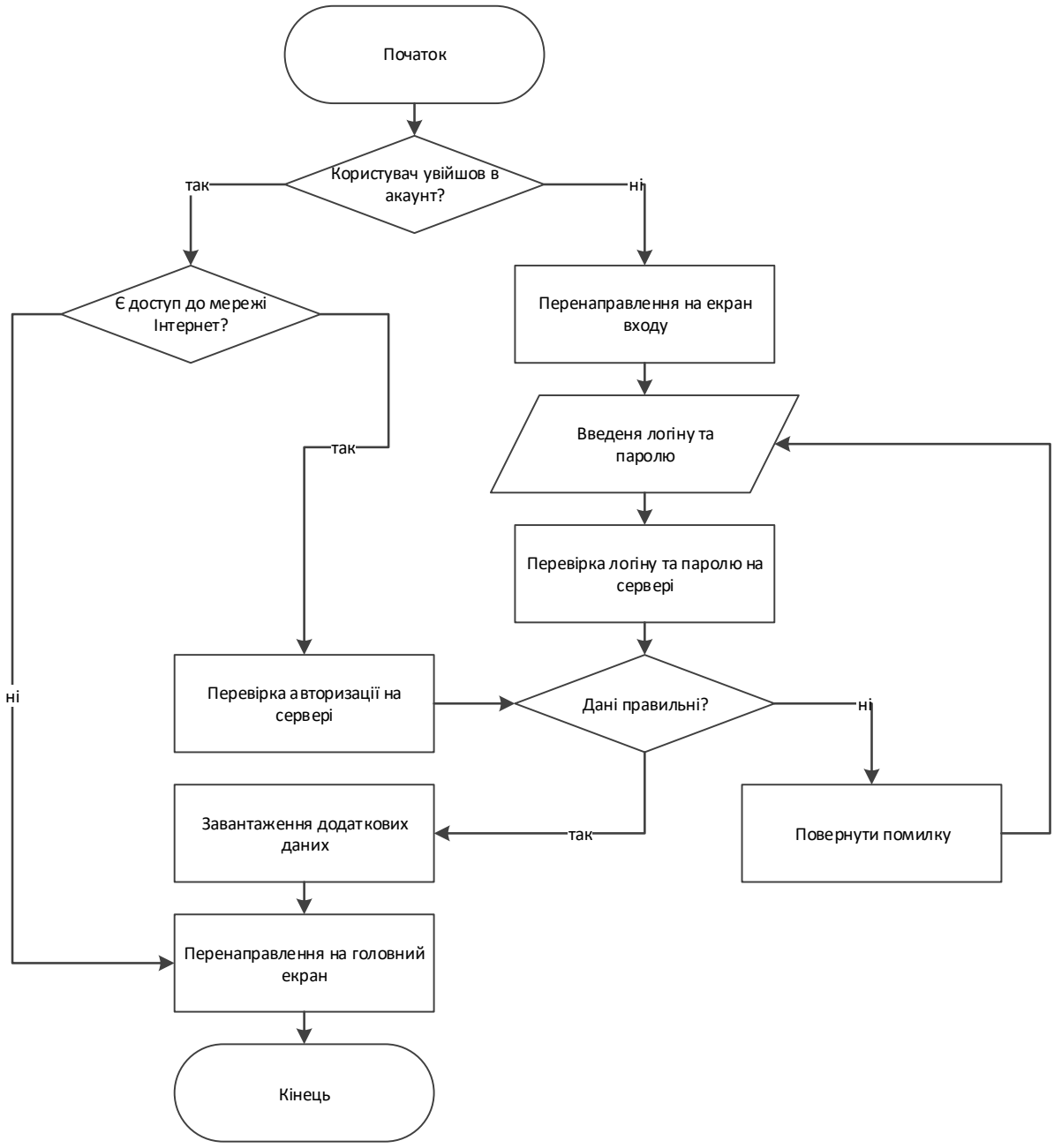

Рисунок 2.8 – Блок-схема алгоритму обробки події авторизації користувача

На наступній блок-схемі (див. рис. 2.9) зображено алгоритм дій у разі зміни користувачем фільтра пошуку, а також дій при при пошуку після переходу на екран списку моделей.

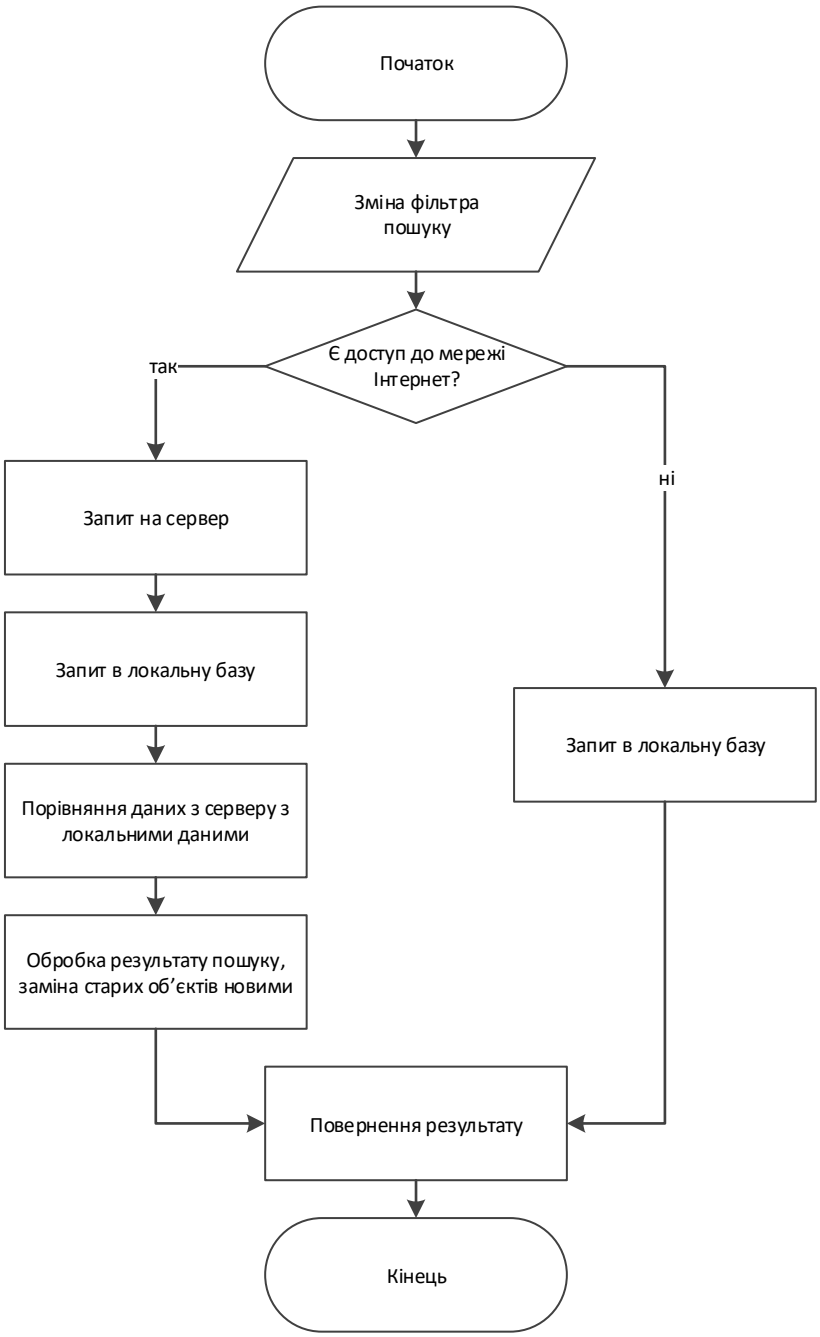

Рисунок 2.9 – Блок-схема алгоритму обробки подій при зміні фільтра пошуку

На третій блок-схемі (див. рис. 2.10) зображено алгоритм дій при відкриванні користувачем форми для редагування. У цьому випадку дані для редагування беруться із списків моделей, а списки типів обладнання завантажуються з локальної бази. Списки типів обладнання мають завантажуватися при старті додатка, тому немає необхідності робити запит на сервер. При винкненні помилки під час збереження даних, можна або вивести її

користувачу, або, якщо помилка була черезу втрату з’єднання, зберігти дані в локальну базу даних для подальшої синхронізації.

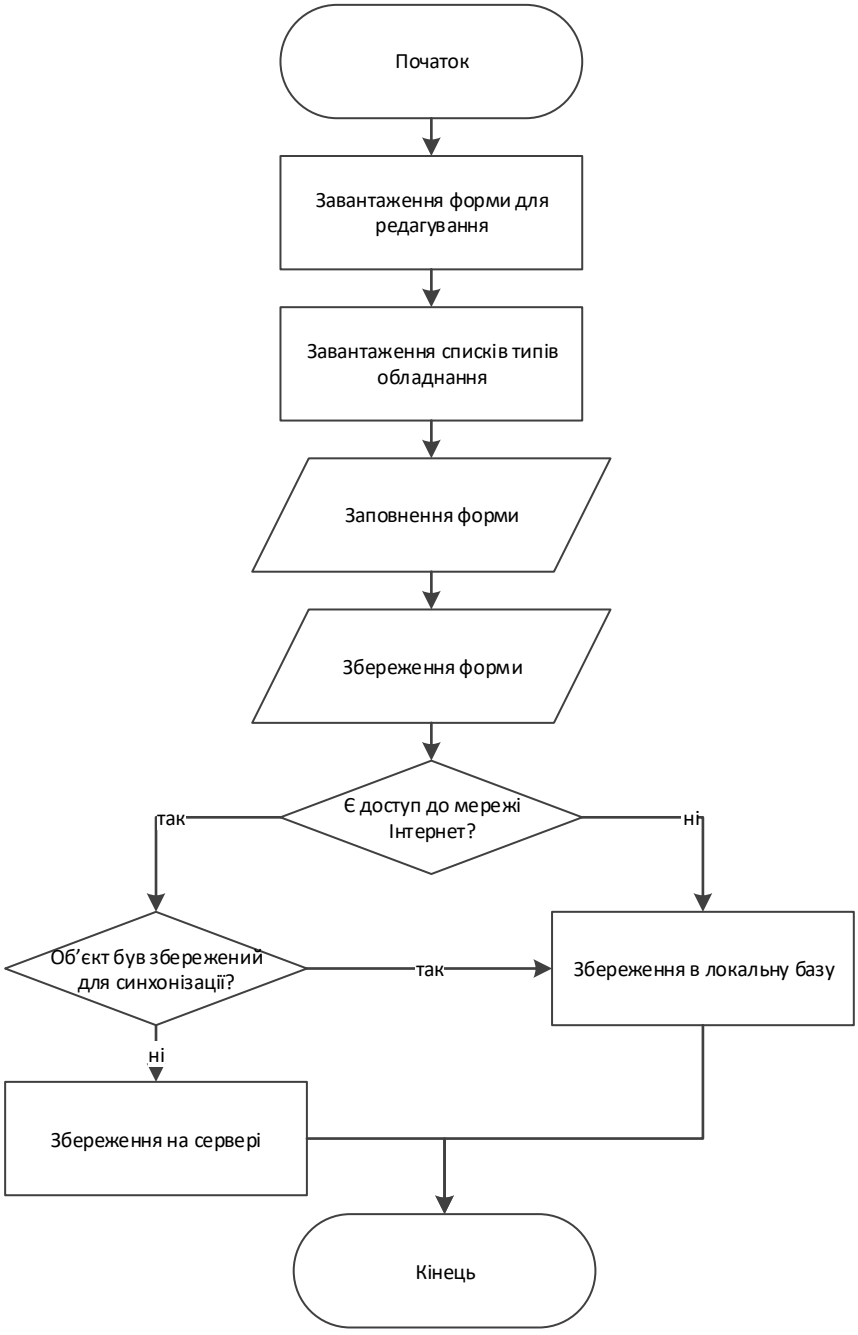

Рисунок 2.10 – Блок-схема алгоритму обробки події редагування

На наступній схемі (див. рис. 2.11) зображено алгоритм синхронізації даних з сервером. Сама синхронізація запускається головним сценарієм у двох випадках: при пові з’єднання з мережою Інтернет або раз в п’ять хвилин. Щоб уникнути помилок, процес синхронізаціє може бути лише один на момент часу.

Рисунок 2.11 – Блок-схема алгоритму синхронізації даних з сервером

На п'ятій блок-схемі (див. рис. 2.12) зображено алгоритм дії у разі, якщо додаток було оновлено. Це потрібно для того, щоб видалити застарілі дані, які сервер вже не зможе прийняти.

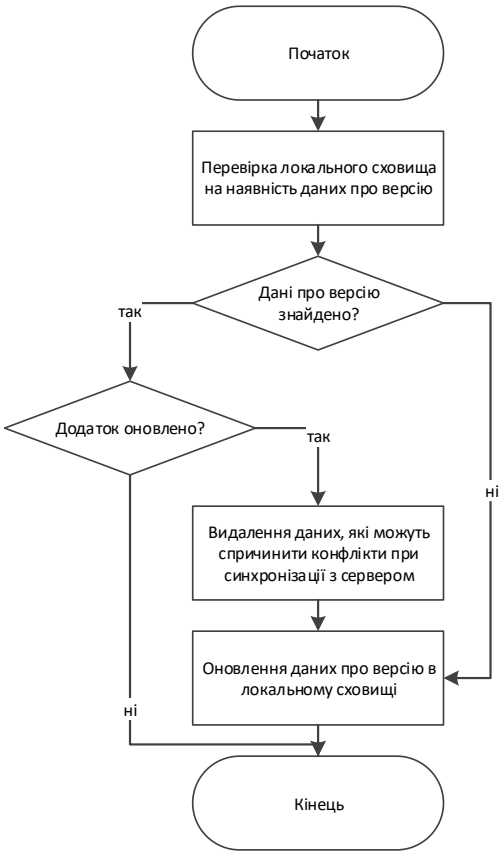

Рисунок 2.12 – Блок-схема алгоритму дій при оновлені додатка

2.2.2 Планування архітектури візуальної частини додатка

Крім сценаріїв контролю за станом та сценаріїїв для синхронизації даних, в додатку повинно бути реалізовано сценарії візуальної частини. Вони будуть описувати поведінку додатка та визначатимуть його вигляд, який буде бачити користувач на екрані свого смартфону.

Головним сценарієм візуальної частини повинен бути сценарій контролю переходів між екранами – маршрутизатор. В ньому повинні бути прописані всі екрани, які застосовуються в додатку, повинен бути реалізований з використанням бібліотеки React Router та має бути «обгорнутим» MobX сховищем для контролю стану маршрутизатора.

Маршрутизатор також повинен бути «обгорнутим» «постачальником» теми. «Постачальником» теми для додатка має бути використано Shoutem UI, який дозволить писати CSS-подібні стилі, а також вже містить в собі набір готових компонентів та стилів. Таким чином тема буде працювати через контекст, але в одночас назви класів стилів можна буде передавати через властивості компонентів.

Наступними у ієрархії візуальної частини є екрани. Це сценарії для візуалізаціїї всіх екранів додатка, які будуть використовуватися в маршрутизаторі. Вони мають бути пов'язані зі сховищем MobX та бізнес-логікою додатка та повині відображати списки даних, форми редагування тощо. До компонентів всередні екрану можна застосовувати стилі Shoutem UI, а також перевизначати стилі для вже стильозованих компонентів.

Щоб декомпозувати код для екранів та максимально його перевикористовувати потрібно реалізовувати компоненти, які будуть виконувати певний ряд дій. Ці компоненти можуть бути як без стану, так і з ним. Також вони можуть бути під’єднанні до бізнес-логіки додатка, якщо це потрібно. До компонентів, як і для екранів можна застосувати стілі Shoutem UI. Наприклад до таких компонентів можна винести:

а) модальні вікна;

б) елементи форми, такі як поля вводу, вибору дати, вибору із списку, автозаповнення;

в) додаткові контейнери;

г) вікно для відображення фото;

д) плеєр для програвання відео;

тощо.

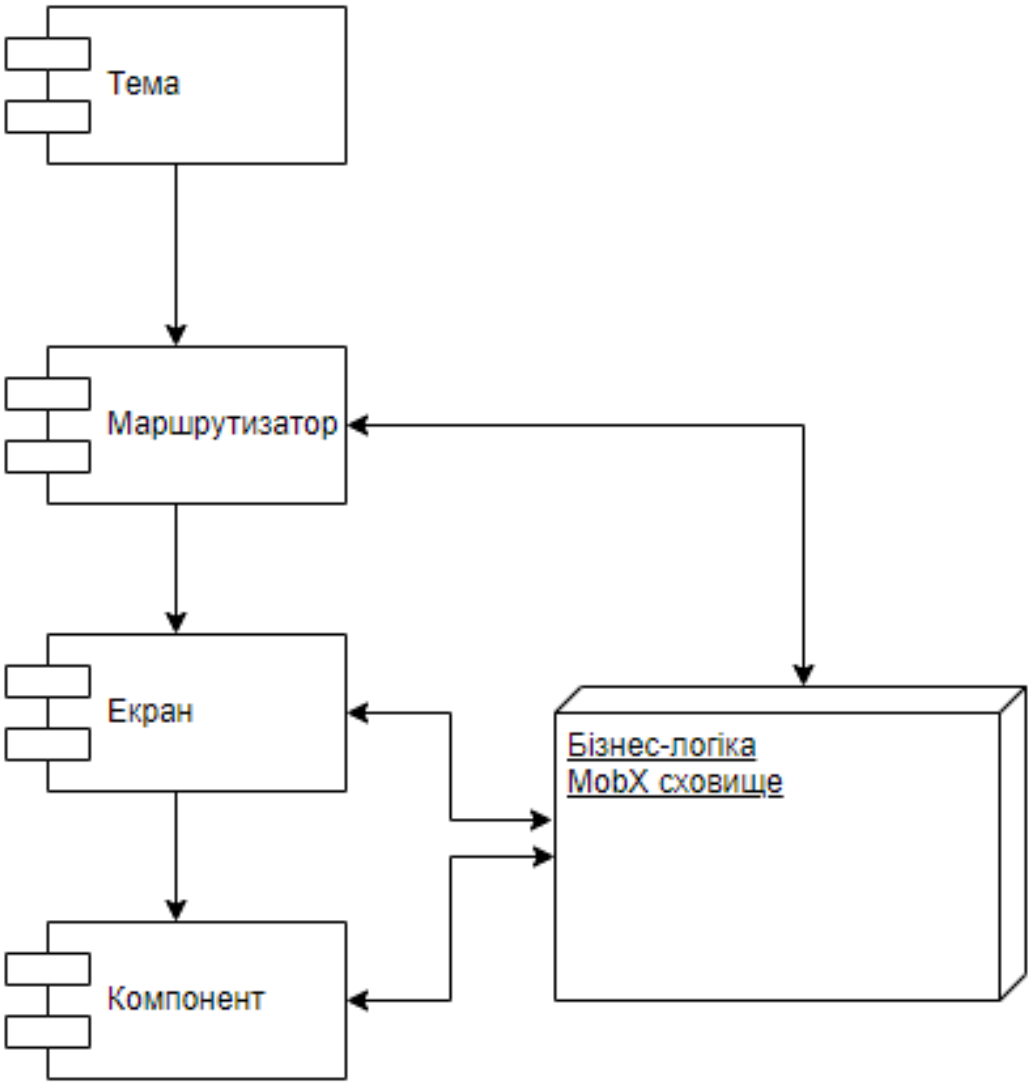

Рисунок 2.13 – Діаграма візуальних компонентів додатка

На наступній блок-схемі (див. рис. 2.14) відображено загальний цикл роботи при візуалізації компонентів. Це спрощена схема роботи React та її можна застосувати до будь-якого додатка.

Рисунок 2.14 – Блок-схема алгоритму циклу візуалізації компонентів

3 РЕАЛІЗАЦІЯ ДОДАТКА

3.1 Реалізація компонентів додатка

Як було описано в 2 розділі, для додатка потрібно створити головний файл сценарію, який буде існувати протягом всієї роботи додатка та буде контроювати основні процеси додатка.

```
class Main {
	@observable isConnected: boolean = false;
	@observable isLoggedIn: boolean = false;
	@observable isLoading: boolean = false;
	@observable wizard: Wizard;
	@observable error: string;
	@observable wasReset: boolean;
	_syncReaction = reaction(() => this.isReachable, reachable => reachable && sync(this.user.id));
	get user() {…}
	@computed get isReachable() {…}
	@action async init() {…}
	@action login(login, password) {…}
	@action logout() {…}
	@action async resetPassword(email) {…}
	@action createWizard(station, project, stations) {…}
	@action disposeWizard() {…}
}
const MainInstance: Main = new Main();
export default MainInstance;
```

Рисунок 3.1 – Вміст головного файлу сценарію

На рисунку 3.1 зображені поля та функції головного сценарію. Поля сценарію позначені декоратором «@observable» [8], це декоратор MobX. За допомогою цих декорторів визначається зміна значення та визначається чи потрібно перемальовувати візуальні компоненти.

Функція «get isReachable» відповідає за перевірку стану з'єднання з мережою Інтернет. Вона позначена декоратором «@computed» [7], і якщо ця функція почне повертати змінене значення, вона спричние виклик «_syncReaction», яка почне синхронізувати дані у випадку, якщо з'явилося з'єднання з мережою інтернет. «_syncReaction» також працює через MobX за допомогою функції «reaction» [9].

Інші функції позначено декоратором «action», він потрібен тому, що вони змінюють поля позначені декоратором «observable».

Розлянемо ці функції:

а) «get user» – повертає об'єкт з даними користувача, який увійшов в систему;

б) «async init» – асинхронно завантажує всі дані, необхідні для роботи додатка;

в) «login» та «logout» – для авторизації та виходу користувача із системи;

г) «resetPassword» – відправляє запит на сервер для скидання паролю;

д) «createWizard» та «disposeWizard» – для створення та видалення сценарію контроля за заповненням покрокових форм.

Останні два рядки коду – це своєрідна реалізація патерну синглтон у JavaScript. На предостанньому рядку створено об'єкт класу «Main», а на останньому рядку цей об'єкт експортується із файлу. Таким чином при імпорті із цього файлу буде доступний лише цей об'єкт, а можливості створити новий не буде.

На наступному рисунку 3.2 зображено сценарій для роботи з локальною базою даних, а саме з NoSQL базою даних PouchDB.

```
…
const SQLiteAdapter = SQLiteAdapterFactory(SQLite);
const storagePlugin = {
        getItem: function(key) {…},
        setItem: function(key, value) {…},
        bulk: function(values) {…},
        removeItem: function(key) {…},
};
PouchDB.plugin(SQLiteAdapter);
PouchDB.plugin(PouchDbFind);
PouchDB.plugin(storagePlugin);
PouchDB.plugin(PouchDbErase);
…
```

Рисунок 3.2 – Код сценарію для роботи з локальною базою даних

На прикладі цього коду можна побачити, що об’єкт для роботи з локальною базою даних ініціалізується за допомогою плагінів. Плагіни «SQLiteAdapter», «PouchDbFind» та «PouchDbErase» – це бібліотеки які призначенні для спрощення роботи з PouchDB, а плагін «storagePlugin» написаний власноруч для розширення функціоналу PouchDB.

SQLiteAdapter – це адаптер PouchDB, який використовує ReactNative SQLite в якості резервного сховища. Оригінальна реалізація PouchDB використовує AsyncStorage, який пише всі дані в файл, це дуже сповільнює запити до бази. За допомогою цього плагіну вдається скоротити час запитів приблизно в 30 разів.

PouchDbFind – значно спрощує код, який потрібно писати для запитів. Робить запити більш подібними до MongoDB запитів.

PouchDbErase – дозволяє видалити повністю базу, це дуже зручно коли наприклад, потрібно швидко позбутися всіх даних.

Власноруч написаний плагін – storagePlugin полегшує виклик наступних операцій: читання, збереження, збереження великої кількості даних, видалення.

На рисунку 3.3 зображено реалізацію сценарію для синхронізації змінених даних з сервером.

```
const getCheckedProjects = async userId => {…};
const cleanDatabase = async (userId, projectIds) => {…};
const syncArchived = async () => {…};
const syncChanged = async () => {…};
const refreshProjects = async (userId, projects, projectIds) => {…};
export const sync = async userId => {…};
```

Рисунок 3.3 – Код сценарію синхронизації

Розглянемо кожну функцію синхронізації окремо:

а) «getCheckedProjects» – функція, яка знаходить проекти взяті користувачем в роботу;

б) «cleanDatabase» – функція, яка чистить локальну базу від непотрібних для синхронізації даних;

в) «syncArchived» – відправляє на сервер дані, які потрібно видалити;

г) «syncChanged» – відправляє на сервер дані, які було змінено;

д) «refreshProjects» – оновлює проекти, які було взято в роботу;

е) «sync» – експортується для використання в головному сценарії та починає весь процес синхронізації.

Процес синхронізації проходить в порядку, описаному в 2 розділі. Спочатку чиститься локальна база, потім синхронизуються видалені та змінені дані і останнім кроком оновлюються дані проектів, які було взято в роботу.

На рисунку 3.4 міститься код із файлу Registry.js, цей клас контролює стани списків моделей, а саме керує додаванням в списки, оновленням та видаленням із списків.

```
class Registry {
        entries= new Map();

        add(list, opts = {}):{…}
        remove(list) {…}
        @action.bound onSave(obj) {…}
        @action.bound onRemove(obj: ManagedObject) {…}
}

const registry: Registry = new Registry();
export default registry;
```

Рисунок 3.4 – Код сценарію для контролю списків

Цей сценарій як і головний, реалізує патерн сінглтон. Розлянемо більш детально поля та функції цього сценарію:

а) «entries» – колекція, в котрій містяться усі відкриті списки на даний момент;

б) «add» – реєструє список, за яким потрібно стежити;

в) «remove» – видаляє список, за яким потрібно стежити;

г) «onSave» – ця функція повинна викликатися кожним об’єктом при збереженні, вона або додає до списка цей об’єкт, або оновлює його;

д) «onRemove» – ця функція повинна викликатися кожним об’єктом при видаленні, вона видаляє цей об’єкт із списка.

Далі розглянемо реалізацію візуальної частини додатка. Як було зазначено в другому розідлі, головним сценарієм є сценарій маршрутизатора, він контроює стан переходів між екранами. Але також є сценаірй компоненту, який є стартовою точкою в роботі всього додатка. Він створює компонент маршрутизатора, підключає до нього тему та викликає функцію «init» головного сценарію «Main», яка завантажує всі дані, необхідні для роботи додатка.

```
@observer
export default class App extends Component<{}> {
        constructor(props, context) {
            super(props, context);

            if (Platform.OS === 'android') {
                UIManager.setLayoutAnimationEnabledExperimental &&
                    UIManager.setLayoutAnimationEnabledExperimental(true);
            }
    }
        componentDidMount() {
            Main.init();
        }

        render() {
            return (
                <StyleProvider>
                    <AppRouter />
                </StyleProvider>
            );
        }
}
```

Рисунок 3.5 – Код стартового сценарію

В конструкторі стартового сценарію, у випадку, якщо це операційна система Android, вмикається анімація для додадтку, в системі iOS вона увімкнена за замовчуванням.

В функції «componentDidMount» [10] викликається головний сценарй для ініціалізації.

Із функції «render» повертається компонент маршрутизатора, який обгорнуто темою, реалізованою за допомогою Shoutem UI. Тут починається візуалізація всього додатка.

```
const SCENES = Actions.create(
        <Lightbox>
          <Scene key="root" navBar={NavBar}>
            <Scene key="SplashScreen" component={SplashScreen} hideNavBar
initial />
            …
            <Drawer key="ProjectWizard">
              <Scene …/>
            </Drawer>
          </Scene>
          …
          <Scene key="PreviewMedia" component={PreviewMedia} />
        </Lightbox>,
);
export default function() {
        return <Router wrapedBy={observer} scenes={SCENES} />;
}
```

Рисунок 3.6 – Код маршрутизатора

Як видно на рисунку 3.6, марщрутизатор створюється за допомогою виклику «Actions.create» [11] із бібліотеки React Router. В ньому є головний елемент – «Lightbox» [13], він призначений для відображення модальних вікон. Всі модальні вікна йдуть після сцени з «key="root"», тобто в цій сцені зберігаються екрани, які не є модальними вікнами, а поза цієї сцени зберігаються лише модальні вікна. У данному випадку, сцена з

«key="PreviewMedia"» є модальним вікном. Також у цієї сцени є властивість «component», сюди передається сценарій екрану.

Сцена з «key="root"», має властивість «navBar», завдяки цієї властивості компонент панелі навігації буде доступний на всіх екранах, окрім тих, які мають властивість «hideNavBar», наприклад, як у сцени з «key="SplashScreen"».

Сцена з «key="SplashScreen"» має властивість «initial», тобто цей екран буде першим, що побачить користувач. Він буде відображатися поки не завантажиться додаток.

Сцени для покрокових форм знаходяться в «Drawer» [12]. «Drawer» – це меню, яке виїжджає, та виступає в ролі майстра покрокових форм, дозволяючи перходити від однієї форми до іншої.

Сам «Router» експортується як функціональний компонент, якому передаюься сцени та «observer» необхідний для зв'зування маршрутизатора з станом MobX.

3.2 Тестування роботи додатка

Всі перераховані раніше вимоги до додатка були реалізовані. В цьому розділі його буду протестовано і буде розглянуто його можливості і особливості.

Для тестування роботи додатка було використано бібліотеку Storybook. Вона дозволяє тестувати елементи додату в ізоляції.

```
configure(() => {require('./stories');}, module);
const StorybookUI = getStorybookUI({port: 7007, onDeviceUI: true});
AppRegistry.registerComponent('test', () => StorybookUI);
export default StorybookUI;
```

Рисунок 3.7 – Конфігурація Storybook

На рисунку 3.7 зображено конфігурацію Storybook, вона завантажує всі компоненти для тестування з директорії «stories» та за допомогою «AppRegistry.registerComponent» реєструє компонент React Native, який не залежний від додатка.

Розглянемо, наприклад тестування компоненту для введення тегів, який використовується при завантаженні медіа файлів.

```
class TagInputStory extends Component {

        this.state = {
          tags: [],
        };

      render() {
        const {tags} = this.state;
        return (
          <ScrollView>
            <TagInput
              placeholder="Search"
              tags={tags}
              onTagsChange={tags => this.setState({tags})}
              needValidate
            />
          </ScrollView>
        );
      }
}
storiesOf('Components', module).add('TagInput', () => <TagInputStory />);
```

Рисунок 3.8 – Код тестування компоненту для введення тегів

Як видно на рисунку 3.8, компонент для введення тегів ізольовано, це дозволяє протестувати його прцездатність у різноманітних випадках. Він завжди повинен правильно додавати та видаляти теги, і при кожній такій події повертати правильний масив «tags», який зберігає введені теги.

Далі на рисунку 3.9 зображено код для тестування компоненту автозаповнення. Він дозволяє протестувати правильність роботи на різних даних.

```
const randomData = getRandomData();

storiesOf('Components', module)
    .add('Autocomplete', () => (

      <View>
        <Autocomplete
          styleName="md-gutter-horizontal"
          options={randomData}
          renderLabel={opt => opt}
          placeholder="Search"
         textValue="None"
        />
      </View>

    ));
```

Рисунок 3.9 – Код тестування компоненту автозаповення

Таким чином за допомогою Storybook, виконано тестування додатка. Наступним розглянемо функціонал додатка.

Почнемо з екрану авторизації. З цього екрану користувач може увійти в систему, а також перейти на екран скидання паролю, якщо він його забув.

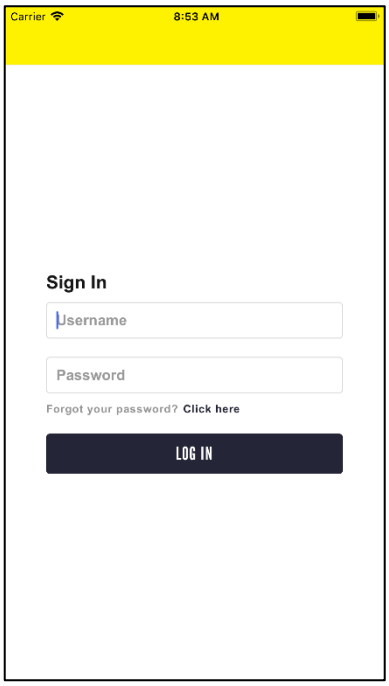

Рисунок 3.10 – Екран авторизації

Після успішного входу в систему користувач автоматично перенаправляється на головний екран (див. рис. 3.11). Тут можна шукати проекти по різним фільтрам (див. рис. 3.12), брати проекти у роботу та дивитися хто працює над проектом (див. рис. 3.13).

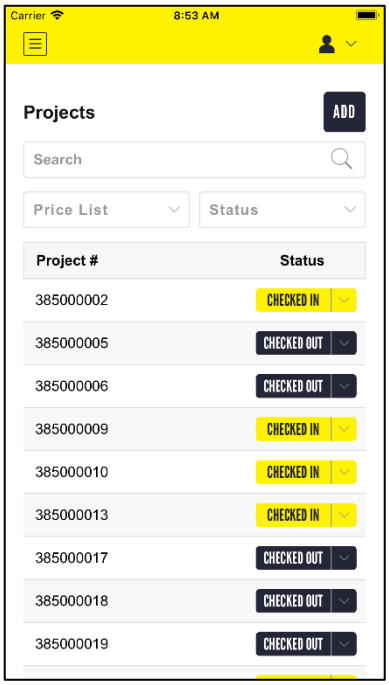

Рисунок 3.11 – Головний екран

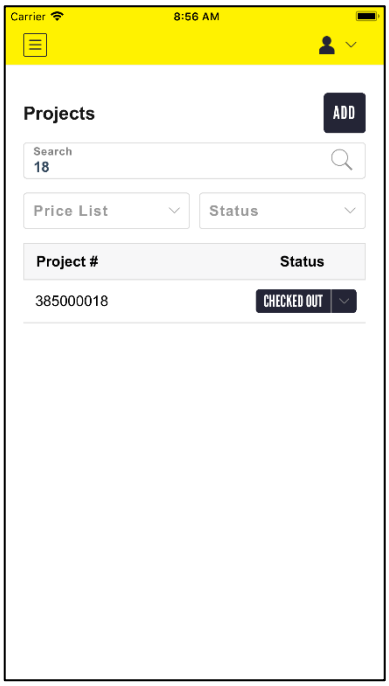

Рисунок 3.12 – Пошук на головному екрані

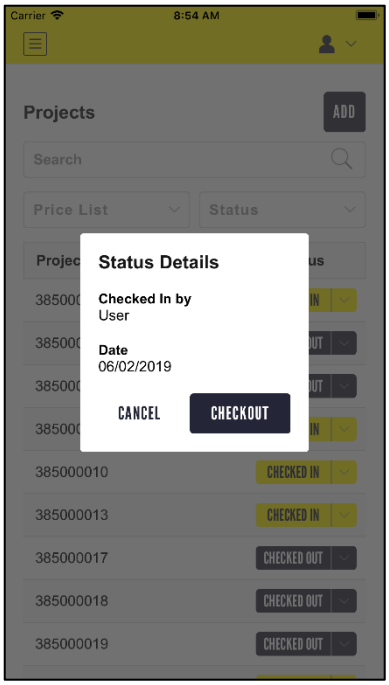

Рисунок 3.13 – Вікно статусу проекту

На рисунку 3.13 зображено вікно статусу проекту, в полі «Checked» відображаєтсья користувач, який взяв або здав проект, в полі «Date» вказана дата коли це відбулося. Для того щоб взяти проект в роботу потрібно натиснути кнопку «CHECKOUT», це дозволить редагувати його дані (див. рис. 3.14).

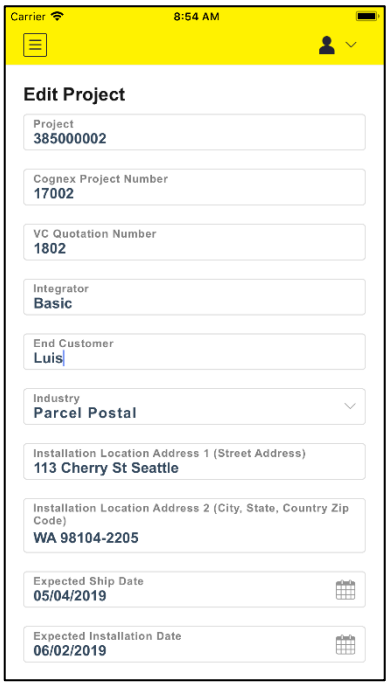

Рисунок 3.14 – Вікно редагування проекту

До проекту можна додавати фото та відео. Щоб це зробити, потрібно з екрану перегляду проекту натиснути кнопку фотоапарата, яка знаходится в правому верхньому куті. Після цього відкриється меню з пропозицією обрати фото чи відео з галереї або зробити знімок з камери (див. рис. 3.15).

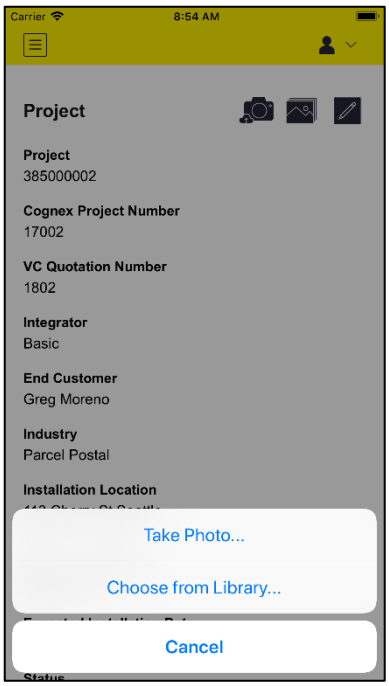

Рисунок 3.15 – Меню камери

Після того, як фото зроблено, до нього можна додати додаткові дані, такі як теги та нотатки (див. рис. 3.16).

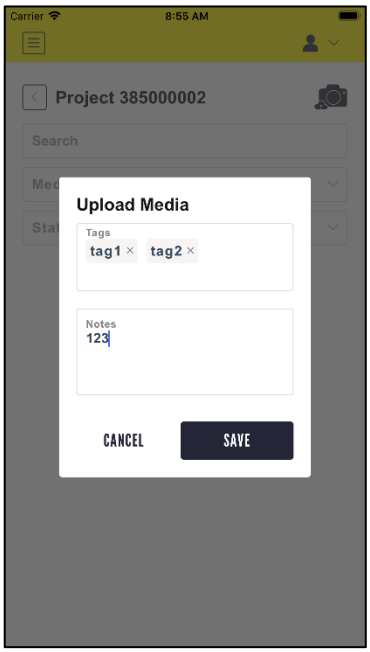

Рисунок 3.16 – Завантаження фото та відео

Після завантаження медіафайлу його дані можна редагувати, а також просто переглядати сам файл (див. рис. 3.17).

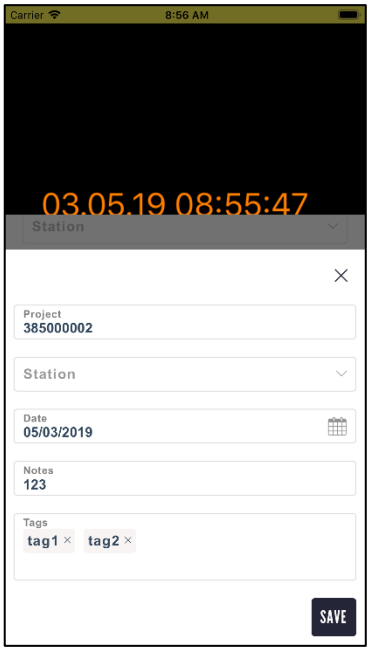

Рисунок 3.17 – Перегляд та редагування даних медіафайлу

Деякі дані проекту можна копіювати, як наприклад цю станцію (див. рис. 3.18-3.19). Станція це частина проекту, з котрою пов’язаний набір даних. При копіюванні станції всі ці дані також копіюються,таким чином скорочується час на заповнення корситувачем інформації по проекту. Щоб скопіювати необхідно з випадаючого спику (див. рис. 3.18) обрати пункт «CLONE». Після цього з'явиться нова станція (див. рис. 3.18-3.19).

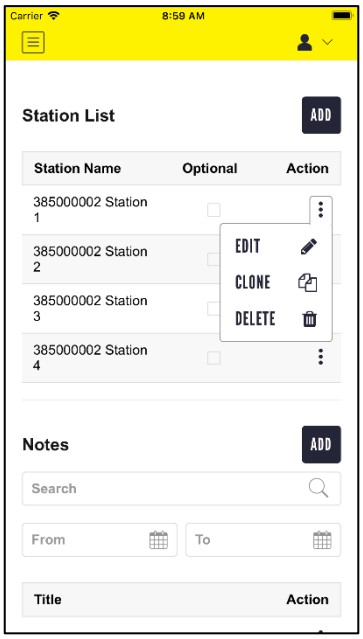

Рисунок 3.18 – Випадаючий список станції

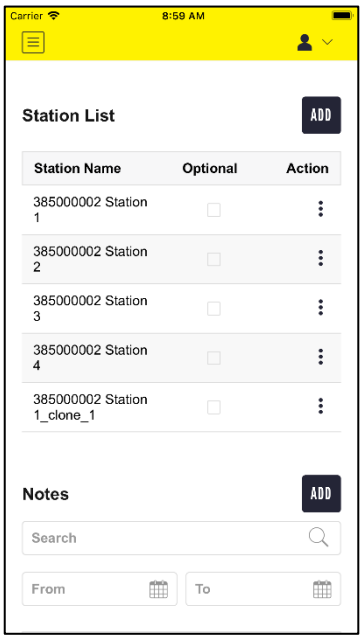

Рисунок 3.19 – Скопійована станція з’явилася у кінці списку

На рисунку 3.18 можна побачити ще два пункти випадаючого списку: «EDIT» та «DELETE». «DELETE» видаляє станцію та всі пов’язані з нею дані. «EDIT» відкриває форми для покрокового редагування (див. рис. 3.20). Ці форми призначені для редагування даних, пов’язаних зі станцією.

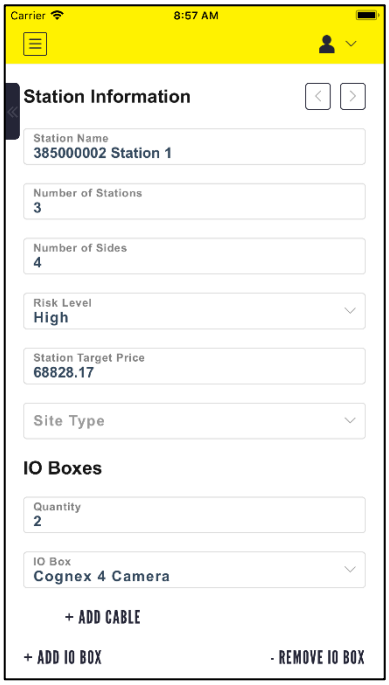

Рисунок 3.20 – Одна із покрокових форм

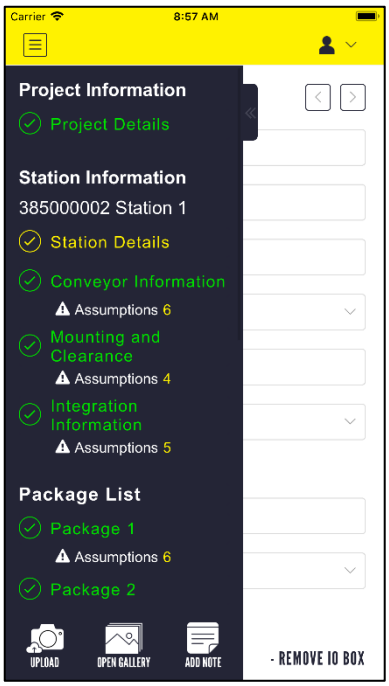

Рисунок 3.21 – Майстер переходів між покровими формами

Пересуватися між цими формами можна за допомогою стрілок «вліво/вправо», вони знаходяться в правому верхньому куті форми (див. рис. 3.20). Або за допомогою майстра переходів між покроковими формами (див. рис. 3.21).

Функціонал додатка повністю реалізовано. Він працює як за наявності мережі Інтернет, так і без неї. Всі екрани розроблено за мокапами, створеними дизайнером, тому рівень користувальницького досвіду дуже високий.

ВИСНОВКИ

В ході виконання дипломної роботи на тему: «Розробка кросплатформного програмного продукту для збирання даних обладнання» було отримано багато нових навичок та знань у області розробки мобільних додатків. Було вивчено роботу React Native у поєднанні з MobX та PouchDB.

Загалом вирішено наступні задачі та проблеми:

а) розробка кросплатформених мобільних додатків за допомогою React Native;

б) вивчення нюансів роботи React Native;

в) детальне дослідження та розбір варіантів синхронізації даних та роботи додатка в автономному режимі;

г) способи уникнення написання неоптимізованого JavaScript коду;

д) планування роботи і функціоналу додатка, відштовхуючись від поставленого технічного завдання;

е) дослідження сфери крупних компаній, які працюють з різноманітним обладнанням.

Розроблений додаток має приємний інтерфейс, швидко та стабільно працює, може оброблювати доволі великі, як для смартфону, об’єми даних без шкоди для самого смартфона.

Основними його перевагами є те, що він може працювати в автономному режимі та є кросплатформеним. Це є важливим для компаній, які працюють з обладнанням, що потребує автоматизовану систему обліку цього обладнання. Завдяки цьому додатку вони зможуть, незважаючи майже ні на що, продовжувати заповнювати необхідні дані, не турбуючись про те, що вони можуть бути загублені.

ПЕРЕЛІК ПОСИЛАНЬ

1. Ейзенман Б. Learning React Native. Ньютон: O`Reilly, 2015. 272 с.

2. Майер Р. Програмування пропозицій для планшетних комп'ютерів та смартфонів. Москва: Эксмо, 2011. 672с.

3. Резіг Д. Секрети JavaScript ніндзя. Москва: Вільямс, 2016. 416 с.

4. Стефанов С. JavaScript. Шаблони. Москва: Вільямс, 2011. 272 с.

5. Фленаган Д. JavaScript. Детальний посібник. Москва: Вільямс, 2013. 1080 с.

6. Introduction to MobX. URL : https://mobx.js.org/getting-started.html (дата звернення: 29.10.2019).

7. MobX computed. URL : https://mobx.js.org/refguide/computed-decorator.html (дата звернення: 11.11.2019).

8. MobX observable. URL : https://mobx.js.org/refguide/observable.html (дата звернення: 11.11.2019).

9. MobX reaction. URL : https://mobx.js.org/refguide/reaction.html (дата звернення: 11.11.2019).

10. React Lifecycle Methods: render and componentDidMount. URL : https://www.codingame.com/playgrounds/8747/react-lifecycle-methods-render-and-componentdidmount (дата звернення: 15.11.2019).

11. React Router Actions. URL : github.com/aksonov/react-native-router-flux/blob/master/docs/API.md#actions (дата звернення: 16.11.2019).

12. React Router Drawer. URL : github.com/aksonov/react-native-router-flux/blob/master/docs/API.md#drawer-drawer-or-scene-drawer (дата звернення: 16.11.2019).

13. React Router Lightbox. URL : github.com/aksonov/react-native-router-flux/blob/master/docs/API.md#lightbox-lightbox (дата звернення: 16.11.2019).

14. What is PouchDB. URL : https://www.javatpoint.com/what-is-pouchdb (дата звернення: 30.10.2019).

ДОДАТОК А

Код головного сценарію

```
import remotedev from 'mobx-remotedev';
import {action, computed, observable, runInAction, reaction} from 'mobx';
import {Actions} from 'react-native-router-flux';
import {NetInfo} from 'react-native';
import {isEmulator} from 'react-native-device-info';
import API from './API';
import MainUser from './models/MainUser';
import {DropdownController} from './models/dropdown';
import {KitsList, UnitsList} from './models/kit';
import Wizard from './Wizard';
import Strings from '../Strings';
import {sync} from './Synchronizer';
import {resetData} from './UpdateChecker';
import BackgroundTimer from 'react-native-background-timer';
import SwaggerProd from '../SwaggerProd';

@remotedev({name: 'Main'})
class Main {
   @observable isConnected: boolean = false;
   @observable isLoggedIn: boolean = false;
   @observable isLoading: boolean = false;
   @observable wizard: Wizard;
   @observable error: string;
   @observable wasReset: boolean;

   _syncReaction = reaction(() => this.isReachable, reachable => reachable && sync(this.user.id));

   get user() {
      return MainUser;
   }

   @computed
   get isReachable(): boolean {
      return this.isConnected && this.isLoggedIn;
   }

   @action
   async init() {
      try {
         this.isConnected = (await NetInfo.isConnected.fetch()) || isEmulator();

         // ---
         // SwaggerProd remains the same for all builds
         // Uncomment "await API.init(...);" for corresponding build
```

```
        await this.user.rehydrateHandle;
        // reset data after update
        this.wasReset = await resetData();

        if (this.wasReset) {
          this.user.resetUser();
        }

        await this._updatePrimaryData(this.user.authToken);
      } catch (error) {
        console.error('Check application init.', error);
        // retry
        setTimeout(() => this.init(), 5000);
      }
    }

    @action
    login(login: string, password: string) {
      if (!this.isConnected) {
        Actions.Info({title: Strings.SIGN_IN, message: Strings.LOG_IN_ERROR});
        return;
      }

      this.isLoading = true;
      this.user.login = login;
      this.user.password = password;

      this.user
        .doLogin()
        .then(() => this._updatePrimaryData(this.user.authToken))
        .catch(err => this._handleAuthError(err.status));
    }

    @action
    logout() {
      if (!this.isConnected) {
        Actions.Info({title: Strings.LOG_OUT, message: Strings.LOG_OUT_ERROR});
        return;
      }

      this.user.doLogout().then(() => this._updatePrimaryData());
    }

    @action
    async resetPassword(email: string): void {
      if (!this.isConnected) {
        Actions.Info({
          title: Strings.FORGOT_PASSWORD,
          message: Strings.FORGOT_PASSWORD_NET_ERROR,
        });
        return;
```

```
        }

        this.isLoading = true;
        try {
            await API.Auth.restorePassword({body: {email}});
            runInAction(() => {
                this.isLoading = false;
                Actions.Info({
                    title: Strings.FORGOT_PASSWORD,
                    message: Strings.FORGOT_PASSWORD_SUCCESS,
                });
            });
        } catch (e) {
            console.log('Reset password error', e);
            Actions.Info({title: Strings.FORGOT_PASSWORD, message:
Strings.FORGOT_PASSWORD_ERROR});
        }
    }

    @action
    createWizard(station, project, stations) {
        if (!this.wizard) {
            this.wizard = new Wizard();
        } else {
            // do not throw error in case of double tap
            console.warn('Wizard already created');
        }
        this.wizard.load(station, project, stations);
        console.log('Wizard created', {...this.wizard});
    }

    @action
    disposeWizard() {
        if (!!this.wizard) {
            console.log('Wizard disposed', {...this.wizard});
            this.wizard.dispose();
            this.wizard = null;
        }
    }

    //
    // private methods
    //
    @action.bound
    async _updatePrimaryData(token?: string) {
        API.authToken = token;
        this.isLoggedIn = !!token;
        this.error = null;

        if (this.isLoggedIn) {
            // load data
            await DropdownController.fetch(true);
```

```
            if (!__DEV__ && DropdownController.isEmpty) {
                // case: first app run
                await DropdownController.fetch();
                await UnitsList.fetch();
                await KitsList.fetch();
            } else {
                // don't wait if not first run
                DropdownController.fetch();
                UnitsList.fetch();
                KitsList.fetch();
            }

            await sync(this.user.id);

            if (this.isLoggedIn) {
                // go to main page
                Actions.Projects({type: 'reset'});
            } else {
                // set error
                runInAction(() => (this.error = Strings.LOGIN_FAILED));
            }
        } else {
            // session lost, reset user and go to login screen
            this.user.resetUser();
            Actions.Login({type: 'reset'});
        }

        this.isLoading = false;
    }

    @action.bound
    _handleAuthError(statusCode) {
        // what kind of error occur?
        // possible cases are expired token, no role permissions etc.

        switch (statusCode) {
            case 403:
                // expired token case
                this._updatePrimaryData();
                break;
            case 400:
            case 404:
                Actions.Info({title: Strings.SIGN_IN, message: Strings.INVALID_USER});
                break;
            default:
                console.warn(`no auth error handler for ${statusCode} error`);
                Actions.Info({title: Strings.SIGN_IN, message: Strings.LOG_IN_ERROR});
        }
        this.isLoading = false;
    }
}
```

```
const MainInstance: Main = new Main();

NetInfo.addEventListener('connectionChange', ({type}) =>
   runInAction(() => (MainInstance.isConnected = type !== 'none')),
);

const SYNC_INTERVAL = 300000;

BackgroundTimer.runBackgroundTimer(() => {
   if (MainInstance.isReachable) {
      sync(MainInstance.user.id);
   } else {
      console.log('Cannot sync, Main is not reachable');
   }
}, SYNC_INTERVAL);

export default MainInstance;
```

ДОДАТОК Б

Код сценарію PouchDB

```
/**
 * @flow
 * @providesModule PouchStorage
 */

import _ from 'lodash';
import SQLiteAdapterFactory from 'pouchdb-adapter-react-native-sqlite';
import PouchDbErase from 'pouchdb-erase';
import PouchDbFind from 'pouchdb-find';
import PouchDB from 'pouchdb-react-native';
import SQLite from 'react-native-sqlite-2';

const getIdFromKey = key => {
   return !_.isEmpty(key) && _.isString(key) ? key : _.toString(key);
};

const SQLiteAdapter = SQLiteAdapterFactory(SQLite);
const storagePlugin = {
   //////// DEVELOPMENT ////////
   _allDocsWithBody: function() {
      return this.allDocs({include_docs: true});
   },

   //////// PUBLIC ////////
   getItem: function(key) {
      const _id = getIdFromKey(key);
      return this.get(_id).catch(error => {
         if (error.status !== 404) {
            console.log('DB getItem: ', _id, error);
         }
         return null;
      });
   },

   setItem: function(key, value) {
      value._rev === undefined && delete value._rev;

      const _id = getIdFromKey(key);

      return this.get(_id)
         .then(doc => this.put({...doc, ...value}))
         .catch(
            error =>
               error.status === 404
```

```
                ? this.put({...value, _id})
                : console.log('error = ', error) || this.setItem(_id, value),
        );
    },

    bulk: function(values) {
        return this._allDocsWithBody().then(({rows: docs}) => {
            const docRevs = docs.reduce((obj, val) => {
                obj[val.doc._id] = val.doc._rev;
                return obj;
            }, {});

            const mappedValues = values.map(val => {
                const _id = getIdFromKey(val.id || val.localId);
                const docRev = docRevs[_id];
                val._id = _id;

                if (!!docRev) {
                    val._rev = docRev;
                }

                val._rev === undefined && delete val._rev;
                return val;
            });

            return this.bulkDocs(mappedValues);
        });
    },

    removeItem: function(key) {
        const _id = getIdFromKey(key);

        return this.get(_id)
            .then(doc => this.remove(doc))
            .catch(error => {
                console.log('DB removeItem: ', _id, error);
                return null;
            });
    },
};

PouchDB.plugin(SQLiteAdapter);
PouchDB.plugin(PouchDbFind);
PouchDB.plugin(storagePlugin);.
PouchDB.plugin(PouchDbErase);

export function createDB(dbName: string): PouchDB {
    return new PouchDB(dbName, {adapter: 'react-native-sqlite', auto_compaction: true});
}

export default createDB('cognexdb');
```

ДОДАТОК В

Код синхронізації

```
import _ from 'lodash';
import {cloneName} from '../utils/utils';
import API from './API';
import ManagedObject from './managedData/ManagedObject';
import ManagedObjectModel from './managedData/ManagedObjectModel';
import {MEDIA_CONTEXT} from './models/media/Media';
import {NOTE_CONTEXT} from './models/note/Note';
import Project, {PROJECT_CONTEXT} from './models/project/Project';
import {STATION_CONTEXT} from './models/station/Station';
import {STATION_CONVEYOR_CONTEXT} from './models/StationConveyor';
import {STATION_INTEGRATION_CONTEXT} from './models/StationIntegration';
import {STATION_MOUNTING_CONTEXT} from './models/StationMounting';
import {VIEW_BARCODE_CONTEXT} from './models/viewBarcode/ViewBarcode';
import {VIEW_CONTEXT} from './models/viewDetails/ViewDetails';
import {VIEW_PACKAGE_CONTEXT} from './models/viewPackage/ViewPackage';
import {mergeRawProject} from './sync/Merger';

const syncTasksIds = [];

const addSyncTask = () => {
   const taskId = _.uniqueId();
   syncTasksIds.push(taskId);
   return taskId;
};

const removeSyncTask = taskId => _.remove(syncTasksIds, id => id === taskId);

const getCheckedProjects = async userId => {
   const projects = await PROJECT_CONTEXT.find({
```

```
        selector: {
            checked_out: true,
            'user.id': userId,
        },
    });
    const projectIds = projects.map(o => o.id);

    return {projects, projectIds};
};

const cleanDatabase = async (userId, projectIds) => {
    console.log('*** DATABASE CLEANING STARTED ***');

    const cleaner = async (context: ManagedObjectModel, filter?: Object) => {
        const objects = await context.find({
            selector: filter || {
                project_id: {$nin: projectIds},
            },
        });

        return Promise.all(objects.map((obj: ManagedObject) => obj.remove()));
    };

    if (_.isEmpty(projectIds)) {
        // case: no data to sync, remove all items,
        // items with hasChanges == true won't be deleted
        await cleaner(PROJECT_CONTEXT, {});
        await cleaner(STATION_CONTEXT, {});
        await cleaner(NOTE_CONTEXT, {});
        await cleaner(MEDIA_CONTEXT, {});
        await cleaner(STATION_CONVEYOR_CONTEXT, {});
        await cleaner(STATION_MOUNTING_CONTEXT, {});
        await cleaner(STATION_INTEGRATION_CONTEXT, {});
        await cleaner(VIEW_CONTEXT, {});
        await cleaner(VIEW_PACKAGE_CONTEXT, {});
```

```
        await cleaner(VIEW_BARCODE_CONTEXT, {});
    } else {
        // case: has data to sync, remove items that not required to be synced
        await cleaner(STATION_CONTEXT);
        await cleaner(NOTE_CONTEXT);
        await cleaner(MEDIA_CONTEXT, {'project.id': {$nin: projectIds}});
    }

    console.log('*** DATABASE CLEANING FINISHED ***');
};

const syncArchived = async () => {
    console.log('*** SYNC ARCHIVED STARTED ***');

    const archiver = async (context: ManagedObjectModel) => {
        const objects = await context.find({
            selector: {
                $and: [{deleted_at: {$exists: true}}, {deleted_at: {$ne: null}}],
            },
        });

        return Promise.all(objects.map((obj: ManagedObject) => obj.archiveRemotely(false)));
    };

    await archiver(STATION_CONTEXT);
    await archiver(NOTE_CONTEXT);
    await archiver(MEDIA_CONTEXT);
    await archiver(VIEW_CONTEXT);
    await archiver(VIEW_PACKAGE_CONTEXT);
    await archiver(VIEW_BARCODE_CONTEXT);

    console.log('*** SYNC ARCHIVED FINISHED ***');
};

const syncChanged = async () => {
```

```
console.log('*** SYNC CHANGED STARTED ***');

const saver = async (
    context: ManagedObjectModel,
    apiSearch: Function,
    isSearchSingle? = false,
    notWait = false,
) => {
    const objects = await context.find({
        selector: {
            $and: [
                {$or: [{deleted_at: {$exists: false}}, {deleted_at: {$eq: null}}]},
                {$or: [{hasChanges: true}, {id: {$exists: false}}, {id: {$eq: null}}]},
            ],
        },
    });

    const existingIds = objects.map(o => o.id).filter(o => !!o);

    // remove objects that has been updated on server later than here
    if (!_.isEmpty(existingIds)) {
        let response = [];

        try {
            if (isSearchSingle) {
                response = await Promise.all(existingIds.map(id => apiSearch({id})));
            } else {
                const filter = {id: {$in: existingIds}};
                const {rows} = await apiSearch({filter: JSON.stringify(filter)});
                response = rows;
            }
        } catch (error) {
            console.log('Sync search error', error);
            return;
        }
```

```
        const newerObjects = context
          .deserialize(response)
          .filter(
            n => !!~_.findIndex(objects, o => n.id === o.id && n.updatedAt > o.updatedAt),
          );

        if (!_.isEmpty(newerObjects)) {
          _.remove(objects, o => !!~_.findIndex(newerObjects, n => n.id === o.id));
          await Promise.all(newerObjects.map((o: ManagedObject) => o.writeLocally()));
        }
      }

      const saveQueue = objects.map(async (o: ManagedObject) => {
        const taskId = addSyncTask();
        const saved = await o.sync();

        if (!saved && o.errorCode === 409) {
          o.changeSyncName();
          await o.sync();
        }

        removeSyncTask(taskId);
      });

      return notWait ? null : Promise.all(saveQueue);
    };

    await saver(STATION_CONTEXT, API.Station.searchStations);
    await saver(STATION_CONVEYOR_CONTEXT, API.StationConveyor.getStationConveyor,
true);
    await saver(STATION_MOUNTING_CONTEXT,
API.StationMounting.getStationMountingClearance, true);
    await saver(STATION_INTEGRATION_CONTEXT,
API.StationIntegration.getStationIntegration, true);
```

```
    await saver(NOTE_CONTEXT, API.ProjectNotes.searchNotes);
    await saver(MEDIA_CONTEXT, API.Media.searchMedia, false, true);
    await saver(VIEW_PACKAGE_CONTEXT, API.StationPackageDetails.searchPackages);
    await saver(VIEW_BARCODE_CONTEXT, API.StationBarcode.searchStationBarcode);
    await saver(VIEW_CONTEXT, API.StationViewDetails.syncStationView);

    console.log('*** SYNC CHANGED FINISHED ***');
};

const refreshProjects = async (userId, projects, projectIds) => {
    console.log('*** REFRESH PROJECTS STARTED ***');

    try {
        let filter = JSON.stringify({id: {$in: projectIds}});
        let serverProjects = await API.Project.syncProjects({filter});

        await Promise.all(
            projects.map(async (project: Project) => {
                const serverProject = _.find(serverProjects, sp => sp.id === project.id);

                if (_.isNil(serverProject)) {
                    await project.remove();
                } else {
                    await mergeRawProject(serverProject);
                }
            }),
        );

        filter = JSON.stringify({
            id: {$notIn: projectIds},
            checked_out: true,
            last_user_id: userId,
        });
        serverProjects = await API.Project.syncProjects({filter});
```

```
        await Promise.all(serverProjects.map(mergeRawProject));
    } catch (error) {
        console.log('Error while refreshing projects');
    }

    console.log('*** REFRESH PROJECTS FINISHED ***');
};

export const sync = async userId => {
    // cannot sync if previous sync wasn't ended or in dev mode
    if (!_.isEmpty(syncTasksIds) || __DEV__) {
        return;
    }

    await syncTask(userId);
};

export const syncTask = async userId => {
    console.log('--*** SYNC STARTED ***--');

    const taskId = addSyncTask();
    const {projects, projectIds} = await getCheckedProjects(userId);

    await cleanDatabase(userId, projectIds);
    await syncArchived();
    await syncChanged();
    await refreshProjects(userId, projects, projectIds);
    removeSyncTask(taskId);

    console.log('--*** SYNC FINISHED ***--');
};
```

ДОДАТОК Г

Код сценарію для контролю списків

```
import ManagedObject from './ManagedObject';
import ManagedObjectList from './ManagedObjectList';
import {action} from 'mobx';
import _ from 'lodash';
import Strings from '../../Strings';

type EntriesList = {
   list: ManagedObjectList,
   listNames: Array<string>,
};

type RegistryEntries = Map<string, Array<EntriesList>>;

type RegistryOptions = {
   listNames: Array<string>,
   // Where to put new entity? At the end of the list or at the top
   insertAtEnd?: boolean,
};

const defaultOptions: RegistryOptions = {
   listNames: ['fetchedObjects'],
   insertAtEnd: true,
};

class Registry {
   entries: RegistryEntries = new Map();

   add(list: ManagedObjectList, opts?: RegistryOptions = {}): void {
      const options = {...defaultOptions, ...opts};
```

```
    const modelName = list.model.modelName;
    let listArray = this.entries.get(modelName);
    if (!listArray) {
        listArray = [{list, options}];
        this.entries.set(modelName, listArray);
    } else {
        listArray.push({list, options});
    }
    console.log(Strings.REG_REGISTERED(modelName), {list, options, registry: this.entries});
}

remove(list: ManagedObjectList): void {
    const modelName = list.model.modelName;
    const listArray = this.entries.get(modelName);
    if (!_.isEmpty(listArray)) {
        _.remove(listArray, {list: {listId: list.listId}});
    }
    console.log(Strings.REG_REMOVED(modelName), {list, registry: this.entries});
}

@action.bound
onSave(obj: ManagedObject): void {
    const modelName = obj.context.modelName;
    const listArray = this.entries.get(modelName);
    if (!_.isEmpty(listArray)) {
        listArray.forEach(entriesList => this._saveObjToLists(entriesList, obj, modelName));
    } else {
        console.log(Strings.REG_NO_LISTS(modelName));
    }
}

@action.bound
onRemove(obj: ManagedObject): void {
    const modelName = obj.context.modelName;
    const listArray = this.entries.get(modelName);
```

```
    if (!_.isEmpty(listArray)) {
      listArray.forEach(entriesList => {
        const {list, options: {listNames}} = entriesList;
        listNames.forEach(listName => {
          _.remove(list[listName], listObj => this._compareObjectsKeys(listObj, obj));
          console.log(Strings.REG_OBJ_REMOVED(modelName, listName), {list, obj});
        });
      });
    } else {
      console.log(Strings.REG_NO_LISTS(modelName));
    }
  }

  _saveObjToLists = (entriesList: EntriesList, obj: ManagedObject, modelName: string) => {
    const {list, options: {listNames, insertAtEnd}} = entriesList;
    listNames.forEach(listName =>
      this._saveObjToList(list, listName, insertAtEnd, obj, modelName),
    );
  };

  _saveObjToList = (
    list: ManagedObjectList,
    listName: string,
    insertAtEnd: boolean,
    obj: ManagedObject,
    modelName: string,
  ) => {
    const idx = _.findIndex(list[listName], listObj => this._compareObjectsKeys(listObj, obj));
    if (list.filterObject(obj, listName)) {
      if (!~idx) {
        // insert new object
        list[listName] = insertAtEnd ? [...list[listName], obj] : [obj, ...list[listName]];
        console.log(Strings.REG_OBJ_ADDED(modelName, listName), {list, obj});
      } else {
        // update existing object
```

```
            list[listName][idx] = obj;
            console.log(Strings.REG_OBJ_UPDATED(idx, modelName, listName), {list, obj});
          }
        } else {
          // remove if exists but doesn't match filters already
          if (!!~idx) {
            _.remove(list[listName], listObj => this._compareObjectsKeys(listObj, obj));
            console.log(Strings.REG_OBJ_UPDATE_FAIL(modelName, listName), {list, obj});
          } else {
            console.log(Strings.REG_OBJ_ADD_FAIL(modelName, listName), {list, obj});
          }
        }
    };

    _compareObjectsKeys = (listObj: ManagedObject, obj: ManagedObject) =>
        (!!listObj.id && !!obj.id && listObj.id === obj.id) ||
        (!!listObj.localId && !!obj.localId && listObj.localId === obj.localId);
}

const registry: Registry = new Registry();
export default registry;
```