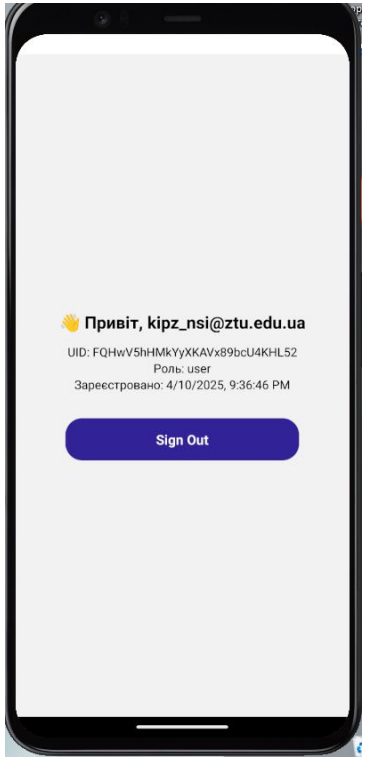
Розробка мобільних додатків

Лекція 10 - Зберігання та управління даними за допомогою Firestore

Розширення функціоналу додатку: збереження даних авторизованого користувача

Базова автентифікація вже реалізована - користувач має можливість входу до застосунку. Проте сам факт авторизації є лише початковим етапом. У більшості мобільних застосунків після входу користувачеві надається доступ до персоналізованого функціоналу, зокрема - можливості зберігати й обробляти власні дані.



Firestore

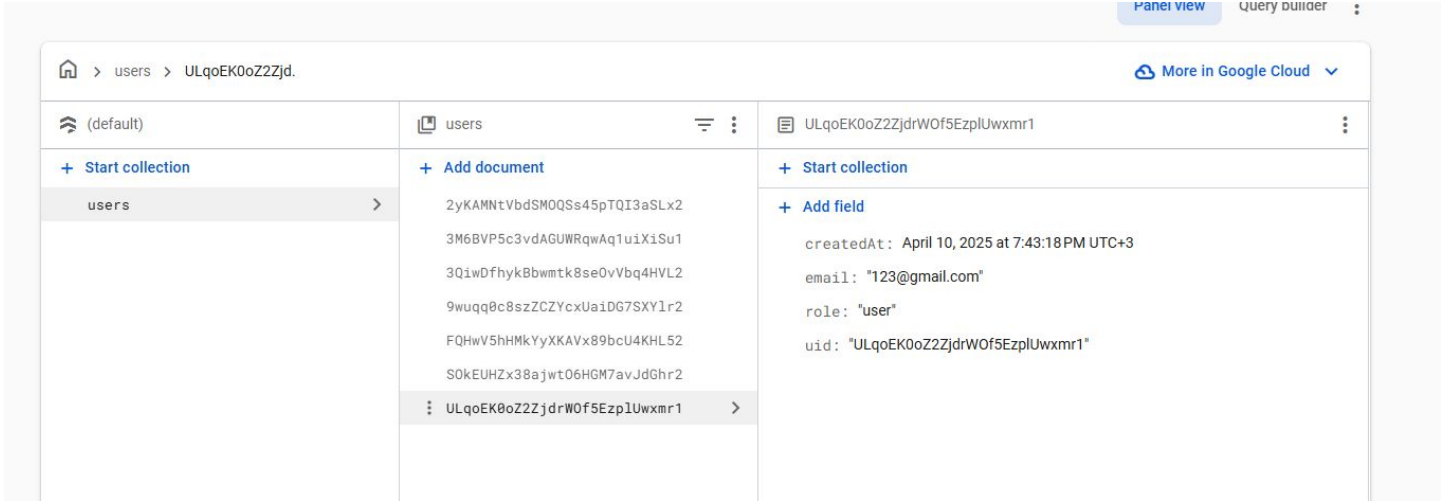
Firestore — це **NoSQL база даних у реальному часі**, що входить до складу Firebase (від Google).

Ключові особливості:

- Хмарна база даних (Cloud)
- Працює у **реальному часі**
- Підтримка **автоматичної синхронізації** між клієнтами
- Надійні **Security Rules** для обмеження доступу
- Масштабована, гнучка та зручна в роботі структура

Структура даних

Firestore дозволяє зберігати дані у вигляді документів і колекцій. Структурно це нагадує вкладені JSON-об’єкти. Кожен документ має унікальний ключ (ID) в межах колекції та може містити поля (рядки, числа, масиви, об’єкти тощо). Наприклад, можемо мати колекцію `"users"` і в ній документ з ID, рівним UID користувача, де зберігається профіль (ім’я, email, фото, тощо). Також можна робити колекції типу `"notes"` де кожен документ містить замітку користувача, і серед полів – `ownerId` або `uid` власника.



Основні обмеження документів у **Firebase Firestore**

Параметр	Максимальне значення	Пояснення
Розмір документа	**1 МіБ** (1,048,576 байтів)	Враховується назва полів, значення та метадані.
Кількість полів	**20,000**	Загальна кількість полів у документі (включаючи вкладені).
Глибина вкладення	**20 рівнів**	Максимальний рівень вкладеності для об'єктів (Map).
ID документа (ключ)	**1,500 байтів**	Не може містити `/`, а також символи `.` або `..`.
Частота запису	**1 запис / сек**	Рекомендований ліміт для одного і того самого документа.

Спеціальні типи даних, які підтримує **Firestore**:

- `string`, `number`, `boolean`
- `timestamp` (через `serverTimestamp()`)
- `array`
- `map` (об'єкт всередині документа)
- `geopoint` (координати для геолокації)
- `reference` (посилання на інший документ)

string, number, boolean

Базові примітиви:

- **string** - текст: `"Hello"`
- **number** - цілі або дробові числа: 42, `3.14`
- **boolean** - логічні значення: `true` або `false`

Ці типи використовуються в більшості звичайних полів.

timestamp (мітка часу)

Використовується для зберігання **дат і часу**. Найчастіше використовується з `serverTimestamp()`:

```
createdAt: serverTimestamp()
```

Це вказує Firestore встановити **час на сервері**, а не на клієнті — що забезпечує точність і синхронність.

array (масив)

Список значень. Може містити елементи будь-якого з підтримуваних типів:

```
tags: ['firebase', 'react', 'nosql']
```

Також підтримуються методи для роботи з масивами:

- `arrayUnion()` - додати елемент без дублювання
- `arrayRemove()` - видалити елемент

map (об’єкт)

Словник або вкладений об’єкт у документі:

```
profile: {
  name: 'Оля',
  age: 28
}
```

Це дозволяє зберігати структуровані дані всередині одного поля.

geopoint (геолокація)

Містить координати широти та довготи. Використовується, якщо потрібно зберігати **місцезнаходження**:

```
location: new GeoPoint(50.45, 30.52) // Київ
```

Може використовуватися для побудови геопошуку (разом із Firestore Extensions або Algolia).

reference (посилання на інший документ)

Це **вказівка на інший документ** у Firestore, подібно до зовнішнього ключа в SQL.

```
author: doc(db, 'users', 'user123')
```

Це дозволяє будувати **зв’язки між колекціями**, наприклад: пост - автор.

Обмеження безкоштовного плану **(Spark)**

Категорія	Безкоштовна квота	Коментар
Зберігання даних	**1 ГБ** (GiB)	Загальний обсяг усіх документів та індексів.
Читання (Reads)	**50,000** на день	Кожен документ у запиті рахується як 1 читання.
Запис (Writes)	**20,000** на день	Створення або оновлення документа.
Видалення (Deletes)	**20,000** на день	Видалення документа.
Пропускна здатність	**10 ГБ** на місяць	Тільки вихідний трафік (egress). Вхідний — безкоштовно.

Підключення **Firebase Firestore** до **Expo-додатку**

Якщо у проєкті вже інтегровано **Firebase Authentication**, підключення **Cloud Firestore** виконується аналогічним чином. У складі бібліотеки `firebase` (версія 9 і вище, модульний підхід) модуль **`firebase/firestore`**, який містить усі необхідні методи для взаємодії з базою даних.

Для початку роботи необхідно імпортувати функції:

```
import { getFirestore, doc, setDoc, getDoc, updateDoc, deleteDoc } from "firebase/firestore";
```

Ці методи забезпечують повноцінну підтримку CRUD-операцій у Firestore: створення, читання, оновлення та видалення документів.

getFirestore

`getFirestore` - це функція з Firebase SDK (версії 9 і вище), яка ініціалізує доступ до Cloud Firestore для вашого застосунку.

Призначення:

Вона створює екземпляр Firestore, прив'язаний до конкретного Firebase App, що дозволяє здійснювати запити до бази даних (читання, запис, оновлення, видалення документів).

```
import { initializeApp } from "firebase/app";
import { getFirestore } from "firebase/firestore";

const firebaseConfig = { /* ключі проекту */ };
const app = initializeApp(firebaseConfig);
const db = getFirestore(app);
```

Взаємодія **Expo**-додатка з **Cloud Firestore**

Пряме хмарне підключення: Додаток взаємодіє з базою даних напряму через інтернет. Усі запити обробляються на віддалених серверах Google (Cloud Infrastructure).

Автоматичне керування з'єднанням: Firebase SDK самостійно підтримує стабільний зв’язок, обробляє повторні спроби запитів та черговість передачі даних.

Ідентичність середовищ: Робота в режимі розробки (**Expo Go**) повністю відповідає поведінці фінальної збірки. Це дозволяє тестувати реальну швидкість та реактивність бази даних ще на етапі написання коду.

Відсутність посередників: Архітектура не потребує власного бекенд-сервера, що спрощує структуру проєкту та знижує затримки.

Створення, читання, оновлення, видалення (CRUD) записів користувача

У Firestore кожна з цих операцій має відповідну функцію:

- `setDoc` / `addDoc` – створення або перезапис документа.
- `getDoc` / `getDocs` – отримання даних (один документ або результат запиту).
- `updateDoc` – оновлення поля(ів) у документі.
- `deleteDoc` – видалення документа.

collection

Метод `collection()` використовується для отримання посилання на колекцію в базі даних Firestore. Це перший крок до читання, запису або прослуховування документів у цій колекції.

```
import { addDoc, collection, serverTimestamp } from "firebase/firestore";
```

Синтаксис:

```
collection(firestore, pathSegments...)
```

- `firestore` - екземпляр Firestore (наприклад, db)
- `pathSegments` - послідовність імен (наприклад, `"users"`, `"userId"`, `"todos"`)

addDoc

Метод `addDoc()` використовується для створення нового документа у вказаній колекції Firestore. Firestore автоматично згенерує унікальний `id` для документа.

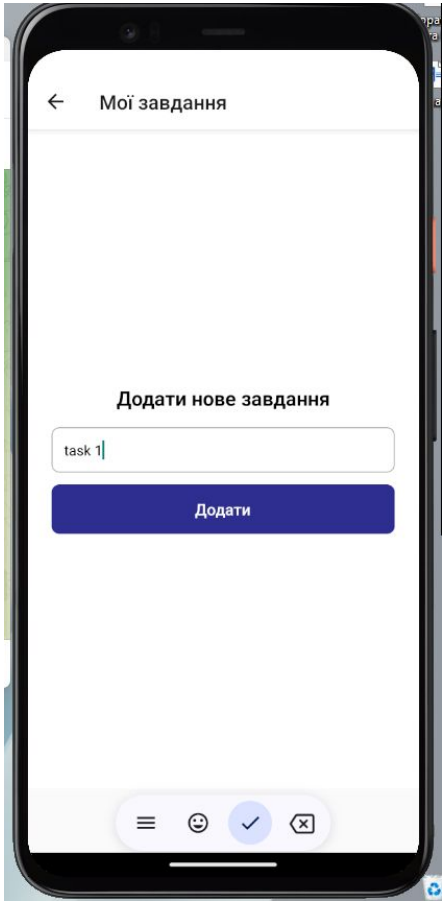
```
import { addDoc, collection, serverTimestamp } from "firebase/firestore";
```

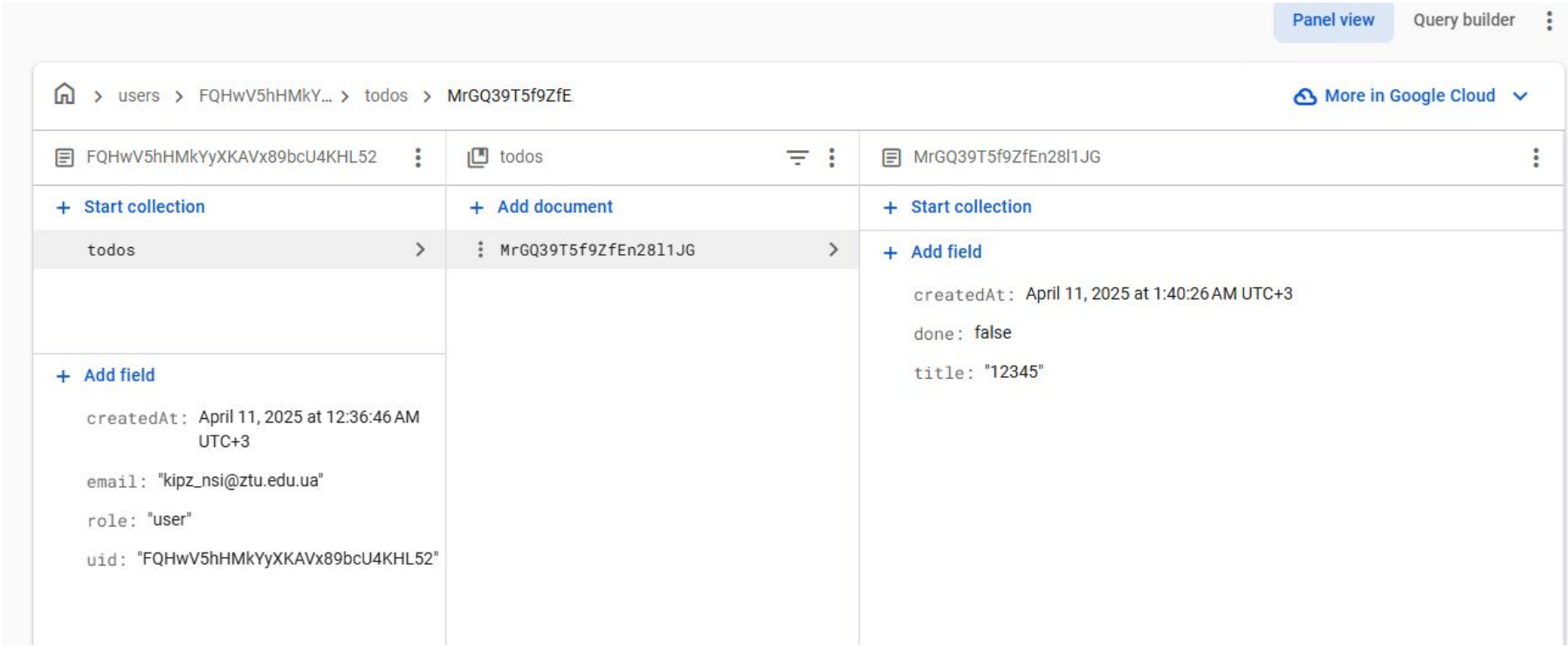
Синтаксис:

```
addDoc(collectionRef, data)
```

- `collectionRef` - посилання на колекцію, отримане за допомогою `collection()`
- `data` - об’єкт з полями та значеннями, які потрібно записати

```
import { addDoc, collection, serverTimestamp } from "firebase/firestore";
import { db } from "../firebase/config";
import { useAuth } from "../contexts/AuthContext";

const TodoScreen = () => {
    const { loggedInUser } = useAuth();
    const [todo, setTodo] = useState( initialState: "");

    const handleAddTodo = async () => {
        if (!todo.trim()) {
            Alert.alert( title: "Помилка",  message: "Поле не може бути порожнім");
            return;
        }

        try {
            const todosRef = collection(db,  path: "users", loggedInUser.uid, "todos")
            await addDoc(todosRef,  data: {
                title: todo,
                done: false,
                createdAt: serverTimestamp(),
            });
            setTodo( value: "");
            Alert.alert( title: "Успіх",  message: "Завдання додано!");
        } catch (error) {
            console.error("error:", error);
            Alert.alert( title: "Помилка",  message: "Не вдалося додати завдання");
        }
    };
```




Panel view
Query builder
users
FQHwV5hHMkY...
todos
MrGQ39T5f9ZfE
More in Google Cloud
FQHwV5hHMkYyXKAVx89bcU4KHL52
todos
MrGQ39T5f9ZfEn28l1JG
Start collection
Add document
Start collection
todos
MrGQ39T5f9ZfEn28l1JG
Add field
createdAt: April 11, 2025 at 1:40:26 AM UTC+3
done: false
title: "12345"
Add field
createdAt: April 11, 2025 at 12:36:46 AM UTC+3
email: "kipz_nsi@ztu.edu.ua"
role: "user"
uid: "FQHwV5hHMkYyXKAVx89bcU4KHL52"

setDoc

Метод `setDoc()` використовується для створення **або перезапису** документа у Firestore з **власним ідентифікатором (ID)**. На відміну від `addDoc()`, `setDoc()` не генерує ID автоматично — його потрібно вказати через `doc()`.

```
await setDoc(doc(db, path: "users", user.uid), data: {
    uid: user.uid,
    email: user.email,
    role: "user",
    createdAt: new Date()
});
```

Синтаксис:

```
setDoc(documentRef, data, options?)
```

- `documentRef` - посилання на документ, створене через `doc()`
- `data` - об’єкт з полями, які потрібно записати
- `options` *(необов'язково)* - `{ merge: true }` для часткового оновлення

З опцією `merge: true`:

```
const userRef = doc(db, "users", loggedInUser.uid);
await setDoc(userRef, {
  lastLogin: serverTimestamp()
}, { merge: true });
```

Це **оновить лише вказані поля**, не затираючи весь документ.

doc()

Метод `doc()` використовується для створення **посилання на документ** у Firestore. Він не читає чи записує дані — лише **вказує точне місце**, де розташований або буде створений документ.

Синтаксис:

```
doc(firestore, pathSegment1, pathSegment2, ...)
```

- `firestore` - екземпляр Firestore (зазвичай db)
- `pathSegment1`, `pathSegment2`, ... - чергування назв колекцій і ID документів

`collection()` - це шлях до "папки", де зберігаються документи.

`doc()` - це шлях до конкретного "файлу" (документа) всередині цієї папки.

getDocs

Метод `getDocs()` використовується для **отримання всіх документів** з вказаної колекції Firestore. Повертає **`QuerySnapshot`**, з якого можна витягнути масив документів за допомогою `docs`.

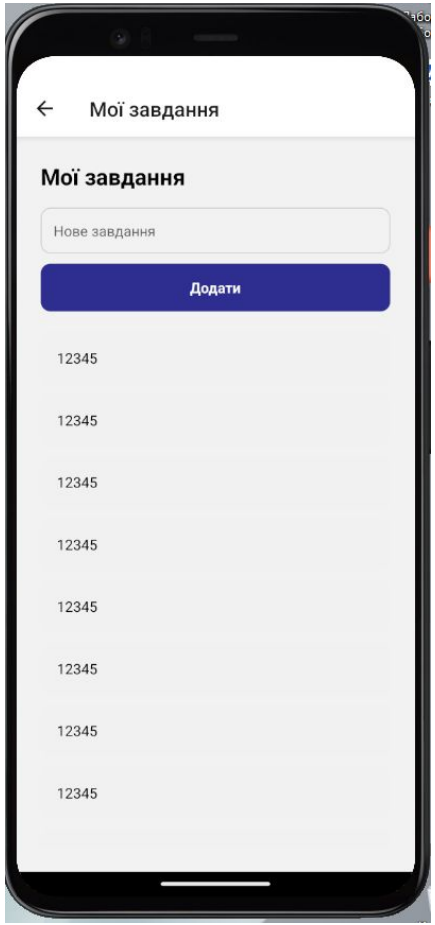
Синтаксис:

```
getDocs(collectionReference)
```

Пояснення:

`collectionReference` - посилання на колекцію, яку створюємо через `collection()`
(наприклад: `collection(db, "users", "abc123", "todos")`)

```
const fetchTodos = async () => {
    try {
        const todosRef = collection(db, path: "users", loggedInUser.uid, "todos");
        const snapshot = await getDocs(todosRef);
        const data = snapshot.docs.map(doc => ({
            id: doc.id,
            ...doc.data(),
        }));
        setTodos(data);
    } catch (error) {
        console.error("Error fetching todos:", error);
    }
};
```



Як перевірити, чи щось знайдено в **getDocs()?**

```
import { collection, getDocs } from "firebase/firestore";

const querySnapshot = await getDocs(collection(db, "todos"));

if (querySnapshot.empty) {
  console.log("Колекція порожня");
} else {
  querySnapshot.forEach((doc) => {
    console.log(doc.id, "=>", doc.data());
  });
}
```

getDoc

Метод **`getDoc()`** використовується для **отримання даних з одного документа** у Firestore. Він повертає **DocumentSnapshot**, з якого можна дістати поля, ID або перевірити, чи документ існує.

Синтаксис:

```
getDoc(documentRef)
```

Аргументи:

`documentRef` - посилання на документ, створене через метод `doc()`.

Як перевірити, чи існує документ у **getDoc()**

```
import { doc, getDoc } from "firebase/firestore";
import { db } from "../firebase/config";

const fetchUser = async (uid) => {
  const userRef = doc(db, "users", uid);
  const userSnap = await getDoc(userRef);

  if (userSnap.exists()) {
    console.log("Користувач:", userSnap.data());
  } else {
    console.log("Користувача не знайдено.");
  }
};
```

updateDoc

Метод `updateDoc()` використовується для **оновлення конкретних полів у документі Firestore**. На відміну від `setDoc()`, він **не перезаписує весь документ**, а лише змінює вказані поля.

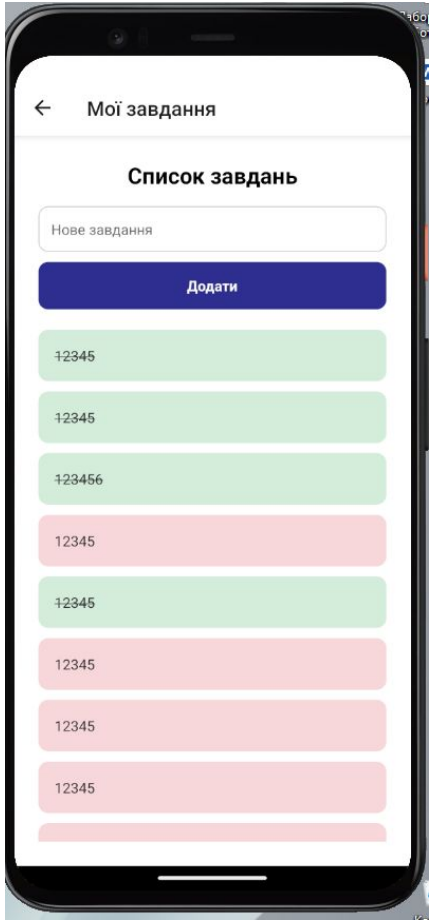
Важливо: Якщо документ не існує - `updateDoc()` поверне помилку.

Синтаксис:

```
import { doc, updateDoc } from 'firebase/firestore';

const docRef = doc(db, 'users', 'userId123');

await updateDoc(docRef, {
  age: 30,
  isActive: true,
});
```

Цей запит оновить **тільки поля age і `isActive`** в документі `users/userId123`.

```
const toggleTodoDone = async (todo) => {
    try {
        const todoRef = doc(db,  path: "users", loggedInUser.uid, "todos", todo.id);
        await updateDoc(todoRef,  data: {
            done: !todo.done,
        });

        fetchTodos();
    } catch (error) {
        console.error("Помилка при оновленні todo:", error);
    }
};
```



Оновлення вкладених полів

Використовується **шлях до поля через крапку**:

```
await updateDoc(docRef, {
  'profile.name': 'Олег',
  'profile.city': 'Львів',
});
```

Це оновить тільки поля `name` і `city` всередині вкладеного об'єкта `profile`.

Додавання в масив: **arrayUnion() / arrayRemove()**

Firestore дозволяє зберігати **масиви** як значення полів у документах. Якщо потрібно **додати елемент до масиву** або **видалити його**, і при цьому уникати дублювання або зайвого коду - можна використовувати **спеціальні функції Firestore**: `arrayUnion()` та `arrayRemove()`.

```
import { arrayUnion, arrayRemove } from 'firebase/firestore';


await updateDoc(docRef, {
  favoriteColors: arrayUnion('blue'),   // додає "blue", якщо ще немає
  tags: arrayRemove('oldTag')           // видаляє "oldTag"
});
```

Особливості

Не потрібно читати масив, модифікувати, а потім записувати заново - `arrayUnion()` і `arrayRemove()` це роблять **на сервері**, атомарно.

Ці операції **безпечні для конкурентного доступу** - якщо кілька користувачів змінюють масив одночасно, не буде конфліктів.

Обмеження

Масив не підтримує вкладених документів (наприклад, `arrayUnion({name: 'x'})`.

Максимальний розмір документа - 1 МБ, тому не варто додавати тисячі елементів.

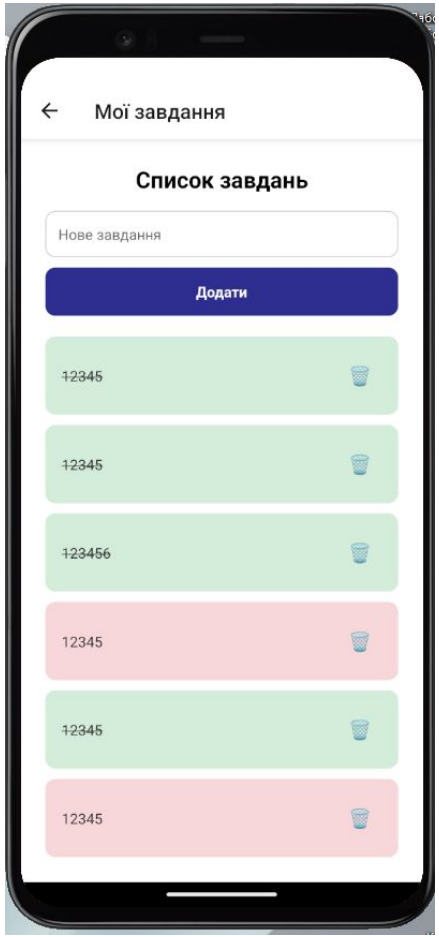
deleteDoc

Метод `deleteDoc()` використовується для **видалення документа** з бази даних Firestore.

Синтаксис:

```
import { deleteDoc, doc } from 'firebase/firestore';

const docRef = doc(db, 'users', 'userId123');

await deleteDoc(docRef);
```

```
const deleteTodo = async (todoId) => {
    try {
        const todoRef = doc(db,  path: "users", loggedInUser.uid, "todos", todoId);
        await deleteDoc(todoRef);
        fetchTodos();
    } catch (error) {
        console.error("Помилка при видаленні todo:", error);
    }
};
```



Важливо:

Документ видаляється повністю, але його підколекції **не видаляються автоматично**.

Наслідки:

- Підколекції залишаються - їх все ще можна отримати через запити
- Вони **не видимі у Firebase Console**, якщо батьківський документ видалено, але **існують**
- Це **створює сміття** у базі, яке займає місце і може вплинути на ліміти

Рішення:

Видаляти вкладені колекції вручну.

Позначити як "видалений"

Іноді замість фізичного видалення краще **логічно "приховати" документ**, додавши поле `isDeleted: true` або `deletedAt`.

Це дозволяє:

- Відновити документ пізніше
- Уникнути втрати історії

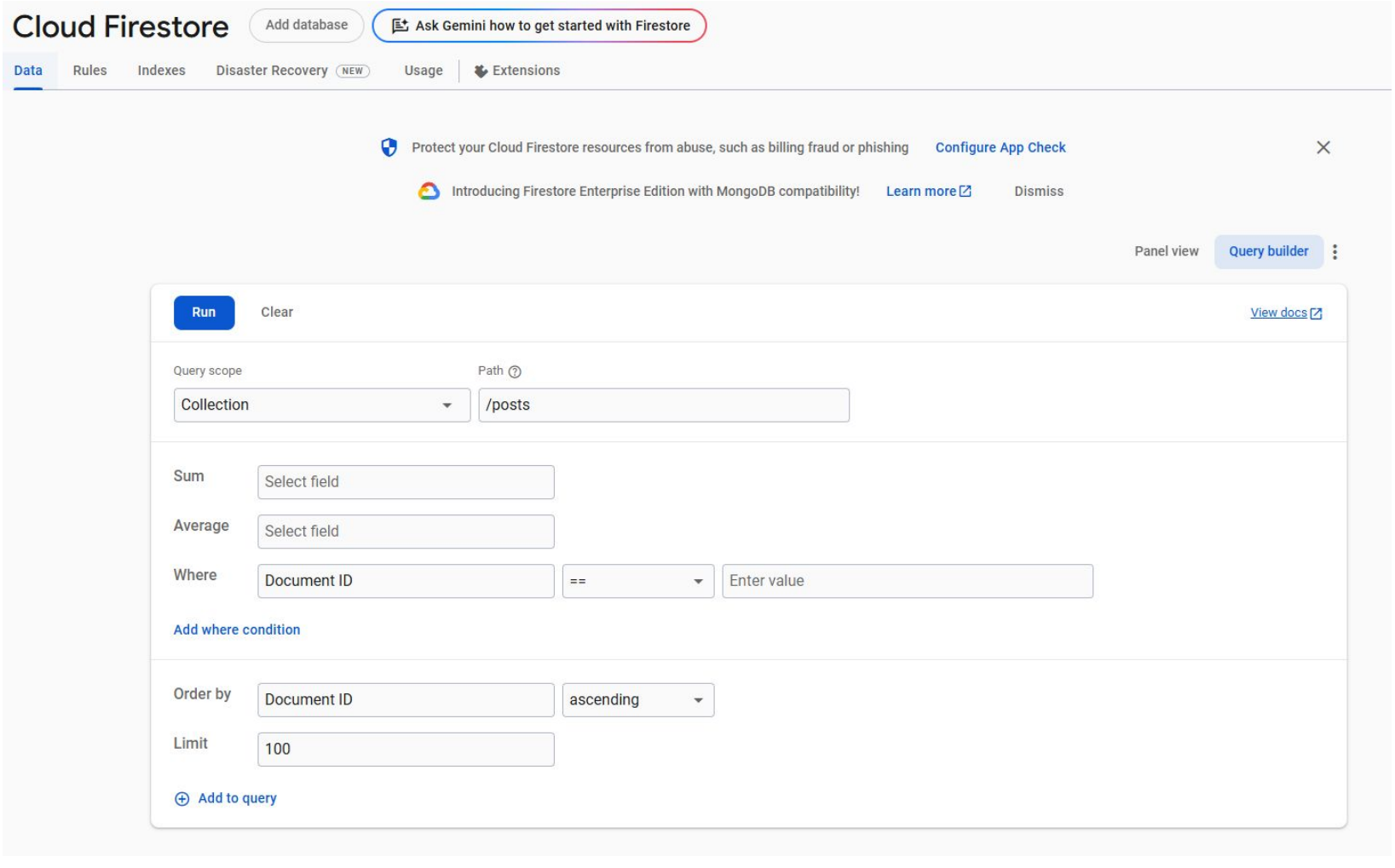
Фільтрація документів у Firestore

Однією з ключових можливостей Firestore є **фільтрація документів**, яка дозволяє отримувати тільки ті записи, які відповідають певним критеріям.

Фільтрація виконується за допомогою **запитів**, які можна комбінувати з різними умовами

Limitations

- Не підтримує **глибокі вкладені фільтри**
- Один документ - **до 1 МБ**
- Один запит - **до 10 умов (where, orderBy і т.д)**


Cloud Firestore
Add database
Ask Gemini how to get started with Firestore
Data
Rules
Indexes
Disaster Recovery
NEW
Usage
Extensions
Protect your Cloud Firestore resources from abuse, such as billing fraud or phishing
Configure App Check
Introducing Firestore Enterprise Edition with MongoDB compatibility!
Learn more
Dismiss
Panel view
Query builder
Run
Clear
View docs
Query scope
Collection
Path
/posts
Sum
Select field
Average
Select field
Where
Document ID
==
Enter value
Add where condition
Order by
Document ID
ascending
Limit
100
Add to query

query() + where()

Метод `query()` у Firebase Firestore дозволяє **комбінувати умови для вибірки документів** з колекції. Він приймає **базову колекцію** та **один або кілька операторів фільтрації / сортування / ліміту**.

`query()` = `collection()` + фільтри `where()`, `orderBy()`, `limit()`, `startAfter()` і тд.

Базовий синтаксис:

```
import { collection, query, where, getDocs } from 'firebase/firestore';

const citiesRef = collection(db, 'cities');

const q = query(citiesRef, where('country', '==', 'Ukraine'));

const snapshot = await getDocs(q);
snapshot.forEach(doc => {
  console.log(doc.id, '=>', doc.data());
});
```

Важливо:

`query()` **не виконує запит одразу** - лише **формує його**. Виконується через `getDocs()` або `onSnapshot()`.

Приклади з `query()`

Комбінування кількох умов:

```
const q = query(
  collection(db, 'cities'),
  where('country', '==', 'Ukraine'),
  where('population', '>', 1000000)
);
```

orderBy()

Метод `orderBy()` дозволяє **сортувати документи за значенням певного поля** у зростаючому (`asc`) або спадному (`desc`) порядку.

Він використовується всередині `query()` і **не виконує запит**, а лише формує його.

```
const fetchTodos = async () => {
    try {
        const todosRef = collection(db, path: "users", loggedInUser.uid, "todos");

        // Створюємо запит із сортуванням за датою
        const q = query(todosRef, orderBy( fieldPath: "createdAt", directionStr: "desc")); // або "asc" — за зростанням

        const snapshot = await getDocs(q);

        const data = snapshot.docs.map((doc) => ({
            id: doc.id,
            ...doc.data(),
        }));

        setTodos(data);
    } catch (error) {
        console.error("Помилка при завантаженні todos:", error);
    }
};
```

Ліміти та пагінація: **limit(), startAfter()**

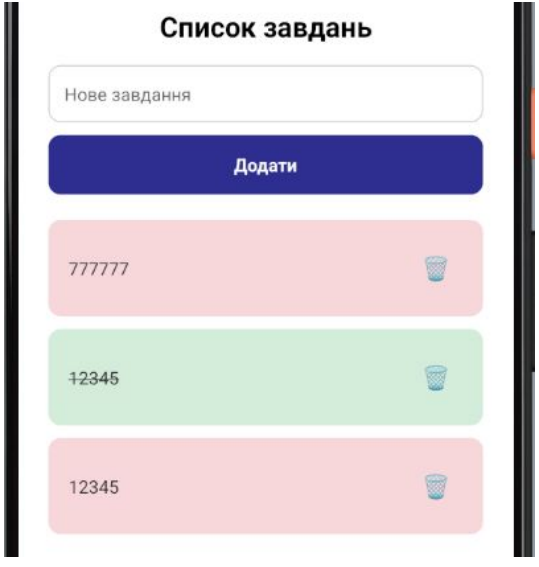
При роботі з великими обсягами даних у Firestore, часто виникає необхідність **обмежувати кількість документів**, які повертаються за один раз, і **завантажувати їх частинами**. Це особливо важливо для реалізації **пагінації**, автопідвантаження (infinite scroll) або просто для покращення продуктивності додатка.

limit()

Метод `limit()` використовується для **обмеження кількості документів**, які повертаються в результаті запиту. Це корисно для:

- Виведення перших *n* записів (наприклад, топ-10)
- Пагінації
- Оптимізації продуктивності запиту

```
const todosRef = collection(db, path: "users", loggedInUser.uid, "todos");

const q = query(
    todosRef,
    orderBy( fieldPath: "createdAt", directionStr: "desc"), // або "asc" для зростання
    limit( limit: 3) // тільки 3 документи
);

const snapshot = await getDocs(q);

const data = snapshot.docs.map((doc) => ({
    id: doc.id,
    ...doc.data(),
}));
```



Методи: **startAt(), startAfter(), endAt(), endBefore()**

Ці методи застосовуються в `query()` і використовуються разом із `orderBy()` для обмеження діапазону результатів.

`startAt(value)`

Починає запит **з вказаного значення** (включно з ним).

```
query(
  collection(db, 'products'),
  orderBy('price'),
  startAt(100)
)
```

`startAfter(value)`

Починає запит **після вказаного значення** (не включає його).

```
query(
  collection(db, 'products'),
  orderBy('price'),
  startAfter(100)
)
```

`endAt(value)`

Завершує запит **на вказаному значенні** (включає його).

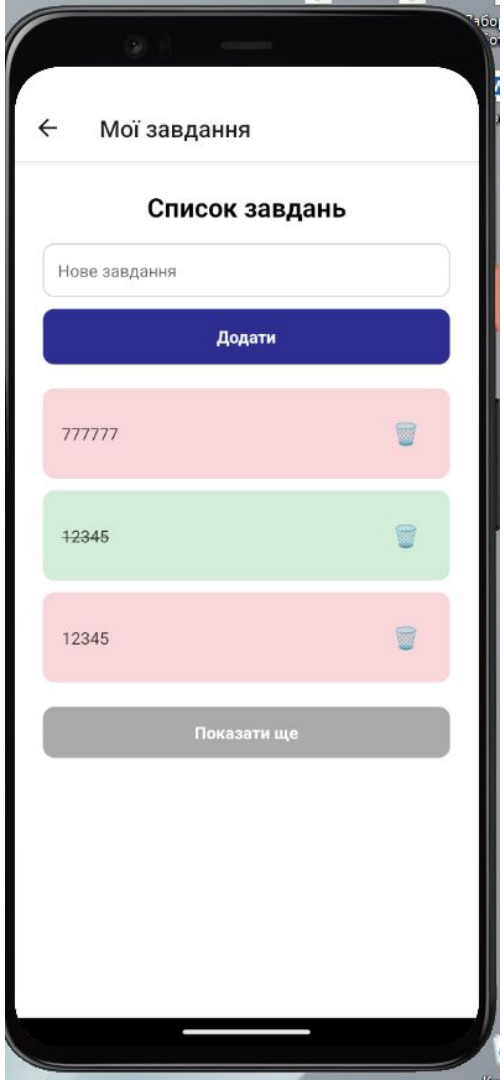
```
query(
  collection(db, 'products'),
  orderBy('price'),
  endAt(500)
)
```

`endBefore(value)`

Завершує запит **перед вказаним значенням** (не включає його).

```
query(
  collection(db, 'products'),
  orderBy('price'),
  endBefore(500)
)
```

```
const fetchMoreTodos = async () => {
    if (!lastVisible || !hasMore) return;

    try {
        const todosRef = collection(db,  path: "users", loggedInUser.uid, "todos");
        const q = query(
            todosRef,
            orderBy( fieldPath: "createdAt",  directionStr: "desc"),
            startAfter(lastVisible),
            limit(PAGE_SIZE)
        );

        const snapshot = await getDocs(q);
        const newTodos = snapshot.docs.map((doc) => ({ id: doc.id, ...doc.data() }));

        setTodos( value: (prev) => [...prev, ...newTodos]);
        setLastVisible(snapshot.docs[snapshot.docs.length - 1]);
        setHasMore( value: snapshot.size === PAGE_SIZE);
    } catch (error) {
        console.error("Помилка пагінації:", error);
    }
};
```



Додаткові можливості

Транзакції

Транзакції - це спосіб **гарантовано виконати запис, використовуючи найактуальнішу інформацію на сервері**. Транзакції **ніколи не виконують часткових записів** - усі операції запису виконуються **лише після успішного завершення транзакції**.

Коли використовувати транзакції?

Транзакції корисні, коли потрібно:

- оновити значення поля **на основі його поточного значення**;
- або на основі значення іншого поля.

Якщо потрібно просто зробити кілька записів без зчитування поточного стану документа - краще використовувати **пакетний запис (batch write)**.

Особливості роботи з транзакціями:

Особливості роботи з транзакціями:

- **Операції читання повинні передувати операціям запису.**
- Функція, яка виконує транзакцію, **може запускатися кілька разів**, якщо під час виконання інші користувачі змінюють ті ж документи.
- **Функції транзакції не повинні напряму змінювати стан додатка**.
- **Транзакції не працюють в офлайн-режимі** - вони вимагають з’єднання з сервером.

Синтаксис транзакції у Firestore

```
import { runTransaction, doc } from "firebase/firestore";

const userRef = doc(db, "users", "user123");

await runTransaction(db, async (transaction) => {
  const userDoc = await transaction.get(userRef);

  if (!userDoc.exists()) {
    throw "Документ не існує!";
  }

  const newBalance = userDoc.data().balance - 100;

  if (newBalance < 0) {
    throw "Недостатньо коштів!";
  }

  transaction.update(userRef, { balance: newBalance });
});
```

Як працює:

1. `runTransaction(...)` — запускає транзакцію
2. Всередині — зчитуємо документ через `transaction.get(...)`
3. Перевіряємо умови
4. Оновлюємо через `transaction.update(...)` або `set(...)`
5. Якщо щось не так — викликаємо `throw` → транзакція скасовується

Важливо:

- Всі операції мають бути в середині `runTransaction(...)`
- Якщо будь-що зміниться під час виконання - Firestore автоматично **повторить транзакцію**
- Максимум **500 зчитувань + записів** в одній транзакції

Метод	Опис
`transaction.get()`	Зчитування документа
`transaction.set()`	Створення або перезапис
`transaction.update()`	Оновлення полів
`transaction.delete()`	Видалення документа

Пакетний запис **(Batch write)**

Якщо під час виконання операцій **не потрібно зчитувати жодних документів**, можна виконати **декілька операцій запису в одному пакеті (batch)**. У пакеті можна комбінувати `set`, `update` або `delete`.

Усі операції в пакеті виконуються **атомарно** - або всі разом, або жодна.

Як це працює:

1. Створит новий екземпляр batch через метод `writeBatch()`.
2. Додати у нього потрібні операції (`set`, `update`, `delete`).
3. Виклик `commit()`, щоби застосувати пакет.

```
import { getDocs, collection, writeBatch, doc } from 'firebase/firestore';
import { db } from './firebase'; // підключення до Firestore

async function deleteAllUsers() {
  const usersRef = collection(db, 'users');
  const snapshot = await getDocs(usersRef);

  const batch = writeBatch(db); // створення нового batch

  snapshot.forEach((docSnap) => {
    const docRef = doc(db, 'users', docSnap.id);
    batch.delete(docRef); // додаємо видалення до batch
  });

  await batch.commit(); // виконуємо всі операції
  console.log('Усі користувачі видалені');
}
```

Зауваження:

- `writeBatch()` може обробити до **500 операцій за раз**. Якщо документів більше — потрібно **розбити на кілька batch-пакетів**.

- Такий спосіб **не видаляє вкладені підколекції**. Їх треба видаляти окремо.

collectionGroup

Метод `collectionGroup()` у Firebase Firestore дозволяє виконувати запити **по всіх підколекціях з однаковим іменем**, незалежно від того, де вони знаходяться у структурі бази.

```
users/{userId}/todos/{todoId}

admins/{adminId}/todos/{todoId}

projects/{projectId}/todos/{todoId}
```

Можна отримати всі `todos` з будь-якої вкладеності, використовуючи:

```
import { collectionGroup, getDocs, query, where } from "firebase/firestore";
import { db } from "./firebase/config";

const fetchAllTodos = async () => {
  const q = query(
    collectionGroup(db, "todos"),
    where("done", "==", false)
  );

  const snapshot = await getDocs(q);

  const todos = snapshot.docs.map((doc) => ({
    id: doc.id,
    ...doc.data(),
  }));

  console.log("Всі незавершені todos:", todos);
};
```

onSnapshot()

Метод `onSnapshot()` використовується для **реєстрації "слухача" (listener)**, який буде автоматично викликатись **кожного разу, коли документ або колекція змінюється на сервері**.

Це дозволяє:

- будувати інтерфейс, що **реагує на зміни в режимі реального часу**;
- уникати ручного оновлення сторінки;
- синхронізувати кількох користувачів (наприклад, у чаті або лайв-таблиці).

Синтаксис: прослуховування документу

```
import { doc, onSnapshot } from 'firebase/firestore';

const docRef = doc(db, 'users', 'user123');

const unsubscribe = onSnapshot(docRef, (doc) => {
  if (doc.exists()) {
    console.log("Дані:", doc.data());
  } else {
    console.log("Документ не існує");
  }
});
```

Синтаксис: прослуховування колекції

```
import { collection, onSnapshot } from 'firebase/firestore';

const unsub = onSnapshot(collection(db, 'posts'), (snapshot) => {
  snapshot.forEach(doc => {
    console.log(doc.id, '=>', doc.data());
  });
});
```

Щоб зупинити слухання:

```
unsub(); // Відписка від оновлень
```

Це важливо робити, коли компоненти **відмонтовуються**, щоб уникати **витоку пам’яті**.

Обробка типів змін

Можна відслідковувати тип змін: `added`, `modified`, `removed`.

```
onSnapshot(collection(db, 'messages'), (snapshot) => {
  snapshot.docChanges().forEach(change => {
    if (change.type === "added") {
      console.log("Додано:", change.doc.data());
    }
    if (change.type === "modified") {
      console.log("Змінено:", change.doc.data());
    }
    if (change.type === "removed") {
      console.log("Видалено:", change.doc.data());
    }
  });
});
```

Reference (посилання на інший документ)

Тип **Reference** у Firestore дозволяє **зберігати посилання на інший документ** в якості значення поля. Це дає змогу:

- будувати **зв’язки між колекціями** (наприклад, `Post` → `User`);
- швидко діставати пов’язану інформацію;
- працювати зі структурою "one-to-one", "many-to-one" тощо.

Як створити посилання:

```
import { doc, setDoc } from 'firebase/firestore';
import { db } from './firebase';

// Створимо посилання на користувача
const userRef = doc(db, 'users', 'user123');

// Додаємо його у документ публікації
await setDoc(doc(db, 'posts', 'post456'), {
  title: 'Мій перший пост',
  author: userRef  // тут — reference
});
```

У результаті поле `author` у документі `posts/post456` буде посиланням на `users/user123`.

Як зчитати й використовувати **Reference**

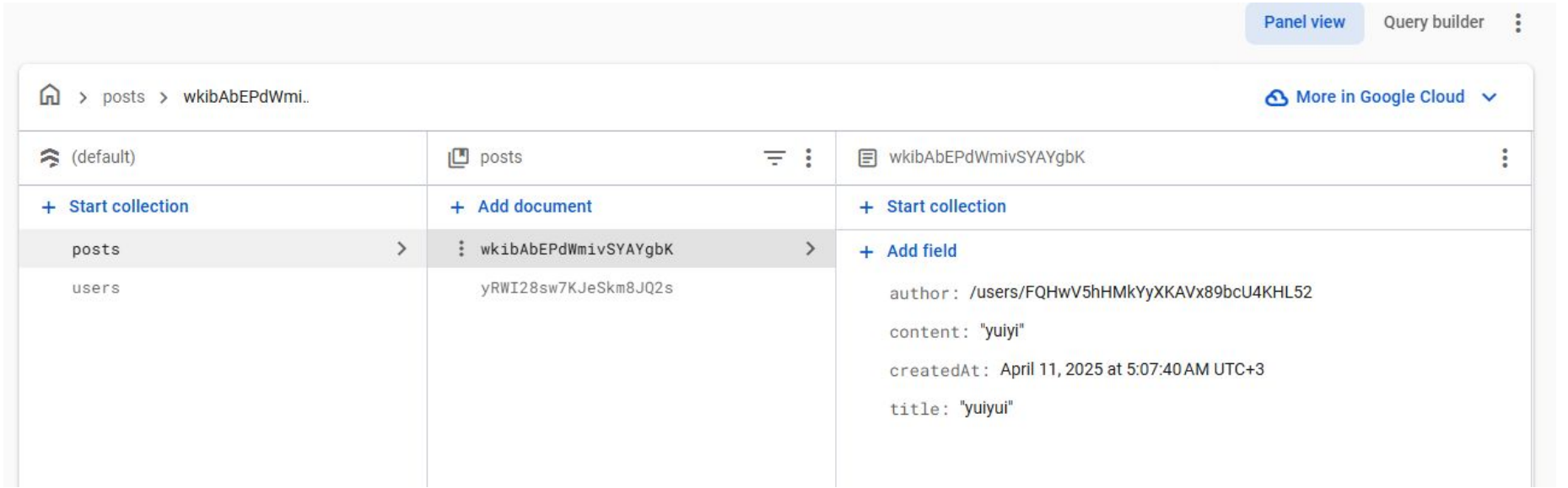
```
import { getDoc } from 'firebase/firestore';

const postDoc = await getDoc(doc(db, 'posts', 'post456'));
const userRef = postDoc.data().author; // Reference

// Тепер отримуємо сам документ користувача
const userDoc = await getDoc(userRef);
console.log(userDoc.data().name); // Наприклад, "Олег"
```

Особливості:

- Reference - це лише **посилання**, Firestore **не робить автоматичний "join"**. Потрібно вручну діставати пов’язаний документ.

- Немає "каскадного видалення" - якщо видалити користувача, поле `author` все ще міститиме посилання, яке веде в нікуди.

- Для UI це може означати необхідність **додаткових запитів**.


Panel view
Query builder
posts
wkibAbEPdWmi..
More in Google Cloud
(default)
posts
wkibAbEPdWmivSYAYgbK
Start collection
Add document
Start collection
posts
wkibAbEPdWmivSYAYgbK
Add field
users
yRWI28sw7KJeSkm8JQ2s
author: /users/FQHwV5hHMkYyXKAVx89bcU4KHL52
content: "yuiyi"
createdAt: April 11, 2025 at 5:07:40 AM UTC+3
title: "yuiyui"

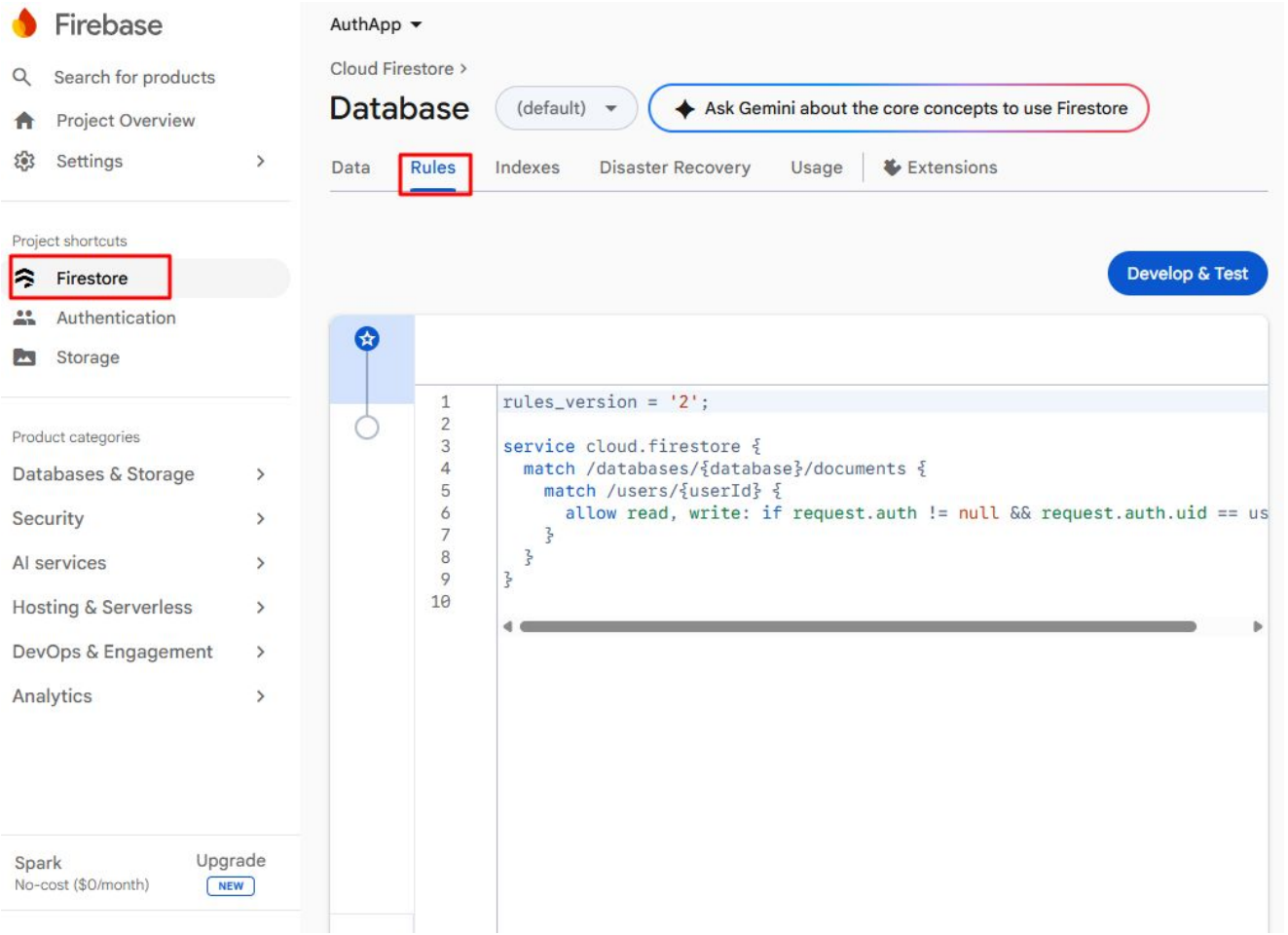
Firestore Security Rules

Security Rules у Firestore - це **механізм контролю доступу до даних**. Вони визначають, **хто, коли та до яких документів або колекцій має доступ**, і які дії дозволено: `read`, `write`, `update`, `delete`.

Правила виконуються **на рівні сервера** і діють **незалежно від коду в додатку**.

Як це працює?

- Перед кожною операцією Firestore перевіряє **відповідні правила доступу**.
- Якщо правила повертають `true` → доступ дозволено.
- Якщо `false` → запит відхиляється з помилкою `PERMISSION_DENIED`.


Firebase
Search for products
Project Overview
Settings
Project shortcuts
Firestore
Authentication
Storage
Product categories
Databases & Storage
Security
AI services
Hosting & Serverless
DevOps & Engagement
Analytics
Spark
No-cost ($0/month)
Upgrade
NEW
AuthApp
Cloud Firestore
Database
(default)
Ask Gemini about the core concepts to use Firestore
Data
Rules
Indexes
Disaster Recovery
Usage
Extensions
Develop & Test
rules_version = '2';
service cloud.firestore {
match /databases/{database}/documents {
match /users/{userId} {
allow read, write: if request.auth != null && request.auth.uid == us
}
}
}

Типи операцій

Тип	Опис
`read`	Включає `get` і `list`
`write`	Включає `create`, `update`, `delete`
`get`	Отримання одного документа
`list`	Отримання списку документів
`create`	Створення нового документа
`update`	Оновлення існуючого документа
`delete`	Видалення документа

Валідація доступу: кожен користувач бачить лише свої дані

З точки зору клієнтського коду, ми завжди підставляємо `user.uid` у запити. Тобто, додаток **не має функціоналу** отримати чужий профіль. Але це лише клієнт – теоретично, зловмисник міг би відкрити консоль та зробити запит до Firestore з іншим id, якщо ваші правила безпеки не захищені. **Безпека має забезпечуватися на стороні сервера теж.** У випадку Firebase, роль сервера виконують **Security Rules**.

Тому, потрібно додати правило , яке каже: в колекції "users" дозволено читати й писати тільки свій документ. Правила пишуться в спеціальному файлі (Firestore Security Rules) і деплояться через Firebase CLI.

Просте правило для цього сценарію:

```
// Firestore rules
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {
    match /users/{userId} {
      allow read, write: if request.auth != null && request.auth.uid == userId;
    }
  }
}
```

Це правило:

- Застосовується до кожного документа в колекції `users` (бо `match /users/{userId}`).
- Змінна `{userId}` буде дорівнювати ID документа, який запитується.
- `request.auth` – об’єкт, що містить дані автентифікації запиту (для запитів з Firebase Auth).
- `request.auth.uid` – UID поточного залогіненого користувача.
- Отже, `allow read, write` (дозволити читання і запис) виконується **тільки якщо** користувач залогінений (`request.auth != null`) **і** UID користувача збігається з ID документа (`request.auth.uid == userId`).

Застосування правил: Після написання, це правило треба розгорнути. До того моменту, поки правило не діє, Firestore за замовчуванням дозволяє **все, якщо в режимі тестування** або **нічого, якщо режим locked**. Можна перевірити правила у Firebase консолі у розділі Firestore Rules. Якщо вони як вище – наш код клієнта повинен успішно працювати, коли юзер залогінений. Спроба ж прочитати чужий документ (наприклад, підставивши інший `userId`) призведе до помилки "Missing or insufficient permissions" на клієнті.