Тема 13. Firebase. Работа в реальном времени, структура данных. Пагинация и оптимизация.

хекслет колледж

Цель занятия:

Научить студентов создавать полноценные приложения с использованием реального времени в Firebase, правильно структурировать данные и оптимизировать базовые запросы для типовых проектов.

Учебные вопросы:

1. Работа в реальном времени
2: Проектирование структуры данных
3. Пагинация и работа со списками

1. Работа в реальном времени

Что такое "реальное время" и зачем оно нужно?
Реальное время - это когда данные в вашем приложении обновляются мгновенно, без необходимости перезагружать страницу или нажимать кнопку "Обновить".

Примеры:

- Чат - вы видите новые сообщения сразу
- Онлайн-счетчик - числа меняются на глазах
- Уведомления - появляются сразу при событии
- Совместная работа - несколько человек видят изменения одновременно

Чем отличается от обычных запросов? Как работает? Когда использовать?

- get() - Один раз получает данные. Загрузка страницы, форма.
- onSnapshot() - Постоянно слушает изменения. Чат, уведомления, счетчики.

Подписка на изменения: onSnapshot()

```
// Базовая схема работы:
// 1. Получаем ссылку на коллекцию
const messagesRef = db.collection('messages');

// 2. Подписываемся на изменения
const unsubscribe = messagesRef.onSnapshot((snapshot) =>
{
  // 3. Этот код выполняется КАЖДЫЙ РАЗ при изменении
данных
  console.log('Получены новые данные!');
```

```
// 4. Обрабатываем изменения
  snapshot.docChanges().forEach((change) => {
    if (change.type === 'added') {
      console.log('Добавлен новый документ:', change.doc.data());
    }
    if (change.type === 'modified') {
      console.log('Изменен документ:', change.doc.data());
    }
    if (change.type === 'removed') {
      console.log('Удален документ:', change.doc.id);
    }
  });
});

// 5. Когда нужно отписаться (например, при выходе со страницы)
// unsubscribe();
```

Основной синтаксис

```
const unsubscribe = messagesRef.onSnapshot(
  (snapshot) => { /* функция при УСПЕХЕ */ },
  (error) => { /* функция при ОШИБКЕ (необязательно) */ }
);
```

onSnapshot() принимает две функции (вторая - необязательная):

- Функция-обработчик успеха - вызывается каждый раз при изменении данных
- Функция-обработчик ошибки - вызывается при проблемах (сеть, права доступа)

Что такое snapshot?

snapshot (с английского - "снимок", "моментальный снимок") - это объект, который содержит:

- Актуальные данные из Firestore на данный момент
- Информацию об изменениях - что добавилось, изменилось, удалилось
- Метаданные - когда данные были обновлены в последний раз

Что ВОЗВРАЩАЕТ onSnapshot()?

onSnapshot() возвращает функцию unsubscribe:

```
const unsubscribe = messagesRef.onSnapshot((snapshot) => {
  // ... ваш код ...
});

// Позже, когда нужно остановить подписку:
unsubscribe(); // Останавливаем слушание
```

Что находится ВНУТРИ snapshot?

Основные свойства и методы snapshot:

```
const unsubscribe = messagesRef.onSnapshot((snapshot) => {
  // 1. Проверяем, есть ли вообще документы
  console.log('Пустая ли коллекция?', snapshot.empty);

  // 2. Сколько документов в коллекции
  console.log('Количество документов:', snapshot.size);

  // 3. ПРОЙДЕМСЯ ПО ВСЕМ ДОКУМЕНТАМ (базовый способ)
  snapshot.forEach((doc) => {
    console.log('Документ ID:', doc.id); // "abc123"
    console.log('Данные документа:', doc.data()); // { text: "Привет", ... }
  });

 // 4. ПОЛУЧИМ МАССИВ ДОКУМЕНТОВ
  const allDocs = snapshot.docs; // Массив документов
  allDocs.forEach((doc) => {
    console.log(doc.id, doc.data());
  });
```

```
// 5. САМОЕ ВАЖНОЕ: КАКИЕ ИМЕННО ИЗМЕНЕНИЯ ПРОИЗОШЛИ?
 snapshot.docChanges().forEach((change) => {
   console.log('Тип изменения:', change.type); // "added", "modified", "removed"
   console.log('ID документа:', change.doc.id);
   console.log('Данные:', change.doc.data());

   if (change.type === 'added') {
     console.log('ДОБАВЛЕН новый документ');
   }
   if (change.type === 'modified') {
     console.log('ИЗМЕНЕН существующий документ');
   }
   if (change.type === 'removed') {
     console.log('УДАЛЕН документ');
   }
 });
```

```
// 6. Метаданные - были ли изменения на сервере
  console.log('Источник данных:', snapshot.metadata.fromCache
    ? 'из кэша (оффлайн)'
    : 'с сервера (онлайн)');
});
```

Основные свойства snapshot

Свойство	Тип	Что содержит	Пример
empty	boolean	Пустая ли коллекция	false - есть документы
size	number	Количество документов	5 - 5 документов
docs	Array	Массив всех документов	[doc1, doc2, doc3]
metadata	Object	Информация о данных	{ fromCache: false }
query	Object	Запрос, который создал snapshot	Ссылка на исходный запрос

Основные методы snapshot

Метод	Что возвращает	Пример
forEach(callback)	void - проходит по всем документам	snapshot.forEach(doc => {})
docChanges()	Array изменений	[{type: 'added', doc: ...}]

Основные свойства error

Свойство	Тип	Что содержит	Пример
code	string	Код ошибки	'permission-denied'
message	string	Текст ошибки	'Missing or insufficient permissions.'
name	string	Название ошибки	'FirebaseError'

docChanges() - основные свойства

docChanges() возвращает массив изменений - какие документы добавились, изменились или удалились с момента последнего snapshot.

Структура каждого изменения (change object)
Каждый элемент массива docChanges() имеет структуру:

```
{
  type: 'added',      // или 'modified', или 'removed'
  doc: DocumentSnapshot,  // сам документ (для 'removed' - последняя версия)
  oldIndex: -1,       // старый индекс в результатах
  newIndex: 0         // новый индекс в результатах
}
```

Подробно о каждом свойстве:

1. type - тип изменения:
'added' - документ появился в результатах запроса
'modified' - документ в результатах изменился
'removed' - документ исчез из результатов запроса

2. doc - сам документ.
Важно: Для type: 'removed' в change.doc будет последняя версия документа перед удалением.

3. oldIndex и newIndex - индексы в массиве
Что значат индексы:
oldIndex - где был документ ДО изменения
newIndex - где находится документ ПОСЛЕ изменения
-1 означает "не в результатах"

2. Проектирование структуры данных

Основные принципы Firestore:

- Firestore — это не SQL база!
- Нет таблиц, есть коллекции и документы
- Нет JOIN между коллекциями
- Дублирование данных — это нормально!

Три типа структур данных

Тип 1: Плоская структура (самая простая)

```
// Все в одной коллекции
users/{userId} {
  name: "Иван",
  email: "ivan@mail.ru",
  role: "user"
}

products/{productId} {
  name: "Ноутбук",
  price: 1000,
  category: "electronics"
}

orders/{orderId} {
  userId: "userId123",
  productId: "productId456",
  date: "2024-01-15"
}
```

Плюсы: Просто, понятно, быстро начинать

Минусы: Сложные запросы, много связей

Тип 2: Вложенные документы

```
// Документ внутри документа
users/{userId} {
  name: "Иван",
  profile: {          // Вложенный объект
    avatar: "url.jpg",
    bio: "Люблю программирование",
    settings: {
      theme: "dark",
      notifications: true
    }
  }
}
```

Плюсы: Все связанные данные в одном месте

Минусы: Ограничение 1MB на документ, сложно обновлять частично

Тип 3: Коллекции внутри документов (субколлекции)

```
// Коллекция внутри документа
chats/{chatId} {
  name: "Общий чат",
  createdAt: "2024-01-15"
}
   ├─── messages/{messageId} {  // Субколлекция!
      text: "Привет всем!",
      userId: "user123",
      timestamp: "2024-01-15T10:30:00"
  }
   ├─── members/{userId} {// Субколлекция!
      role: "admin",
      joinedAt: "2024-01-15"
  }
```

Плюсы: Логическая группировка, легкое управление правами
Минусы: Сложнее запрашивать данные из разных субколлекций

Золотые правила проектирования

1. ДУБЛИРУЙТЕ данные для быстрого доступа

```
// ❌ ПЛОХО: Только ссылка
posts/{postId} {
  title: "Мой пост",
  authorId: "userId123"  // Придется делать второй запрос
}

// ✅ ХОРОШО: Дублируем часто используемые данные
posts/{postId} {
  title: "Мой пост",
  authorId: "userId123",
  authorName: "Иван Петров",     // ДУБЛИРУЕМ!
  authorAvatar: "avatar.jpg"     // ДУБЛИРУЕМ!
}

// Теперь можем показывать пост БЕЗ дополнительных запросов!
```

2. Храните то, что ЧАСТО ЗАПРАШИВАЕТЕ

```
// Пример для приложения блога
posts/{postId} {
  title: "Как научиться программировать",
  content: "Очень длинный текст...",
  authorId: "user123",

  // Дублируем для быстрых запросов:
  authorName: "Мария",
  preview: "В этой статье я расскажу...",  // Для списков статей
  readTime: 5,                             // Для отображения
  tags: ["программирование", "обучение"],  // Для фильтрации
  commentsCount: 15,                       // Не нужно считать каждый раз!
  likesCount: 42,
  lastActivity: "2024-01-15T10:30:00"      // Для сортировки
}
```

3. Избегайте глубокой вложенности

```
// ❌ СЛИШКОМ ГЛУБОКО
users/{userId}/posts/{postId}/comments/{commentId}/replies/{replyId}

// ✅ ЛУЧШЕ
posts/{postId} {
  // данные поста
}
comments/{commentId} {
  postId: "postId",
  // данные комментария
}
replies/{replyId} {
  commentId: "commentId",
  // данные ответа
}
```

Пример: Простая структура данных интернет-магазина

Всего 5 коллекций (минимальный набор):

1. Коллекция products - Товары

```
products/{productId} {
  name: "iPhone 15",                  // Название
  price: 999,                         // Цена
  description: "Новый iPhone...",     // Описание
  category: "phones",                 // Категория
  image: "iphone15.jpg",              // Картинка
  inStock: true,                      // В наличии
  rating: 4.8,                        // Средний рейтинг
  reviewCount: 125,                   // Количество отзывов
  createdAt: "2024-01-15"             // Дата добавления
}
```

2. Коллекция users - Пользователи

```
users/{userId} {
  email: "ivan@mail.ru",        // Email
  name: "Иван Петров",          // Имя
  phone: "+79161234567",        // Телефон
  address: "ул. Ленина, 1",     // Адрес
  createdAt: "2024-01-01",      // Дата регистрации
  role: "customer"              // Роль
}
```

3. Коллекция orders - Заказы

```
orders/{orderId} {
  userId: "user123",                 // ID покупателя
  userName: "Иван Петров", // Имя покупателя (дублируем)
  userEmail: "ivan@mail.ru",  // Email покупателя (дублируем)

  items: [                           // Товары в заказе
    {
      productId: "prod1",
      name: "iPhone 15",             // Название (дублируем)
      price: 999,                    // Цена на момент заказа
      quantity: 1,
      image: "iphone15.jpg"          // Картинка (дублируем)
    },
    {
       ....
    }
  ],
  total: 1049,                       // Общая сумма
  status: "processing",              // Статус: cart/processing/shipped/delivered
  address: "ул. Ленина, 1",    // Адрес доставки
  createdAt: "2024-01-15T10:30:00" // Дата заказа
}
```

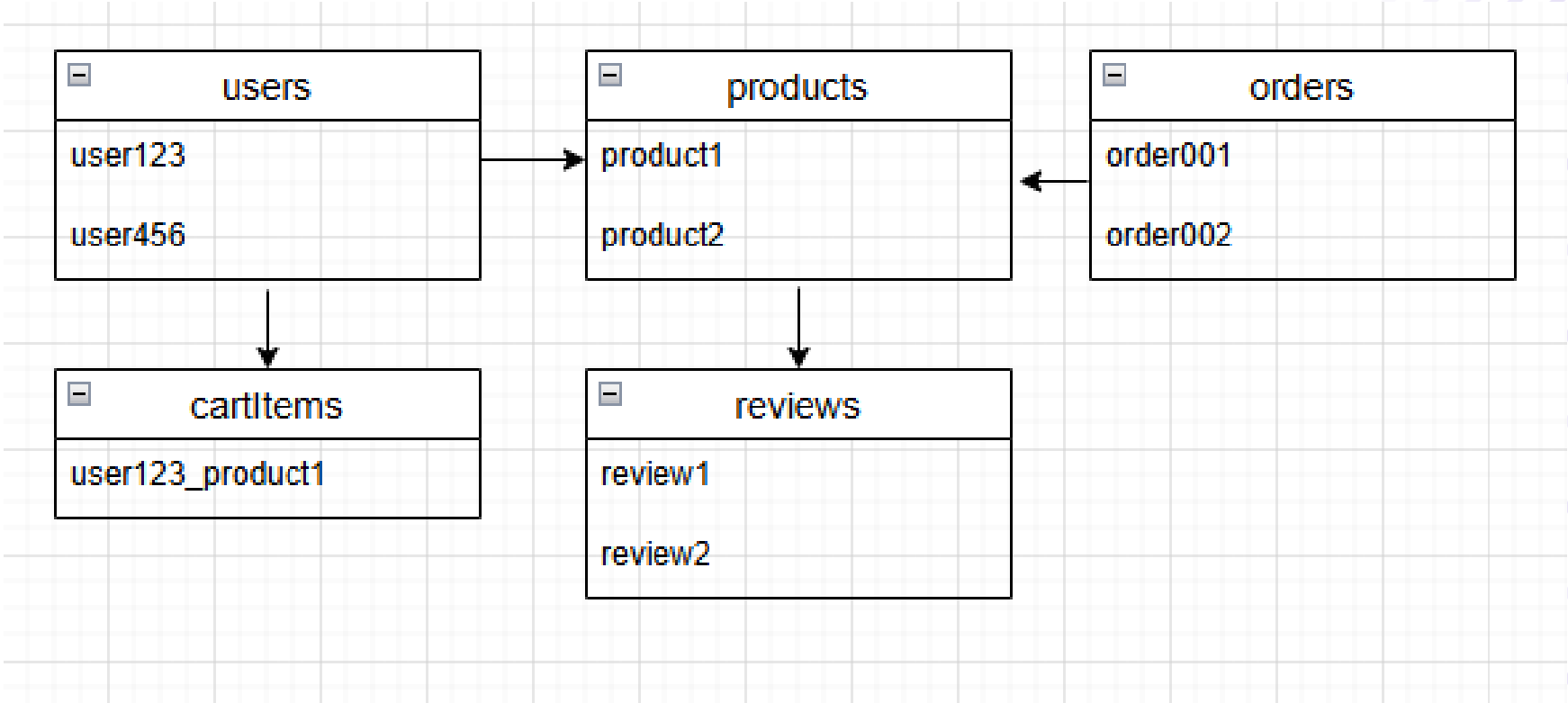
4. Коллекция reviews - Отзывы

```
reviews/{reviewId} {
  productId: "prod1",              // ID товара
  productName: "iPhone 15",   // Название товара (дублируем)

  userId: "user456",                // ID автора
  userName: "Анна",                 // Имя автора (дублируем)

  rating: 5,                              // Оценка 1-5
  text: "Отличный телефон!", // Текст отзыва
  createdAt: "2024-01-15"       // Дата отзыва
}
```

5. Коллекция cartItems - Корзина пользователя

```
cartItems/{userId_productId} {   // Составной ID: "user123_prod1"
  userId: "user123",                    // ID пользователя
  productId: "prod1",                   // ID товара
  productName: "iPhone 15",             // Название (дублируем)
  price: 999,                           // Цена (дублируем)
  image: "iphone15.jpg",                // Картинка (дублируем)
  quantity: 1,                          // Количество
  addedAt: "2024-01-15T10:30:00"   // Дата добавления
}
```

Связи между коллекциями:



Примеры запросов (покупатель):

1. Получить все товары категории "phones"

```
db.collection('products')
  .where('category', '==', 'phones')
  .where('inStock', '==', true)
  .orderBy('price')
  .get()
```

2. Получить корзину пользователя

```
db.collection('cartItems')
  .where('userId', '==', currentUserId)
  .get()
```

3. Получить заказы пользователя

```
db.collection('orders')
  .where('userId', '==', currentUserId)
  .orderBy('createdAt', 'desc')
  .get()
```

4. Получить отзывы о товаре

```
db.collection('reviews')
  .where('productId', '==', productId)
  .orderBy('createdAt', 'desc')
  .get()
```

Примеры запросов (админ):

```
// 1. Добавить новый товар
await db.collection('products').add({
  name: "Новый товар",
  price: 1000,
  description: "Описание товара",
  category: "phones",
  image: "image.jpg",
  inStock: true,
  rating: 0,
  reviewCount: 0,
  createdAt: new Date().toISOString()
})
```

2. Все заказы (с пагинацией)

```
db.collection('orders')
  .orderBy('createdAt', 'desc')
  .limit(50)
  .get()
```

3. Заказы по статусу

```
// Новые заказы (в обработке)
db.collection('orders')
  .where('status', '==', 'processing')
  .orderBy('createdAt', 'desc')
  .get()

// Доставленные заказы
db.collection('orders')
  .where('status', '==', 'delivered')
  .orderBy('createdAt', 'desc')
  .get()
```

4. Все пользователи

```
db.collection('users')
  .where('role', '==', 'customer')
  .get()
```

5. Изменить статус заказа

```
db.collection('orders').doc('orderId').update({
  status: 'delivered',
  updatedAt: new Date().toISOString()
})
```

Чек-лист проектирования

Перед созданием структуры ответьте на вопросы:

- Какие данные чаще всего запрашиваются вместе? → Дублируйте их
- Как пользователи будут искать данные? → Добавьте поля для поиска
- Нужна ли сортировка? → Добавьте поля для сортировки
- Часто ли меняются данные? → Вынесите в отдельные документы
- Нужны ли счетчики? → Храните предварительно вычисленные значения
- Какой максимальный размер данных? → Ограничьте массивы и вложенности

3. Пагинация и работа со списками

Что такое пагинация?
Пагинация — это когда мы показываем данные не все сразу, а по частям (страницами).

Пример без пагинации:
// ❌ ПЛОХО: Загрузит ВСЕ товары сразу

```
db.collection('products').get()

// Если товаров 1000 → медленно, много трафика
```

Пример с пагинацией:

```
// ✅ ХОРОШО: Загрузит только 10 товаров
db.collection('products')
  .orderBy('name')
  .limit(10)
  .get()
// Потом еще 10, потом еще...
```

Три ключевых метода

1. .limit(n) — сколько показать

// Показать 10 товаров
db.collection('products').limit(10).get()

// Показать 5 заказов
db.collection('orders').limit(5).get()

2. .orderBy('поле') — как отсортировать

```
// orderBy Обязательно для пагинации!
db.collection('products')
  .orderBy('price')      // по цене
  .limit(10)
  .get()

// Можно сортировать по-разному:
.orderBy('createdAt', 'desc')   // новые сначала
.orderBy('rating', 'desc')      // лучшие сначала
.orderBy('name')                // по алфавиту
```

2. .orderBy('поле') — как отсортировать

```
// orderBy Обязательно для пагинации!
db.collection('products')
  .orderBy('price')      // по цене
  .limit(10)
  .get()
// Можно сортировать по-разному:
.orderBy('createdAt', 'desc')   // новые сначала
.orderBy('rating', 'desc')      // лучшие сначала
.orderBy('name')                // по алфавиту
```

3. .startAfter(документ) — с чего продолжить

```
// Показать следующие 10 после последнего
db.collection('products')
  .orderBy('name')
  .startAfter(lastProduct)  // последний  загр. товар
  .limit(10)
  .get()
```

Важно:

- Всегда используйте orderBy() — без сортировки пагинация не работает
- Всегда используйте limit() — иначе загрузите всё
- .startAfter() для продолжения — показывает что дальше
- Сохраняйте последний документ — чтобы знать, с чего продолжать

Простой алгоритм пагинации

Шаг 1: Первая загрузка

```
const firstPage = await db.collection('products')
  .orderBy('name')
  .limit(10)
  .get();
// Показываем эти 10 товаров
```

Шаг 2: Сохраняем последний

```
const lastProduct = firstPage.docs[firstPage.docs.length - 1];
```

Шаг 3: Следующая страница

```
const nextPage = await db.collection('products')
  .orderBy('name')
  .startAfter(lastProduct)  // начинаем после последнего
  .limit(10)
  .get();
// Показываем еще 10 товаров
```

4. Оптимизация и отладка Firebase

Три главных правила оптимизации:
Правило 1: Используйте select() - берите только нужные поля

```
// ❌ ПЛОХО: Берем ВСЕ поля (10+ полей)
db.collection('products').get()
// Загружаем: name, price, description, category, image, rating, reviewCount...

// ✅ ХОРОШО: Берем только 3 нужных поля
db.collection('products')
  .select('name', 'price', 'image')  // Только эти поля
  .limit(20)
  .get()
// Загружаем только: name, price, image
```

Правило 2: Дублируйте данные для быстрого доступа

// Вместо того чтобы делать 3 запроса:
// 1. Получить отзыв
// 2. Получить имя автора (по userId)
// 3. Получить название товара (по productId)

// Мы сразу сохраняем все в отзыве:

```
reviews/{reviewId} {
  text: "Отличный товар!",
  userId: "user123",
  userName: "Иван",          // ← ДУБЛИРУЕМ имя
  productId: "prod1",
  productName: "iPhone 15",   // ← ДУБЛИРУЕМ название
  rating: 5
}
```

// Теперь нужен только 1 запрос!

Правило 3: Считайте заранее - храните счетчики

```
// ❌ ПЛОХО: Считать каждый раз
const reviewsSnapshot = await db.collection('reviews')
  .where('productId', '==', 'prod1')
  .get();
const reviewCount = reviewsSnapshot.size; // Дорогая операция!

// ✅ ХОРОШО: Хранить счетчик в товаре
products/{productId} {
  name: "iPhone 15",
  reviewCount: 42,     // ← Уже посчитано
  rating: 4.8          // ← Уже вычислен
}

// Обновление счетчика:
await db.collection('products').doc('prod1').update({
  reviewCount: firebase.firestore.FieldValue.increment(1)
});
```

Как отлаживать в браузере

1. Смотрите запросы в Network вкладке

- Откройте DevTools (F12)
- Перейдите на вкладку Network
- Фильтруйте по firestore.googleapis.com
- Смотрите какие запросы идут

Что смотреть:

- Сколько запросов делается?
- Большие ли данные приходят?
- Есть ли ошибки (красный цвет)?

2. Логируйте операции в консоли

```
console.log('Начало загрузки товаров');
const snapshot = await db.collection('products').get();
console.log(`Загружено товаров: ${snapshot.size}`);
console.log('Загрузка завершена');

// Проверяем конкретные данные
snapshot.forEach(doc => {
  console.log(`Товар: ${doc.data().name}, Цена: ${doc.data().price}`);
});
```

Выводы

Реальное время:
- onSnapshot() для данных, которые меняются в реальном времени
- Всегда сохранять unsubscribe функцию для остановки подписки
- docChanges() показывает только изменения (added/modified/removed)

Структура данных:
- Firestore ≠ SQL: дублирование данных нормально
- Дублировать часто запрашиваемые вместе данные (имя в заказе)
- Вложенные объекты для редко меняющихся данных (настройки)
- Подколлекции для списков, которые растут (заказы, сообщения)
- Хранить предвычисленные счетчики (reviewCount, rating)

Пагинация:
- Обязательно: orderBy() + limit()
- Для продолжения: .startAfter(lastDoc)
- Без orderBy() пагинация не работает

Оптимизация:
- select() — брать только нужные поля
- Один запрос на 10 документов лучше 10 запросов по одному

Контрольные вопросы:

- В чем разница между get() и onSnapshot()?
- Зачем нужна функция unsubscribe() и когда ее вызывать?
- Какие три типа изменений показывает docChanges()?
- Почему в NoSQL базах можно дублировать данные?
- Когда использовать вложенный объект, а когда подколлекцию?
- Какие три метода Firestore обязательны для пагинации?
- Как метод select() помогает оптимизировать запросы?
- Почему в заказе нужно дублировать имя пользователя и название товара?

хекслет колледж

@HEXLY.KZ