Making TanStack Query feel more like the Cloud Firestore client-side SDK

Daniel Giljam

Degree Thesis

Information Technology

2023

Degree Thesis

(Author) Daniel, Giljam

Making TanStack Query feel more like the Cloud Firestore client-side SDK.

Arcada University of Applied Sciences: Information Technology, 2023.

Identification number:

8801

Commissioned by:

N/A

Abstract:

The goal of this thesis is to come up with a solution which can take care of query cache updating in a generic and robust way in web applications which use TanStack Query. Query cache is an abstraction in TanStack Query, which is a JavaScript library for handling asynchronous read and write operations and representing their state in a user interface. After examining various open-source technologies, it is the conclusion of this thesis that Orbit.js is the solution to the problem. Orbit.js is a composable data framework for ambitious web applications. An accompanying result of this thesis is the publicly available GitHub project, https://github.com/DanielGiljam/tanstack-query-with-orbitjs, which contains a library for using TanStack Query together with Orbit.js as well as two example applications, one which demonstrates the problems with query cache updating when using TanStack Query and another one which demonstrates the solution of using TanStack Query together with Orbit.js with the help of the library for integrating the two. A web page edition of the thesis is also available at https://danielgiljam.com/degree-thesis. It provides the best reading experience for this thesis.

Keywords:

TanStack Query
React Query
Orbit.js
Ambitious web applications
ORM (Object-Relational Mapping)
React.js
TypeScript
JavaScript
Plugins
Integrations
Extensions
NPM packages
Cloud Firestore
Firebase
Real-time data

Lärdomsprov

(Författare) Daniel, Giljam

Making TanStack Query feel more the Cloud Firestore client-side SDK.

Yrkeshögskolan Arcada: Informationsteknik, 2023.

Identifikationsnummer:

8801

Uppdragsgivare:

N/A

Sammandrag:

Målet med det här arbetet är att ta fram en lösning som kan hantera query cache uppdatering på ett generellt och robust sätt i webbapplikationer som använder TanStack Query. Query cache är en abstraktion som förekommer i TanStack Query som är ett JavaScript-bibliotek för att hantera asynkrona läs- och skrivoperationer och återge deras tillstånd i ett användargränssnitt. Efter att ha undersökt olika open-source teknologier är slutsatsen i detta arbete att Orbit.js är lösningen. Orbit.js är ett komponerbart dataramverk för ambitiösa webbapplikationer. Ett ackompanjerande resultat till detta arbete är GitHub-projektet, https://github.com/DanielGiljam/tanstack-query-with-orbitjs, som innehåller ett bibliotek för att använda TanStack Query tillsammans med Orbit.js samt två exempelapplikationer, en som demonstrerar problemen med query cache uppdatering då man använder TanStack Query och en annan som demonstrerar lösningen att använda TanStack Query tillsammans med Orbit.js med hjälp av biblioteket för att integrera de två. En webbsideutgåva av arbetet finns också tillgänglig på https://danielgiljam.com/degree-thesis. Den erbjuder bästa läsupplevelsen för arbetet.

Nyckelord:

TanStack Query
React Query
Orbit.js
Ambitious web applications
ORM (Object-Relational Mapping)
React.js
TypeScript
JavaScript
Plugins
Integrations
Extensions
NPM packages
Cloud Firestore
Firebase
Real-time data

Contents

1 Introduction 4

1.1 Prerequisites 4

1.2 Delimitations 5

1.2.1 Delimitation #1 5

1.2.2 Delimitation #2 5

2 Background 6

2.1 TanStack Query 6

2.2 The Cloud Firestore client-side SDK 7

2.2.1 Cloud Firestore 7

2.2.2 “Real-time” databases 8

3 Problem 9

3.1 What are query cache updaters? 10

3.2 Why do we need query cache updaters? 10

3.2.1 How TanStack Query achieves almost always-up-to-date auto-managed queries with its default configuration 11

4 Research questions 12

5 Goal 13

6 Hypothesis 14

7 Methods 15

7.1 Research 15

7.1.1 Finding related work 15

7.1.2 Comparing related work 15

7.1.3 Assessing related work 17

7.2 Implementation 18

7.2.1 Developing an example application 18

7.2.2 Solving the problem 19

7.2.3 Packaging the solution as a library 20

8 Related work 21

8.1 Overview 21

8.1.1 CushionDB 21

8.1.2 Dexie.js 21

8.1.3 Orbit.js 21

8.1.4 SQLite as a WebAssembly module 22

8.2 Comparison 22

8.2.1 Modularity 23

8.2.2 Developer experience 24

8.2.3 UI framework integrations 24

8.2.4 ORM capabilities 26

8.2.5 Consideration of asynchronicity and concurrency 26

8.2.6 Offline support 27

8.3 Analysis 27

9 Results....29
9.1 A concrete example of the problem 29
9.1.1 Initial code 29
9.1.2 Updating the UI when new chat messages arrive 33
9.1.3 Handling the case where the chat rooms doesn't exist in the `["chat-rooms"]` query's cached data 36
9.1.4 Dealing with concurrency 38
9.1.5 Summing it up 40
9.2 The solution 40
9.2.1 `@tanstack-query-with-orbitjs/core` 40
9.2.2 `@tanstack-query-with-orbitjs/react` 42
10 Conclusion....43
10.1 Further development 43
11 References 45
12 Appendices 50
12.1 Appendix #1: Summary in Swedish.... 50

1 Introduction

This thesis is about coming up with a way to make TanStack Query (*TanStack Query*, n.d.) feel more like the Cloud Firestore (*Cloud Firestore | Store and Sync App Data at Global Scale*, n.d.) client-side SDK.

The Cloud Firestore client-side SDK provides an unparalleled interface, in terms of elegance, for dealing with "real-time" data in the client-side code. Combining the positive traits of the Cloud Firestore client-side SDK with TanStack Query could hypothetically solve the most prominent problem I face when developing with TanStack Query.

The most prominent problem I face when developing with TanStack Query is that if I opt-out of revalidation, then I have to write huge amounts of boilerplate-heavy fragile query cache updating code which is tightly coupled with both the data model and the UI of the app. I identify that there's a missing piece here — a need for a solution which can take care of the query cache updating in a generic and robust way. My hypothesis is that the recipe for that solution can be derived from the Cloud Firestore client-side SDK.

NOTE: Do not waste expensive ink and the nature's resources on printing this paper. The digital version provides a much better reading experience thanks to hyperlinks and cross-references, which this paper makes extensive use of. In fact, you might want to check out the web page edition of this thesis: https://danielgiljam.com/degree-thesis. It provides the ultimate reading experience with responsive screen width (mobile support) and dark theme support.

The source code for the web page edition of this thesis, as well as the examples and libraries produced as results of this thesis can be found on GitHub: https://github.com/DanielGiljam/tanstack-query-with-orbitjs.

1.1 Prerequisites

I assume that the reader is familiar with current web application development practices, patterns, frameworks, libraries, and tooling, and that the reader has some experience working with TanStack Query or similar libraries, such as SWR by Vercel (*React Hooks for Data Fetching – SWR*, 2023).

1.2 Delimitations

1.2.1 Delimitation #1

In this thesis, the underlying assumption is that TanStack Query is part of our frontend stack, and that is not going to change.

1.2.2 Delimitation #2

In this thesis, the underlying assumption is that we cannot use Cloud Firestore as part of our stack.

2 Background

2.1 TanStack Query

TanStack Query (formerly known as React Query) by Tanner Linsley (*Tannerlinsley.Com*, n.d.) has 33,984 stars on GitHub as of 11th of April 2023 (*TanStack/Query on GitHub*, 2019/2023). In the State of JS Survey 2022, 28% of respondents said that had used it as a data fetching library (*State of JavaScript 2022: Other Tools*, n.d.). It is part of the increasingly popular open-source full stack solution T3 Stack. TanStack Query is trusted by numerous big companies in production (*TanStack Query*, n.d.).

I've used TanStack Query for a while now in several projects that I work on, and I think it's a great library which elegantly solves many problems and challenges commonly faced when developing web applications. I hope it becomes even more popular and widely adopted. I think it's a positive technology with a positive impact on the web development community and industry.

It provides the following value:

- Solution for managing the state of async read and writes and accurately reflecting it in the UI.
    - Very complex wheel to try and re-invent yourself.
- Separation of concerns.
    - UI components can be written in a truly declarative fashion thanks to its great API design.
- Good developer experience.
    - Well-designed API which provides both high-level access and low-level access.
    - Good documentation (*Overview | TanStack Query Docs*, n.d.).
    - Fully-fledged type definitions (TypeScript (*TypeScript: JavaScript With Syntax For Types.*, n.d.) support).
    - Graphical "devtools" (*Devtools | TanStack Query Docs*, n.d.), which is great for debugging while developing.

But while it solves many problems and challenges commonly faced when developing web applications, it does not solve all of them. Or by solving some problems and challenges, it takes you to the next level and "unlocks" some new problems and challenges.
In this thesis, I will address what I found to be the most prominent problem which TanStack Query does not solve.

2.2 The Cloud Firestore client-side SDK

The Cloud Firestore client-side SDK is — as the name suggests — a SDK for interacting with Cloud Firestore from a client application. Cloud Firestore is a "real-time" database. The downside of Cloud Firestore is that it's part of Google's (*Google - About Google, Our Culture & Company News*, n.d.) proprietary and closed-source backend-as-a-service platform Firebase (*Firebase*, n.d.), which disqualifies its use in a lot of projects, due to reasons such as costs, the risk of vendor lock-in, to name a few.

The upside is that it provides an unparalleled interface, in terms of elegance, for dealing with "real-time" data in the client-side code. Combining the positive traits of the Cloud Firestore client-side SDK with TanStack Query could hypothetically solve the most prominent problem I face when developing with TanStack Query.

2.2.1 Cloud Firestore

Cloud Firestore is a NoSQL document database which is hosted in the cloud and part of Firebase, an app development platform provided by Google. Its counterparts in AWS (*Cloud Computing Services - Amazon Web Services (AWS)*, n.d.) land and Azure (*Cloud Computing Services | Microsoft Azure*, n.d.) land are commonly viewed as being DynamoDB (*Fast NoSQL Key-Value Database – Amazon DynamoDB – Amazon Web Services*, n.d.) and CosmosDB (*Azure Cosmos DB*, n.d.), respectively, although there are significant differences between the databases in terms of their design and functionality.

What in my opinion mostly distinguishes Cloud Firestore from similar solutions is its "real-time" functionality — a set of capabilities inherited from its predecessor / sibling product Firebase Realtime Database (*Firebase Realtime Database | Store and Sync Data in Real Time*, n.d.), for which the "real-time" functionality was the main selling point back in the day.

2.2.2 “Real-time” databases

A “real-time” database — such as Firebase Realtime Database — is a database which provides a mechanism for clients to subscribe to be notified about changes to the data in the database. In the case of Firebase Realtime Database as well as Cloud Firestore, the implementation of this mechanism is by design hidden to the consumer of the service(s). Google’s backend and the client-side SDK work in tandem to provide a high-level way of consuming "real-time" data.

In Cloud Firestore’s client-side SDK, this is done by using the `onSnapshot` API (*Get Realtime Updates with Cloud Firestore*, n.d.). It lets you listen to when the result of a database query changes.

3 Problem

The most prominent problem I face when developing with TanStack Query is that if I opt-out of revalidation, then I have to write huge amounts of boilerplate-heavy fragile query cache updating code which is tightly coupled with both the data model and the UI of the app. I identify that there's a missing piece here — a need for a solution which can take care of the query cache updating in a generic and robust way.

Query cache updaters — when implemented as suggested in TanStack Query's documentation — are bad because:

- They are difficult to write.
  - Writing a bug-free query cache updater requires the developer to take into consideration all permutations of what the state of the query could be when the query cache updater is being run.
    - The developer must have a profound understanding of how TanStack Query works to know what cases to consider and how to consider them.
    - This results in a lot of boilerplate code.
  - The complexity of writing a bug-free (robust) real-life-use-case query cache updater isn't clearly conveyed in TanStack Query's documentation.
    - Perhaps a very intelligent and experienced developer could pick it up from between the lines in the documentation, but for most developers, it's something that they must discover for themselves through trial and error.
- They are difficult to maintain for the same reasons that they are difficult to write.
- They are tightly coupled with the data model ***AND*** with the UI in the application, so they need to be updated whenever the data model or the UI changes — which is another way of saying they need to be updated very frequently.

This prevents TanStack Query from scaling well in larger, more complex applications.

See chapter *9.1 Results: A concrete example of the problem* for a concrete example of the problem.

3.1 What are query cache updaters?

"Query cache updaters" is the term I use to refer to functions that update the `QueryCache` (*QueryCache | TanStack Query Docs*, n.d.). Query cache updaters are needed when you want to update the result of a query without having to re-fetch the query. See Updates from Mutation Responses (*Updates from Mutation Responses | TanStack Query Docs*, n.d.) for more information. Note that a query cache updater doesn't always have to be used in conjunction with a mutation. A query cache updater can be used anywhere where you have access to the `QueryClient` (*QueryClient | TanStack Query Docs*, n.d.) object. For example, in a chat application, you could use a query cache updater in conjunction with a `WebSocket` (*WebSocket - Web APIs | MDN*, 2023) message handler to update the query which holds the list of chat messages whenever a new chat message arrives.

3.2 Why do we need query cache updaters?

It may seem counter-intuitive that we need query cache updaters, especially as TanStack Query describes itself on its home page as giving us "always-up-to-date auto-managed queries". This is half-true, in that out-of-the-box, TanStack Query is configured in a way where it will constantly revalidate our queries (*Important Defaults | TanStack Query Docs*, n.d.), effectively resulting in them being almost "always-up-to-date".

You will most likely opt-out of revalidation — in other words, tweak your configuration so that queries are not revalidated automatically — in order to avoid that the app is making too many unnecessary network requests. Instead, you will manually update the query cache when something has changed. For manually updating the query cache when something has changed, you need to write query cache updaters.

3.2.1 How TanStack Query achieves almost always-up-to-date auto-managed queries with its default configuration

It achieves almost always-up-to-date auto-managed queries through frequent revalidation of the queries. By default (*Important Defaults | TanStack Query Docs*, n.d.), TanStack Query will revalidate a query when:

- An observer is added.
- The window is refocused.
- The network is reconnected.

This results in queries being revalidated very often. For bandwidth and performance reasons, it may not be desirable to be hitting the backend with network requests (or whatever data source with whatever way they're being queried in the query functions) that often.

Thus, in real-word use-cases, the configuration is usually tweaked so that queries are not invalidated automatically and instead they're manually invalidated when the developer knows something has changed (e.g., upon receiving a socket message or when a mutation has been successfully executed).

However, even when controlling when queries are invalidated, a lot of unnecessary and expensive network requests can still incur. For example, if all the relevant data regarding a change was provided in the socket message informing the client about the change, then, in theory, no network requests need to be made — only the query cache needs to be updated.

4 Research questions

RQ1: How can we take care of the query cache updating in a generic and robust way?

5 Goal

Come up with a solution which can take care of the query cache updating in a generic and robust way.

Generic as in decoupled and deduplicated (reducing the amount of boilerplate needed for each query cache updater). Robust as in hiding the complexity and writing it only once with meticulous care and precision.

6 Hypothesis

The Cloud Firestore client-side SDK provides an unparalleled interface, in terms of elegance, for dealing with "real-time" data in the client-side code. Combining the positive traits of the Cloud Firestore client-side SDK with TanStack Query could hypothetically solve the most prominent problem I face when developing with TanStack Query.

The most prominent problem I face when developing with TanStack Query is that if I opt-out of revalidation, then I have to write huge amounts of boilerplate-heavy fragile query cache updating code which is tightly coupled with both the data model and the UI of the app. I identify that there's a missing piece here — a need for a solution which can take care of the query cache updating in a generic and robust way. My hypothesis is that the recipe for that solution can be derived from the Cloud Firestore client-side SDK.

7 Methods

In brief, my method can be broken into two parts of which each part can further be broken into steps:

1. Research
    a. Find related work.
    b. Review related work.
    c. Assess related work.
2. Implementation
    a. Develop an example application which demonstrates the problem.
    b. Develop the solution to the problem in the example application.
    c. Extract the solution from the example application into a library.

7.1 Research

I explore related work to gain a greater insight into potential ways of solving the problem and to find pre-existing solutions to potentially draw from, use in, integrate into, or embed into my solution.

7.1.1 Finding related work

I search for related work with the initial search terms:

- "cloud firestore client-side sdk features but open-source and not google"
- "open-source, non-google solutions for listening to queries like in cloud firestore client-side sdk"

7.1.2 Comparing related work

I compare related work based on the following items:

- Modularity
    - **Full stack**
      The solution is full stack, meaning it comes with both a server-side component and a client-side component.

- **Backend-agnostic**

  The solution is backend-agnostic, meaning it can be used with any kind of backend.
- **Modular**

  The solution is modular, meaning it can be incrementally adopted or used in a partial manner. The solution is shipped more like a toolkit where you can pick and use the tools you like and discard the rest and less like a solution that you either must buy into entirely or not use at all.

- Developer experience
  - **1st-class TypeScript support**

    The solution is written in TypeScript and it ships with detailed type definitions.
  - **Graphical developer tools**

    There are official graphical developer tools to be used with the solution.
- UI framework integrations
  - **1st-class React support**

    The solution has an official React.js (*React*, n.d.) adapter.
  - **1st-class Solid support**

    The solution has an official SolidJS (*SolidJS*, n.d.) adapter.
  - **1st-class Vue support**

    The solution has an official Vue.js (*Vue.Js - The Progressive JavaScript Framework | Vue.Js*, n.d.) adapter.
  - **1st-class Svelte support**

    The solution has an official Svelte (*Svelte • Cybernetically Enhanced Web Apps*, n.d.) adapter.
- ORM (Object-Relation Mapping) capabilities
  - **Relationship tracking**

    The solution "understands" relationships in your data.
  - **Live queries**

    The solution provides a mechanism to listen for when the result of a query for data changes.

- Consideration of asynchronicity and concurrency
    - **Optimistic updates**

      The solution provides a dedicated mechanism for optimistic updates and the use-case of optimistic updates was taken into consideration in the design of the solution.
    - **Browser tab sync**

      The solution provides a way to synchronize the state between browser tabs.
- Offline support
    - **Offline-first**

      The solution embraces the offline-first use-case and can be used to power offline-first web experiences.
    - **Data persistence**

      The solution provides a mechanism to persist data.
    - **Create when offline, publish when online**

      The solution provides a mechanism to create data when offline, persist it locally, and publish it to any potential remote data source once the client's network is reconnected (the remote data source becomes available again).

TanStack Query and the Cloud Firestore client-side SDK are included in the comparison for the sake of reference.

The items are a union of key features of the solutions being compared.

The purpose of the comparison is to map out where each solution predominantly exists in the problem space and how much territory it covers.

7.1.3 Assessing related work

I decide on how to proceed with my solution which can take care of the query cache updating in a generic and robust way.

I assess if and to what degree related work can be utilized in my solution. I want to avoid reinventing the wheel, so, if possible, my solution should be an integration of pre-existing solution and not an entirely new solution.

7.2 Implementation

I first solve the problem in an example application which uses TanStack Query. I then extract the solution into a separate library.

7.2.1 Developing an example application

I develop an example application which uses TanStack Query, and which acts as a sandbox for developing the solution to the problem as well as a practical example of the problem, prior to the solution being developed in it.

The application should resemble a typical chat application, the likes of

- Slack (*Slack Is Your Productivity Platform*, n.d.),
- Discord (*Discord | Your Place to Talk and Hang Out*, n.d.),
- Telegram (*Telegram – a New Era of Messaging*, n.d.),
- or WhatsApp (*WhatsApp*, n.d.).

The UI is expected to look like this:

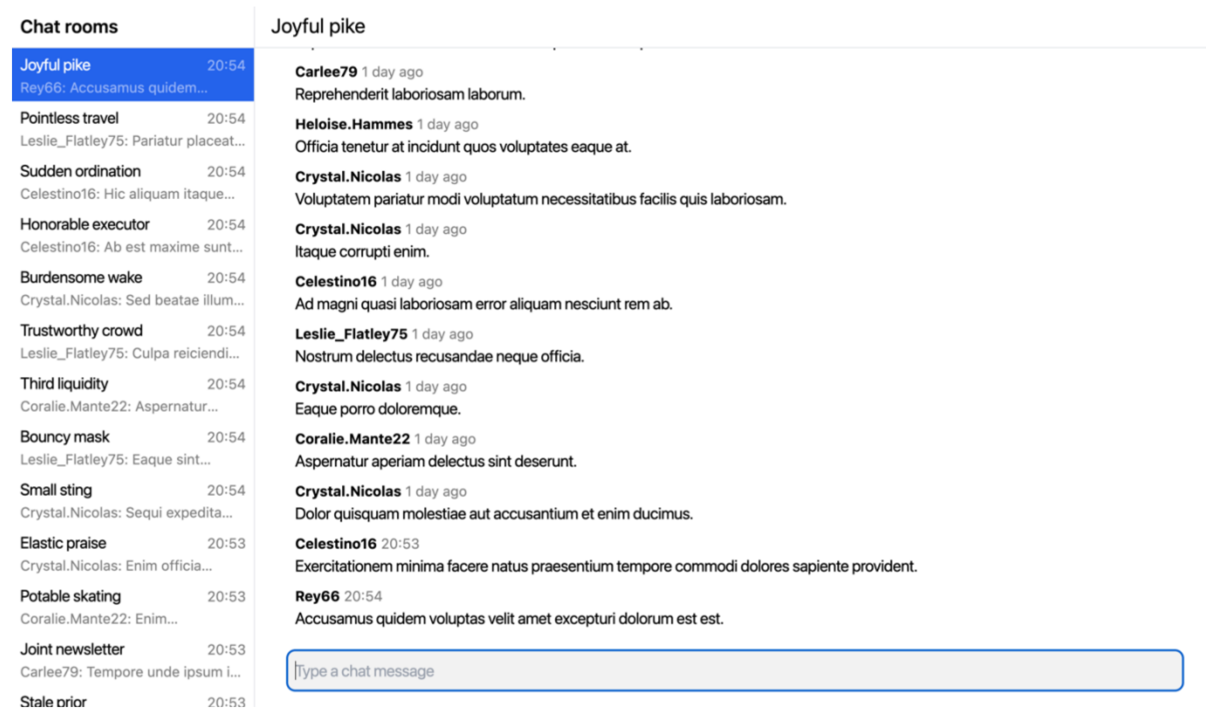

Figure 1. Example chat application UI

On the left-hand side, you have a list of chat rooms. On the right-hand side, you have a list of messages in the currently selected chat room.

Detailed requirements

- It should be able to fetch and display a list of chat rooms.
  - Ordered by when the latest chat message was sent, descending.
  - A preview of the latest chat message should be displayed in the chat room list item.
  - The list should be an “infinite scroll” list, meaning it should load more chat rooms as the user scrolls down.
- It should let the user select a chat room to view the messages in that room and send new messages to that room.
- It should be able to fetch and display a list of chat messages in the currently selected chat room.
  - Ordered by when the chat message was sent, descending.
  - The list should be an “infinite scroll” list, meaning it should load more chat messages as the user scrolls to the end of the list.
- It should let the user send a new chat message to the currently selected room.
- It should automatically and as quickly as possible update the UI to reflect changes to the data when new chat messages arrive or when the user has sent a new chat message.

7.2.2 Solving the problem

I solve the problem in the example application.

This method was chosen due to its agile attributes. Carving out the solution this way benefits from quick iteration and a tight feedback loop and you stay close to reality as your development server provides you with real-time feedback on whether your solution works in practice or not and whether it actually solves the problem or not.

7.2.3 Packaging the solution as a library

I extract the solution from the example application into a separate library, and package it in a way which allows it to be consumed and used in any application that uses TanStack Query, with minimal additional configuration.

8 Related work

8.1 Overview

8.1.1 CushionDB

CushionDB (*CushionDB*, n.d.) is a small open-source project created by three software developers: Avshar Kirksall (*Avshrk on GitHub*, n.d.), Daniel Rote (*Drote on GitHub*, n.d.) and Jaron Truman (*Jtruman88 on GitHub*, n.d.). It describes itself as an "open-source database for progressive web applications". It especially focuses on offline-first data management and synchronization.

8.1.2 Dexie.js

Dexie.js (*Dexie.Js - Minimalistic IndexedDB Wrapper*, n.d.) is one of the most popular open-source libraries for interacting with IndexedDB (*IndexedDB API - Web APIs | MDN*, 2023), the web browser's built-in database for storing and retrieving large amounts of structured data. The author of Dexie.js is David Fahlander (*Dfahlander on GitHub*, n.d.). It describes itself as a "Minimalistic Wrapper for IndexedDB". Arguably, it isn't that minimalistic. But it has a lot of neat features such as live queries (*LiveQuery()*, n.d.) and browser tab sync (*Dexie.on.Storagemutated*, n.d.), to name a few.

8.1.3 Orbit.js

Orbit.js (*Orbit.Js - The Universal Data Layer | Orbit.Js*, n.d.) is an open-source project from Cerebris Corporation (*Cerebris :: Developers of Ambitious Web Applications*, n.d.), a "small company with a BIG open source presence" (*Cerebris :: Projects*, n.d.). It describes itself as

- "The Universal Data Layer" (website tagline),
- "Composable data framework for ambitious web applications" (description on GitHub (*Orbitjs/Orbit on GitHub*, 2013/2023)),
- and "Orbit is a composable data framework for managing the complex needs of today's web applications" (first sentence in the `README.md`).

The author and core maintainer of Orbit.js is Dan Gebhardt (*Cerebris :: Projects*, n.d.; *Dgeb on GitHub*, n.d.), Principal Software Engineer at and Co-Founder of Cerebris Corporation (*Dan Gebhardt | LinkedIn*, n.d.). He's also a core maintainer of Ember.js (*Ember.Js - A Framework for Ambitious Web Developers*, n.d.) and Glimmer.js (*Glimmer*, n.d.) and the JSON:API (*JSON:API — A Specification for Building APIs in JSON*, n.d.) specification.

8.1.4 SQLite as a WebAssembly module

`wa-sqlite` (Hashimoto, 2021/2023) by Ryo Hashimoto (*Rhashimoto on GitHub*, n.d.) is a WebAssembly (*WebAssembly*, n.d.) build of SQLite (*SQLite Home Page*, n.d.) which effectively brings a fully-fledged relational database to the web platform.

8.2 Comparison

Table 1. Related work: Comparison: Modularity

	Query	Firestore	CushionDB	Dexie.js	Orbit.js	SQLite
Full stack		✓	✓			
Backend-agnostic	✓			✓	✓	✓
Modular					✓	

Table 2. Related work: Comparison: Developer experience

	Query	Firestore	CushionDB	Dexie.js	Orbit.js	SQLite
1st-class TypeScript support	✓	✓		✓	✓	✓
Graphical developer tools	✓					

Table 3. Related work: Comparison: UI framework integrations

	Query	Firestore	CushionDB	Dexie.js	Orbit.js	SQLite
1st-class React support	✓			✓		
1st-class Solid support	✓					
1st-class Vue support	✓			✓		
1st-class Svelte support	✓			✓		

Table 4. Related work: Comparison: ORM capabilities

	Query	Firestore	CushionDB	Dexie.js	Orbit.js	SQLite
Relationship tracking					✓	✓
Live queries		✓		✓	✓	

Table 5. Related work: Comparison: Consideration of asynchronicity and concurrency

	Query	Firestore	CushionDB	Dexie.js	Orbit.js	SQLite
Optimistic updates	✓	✓			✓	
Browser tab sync	✓	✓		✓		

Table 6. Related work: Comparison: Offline support

	Query	Firestore	CushionDB	Dexie.js	Orbit.js	SQLite
Offline-first		✓	✓		✓	
Data persistence	✓	✓	✓	✓	✓	✓
Creating when offline, publish when online		✓	✓		✓	

Query stands for TanStack Query.

Firestore stands for the Cloud Firestore client-side SDK.

SQLite stands for SQLite as a WebAssembly module.

8.2.1 Modularity

Full stack

Firestore and CushionDB are both entirely full stack solutions. TanStack Query, Orbit.js and SQLite are entirely client-side solutions. Dexie.js is a client-side solution, but on Dexie.js' home page, as of writing this, you can find a link to Dexie Cloud$^{\text{BETA}}$ (*Dexie Cloud*, n.d.), a cloud-hosted sync service for Dexie.js.

Backend-agnostic

Firestore and CushionDB are not backend-agnostic solutions. TanStack Query, Dexie.js, Orbit.js and SQLite are backend-agnostic solutions.

Modular

Orbit.js is the only solution out of the bunch that is truly modular in the sense that it's shipped as a toolkit where you can pick and use the tools you like and discard the rest and less like a solution that you either must buy into entirely or not at all.

8.2.2 Developer experience

1st-class TypeScript support

TanStack Query, Firestore, Dexie.js and Orbit.js are all written in TypeScript. The source for the JavaScript bindings in `wa-sqlite` is written in JavaScript, but the library ships with its own TypeScript type definition declaration files. CushionDB is written in JavaScript.

Graphical developer tools

TanStack Query is the only solution out of the bunch that has graphical developer tools.

8.2.3 UI framework integrations

1st-class React support

TanStack Query and Dexie.js both have 1st-class support for React:

- React Query | TanStack Query Docs (*React Query | TanStack Query Docs*, n.d.)
- Get started with Dexie in React (*Get Started with Dexie in React*, n.d.)

The GitHub organization Orbit.js (*Orbit.Js*, n.d.) has a small package called `react-orbit` (*Orbitjs/React-Orbit on GitHub*, 2019/2023), but it's very minimal, so I would categorize it as an example of how to use Orbit.js with React instead of viewing it as Orbit.js having 1st-class React support.

Firestore, CushionDB and SQLite don't have 1st-class support for React.

1st-class Solid support

TanStack Query is the only solution out of the bunch which has 1st-class support for Solid:

- Solid Query | TanStack Query Docs (*Solid Query | TanStack Query Docs*, n.d.)

1st-class Vue support

TanStack Query and Dexie.js both have 1st-class support for Vue:

- Vue Query | TanStack Query Docs (*Vue Query | TanStack Query Docs*, n.d.)
- Get started with Dexie in Vue (*Get Started with Dexie in Vue*, n.d.)

Firestore, CushionDB, Orbit.js and SQLite don't have 1st-class support for Vue.

1st-class Svelte support

TanStack Query and Dexie.js both have 1st-class support for Svelte:

- Svelte Query | TanStack Query Docs (*Svelte Query | TanStack Query Docs*, n.d.)
- Get started with Dexie in Svelte (*Get Started with Dexie in Svelte*, n.d.)

Firestore, CushionDB, Orbit.js and SQLite don't have 1st-class support for Svelte.

8.2.4 ORM capabilities

Relationship tracking

Orbit.js is aware of relationships in your data. SQLite is naturally aware of relationships in your data as it's a fully-fledged relational database which lets you interact with it using SQL (Structured Query Language). TanStack Query, Firestore, CushionDB and Dexie.js are not aware of relationships in your data.

Live queries

Firestore has live queries in the form of the `onSnapshot` API. Dexie.js and Orbit.js have live queries:

- liveQuery() | dexie.org
- Live Queries | Orbit.js (*Live Queries | Orbit.Js*, n.d.)

TanStack Query, CushionDB and SQLite don't have live queries.

8.2.5 Consideration of asynchronicity and concurrency

Optimistic updates

TanStack Query, Firestore and Orbit.js provide mechanisms for doing optimistic updates. CushionDB, Dexie.js and SQLite don't provide any mechanisms for doing optimistic updates.

Browser tab sync

Firestore and Dexie.js synchronize their states across browser tabs. TanStack Query has a plugin (*BroadcastQueryClient (Experimental) | TanStack Query Docs*, n.d.), which is as of writing this annotated as "experimental", which synchronizes the `QueryClient`'s state across browser tabs. CushionDB, Orbit.js and SQLite don't synchronize their states across browser tabs.

8.2.6 Offline support

Offline-first

Firestore, CushionDB, and Orbit.js were designed with the enablement of offline-first web experiences in mind. TanStack Query, Dexie.js and SQLite were not designed with the enablement of offline-first web experiences in mind.

Data persistence

All the solutions provide mechanisms to persist data.

Creating when offline, publish when online

Firestore, CushionDB, and Orbit.js provide mechanisms for creating data when offline and publishing it when online. TanStack Query, Dexie.js and SQLite don't provide mechanisms for creating data when offline and publishing it when online, the exception being Dexie.js, if you use it with Dexie Cloud[BETA].

8.3 Analysis

Let's take a closer look at my hypothesis that a recipe for the solution can be derived from the Cloud Firestore client-side SDK. I initially asked myself the question: "What enables the Cloud Firestore client-side SDK to have an API such as the `onSnapshot` API, which lets you listen to when the result of a query changes?". But that is not the right question to ask, because the framework-agnostic core of TanStack Query has a very similar API to the Cloud Firestore client-side SDK's `onSnapshot` API in that you can subscribe to be notified when the result of query changes. So, the answer to that question is simply: "The same thing that enables it in TanStack Query: observables and observers.".

What is then the difference between TanStack Query and the Cloud Firestore client-side SDK? Both have queries and both let you listen to when the result of a query changes.

Let's explore the meaning of the word *query* in the different libraries. In general, *query* is, in the context of software development, a request for specific information from a database or other data storage system. In TanStack Query, a *query* can be more specifically described as an abstraction for an asynchronous read operation, its state, and its cached result. In the Cloud Firestore client-side SDK, a *query* can be more specifically described as an abstraction which describes the request for specific information itself, meaning the request itself, in a structured way, which the library understands. The Cloud Firestore client-side SDK has a query language, while TanStack Query does not.

In TanStack Query, the meaning of the key that identifies a query is of no interest to the library. It only sees the key as a pointer to some value in the `QueryCache`, which is essentially a key-value store. Similarly, the data stored as the result of a query is opaque to TanStack Query. In contrast, the Cloud Firestore client-side SDK understands the data that flows through the library. This is what allows the Cloud Firestore client-side SDK to know based on a socket message informing about a change, how the in-memory data should be mutated to reflect the change and which query listeners should be notified.

I have decided that Orbit.js is the solution to taking care of the query cache updating in a generic and robust way. I will implement a solution for using the solution (Orbit.js) together with TanStack Query.

Orbit.js is the solution because:

- It's modular/composable. It can be used in a way where it just solves the query cache updating problem, but changes little else about the application, or you can go all in, and use it to power offline-first web experiences.
- It has a query language, and it understands the data that flows through it, and it lets you listen to when the result of query changes, like the Cloud Firestore client-side SDK. Additionally, it lets you describe relationships in your data, which the Cloud Firestore client-side SDK doesn't let you do to the same degree.
- It's an entirely client-side solution and it's backend-agnostic. The solution doesn't require the entire stack to change.

9 Results

9.1 A concrete example of the problem

Thanks to the method I chose when implementing the solution, there's now a concrete example of the problem, which can be used to demonstrate the problem in a practical step-by-step manner.

A practical step-by-step demonstration of the problem follows below.

Imagine you're building the chat application described in *7.2.1 Methods: Implementation: Developing an example application*, using React.js and TanStack Query.

9.1.1 Initial code

You would probably start out by creating an `App` component that looks something like this:

```
export const App = () => {
    const [
        selectedChatRoomId,
        setSelectedChatRoomId,
    ] = React.useState<string | null>(null);
    return (
        <div>
            <ChatRoomList
                selectedChatRoomId={selectedChatRoomId}
                setSelectedChatRoomId={setSelectedChatRoomId}
            />
            <ChatRoom chatRoomId={selectedChatRoomId} />
        <div/>
    );
};
```

Figure 2. src/components/App.tsx (initial)

Note: for the sake of brevity only the most relevant source code for this example is shown. For the full source code, see the GitHub repository.

Then you would go on to implement the `ChatRoomList` component…

```
const queryFn = async () => {
    const response = await fetch("/api/chat-rooms?count=10");
    return response.json();
};

export const ChatRoomList = ({
    selectedChatRoomId,
    setSelectedChatRoomId
}: ChatRoomListProps) => {
    const {data: chatRooms} = useQuery({
        queryKey: ["chat-rooms"],
        queryFn,
    });
    return (
        <ul>
            {chatRooms?.map((chatRoom) => (
                // Assumed stateless component,
                // implementation not of interest in this example
                <ChatRoomListItem
                    key={chatRoom.id}
                    chatRoom={chatRoom}
                    selected={chatRoom.id === selectedChatRoomId}
                    onClick={() => setSelectedChatRoomId(chatRoom.id)}
                />
            ))}
        <ul/>
    );
};
```

Figure 3. src/components/ChatRoomList.tsx

…and the `ChatRoom` component.

```
export const ChatRoom = ({chatRoomId}: ChatRoomProps) => {
    return (
        <div>
            <ChatMessageList chatRoomId={chatRoomId} />
            <ChatMessageInput chatRoomId={chatRoomId} />
        <div/>
    );
};
```

Figure 4. src/components/ChatRoom.tsx

Now, there are two more components to implement.

The `ChatMessageList` component…

```
const queryFn = async (
    ctx: QueryFunctionContext<["chat-messages", string]>,
) => {
    const response = await fetch(
        `/api/chat-room/${ctx.queryKey[1]}/chat-messages?count=10`,
    );
    return response.json();
};

export const ChatMessageList = ({chatRoomId}: ChatMessageListProps) => {
    const {data: chatMessages} = useQuery({
        queryKey: ["chat-messages", chatRoomId],
        queryFn,
    });
    return (
        <ul>
            {chatMessages?.map((chatMessage) => (
                // Assumed stateless component,
                // implementation not of interest in this example
                <ChatMessageListItem
                    key={chatMessage.id}
                    chatMessage={chatMessage}
                />
            ))}
        <ul/>
    );
};
```

Figure 5. src/components/ChatMessageList.tsx

…and the `ChatMessageInput` component.

```
const mutationFn = async (
    {chatRoomId, text}: {chatRoomId: string; text: string},
) => {
    const response = await fetch(
        `/api/chat-room/${chatRoomId}/chat-message`,
        {
            method: "POST",
            headers: {
                "Content-Type": "application/json",
            },
            body: JSON.stringify({text}),
        },
    );
    return response.json();
};

export const ChatMessageInput = ({chatRoomId}: ChatMessageInputProps) => {
    const [text, setText] = React.useState("");
    const {mutate: sendChatMessage, isLoading} = useMutation({
        mutationFn,
        onSuccess: () => setText(""),
    });
    const onChange = (event: React.ChangeEvent<HTMLInputElement>) => {
        setText(event.target.value);
    };
    const onKeyDown = (event: React.KeyboardEvent<HTMLInputElement>) => {
        if (event.key === "Enter") {
            sendChatMessage({chatRoomId, text});
        }
    };
    return (
        <input
            value={text}
            disabled={isLoading}
            onChange={onChange}
            onKeyDown={onKeyDown}
        />
    );
};
```

Figure 6. src/components/ChatMessageInput.tsx

9.1.2 Updating the UI when new chat messages arrive

You now have a chat application that works to a certain extent but is limited in its functionality in that it doesn't update the UI when new chat messages arrive or when the user sends new chat messages, which is something users have come to expect from modern day chat applications.

Updating the UI when new chat messages arrive can be implemented fairly easily.

In the `App` component, a [`useEffect` hook](#) (*UseEffect – React*, n.d.) can be added that subscribes to a [`WebSocket`](#) connection that receives a message whenever there is a new chat message. Then, using the [`QueryClient`](#), we can [invalidate the queries](#) (*Query Invalidation | TanStack Query Docs*, n.d.), causing the queries to be re-fetched and the UI to be updated.

```
export const App = () => {
    const queryClient = useQueryClient();
    const [
        selectedChatRoomId,
        setSelectedChatRoomId,
    ] = React.useState<string | null>(null);
    React.useEffect(() => {
        const socket = getSocket();
        socket.on("new-chat-message", () => {
            void queryClient.invalidateQueries();
        })
        return () => socket.disconnect();
    }, [queryClient]);
    return (
        <div>
            <ChatRoomList
                selectedChatRoomId={selectedChatRoomId}
                setSelectedChatRoomId={setSelectedChatRoomId}
            />
            <ChatRoom chatRoomId={selectedChatRoomId} />
        <div/>
    );
};
```

Figure 7. src/components/App.tsx (with socket connection)

However, this is very inefficient. Calling [`QueryClient#invalidateQueries`](#) (*QueryClient#invalidateQueries | TanStack Query Docs*, n.d.) is effectively causing the app to redo all the API requests that it has done so far, which is a lot of unnecessary work.

A better alternative would be to only invalidate the queries that are affected by the new chat message. That is:

- The `["chat-messages"]` query for the chat room that the new message arrived in.
- The `["chat-rooms"]` query, because the chat rooms are sorted according to when the latest message arrived in the room, and a preview of the latest message is shown in the chat room list item.

Let's assume that the new chat message is included in the socket message, so the information about the chat room that the new message arrived in can be retrieved from the socket message.

```
socket.on("new-chat-message", (chatMessage: ChatMessage) => {
    void queryClient.invalidateQuery(
        ["chat-messages", chatMessage.chatRoomId],
    );
    void queryClient.invalidateQuery(
        ["chat-rooms"],
    );
});
```

Figure 8. Invalidating specific queries

Frankly, this is still inefficient. As the new chat message in included in the socket message, in most cases, all the information necessary is already present in the app's memory in some shape or another, and there is theoretically no need to make any additional API requests.

"In most cases", because no assumption was made that the chat room itself would be included in the socket message, and in case it doesn't exist in cache from before, it would have to be fetched separately before the update can be applied.

QueryClient#setQueryData (*QueryClient#setQueryData | TanStack Query Docs*, n.d.) can be used to manually update the queries' cached data and avoid doing any network requests.

```
socket.on("new-chat-message", (chatMessage: ChatMessage) => {
    queryClient.setQueryData(
        ["chat-messages", chatMessage.chatRoomId],
        (data) => [
            chatMessage,
            // data might be undefined if the query
            // doesn't exist from before
            ...(data ?? []),
        ],
    );
    queryClient.setQueryData(["chat-rooms"], (data) => {
        // data might be undefined if the query
        // doesn't exist from before
        if (data == null) {
            return;
        }
        // make a copy of the array which we are allowed to mutate
        const newData = [...data];
        const chatRoomIndex = newData.findIndex(
            (chatRoom) => chatRoom.id === chatMessage.chatRoomId,
        );
        // remove the chat room from the array
        const [chatRoom] = newData.splice(chatRoomIndex, 1);
        // prepend the a clone of chat room object to the array
        // where latestChatMessage is set to the new chat message
        newData.unshift({
            ...chatRoom,
            latestChatMessage: chatMessage,
        });
        return newData;
    });
});
```

Figure 9. Using QueryClient#setQueryData

As hinted earlier in the text, this only works in most cases. The case where the chat room doesn't exist in the `["chat-rooms"]` query's cached data is not accounted for. It cannot be assumed that the `["chat-rooms"]` query's cached data contains all chat rooms. In the example implementation of the `ChatRoomList` component (*Figure 3.*) the `["chat-rooms"]` query's cached data will contain the 10 most recently active chat rooms if the query resolved successfully.

In the real implementation which can be found in the GitHub repository and which meets the requirements listed in *7.2.1 Methods: Implementation: Developing an example application*, "infinite scrolling" has been implemented, which allows the user to view rooms past the 10

first. But even though more than 10 chat rooms may exist in cache, we cannot assume that it's all chat rooms.

There are several ways to address this. Ignoring the case where the chat room doesn't exist in the `["chat-rooms"]` query's cached data isn't one of those ways. Even if the new chat message belongs to a room which wasn't among the 10 first rooms, that room is now among the 10 first rooms, since the order of the rooms is determined by when chat messages last arrived in them.

In the example app, reverting to calling `queryClient.invalidateQuery(["chat-rooms"])` won't do much harm. Arguably, the added complexity of manually trying to update the cached data isn't worth it in this case.

But for real chat apps that implement pagination, the situation is arguably different. It might then be a question of potentially re-fetching hundreds of chat rooms just because one chat room got a new message.

9.1.3 Handling the case where the chat rooms doesn't exist in the `["chat-rooms"]` query's cached data

The easiest solution from the point of view of a frontend developer is probably if the backend makes a change to include the chat room object as well as the chat message object in the socket message which informs the client about the new chat message.

But the socket service adapting this way to accommodate to very specific frontend needs is a luxury which cannot be expected every time a case like this is encountered. The socket service has performance and efficiency considerations of its own that it needs to care about, and bundling more data in the socket messages goes strictly against those considerations.

So, as a frontend developer, you might as well accept that you need to be able to handle this in the frontend.

The first step is to go asynchronous.

```ts
export const onNewChatMessage = async (
    queryClient: QueryClient,
    chatMessage: ChatMessage,
) => {
    queryClient.setQueryData(
        ["chat-messages", chatMessage.chatRoomId],
        (data) => [
            chatMessage,
            // data might be undefined if the query
            // doesn't exist from before
            ...(data ?? []),
        ],
    );
    const data = queryClient.getQueryData(["chat-rooms"]);
    // make a copy of the array which we are allowed to mutate
    const newData = [...(data ?? [])];
    const chatRoomIndex = newData.findIndex(
        (chatRoom) => chatRoom.id === chatMessage.chatRoomId,
    );
    let chatRoom: ChatRoom;
    // if chat room doesn't exist, fetch it
    if (chatRoomIndex === -1) {
        try {
            chatRoom = await fetchChatRoom(chatMessage.chatRoomId);
        } catch {
            // if fetching the chat room fails, we surrender (for now)
            return;
        }
    }
    // else remove it from the array
    else {
        const staleChatRoom = newData.splice(chatRoomIndex, 1)[0];
        chatRoom = {
            ...staleChatRoom,
            latestChatMessage: chatMessage,
        };
    }
    // prepend chat room to the array
    newData.unshift(chatRoom);
    queryClient.setQueryData(["chat-rooms"], newData);
};
```

Figure 10. src/socket-message-handlers/onNewChatMessage.ts (initial)

```ts
socket.on("new-chat-message", (chatMessage: ChatMessage) => {
    queryClient.setQueryData(
        ["chat-messages", chatMessage.chatRoomId],
        … (6 hidden lines)
    );
    queryClient.setQueryData(["chat-rooms"], (data) => {
        … (19 hidden lines)
    });
    void onNewChatMessage(queryClient, chatMessage);
});
```

Figure 11. Going asynchronous

9.1.4 Dealing with concurrency

The `onNewChatMessage` function (*Figure 10.*) does not take into consideration if any of the queries whose cached data it's modifying are currently in-flight. This could affect the outcome in different unwanted ways.

To dodge this potential bullet, the query which the function is currently operating on must be cancelled if it's currently in-flight. It can be done by using [`QueryClient#cancelQueries`]() (*QueryClient#cancelQueries | TanStack Query Docs*, n.d.). Then after doing the synchronous query cache update, if the query was being fetched prior to the update, the function must call [`QueryClient#refetchQueries`]() (*QueryClient#refetchQueries | TanStack Query Docs*, n.d.) to make sure that whatever was being fetched prior to the update still gets fetched in the end.

```
export const onNewChatMessage = async (…) => {
    const wasFetchingChatMessages =
        queryClient.isFetching(["chat-messages", chatMessage.chatRoomId]) > 0;
    if (wasFetchingChatMessages) {
        await queryClient.cancelQueries([
            "chat-messages",
            chatMessage.chatRoomId,
        ]);
    }
    queryClient.setQueryData(
        ["chat-messages", chatMessage.chatRoomId],
        … (6 hidden lines)
    );
    if (wasFetchingChatMessages) {
        void queryClient.refetchQueries([
            "chat-messages",
            chatMessage.chatRoomId,
        ]);
    }
    const wasFetchingChatRooms = queryClient.isFetching(["chat-rooms"]) > 0;
    if (wasFetchingChatRooms) {
        await queryClient.cancelQueries(["chat-rooms"]);
    }
    const data = queryClient.getQueryData(["chat-rooms"]);
    // make a copy of the array which we are allowed to mutate
    const newData = [...(data ?? [])];
    … (22 hidden lines)
    queryClient.setQueryData(["chat-rooms"], newData);
    if (wasFetchingChatMessages) {
        void queryClient.refetchQueries(["chat-rooms"]);
    }
};
```

Figure 12. Dealing with concurrency – Part 1

Since `fetchChatRoom` is asynchronous, by the time it's finished, the cached data for the `["chat-rooms"]` query might already have changed and differ from what's in `newData`.

```
export const onNewChatMessage = async (…) => {
    … (24 hidden lines)
    constlet data = queryClient.getQueryData(["chat-rooms"]);
    // make a copy of the array which we are allowed to mutate
    constlet newData = [...(data ?? [])];
    constlet chatRoomIndex = newData.findIndex(
        (chatRoom) => chatRoom.id === chatMessage.chatRoomId,
    );
    let chatRoom: ChatRoom;
    // if chat room doesn't exist, fetch it
    if (chatRoomIndex === -1) {
        try {
            // synchronous handling of socket message ends here
            chatRoom = await fetchChatRoom(chatMessage.chatRoomId);
        } catch {
            // if fetching the chat room fails, we surrender (for now)
            return;
        }
        // we need to check for in-flight queries and cancel them again
        // since we "left" the synchronous execution context
        wasFetchingChatRooms = queryClient.isFetching(["chat-rooms"]) > 0;
        if (wasFetchingChatRooms) {
            await queryClient.cancelQueries(["chat-rooms"]);
        }
        data = queryClient.getQueryData(["chat-rooms"]);
        newData = [...(data ?? [])];
        chatRoomIndex = newData.findIndex(
            (chatRoom) => chatRoom.id === chatMessage.chatRoomId,
        );
        if (chatRoomIndex !== -1) {
            // we can assume that the chat room we just fetched
            // is as up-to-date or more up-to-date than the one
            // that was added to the cache while we were fetching
            newData[chatRoomIndex] = chatRoom;
        } else {
            // prepend chat room to the array
            newData.unshift(chatRoom);
        }
        // we cannot assume that the chat room we just fetched
        // should still go to the top of the list
        newData.sort(latestChatMessageCreatedAtDescendingCompareFn);
    }
    // else remove it from the array
    else {
        … (4 hidden lines)
        // prepend chat room to the array
        newData.unshift(chatRoom);
    }
    // prepend chat room to the array
    newData.unshift(chatRoom);
    queryClient.setQueryData(["chat-rooms"], newData);
    if (wasFetchingChatMessages) {
        void queryClient.refetchQueries(["chat-rooms"]);
    }
};
```

Figure 13. Dealing with concurrency – Part 2

Now that concurrency has been taken into consideration, this particular query cache updater is finished.

9.1.5 Summing it up

The `onNewChatMessage` function was just one query cache updater. As the app grows, the need to create more of these updaters will arise, and while the code can be organized into neat file structures and parts of it can be extracted into helper functions that can be reused across those files, there will still no doubt be a lot of query cache updating code to maintain.

A single forgotten conditional statement or unhandled edge-case in any of these query cache updaters can lead to a cascade of bugs that are hard to track down and fix.

Whenever there are changes to the data model or the UI of the application, it's likely that some changes need to be made to the query cache updaters as well, since they are tightly coupled with both the data model and the UI of the application.

This code is not something you should have to write and maintain yourself, especially since there are other solutions (such as Orbit.js and the Cloud Firestore client-side SDK) that prove that the desired functionality can be achieved in other ways which eliminate the need for you to write this kind of code and — in addition to that — work more reliably.

9.2 The solution

The solution comprises of two libraries:

- `@tanstack-query-with-orbitjs/core`
- `@tanstack-query-with-orbitjs/react`

9.2.1 `@tanstack-query-with-orbitjs/core`

`@tanstack-query-with-orbitjs/core` is effectively an extension of, a wrapper of, a flavor of or a preset for `@tanstack/query-core`; the UI framework agnostic core of TanStack Query (*@tanstack/Query-Core on GitHub*, n.d.).

`@tanstack-query-with-orbitjs/core` is also UI framework agnostic. As the name suggests, it's a library for using TanStack Query together with Orbit.js. It exports the following items:

- `LiveQueryClient`
  Replacement for `QueryClient`. (Extends `QueryClient` class.)
- `LiveQueryClientConfig`
  Type definition. (Extends `QueryClientConfig` (*QueryClientConfig in @tanstack/Query-Core on GitHub*, n.d.) interface.)
- `QueryMeta`
  Module augmented, and declaration merged more specific version of the `QueryMeta` (*QueryMeta in @tanstack/Query-Core on GitHub*, n.d.) interface.
- `GetQueryOrExpressions`
  Type definition for a function signature which has a central role when using the library.
- `LiveQueryObserver`
  Replacement for `QueryObserver` (*QueryObserver | TanStack Query Docs*, n.d.). (Extends `QueryObserver` class.)
- `LiveInfiniteQueryObserver`
  Replacement for `InfiniteQueryObserver` (*InfiniteQueryObserver | TanStack Query Docs*, n.d.). (Extends `InfiniteQueryObserver` class.)

Using `@tanstack-query-with-orbitjs/core` differs from using `@tanstack/query-core` in that the `LiveQueryClient` constructor requires that you pass it a reference to an Orbit.js `MemorySource` (*Class: MemorySource<QO, TO, QB, TB, QRD, TRD> | Orbit.Js*, n.d.) and in that you don't pass `queryFn`s (*Query Functions | TanStack Query Docs*, n.d.) to observers when instantiating them. Instead, the library makes use of the meta object that `@tanstack/query-core` associates with each query. In the meta object, you specify a `getQueryOrExpressions` function which returns the query or expression which the default `queryFn` (*Default Query Function | TanStack Query Docs*, n.d.) uses to query the memory source and which is used to create an Orbit.js live query which automatically keeps your query up to date.

For more information, check out the library's `README.md`, where how to use the library is covered in greater detail, or check out the source code for the example chat application which uses this solution.

9.2.2 `@tanstack-query-with-orbitjs/react`

`@tanstack-query-with-orbitjs/react` contains React bindings for using `@tanstack-query-with-orbitjs/core` in a React application. It exports:

- **`useLiveQuery`**
  Same as `useQuery` (*UseQuery | TanStack Query Docs*, n.d.), but for when using `@tanstack-query-with-orbitjs/core`.
- **`useLiveInfiniteQuery`**
  Same as `useInfiniteQuery` (*UseInfiniteQuery | TanStack Query Docs*, n.d.), but for when using `@tanstack-query-with-orbitjs/core`.
- **`useLiveQueryClient`**
  Same as `useQueryClient` (*UseQueryClient | TanStack Query Docs*, n.d.), but for when using `@tanstack-query-with-orbitjs/core`.

For more information, check out the library's `README.md`, where how to use the library is covered in greater detail, or check out the source code for the example chat application which uses this solution.

10 Conclusion

I reached my goal of coming up with a solution that can take of the query cache updating in a generic and robust way.

The solution is to use Orbit.js. I additionally developed a library for using Orbit.js together with TanStack Query which combines the best of both worlds:

- The excellent UI framework integrations and asynchronous state management of TanStack Query
- The amazing data orchestration capabilities of Orbit.js

Orbit.js eliminates the need for writing query cache updaters. However, you still need to write updaters, where you tell Orbit about changes to data. While these updaters aren't as concise as I had expected and the number of lines of code might not differ all too much from the number of lines of code of a query cache updater, the code in an updater is significantly more expressive than the code in a query cache updater and the code in an updater is also more static and decoupled, especially from the UI of the application.

10.1 Further development

Orbit.js is an exciting library in many ways and I feel like its potential hasn't been fully tapped into by the web development and industry. Orbit.js is designed in a way which screams "extend me!", with hooks and slots to be found in every corner, that can be used for customizing it and adding features and functionality on top of its already super powerful and solid core. I could potentially see an entire ecosystem of solutions forming around Orbit.js, which would greatly benefit developers who work on large-scale, complex web applications.

At a certain point, however, Orbit.js won't be enough. For really complex web applications this might be the case. I predict the query language and the data model is where Orbit.js would first fall short. Not in the data orchestration features. In that department I've yet to imagine a better solution. But while Orbit.js lets you describe relationships in your data and query based on relationships, it comes with some limitations and its nowhere close to being as good as fully-fledged SQL.

For really complex web applications, I think it would be worthwhile investigating SQLite as a WebAssembly module.

11 References

Avshrk on GitHub. (n.d.). GitHub. Retrieved May 22, 2023, from https://github.com/avshrk

Azure Cosmos DB. (n.d.). Retrieved May 22, 2023, from https://cosmos.azure.com/

BroadcastQueryClient (Experimental) | TanStack Query Docs. (n.d.). Retrieved May 22, 2023, from https://tanstack.com/query/v4/docs/plugins/broadcastQueryClient

Cerebris: Developers of Ambitious Web Applications. (n.d.). Retrieved May 22, 2023, from https://www.cerebris.com/

Cerebris: Projects. (n.d.). Retrieved May 5, 2023, from https://www.cerebris.com/projects/

Class: MemorySource<QO, TO, QB, TB, QRD, TRD> | Orbit.js. (n.d.). Retrieved May 22, 2023, from https://orbitjs.com/docs/api/memory/classes/MemorySource

Cloud Computing Services | Microsoft Azure. (n.d.). Retrieved May 22, 2023, from https://azure.microsoft.com/

Cloud Computing Services—Amazon Web Services (AWS). (n.d.). Amazon Web Services, Inc. Retrieved May 22, 2023, from https://aws.amazon.com/

Cloud Firestore | Store and sync app data at global scale. (n.d.). Firebase. Retrieved May 22, 2023, from https://firebase.google.com/products/firestore

CushionDB. (n.d.). Retrieved May 22, 2023, from https://cushiondb.github.io/

Dan Gebhardt | LinkedIn. (n.d.). Retrieved May 5, 2023, from https://www.linkedin.com/in/dgeb/

Default Query Function | TanStack Query Docs. (n.d.). Retrieved May 22, 2023, from https://tanstack.com/query/v4/docs/guides/default-query-function

Devtools | TanStack Query Docs. (n.d.). Retrieved May 4, 2023, from https://tanstack.com/query/v4/docs/devtools

Dexie Cloud. (n.d.). Retrieved May 22, 2023, from https://dexie.org/cloud/

Dexie.js—Minimalistic IndexedDB Wrapper. (n.d.). Retrieved May 22, 2023, from https://dexie.org/

Dexie.on.storagemutated. (n.d.). Retrieved May 22, 2023, from https://dexie.org/docs/Dexie/Dexie.on.storagemutated

Dfahlander on GitHub. (n.d.). GitHub. Retrieved May 22, 2023, from https://github.com/dfahlander

Dgeb on GitHub. (n.d.). GitHub. Retrieved May 22, 2023, from https://github.com/dgeb

Discord | Your Place to Talk and Hang Out. (n.d.). Discord. Retrieved May 26, 2023, from https://discord.com/

Drote on GitHub. (n.d.). GitHub. Retrieved May 22, 2023, from https://github.com/drote

Ember.js—A framework for ambitious web developers. (n.d.). Retrieved May 22, 2023, from https://emberjs.com/

Fast NoSQL Key-Value Database – Amazon DynamoDB – Amazon Web Services. (n.d.). Amazon Web Services, Inc. Retrieved May 22, 2023, from https://aws.amazon.com/dynamodb/

Firebase. (n.d.). Firebase. Retrieved May 22, 2023, from https://firebase.google.com/

Firebase Realtime Database | Store and sync data in real time. (n.d.). Firebase. Retrieved May 22, 2023, from https://firebase.google.com/products/realtime-database

Get realtime updates with Cloud Firestore. (n.d.). Firebase. Retrieved May 22, 2023, from https://firebase.google.com/docs/firestore/query-data/listen

Get started with Dexie in React. (n.d.). Retrieved May 26, 2023, from https://dexie.org/docs/Tutorial/React

Get started with Dexie in Svelte. (n.d.). Retrieved May 26, 2023, from https://dexie.org/docs/Tutorial/Svelte

Get started with Dexie in Vue. (n.d.). Retrieved May 26, 2023, from https://dexie.org/docs/Tutorial/Vue

Glimmer. (n.d.). Retrieved May 22, 2023, from https://glimmerjs.com/

Google—About Google, Our Culture & Company News. (n.d.). Retrieved May 22, 2023, from https://about.google/

Hashimoto, R. (2023). *Wa-sqlite on GitHub* [JavaScript]. https://github.com/rhashimoto/wa-sqlite (Original work published 2021)

Important Defaults | TanStack Query Docs. (n.d.). Retrieved May 4, 2023, from https://tanstack.com/query/v4/docs/guides/important-defaults

IndexedDB API - Web APIs | MDN. (2023, March 21). https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API

InfiniteQueryObserver | TanStack Query Docs. (n.d.). Retrieved May 26, 2023, from https://tanstack.com/query/v4/docs/reference/InfiniteQueryObserver

JSON:API — A specification for building APIs in JSON. (n.d.). Retrieved May 22, 2023, from https://jsonapi.org/

Jtruman88 on GitHub. (n.d.). GitHub. Retrieved May 22, 2023, from https://github.com/jtruman88

Live queries | Orbit.js. (n.d.). Retrieved May 22, 2023, from https://orbitjs.com/docs/querying-data#live-queries

LiveQuery(). (n.d.). Retrieved May 22, 2023, from https://dexie.org/docs/liveQuery()

Orbit.js. (n.d.). GitHub. Retrieved May 22, 2023, from https://github.com/orbitjs

Orbitjs/orbit on GitHub. (2023). [TypeScript]. Orbit.js. https://github.com/orbitjs/orbit (Original work published 2013)

Orbitjs/react-orbit on GitHub. (2023). [JavaScript]. Orbit.js. https://github.com/orbitjs/react-orbit (Original work published 2019)

Orbit.js—The Universal Data Layer | Orbit.js. (n.d.). Retrieved May 22, 2023, from https://orbitjs.com/

Overview | TanStack Query Docs. (n.d.). Retrieved May 22, 2023, from https://tanstack.com/query/v4/docs/overview

Query Functions | TanStack Query Docs. (n.d.). Retrieved May 22, 2023, from https://tanstack.com/query/v4/docs/guides/query-functions

Query Invalidation | TanStack Query Docs. (n.d.). Retrieved May 22, 2023, from https://tanstack.com/query/v4/docs/guides/query-invalidation

QueryCache | TanStack Query Docs. (n.d.). Retrieved May 22, 2023, from https://tanstack.com/query/v4/docs/reference/QueryCache

QueryClient | TanStack Query Docs. (n.d.). Retrieved May 22, 2023, from https://tanstack.com/query/v4/docs/reference/QueryClient

QueryClient#cancelQueries | TanStack Query Docs. (n.d.). Retrieved May 22, 2023, from https://tanstack.com/query/v4/docs/reference/QueryClient#queryclientcancelqueries

QueryClientConfig in @tanstack/query-core on GitHub. (n.d.). GitHub. Retrieved May 22, 2023, from https://github.com/TanStack/query/tree/v4.16.1/packages/query-corehttps://github.com/TanStack/query/tree/v4.16.1/packages/query-core/src/types.ts#L708-L713

QueryClient#invalidateQueries | TanStack Query Docs. (n.d.). Retrieved May 22, 2023, from https://tanstack.com/query/v4/docs/reference/QueryClient#queryclientinvalidatequeries

QueryClient#refetchQueries | TanStack Query Docs. (n.d.). Retrieved May 22, 2023, from https://tanstack.com/query/v4/docs/reference/QueryClient#queryclientrefetchqueries

QueryClient#setQueryData | TanStack Query Docs. (n.d.). Retrieved May 22, 2023, from https://tanstack.com/query/v4/docs/reference/QueryClient#queryclientsetquerydata

QueryMeta in @tanstack/query-core on GitHub. (n.d.). GitHub. Retrieved May 22, 2023, from https://github.com/TanStack/query/tree/v4.16.1/packages/query-corehttps://github.com/TanStack/query/tree/v4.16.1/packages/query-core/src/types.ts#L51-L53

QueryObserver | TanStack Query Docs. (n.d.). Retrieved May 26, 2023, from https://tanstack.com/query/v4/docs/reference/QueryObserver

React. (n.d.). Retrieved May 22, 2023, from https://react.dev/

React Hooks for Data Fetching – SWR. (2023, May 9). https://swr.vercel.app/

React Query | TanStack Query Docs. (n.d.). Retrieved May 26, 2023, from https://tanstack.com/query/v4/docs/react/overview

Rhashimoto on GitHub. (n.d.). GitHub. Retrieved May 22, 2023, from https://github.com/rhashimoto

Slack is your productivity platform. (n.d.). Slack. Retrieved May 26, 2023, from https://slack.com

Solid Query | TanStack Query Docs. (n.d.). Retrieved May 26, 2023, from https://tanstack.com/query/v4/docs/solid/overview

SolidJS. (n.d.). Retrieved May 22, 2023, from https://www.solidjs.com

SQLite Home Page. (n.d.). Retrieved May 22, 2023, from https://sqlite.org/index.html

State of JavaScript 2022: Other Tools. (n.d.). Retrieved May 4, 2023, from https://2022.stateofjs.com/en-US/other-tools/

Svelte • Cybernetically enhanced web apps. (n.d.). Retrieved May 22, 2023, from https://svelte.dev/

Svelte Query | TanStack Query Docs. (n.d.). Retrieved May 26, 2023, from https://tanstack.com/query/v4/docs/svelte/overview

Tannerlinsley.com. (n.d.). Retrieved May 22, 2023, from https://tannerlinsley.com/

TanStack Query. (n.d.). Retrieved May 4, 2023, from https://tanstack.com/query

TanStack/query on GitHub. (2023). [TypeScript]. TanStack. https://github.com/TanStack/query (Original work published 2019)

@tanstack/query-core on GitHub. (n.d.). GitHub. Retrieved May 22, 2023, from https://github.com/TanStack/query/tree/v4.16.1/packages/query-core

Telegram – a new era of messaging. (n.d.). Telegram. Retrieved May 26, 2023, from https://telegram.org/

TypeScript: JavaScript With Syntax For Types. (n.d.). Retrieved May 22, 2023, from https://www.typescriptlang.org/

Updates from Mutation Responses | TanStack Query Docs. (n.d.). Retrieved May 22, 2023, from https://tanstack.com/query/v4/docs/guides/updates-from-mutation-responses

UseEffect – React. (n.d.). Retrieved May 22, 2023, from https://react.dev/reference/react/useEffect

UseInfiniteQuery | TanStack Query Docs. (n.d.). Retrieved May 26, 2023, from https://tanstack.com/query/v4/docs/react/reference/useInfiniteQuery

UseQuery | TanStack Query Docs. (n.d.). Retrieved May 26, 2023, from https://tanstack.com/query/v4/docs/react/reference/useQuery

UseQueryClient | TanStack Query Docs. (n.d.). Retrieved May 26, 2023, from https://tanstack.com/query/v4/docs/react/reference/useQueryClient

Vue Query | TanStack Query Docs. (n.d.). Retrieved May 26, 2023, from https://tanstack.com/query/v4/docs/vue/overview

Vue.js—The Progressive JavaScript Framework | Vue.js. (n.d.). Retrieved May 22, 2023, from https://vuejs.org/

WebAssembly. (n.d.). Retrieved May 22, 2023, from https://webassembly.org/

WebSocket—Web APIs | MDN. (2023, March 16). https://developer.mozilla.org/en-US/docs/Web/API/WebSocket

WhatsApp. (n.d.). WhatsApp.Com. Retrieved May 26, 2023, from https://www.whatsapp.com

12 Appendices

12.1 Appendix #1: Summary in Swedish

Introduktion

Det här examensarbetet handlar om att komma på ett sätt att få TanStack Query att kännas mera som Cloud Firestore:s client-side SDK.

Cloud Firestore:s client-side SDK erbjuder ett oöverträffat gränssnitt, i mån om elegans, för att hantera ”realtidsdata” i client-side kod. Att kombinera Cloud Firestore:s client-side SDK:s positiva egenskaper med TanStack Query kunde hypotetiskt lösa det största problemet jag stöter på då jag utvecklar med TanStack Query.

Det största problemet jag stöter på då jag utvecklar med TanStack Query är att om jag väljer bort revalidering, så måste jag skriva stora mängder bräcklig query cache uppdateringskod som är starkt kopplad till både applikationens datamodell samt dess användargränssnitt. Jag inser att en pusselbit saknas här — det behövs en lösning som kan ta hand om query cache uppdatering på ett generellt och robust sätt. Min hypotes är att receptet för en sådan lösning kunde härledas från Cloud Firestore:s client-side SDK.

En webbsideutgåva av examensarbetet finns på https://danielgiljam.com/degree-thesis. Den erbjuder bästa läsupplevelsen.

Källkoden för webbsideutgåvan av examensarbetet, samt exemplen och biblioteken som producerats som resultat av examensarbetet, finns tillgängligt på GitHub: https://github.com/DanielGiljam/tanstack-query-with-orbitjs.

Förutsättningar och avgränsningar

Jag utgår från att läsaren är bekant med samtida webbapplikationsutvecklingspraxis, -mönster, -ramverk, -bibliotek, och -verktyg, och att läsaren har erfarenhet med att använda TanStack Query eller liknande bibliotek, som till exempel SWR från Vercel.

Avgränsning #1

En utgångspunkt i detta examensarbete är att TanStack Query är en orubblig del av frontend stack:en, så lösningen kan inte vara att inte använda TanStack Query överhuvudtaget. Orsaken till detta är att TanStack Query överlag har flera fördelar än nackdelar och att biblioteket tillför mera värde till utvecklare än vad det stjäl värde från dem i form av vassa kanter, såsom query cache uppdatering. Men att redogöra för för- och nackdelar med TanStack Query är ett helt skilt arbete i sig och utanför ramen för det här arbetet och definitivt inte syftet med det här arbetet.

Avgränsning #2

En utgångspunkt i det här examensarbetet att man inte får använda Cloud Firestore som del av stack:en. Lika som med utgångspunkten gällande TanStack Query, är det inte syftet med det här arbetet att motivera varför Cloud Firestore inte får vara del av stack:en — den motiveringen kunde vara ett helt skilt arbete i sig — men kort sagt är orsaken att Cloud Firestore är en icke-standard, proprietär molndatabaslösning från Google och detta sätter käppar i hjulet då det kommer till budget-, regelverk- och kompatibilitetskrav i många projekt eller att Cloud Firestores proprietära natur går i konflikt med värden som projektet står bakom, om projektet till exempel står bakom värden som öppna teknologistandarder och öppen källkod.

Bakgrund

TanStack Query

TanStack Query (tidigare känt som React Query) är ett populärt open-source JavaScript-bibliotek från Tanner Linsley. Bibliotekets funktion är att hantera asynkrona läs- och skrivoperationer och återge deras tillstånd i ett användargränssnitt.

Cloud Firestore:s client-side SDK

Cloud Firestore är en ”realtidsdatabas” som är en del av Googles ”backend-as-a-service” plattform Firebase. Cloud Firestore, liksom dess föregångare Firebase Realtime Database, har ett mycket elegant gränssnitt för att hantera ”realtidsdata”, nämligen `onSnapshot` API, som låter en lyssna till när resultatet på en query ändras.

Problem och forskningsfråga

Det största problemet jag stöter på då jag utvecklar med TanStack Query är att om jag väljer bort revalidering, så måste jag skriva stora mängder bräcklig query cache uppdateringskod som är starkt kopplad till både applikationens datamodell samt dess användargränssnitt. Jag inser att en pusselbit saknas här — det behövs en lösning som kan ta hand om query cache uppdatering på ett generellt och robust sätt.

Query cache uppdaterare är jobbiga, svåra och farliga att skriva. Jobbiga, för att de kräver mycket boilerplate kod och de är starkt kopplade med både applikationens datamodell och dess användargränssnitt. Svåra, för att man måste beakta samtidighet (”concurrency”) då man skriver dem och för att man måste ha en djup förståelse av hur TanStack Query fungerar för att kunna beakta alla permutationer av vad omgivande tillstånd potentiellt är då query cache uppdateraren körs. Farliga, för att sannolikheten att man introducerar buggar då man skriver dem är hög och utvecklingstakten lider som följd av mängden underhåll som query cache uppdaterare kräver.

Query cache uppdaterare är termen jag använder för att hänvisa till funktioner som uppdaterar `QueryCache`. Query cache uppdaterare behövs då man vill uppdatera resultatet av en query utan att den underliggande asynkrona läsoperationen som producerade query:ns resultat i första fallet körs igen. Med andra ord, man vill modifiera det cache:ade query resultatet. Till exempel, i en chattapplikation, är det logiskt att använda query cache uppdaterare i samband med hantering av WebSocket meddelanden, till exempel för att uppdatera resultatet för query:n på listan på chattmeddelanden, för att lägga till ett nytt chattmeddelande, då det anländer ett WebSocket meddelande som informerar klienten om ett nytt chattmeddelande.

Att revalidera query:s, det vill säga att köra den underliggande asynkrona läsoperationen igen för att få ett uppdaterat query resultat, är inte hållbart mönster i större och mer komplicerade webbapplikationer. All onödig beräkning och bandbreddsanvändning som detta tillvägagångssätt innebär leder till allvarliga prestandaproblem som i högsta grad är synliga för slutanvändaren. Därför behövs query cache uppdaterare, speciellt i större och mer komplicerade webbapplikationer.

Forskningsfrågor

FF1: Hur kan man hantera query cache uppdatering på ett generellt och robust sätt?

Mål

Ta fram en lösning som kan hantera query cache uppdatering på ett generellt och robust sätt.

Generellt som i att koppla loss (minska på starka kopplingar) och minska på mängden boilerplate som varje utvecklare behöver skriva. Robust som i att gömma komplexiteten och skriva det en gång väldigt noggrant och korrekt.

Hypotes

Receptet för lösningen kan härledas från Cloud Firestore:s client-side SDK.

Metod

Min metod består av två delar varav varje del kan brytas ner i tre delsteg.

1. Forskning
    a. Hitta existerande lösningar
    b. Jämför existerande lösningar
    c. Besluta till vilken mån lösningen på problemet jag försöker lösa kan bygga på existerande lösningar
2. Implementation
    a. Utveckla en exempelapplikation som demonstrerar problemet
    b. Utveckla lösningen i exempelapplikationen
    c. Extrahera lösningen från exempelapplikationen till ett separat bibliotek

Resultat

Lösningen som jag kommer fram till och väljer att föreskriva i mitt arbete, är att använda Orbit.js, ett komponerbart dataramverk för ambitiösa webbapplikationer. Anledningen till att lösningen är att använda Orbit.js är att Orbit.js i princip är ett ORM-bibliotek. ORM står för

Object Relational Mapping vilket är en teknik där man gör relationsdata tillgängligt i ett objektorienterat programmeringsspråk. Jag härledde på basen av min hypotes att ett potentiellt sätt att hantera query cache uppdatering på ett generellt och robust sätt är med hjälp av ett ORM-bibliotek.

Andra resultat av mitt arbete är:

- `@tanstack-query-with-orbitjs/core` och `@tanstack-query-with-orbitjs/react`
  Bibliotek som jag utvecklat för att använda TanStack Query tillsammans med Orbit.js.
- `example-chat-application-with-tanstack-query`
  Exempelapplikation som demonstrerar problemet med query cache uppdaterare.
- `example-chat-application-with-tanstack-query-and-orbitjs`
  Exempelapplikation som demonstrerar lösningen att använda TanStack Query och Orbit.js tillsammans med hjälp av mitt bibliotek för att integrera de två.

Slutsats

Koden för query cache uppdaterare blev mera expressiv än innan.

Koden för query cache uppdaterare blev inte så mycket kortare än innan som jag hade förväntat mig att den skulle ha blivit.

Ambitiösa webbapplikationer behöver en lokal backend, det vill säga, i ambitiösa webbapplikationer kan det löna sig att fördela ansvaret till den grad att frontendutvecklarens mentala modell inte är att användargränssnittskoden kallar på backenden direkt, utan istället att användargränssnittskoden kallar på en lokal backend som i sin tur kallar på backenden.

Vidare utveckling kunde vara att utforska hur SQLite som WebAssembly modul kunde vara en hörnsten i en lokal backend.